Linux Kernel, Board Support and Driver Development — BeaglePlay variant

Root Commit

June 13, 2026

Contents

1 How to use this document **6**

1.1 Copying code snippets . . . 6

2 Licensing information **7**

2.1 License . . . 7

2.2 Sources . . . 7

3 Preparing Your Environment **8**

3.1 Goals . . . 8

3.2 PC performance check . . . 8

3.3 GNU/Linux distribution check . . . 8

3.4 Matrix channel check . . . 8

3.5 Download and extract lab archive . . . 8

4 BeaglePlay Setup **9**

4.1 Goals . . . 9

4.2 Connect the serial line . . . 9

4.3 Access the serial line . . . 9

4.4 Terminal emulator . . . 10

4.5 Power-up board and access U-Boot . . . 10

4.5.1 Flashing the bootloader from recovery mode . . . 10

4.6 Networking setup . . . 11

4.6.1 Direct connection to your PC . . . 11

4.6.2 Networking setup on the PC side . . . 12

4.6.3 Networking setup on the board side . . . 12

4.7 tftp communication . . . 13

5 Fetching Kernel Sources **14**

5.1 Goals . . . 14

5.2 Install required packages . . . 14

5.3 Git configuration . . . 14

5.4 Cloning Git kernel sources . . . 14

6 Exploring Kernel Source Code **15**

6.1 Goals . . . 15
6.2 Setup . . . 15
6.3 Check version for the default branch . . . 15
6.4 Choose a specific kernel version . . . 15
6.5 Find names of kernel maintainers . . . 15
6.6 Exploring sources manually . . . 16
6.7 Exploring sources with VS Code (optional) . . . 16
6.8 Exploring sources with the Elixir Cross Referencer . . . 16

7 Kernel Configuration and Compiling 17
7.1 Goals . . . 17
7.2 Environment setup . . . 17
7.3 Kernel configuration . . . 17
7.4 Kernel compiling . . . 17

8 Kernel Booting 18
8.1 Goals . . . 18
8.2 Loading and booting the kernel . . . 18
8.3 Automating the boot sequence . . . 19
8.4 Booting the board through NFS . . . 19
8.4.1 Server setup . . . 19
8.4.2 Client setup . . . 20
8.5 Save kernel configuration . . . 20

9 Writing modules 21
9.1 Goals . . . 21
9.2 Setup . . . 21
9.3 Writing a module . . . 21
9.4 Building your module . . . 21
9.5 Testing your module . . . 22
9.6 Adding a parameter to your module . . . 22
9.7 Adding time information . . . 22
9.8 Following Linux coding standards . . . 22
9.9 Adding the hello_version module to the kernel sources . . . 22
9.10 Create a kernel patch . . . 23

10 Describing Hardware — LEDs 24
10.1 Goals . . . 24
10.2 Setup . . . 24
10.3 Create a custom device tree . . . 24
10.4 Setting the board's model name . . . 24
10.5 Driving LEDs . . . 25
10.5.1 Device Tree properties . . . 25

10.5.2 /sys interface for LEDs . . . 25
10.5.3 Using labels and phandles . . . 25
10.6 Saving changes . . . 26

11 Describing Hardware Devices — I2C **27**
11.1 Download documentation . . . 27
11.2 Managing I2C buses and devices . . . 27
11.2.1 Available buses . . . 27
11.3 Configuring pin muxing . . . 29
11.3.1 Probing the different buses . . . 29
11.3.2 Understanding `i2c-5` pin muxing settings . . . 30
11.4 Detecting I2c devices on i2c5 . . . 30
11.5 Add DT description for the device . . . 32
11.5.1 Generic guidelines . . . 32
11.5.2 Declare the gamepad device . . . 32

12 Basic I2C driver **33**
12.1 Objectives . . . 33
12.2 Create I2C driver . . . 33
12.3 Driver loading tests . . . 33
12.4 Polishing code . . . 33

13 I2C Communication **34**
13.1 Objectives . . . 34
13.2 Gamepad initialization . . . 34
13.2.1 How to interact with the device . . . 34
13.2.2 Routine to write a register . . . 35
13.2.3 Routine to write a single byte . . . 35
13.2.4 Add defines . . . 36
13.2.5 Device initialization code . . . 37
13.3 Reading device data . . . 37

14 Input Interface **38**
14.1 Goals . . . 38
14.2 Input and joystick interface support . . . 38
14.3 Allocate and register an input interface . . . 38
14.4 Add missing `struct input_dev` fields . . . 38
14.5 Implement polling function . . . 39
14.5.1 Register polling function . . . 39
14.5.2 Need for pointers between structures . . . 39
14.5.3 Polling function code . . . 40
14.6 Cleaning up . . . 41
14.7 Testing . . . 41

14.7.1 Testing with `evtest` . . . 41
14.7.2 Testing with games . . . 41

15 Accessing I/O memory and ports **42**
15.1 Goals . . . 42
15.2 Setup . . . 42
15.3 Add UART devices . . . 42
15.4 Operate a platform device driver . . . 43
15.5 Create a device private structure . . . 43
15.6 Get a base virtual address for your device registers . . . 44
15.7 Device initialization . . . 44
15.7.1 Accessing device registers . . . 44
15.7.2 Power management initialization . . . 44
15.7.3 Line and baud rate configuration . . . 45
15.7.4 FIFOs reset . . . 45
15.8 Standalone write routine . . . 45
15.9 Driver sanity check . . . 46

16 Output-only misc driver **47**
16.1 Objectives . . . 47
16.2 Misc driver registration . . . 47
16.3 Implement the write() routine . . . 48
16.4 Module reference counting . . . 49
16.5 Ioctl operations . . . 49

17 Sleeping and handling interrupts **51**
17.1 Goals . . . 51
17.2 Setup . . . 51
17.3 Register the handler . . . 51
17.4 Enable and filter the interrupts . . . 52
17.5 Sleeping, waking up and communication . . . 52

18 Concurrency prevention and locking **54**
18.1 Goals . . . 54
18.2 Concurrency issue . . . 54
18.3 Why does this happen? . . . 54
18.4 Making the write counter atomic . . . 55
18.5 Issue with the number of transmitted bytes . . . 55

19 Kernel debugging **57**
19.1 Goals . . . 57
19.2 Make debug messages more explicit . . . 57
19.3 dev_dbg() and dynamic debugging . . . 57

19.4 debugfs 58

20 Kernel crash analysis **59**

20.1 Goals 59

20.2 Setup 59

20.3 Analyzing the crash message 59

20.4 Locating the exact line where the error happens 59

1 How to use this document

1.1 Copying code snippets

This PDF document includes code snippets that you can copy and paste to source files.

Unfortunately, if you open this document with a viewer such as `evince` or `okular`, the pasted code may have extra linebreaks that you will have to repair manually.

This doesn't happen if you open the PDF document with a regular web browser such as `firefox` or `chromium`:

```
$ firefox document.pdf
```

2 Licensing information

This document was forked from Bootlin's training materials (https://bootlin.com/docs) in 2025 and now maintained independently.

2.1 License

2.2 Sources

LaTeX sources for this document are shared at https://gitlab.com/rootcommit/training-materials

3 Preparing Your Environment

3.1 Goals

Prepare your computer for the practical labs that follow

3.2 PC performance check

To avoid trouble and wasting a lot of time later make sure that your PC has:

- At least 2 physical CPU cores (4 or more are better)
- At least 8 GB of RAM
- At least 40 GB of free disk space
- High-speed connection to the Internet

Also double check that you have all the required hardware for the labs: https://rootcommit.com/training/linux-kernel-hardware/. Note that in an in-person setting, the electronic board and its accessories are provided by the instructor, and not all at once.

3.3 GNU/Linux distribution check

Make sure that you're using a distribution that's supported by the Yocto Project: https://docs.yoctoproject.org/ref-manual/system-requirements.html#supported-linux-distributions

This list is a bit arbitrary since we won't use the Yocto Project, but it corresponds to a list of well supported GNU/Linux distributions.

3.4 Matrix channel check

At this point, you should also be connected to the session's Matrix channel, if you are attending an online course. You should have received instructions when your participation to our course was confirmed, but your instructor can still help you complete this step.

3.5 Download and extract lab archive

You need a directory structure for the labs, which contains a few ready-made files for your work:

```
cd $HOME
wget https://rootcommit.com/pub/training/linux-kernel/kernel-labs.tar.xz
tar xf kernel-data.tar.xz
```

You are now ready to run the next labs.

4 BeaglePlay Setup

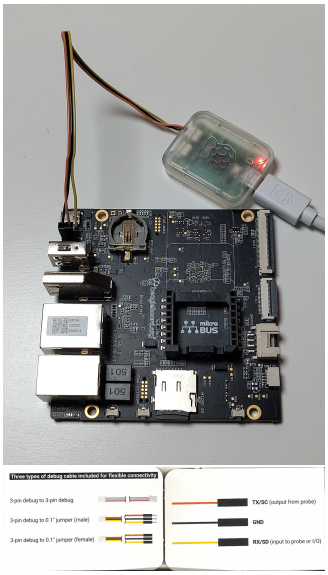
4.1 Goals

The goals of this lab are to prepare your board and PC to allow to:

- Access the serial console on the board
- Boot the board on the U-Boot bootloader
- Set up a tftp server on your PC, allowing your board to load binaries over the network.

4.2 Connect the serial line

The first thing to do is to connect the board UART port (between the USB-C and USB-A) port. This way, we will have access to the first messages from the bootloader and from the kernel, and will be able to interact with the system even before we set up networking.



You can either use a regular USB-serial dongle, or use the Raspberry Pi Debug Probe if you have one. In any case, you should:

- Connect the `GND` pin of the cable to the `GND` pin of the board.
- Connect the `TX` pin of the cable to the `RX` pin of the board (UART pins are crossed!).
- Connect the `RX` pin of the cable to the `TX` pin of the board.

Later, when we boot the board, if you don't get anything on the line, it's probably because you swapped the signal pins.

4.3 Access the serial line

Now connect your USB-serial cable to your PC, and run the `sudo dmesg` command to check that a new serial device file has appeared:

For the Raspberry Pi Debug Probe, you should have a `/dev/ttyACMx` device file:

```
[311822.061148] cdc_acm 5-1.4.3:1.1: ttyACM0: USB ACM device
```

And for a more traditional USB-serial dongle, you're likely to get a `/dev/ttyUSBx` device file:

```
[311990.458934] usb 5-1.4.4: pl2303 converter now attached
to ttyUSB0
```

Now, you want to have permission to access this file as a regular user. It's nice not to need `sudo` to access serial lines. So, check the UNIX group for this file:

```
ls -la /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 0 Mar 14 06:43 /dev/ttyUSB0
```

Here on Ubuntu, we need to add our user to the `dialout` group:

```
sudo adduser $USER dialout
```

At least in Ubuntu 24.04, you have to reboot for the change to be effective. In theory, logging out and logging back in should be sufficient, so that could work with other distributions.

4.4 Terminal emulator

You need a terminal emulator program to use the serial line. One of the easiest ones is `tio`:

```
sudo apt install tio
```

Then, just pass `tio` the name of your serial device file:

```
tio /dev/ttyACM0
```

Hit `[Ctrl][t] + [q]` whenever you want to exit `tio`.

4.5 Power-up board and access U-Boot

Power up the board a USB-C cable connected to a computer or to a USB power supply.

⚠ **Don't connect the board to a USB hub that doesn't have its own power supply. This may not supply enough power to the board.**

In the terminal running `tio`, you should see U-Boot booting.

You can access the U-Boot shell before the board continues booting to Linux by pressing a key in the terminal.

The U-Boot prompt should look like:

```
=>
```

Now, check that the below command works fine:

```
=> saveenv
```

If it doesn't, it's probably because you are doing the labs on your own, and the bootloader hasn't been adapted for you by your instructor. If that's the case, go through the new section. Otherwise, skip it.

4.5.1 Flashing the bootloader from recovery mode

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 ~/kernel-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
```

You can ignore 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
```

You should see the progress of the copy on the serial console. Be patient!

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

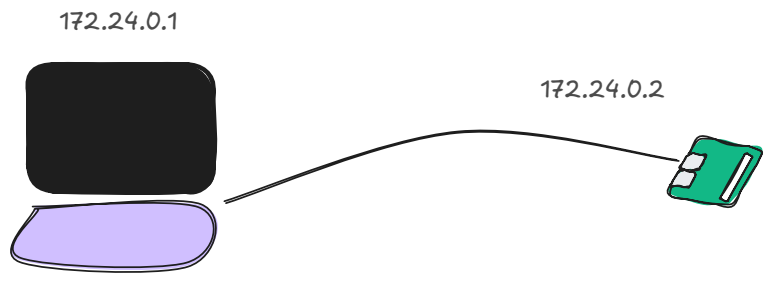
4.6 Networking setup

In this section, we are going to connect the board to our PC.

4.6.1 Direct connection to your PC

This solution should always be available if you have a spare Ethernet port on your PC (typically if it is connected to the Internet via WLAN), or if you have a USB to Ethernet adapter (part of the recommended accessories for this course).

Since we're going to connect the board directly to the PC, there will be no other networking equipment, no DHCP server assigning dynamic IP addresses, so we will need to use arbitrary static IP addresses:



⚠ **The IP address and subnet that you choose must not collide with the address range of your local network. Otherwise, you may end up with two different machines having the same IP address, which causes very hard to debug networking issues. Sometimes packets go through, sometimes they don't... and for a good reason!**

4.6.2 Networking setup on the PC side

The first thing is to find the name the corresponding networking device. Type the `ip a` command to list all networking devices on your system:

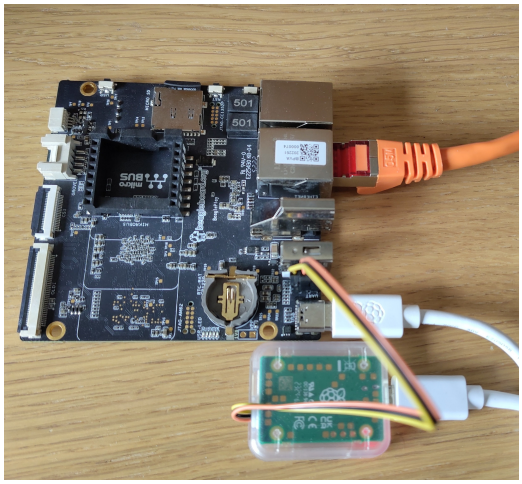
- `enp` devices correspond to internal network cards (`p` like PCI)
- `enx<macaddr>` correspond to USB networking dongles
- `wl` devices correspond to Wireless LAN adapters

Now that your network interface is identified, assign a static IPv4 address to it. Use the below command, changing the IP address and network interface name:

```
nmcli con add type ethernet ifname enx6c1ff71acf05 ip4 172.24.0.1/24
```

4.6.3 Networking setup on the board side

Connect your board to your PC through an Ethernet cable.



U-Boot supports networking by setting specific environment variables:

- `ipaddr`: IP address of the board running U-Boot
- `serverip`: IP address of servers U-Boot will connect to

So, let's set these variables according to our IP settings:

```
=> setenv ipaddr 172.24.0.2
=> setenv serverip 172.24.0.1
=> saveenv
```

Still in the U-Boot shell, let's check the communication:

```
=> ping 172.24.0.1
```

ℹ You won't be able to ping your board from your PC. By default, U-Boot has a very lightweight networking stack that doesn't support replying to pings from other machines.

4.7 tftp communication

We are going install a TFTP server to let the board download files generated on your PC:

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

Now, create a simple `hello.txt` file under `/srv/tftp/` and try to download it from U-Boot on your board:

```
=> tftp 0x80000000 hello.txt
```

If this works, you can check dump the RAM contents at the specified loading address:

```
=> md 0x80000000
80000000: 6c6c6548 6f57206f 0a646c72 ffff0000    Hello World.....
80000010: ffffffff ffffffff ffffffff ffffffff    ................
80000020: ffffffff ffffffff ffffffff ffffffff    ................
80000030: ffffffff ffffffff ffffffff ffffffff    ................
80000040: ffffffff ffffffff ffffffff ffffffff    ................
80000050: ffffffff ffffffff ffffffff ffffffff    ................
80000060: ffffffff ffffffff ffffffff ffffffff    ................
80000070: ffffffff ffffffff ffffffff ffffffff    ................
80000080: ffffffff ffffffff ffffffff ffffffff    ................
80000090: ffffffff ffffffff ffffffff ffffffff    ................
800000a0: ffffffff ffffffff ffffffff ffffffff    ................
800000b0: ffffffff ffffffff ffffffff ffffffff    ................
800000c0: ffffffff ffffffff ffffffff ffffffff    ................
800000d0: ffffffff ffffffff ffffffff ffffffff    ................
800000e0: ffffffff ffffffff ffffffff ffffffff    ................
800000f0: ffffffff ffffffff ffffffff ffffffff    ................
```

5 Fetching Kernel Sources

5.1 Goals

Download the Git sources for the Linux kernel.

5.2 Install required packages

We need to install Git to fetch the sources of the Linux kernel. We could have worked with a tar archive of the sources, but it wouldn't have allowed us to create our own branches and update them according to changes upstream.

```
sudo apt install git
```

5.3 Git configuration

As we are going to create git commits and eventually share them, it's important to associate your name and e-mail address to them.

Here is an example:

```
git config --global user.name "Isaac Newton"
git config --global user.email "isaac@apple.com"
```

Including a name and e-mail address in a commit (`git commit -s`) is actually required to send contributions to the Linux kernel.

5.4 Cloning Git kernel sources

Create the `$HOME/kernel-labs/src` directory and go into it.

Let's clone the stable kernel tree, as we want to work with kernel release branches:

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

Then wait... This corresponded to 5.4 GB of data to download when 6.15 was the latest version!

i If you are part of an on-site training course with shared and relatively slow Internet access, your instructor will share a `linux-git.tar.xz` archive to extract into `$HOME/kernel-labs`.

You are now ready to use the kernel sources.

6 Exploring Kernel Source Code

6.1 Goals

- Choose a kernel version
- Get familiar with kernel sources
- Look for information in sources

6.2 Setup

Stay in the Linux kernel source directory.

6.3 Check version for the default branch

Open the Makefile file to check the version corresponding to the default branch.

6.4 Choose a specific kernel version

In these practical labs, we are going to work with Linux 6.14.

Run the `git branch -a` command to find the name of the corresponding branch in the upstream tree, and check it out in your local copy:

```
git checkout -b 6.14 origin/<branch>
```

Check with exact Linux 6.14.y version you're at.

6.5 Find names of kernel maintainers

Look for the maintainer of the drivers/pinctrl/pinctrl-th1520.c file.

First, do this in a manual way, using information provided in the lectures.

Then, you can use the `get_maintainer.pl` script to perform the same search:

```
./scripts/get_maintainer.pl drivers/pinctrl/pinctrl-th1520.c
```

Note that the `get_maintainer.pl` script is more powerful than this. You can also give it a patch and it will find all the maintainers for the files touched by this patch.

We will get back to this later.

6.6 Exploring sources manually

Use Linux command line tools to solve these quick challenges:

- Look for an SVG diagram about the I2C bus, and view it.
- Find the number of `.c` and `.rs` (Rust) files in your currrent version.
- Count the number of files with a Microsoft copyright.
  Tip: use the `git grep` command instead of `grep`
- Look for the definition of the `request_irq()` function

Don't hesitate to ask your instructor for hints!

6.7 Exploring sources with VS Code (optional)

For this section, we need to install VS Code, if you don't have it yet:

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

Then, start VS code from inside the kernel sources:

```
code .
```

Install the recommended C/C++ extensions as suggested by the interface.

Then, open the `drivers/parport/parport_atari.c` file and look for:

- The definition of the `partport` structure
- The value of `ENODEV`
- The definition of the `local_irq_save()` macro
- You can also look for the definition of the `request_irq()` function, but unfortunately, it will be quite CPU intensive for VS Code to find out. The next solution will be easier.

6.8 Exploring sources with the Elixir Cross Referencer

Now, let's use the Elixir Cross Referencer! Go to https://elixir.bootlin.com and perform the below searches:

- Look for the definition of the `request_irq()` function again. Also look for its documentation!
- Open `drivers/parport/parport_atari.c` and look again for
  - The definition of the `local_irq_save()` macro
  - The value of `ENODEV`
- Look for the definition of the `CONFIG_DRM` configuration parameter and then follow links to find the definition and references of the `CONFIG_HDMI` parameter.

This solution is a great companion to understand the kernel code, in read-only mode only though.

Your instructor will demonstrate more features during the lectures too.

7 Kernel Configuration and Compiling

7.1 Goals

Let's configure and compile our kernel for the BeaglePlay board.

7.2 Environment setup

Stay in the `/kernel-labs/src/linux` directory.

First, install the packages needed to compile Linux with Clang:

```
sudo apt install llvm clang lld
```

Then, set the target architecture and cross-compiler for our board, knowing that the BeaglePlay is an ARM64 board.

To avoid issues changing terminals, we advise you to store these settings in your `~/.bashrc` file.

7.3 Kernel configuration

First apply the default configuration for the ARM64 kernel, called `defconfig`.

Then start the configuration interface of your choice, and set kernel settings as follows:

- In `Platform Selection`, disable support for all SoCs families except `Texas Instruments Inc. K3 multicore SoC architecture`
- Find the `DRM` symbol and disable it. It corresponds to display and GPU drivers. We don't need such functionality, and this will save a substantial amount of compile time.
- Also disable `LEDS_GPIO` which we will enable in due time.
- Make sure that your kernel is configured with `ROOT_NFS` as we will boot the kernel on an NFS exported root filesystem.

7.4 Kernel compiling

Start compiling your kernel, using all the CPU threads available on your host computer.

Then, compiling the kernel will take time. Let's get back to lectures in the meantime.

8 Kernel Booting

8.1 Goals

In this section, we mainly want to boot the kernel we've just built, on the BeaglePlay board. We will make it use a root filesystem shared with the PC host through NFS (Network File System).

8.2 Loading and booting the kernel

The first thing is to load the kernel and device tree on the board.

So, let's copy these generated files to `/srv/tftp`:

- `arch/arm64/boot/Image.gz`: compressed kernel
- `arch/arm64/boot/dts/ti/k3-am625-beagleplay.dtb`: board device tree

Then let's try to load the kernel in RAM:

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

This kernel is compressed, and on ARM64, the Linux kernel doesn't support decompressing the kernel, so this is up to the bootloader. So, let's ask U-Boot to decompress Linux at 0x82000000. That's 32 MB away from the address of the compressed kernel. As the compressed kernel is currently 10 MB big, that's beyond the end of the compressed kernel.

```
unzip 0x80000000 0x82000000
```

Last but not least, let's load the board device tree too:

```
tftp 0x86000000 k3-am625-beagleplay.dtb
```

`0x86000000` is 64 MB away from the start of the decompressed kernel. This should leave enough space even for bigger kernels.

Now let's try to boot our kernel, passing both the address of the uncompressed image and of the device tree:

```
booti 0x82000000 - 0x86000000
```

The kernel should boot on the serial console, but ultimately panic because no root filesystem has been specified yet:

```
[    2.405751] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
```

Anyway, that's good enough for now.

8.3 Automating the boot sequence

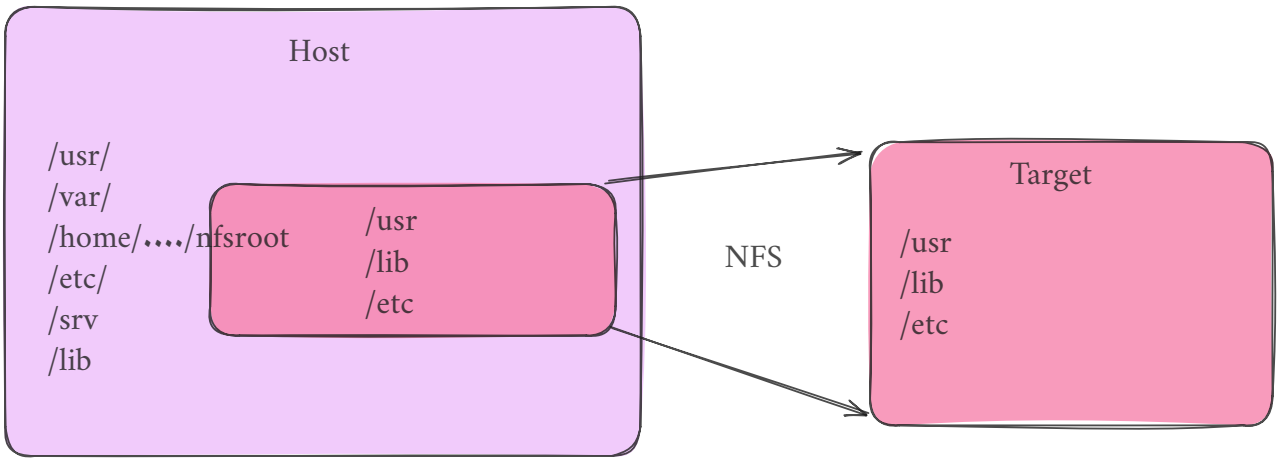
Reset the board and get back to U-Boot. We will define the `bootcmd` variable, which contains all the commands to run automatically at boot time, if the countdown is not interrupted:

```
setenv bootcmd 'tftp 0x80000000 Image.gz; unzip 0x80000000 0x82000000; tftp 0x86000000 k3-am625-beagleplay.dtb; booti 0x82000000 - 0x86000000'
saveenv
```

Reset the board and make sure that it boots Linux automatically now.

8.4 Booting the board through NFS

We are going to use the Network File System (NFS) to boot our board on a directory on the host PC, exported through the network:



8.4.1 Server setup

We first need to install a server package on the PC host:

```
sudo apt install nfs-kernel-server
```

We then need to create the directory to share with the board:

```
cd ~/kernel-labs/
mkdir nfsroot
cd nfsroot
tar xf ../data/nfsroot-genericarm64.rootfs.tar.xz
```

Then, we must explicitly allow this directory to be shared with the board, by adding this line to the `/etc/exports` file:

```
/home/<user>/kernel-labs/nfsroot 172.24.0.2(rw,no_root_squash,no_subtree_check)
```

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

Last but not least, we need to tell the NFS server to reload this file:

```
sudo exportfs -r
```

8.4.2 Client setup

On the NFS client (board) side, you have to set the kernel command line through the `bootargs` variable in U-Boot (set `bootargs` **in a single line**):

```
setenv bootargs root=/dev/nfs rw ip=172.24.0.2:::::eth0 console=ttyS2,115200
nfsroot=172.24.0.1:/home/<user>/kernel-labs/nfsroot,nfsvers=3,tcp
saveenv
```

Type `boot` or reset the board.

This time, it should boot to a Linux command line.

Log in a `root`, with an empty password.

Tip: If you have issues, a good way to investigate them is to look at `/var/log/syslog`. The server will let you know what was wrong with the client's request, and if there is no information from the server, it means that the client didn't even manage to contact it (networking problem?).

8.5 Save kernel configuration

In case you want to share your working kernel configuration with others, a good way is to store it as a new `defconfig` file in the kernel sources:

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

Then, people can reuse the same configuration by typing:

```
make beagleplay_defconfig
```

9 Writing modules

9.1 Goals

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.

9.2 Setup

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

9.3 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.

Tip: Have a look at init/version-timestamp.c. It contains all the details you need. In particular, you are interested in the "release" information which contains the actual kernel version number.

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.

9.4 Building your module

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

9.5 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.

9.6 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.

9.7 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!

9.8 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:

```
~/kernel-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`.

9.9 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 6.14
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 menuconfig` 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 ~/kernel-labs/src/linux
git status
git add -A
git commit -as
```

- `git add -A` adds new files and modified 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.

9.10 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 6.14
```

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

Now, check the patch for compliance against Linux coding rules:

```
./scripts/checkpatch.pl 0001-<description>patch
```

You may have to make changes to the sources files to keep `checkpatch.pl` happy:

```
git add <modified-files>
git commit --amend
```

And run `git format-patch` again, until `checkpatch.pl` is fully satisfied with your changes.

Tip: `checkpatch.pl` expects `Kconfig` help information to be at least 4 lines long!

10 Describing Hardware — LEDs

10.1 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.

10.2 Setup

Go to the `~/kernel-labs/src/linux` directory. Check out the `6.14` branch.

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

10.3 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` (`make dtbs` to just compile the DTBs). Now, copy the new DTB to the tftp server home directory, change the DTB file name in the U-Boot configuration[1], and boot the board.

10.4 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:

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

- 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.

10.5 Driving LEDs

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

10.5.1 Device Tree properties

Open the original BeaglePlay DTS and find where these LEDS are defined.

The five LEDs are actually supposed to be triggered by a driver matching the compatible `gpio-leds` string. This is a generic driver which supports LEDs connected to GPIOs. But as you can see, despite being part of the current Device Tree (see arch/arm64/boot/dts/ti/k3-am625-beagleplay.dts), the LEDs remain off. The reason for that is the absence of trigger for these nodes: nothing actually drives the LEDs even if they are described. So you can now recompile your kernel with `CONFIG_LEDS_GPIO=y`.

After rebooting, you should now see `USR_LED0` blink with the CPU activity, and the others staying off. If you look at the bindings documents Documentation/devicetree/bindings/leds/common.yaml you'll see the values we can assign to the `default-trigger` property to control the state of the LEDs.

10.5.2 /sys interface for LEDs

In particular, we'd like to use the `usb-host` trigger to show USB host activity on `LED_USR4`.

Before doing that, let's experiment with the userspace interface to change the triggers:

```
# cd /sys/class/leds/
# ls -l
```

From the subdirectory names, you can recognize the LEDS defined in the board device tree.

Therefore, `:wlan` must correspond to `led-4` in the DTS.

Go into the `:wlan` subdirectory and run:

```
# cat triggers
```

You can check that `usb-host` is not part of the triggers yet. No problem, add `CONFIG_USB_LED_TRIG=y` to your kernel and reboot.

The `usb-host` should now be available:

```
# cat /sys/class/leds/:wlan/trigger
```

Let's enable it:

```
# echo usb-host > /sys/class/leds/:wlan/trigger
```

Connect a USB device to the board, and you should see `LED_USR4` blink.

If that's a USB key, you can create intensive activity by reading the whole storage:

```
# cat /dev/sda > /dev/null
```

10.5.3 Using labels and phandles

How to make the setting permanent? That's where we need to modify the device tree. Unfortunately, the BeaglePlay DTS doesn't offer aliases for the LED nodes to refer to in an enclosing DTS.

So, having no choice but to modify a file maintained by other people, add such a label to `led-4` in the standard BeaglePlay DTS, and add a phandle to this label in your custom device tree.

Then, add the missing property that makes `led-4` blink according to USB host activity. Update your device tree and check that the board behaves as expected.

10.6 Saving changes

Commit your changes in your branch:

```
git status
git add -A .
git commit -as
```

Through the upcoming labs, we will get many more opportunities to tweak the Device Tree.

11 Describing Hardware Devices — I2C

11.1 Download documentation

Download the TI AM62x SoC Technical Reference Manual (TRM) available at https://www.ti.com/lit/ug/spruiv7a/spruiv7a.pdf. How many pages 😉?

Also download the SOC datasheet at https://www.ti.com/lit/gpn/am625 (`am625.pdf`). It is shorter (!) and contains information about pin assignments.

We will need these documents several times during our labs.

11.2 Managing I2C buses and devices

The next thing we want to do is connect an Adafruit Mini I2C Gamepad 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.

11.2.1 Available buses

Let's see which I2C buses Linux sees:

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

This shows 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`

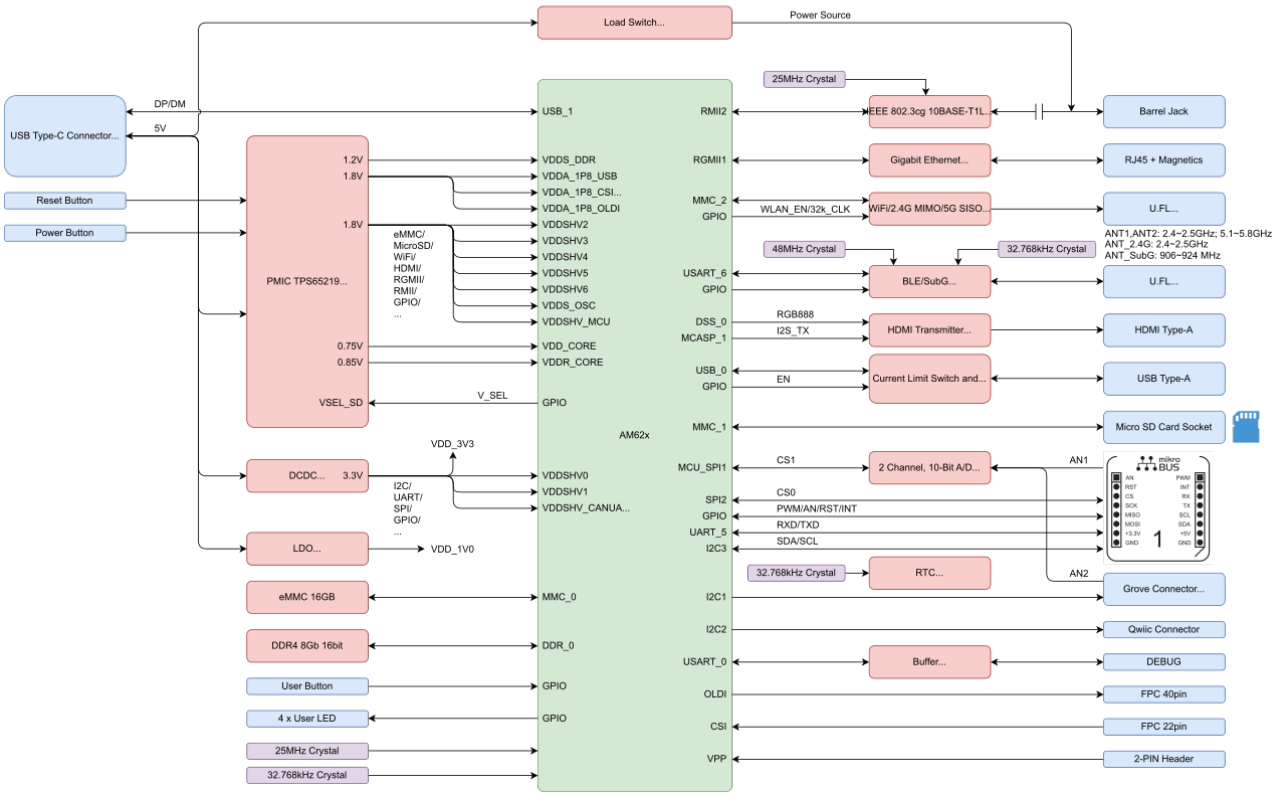
Now let's double check the addresses by looking at the SoC TRM (`spruiv7a.pdf`), in the `2.1 Memory Maps` section:

- `i2c-0`: `0x20000000` indeed corresponds to I2C0
- `i2c-1`: `0x20010000` indeed corresponds to I2C1
- `i2c-2`: `0x20020000` indeed corresponds to I2C2
- `i2c-3`: `0x20030000` indeed corresponds to I2C3
- `i2c-5`: `0x04900000` indeed corresponds to `MCU_I2C0`

Now, we want to use I2C from the QWIIC connector because it is easy to use and error proof.

If you look at this diagram from https://docs.beagleboard.org/boards/beagleplay/03-design.html, the QWIIC connector is connected to I2C2:

BeaglePlay System Block Diagram



However, this turns out to be wrong as described on https://forum.beagleboard.org/t/beagleplay-qwiic-connected-to-mcu-i2c0-not-i2c2/42185

So, we will try to use `i2c-5`, corresponding to `MCU_I2C0`.

11.3 Configuring pin muxing

11.3.1 Probing the different buses

The Beagle Play device tree already correctly configures the pin muxing state for the `i2c-5` bus. Before proceeding with this lab, we ask you to delete this pin muxing configuration by adding these lines to your custom device tree:

```
&mcu_i2c0 {
        /delete-property/ pinctrl-0;
        /delete-property/ pinctrl-names;
};
```

As you can see, the device tree language allows to delete existing properties. This works with nodes too (`/delete-node/`).

Update the device tree and reboot your board.

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 the tool uses 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
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-0 using receive byte commands.
I will probe address range 0x08-0x77.
Continue? [Y/n]
     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 `i2c-5` with `i2cdetect -r 5`.

You will see that the command will not fail but is very slow. That's because the corresponding signals are not exposed yet to the outside connectors through pin muxing. It seems the communication times out for each possible device address.

Now, remove the `/delete-property/` lines, and reboot the board with the update.

11.3.2 Understanding `i2c-5` pin muxing settings

Using the Elixir Cross Referencer, look for the value set to `pinctrl-0` in the board DTS: `i2c_qwiic_pins_default`:

```
i2c_qwiic_pins_default: i2c-qwiic-default-pins {
 pinctrl-single,pins = <
  AM62X_MCU_IOPAD(0x0044, PIN_INPUT, 0) /* (A8) MCU_I2C0_SCL */
  AM62X_MCU_IOPAD(0x0048, PIN_INPUT, 0) /* (D10) MCU_I2C0_SDA */
 >;
};
```

Here are details about the values:

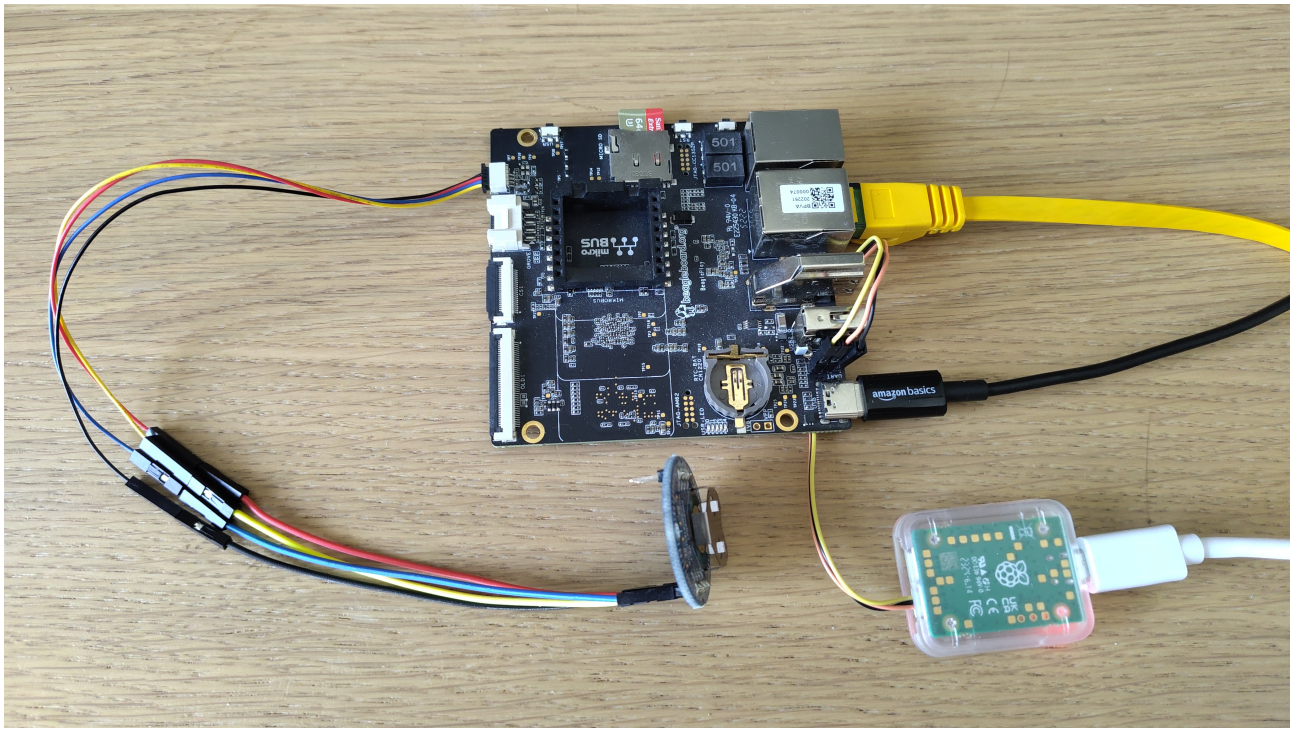
- `0x0044` and `0x0048` are the offsets of the pad configuration registers to control muxing on the corresponding external package pins (`A8` and `D10`)
- `MCU_I2C0_SCL` and `MCU_I2C0_SDA` are the names of the internal signals that need to be selected.
- Muxing mode 0, is set for both pins.
- `PIN_INPUT` declares the pins as input.

You can find all possible configurations by opening the SoC datasheet (`am625.pdf`), looking for `MCU_I2C0_SCL` and `MCU_I2C0_SDA`, in the `6-1. Pin Attributes` section.

11.4 Detecting I2c devices on i2c5

Run `i2c-detect -r 5` to detect devices connected to our bus. Nothing should be detected, this bus is for external devices only, and we haven't connected any device yet.

Now, before connecting our gamepad, let's connect another I2C device, like the Adafruit Chronodot V2.1 device provided by your instructor, or any other you have in stock:

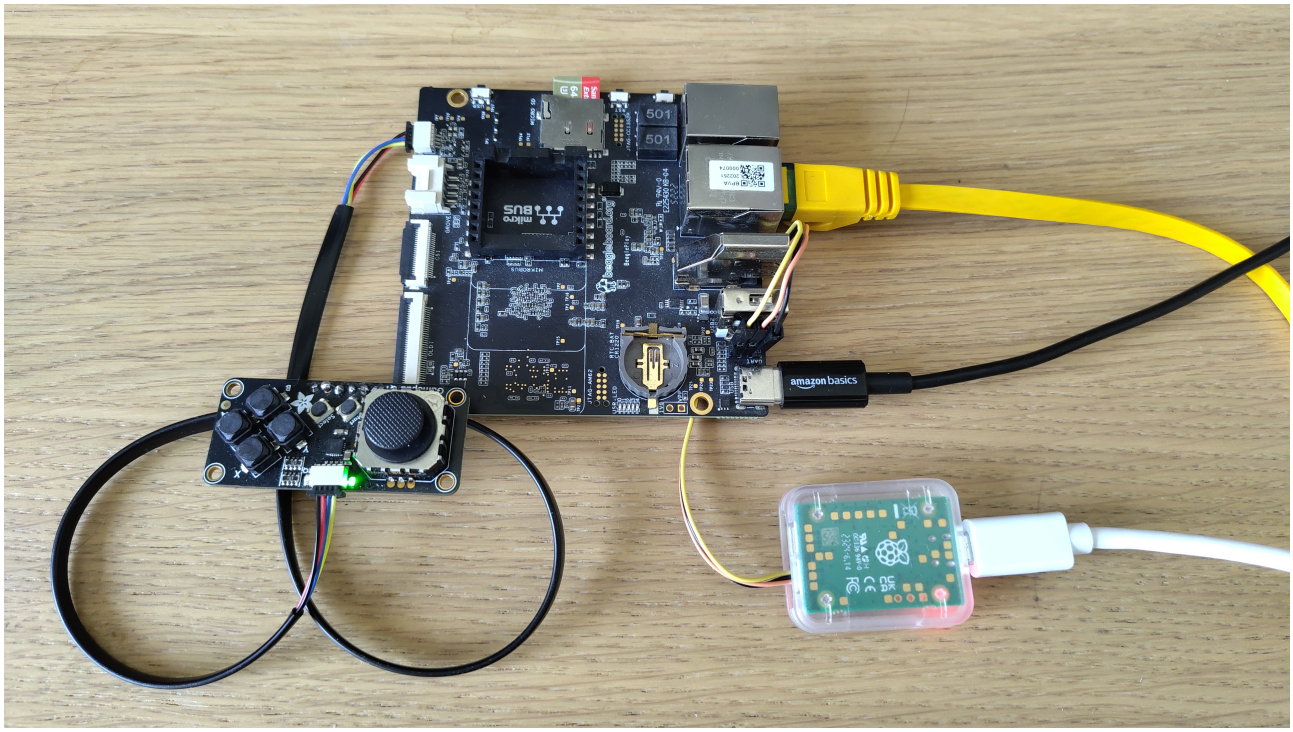



```
# i2cdetect -r 5
WARNING! This program can confuse your I2C bus, cause data loss and worse!
```

```
I will probe file /dev/i2c-5 using receive byte commands.
I will probe address range 0x08-0x77.
Continue? [Y/n]
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

This proves that the `i2c-5` bus works as expected and is able to detect devices.

Let's connect our gamepad now, using a 4 wire JST cable:



Let's try to detect it:

```
# i2cdetect -r 5
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-5 using receive byte commands.
I will probe address range 0x08-0x77.
Continue? [Y/n]
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Ouch. There is actually an I2C compliance issue here with this device, preventing detection with

`i2cdetect`. The device will work though, once we develop a Linux driver for it.

The issue has been reported on the Adafruit forums and on the linux-i2c mailing list: https://forums.adafruit.com/viewtopic.php?p=1052577.

11.5 Add DT description for the device

11.5.1 Generic guidelines

Before describing your gamepad 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 `adafruit,gamepad`.
- 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 gamepad. You will find the I2C slave address of the gamepad on the datasheet at https://cdn-learn.adafruit.com/downloads/pdf/gamepad-qt.pdf. In case the Adafruit website is down, the document is also available on https://rootcommit.com/pub/training/docs/gamepad-qt.pdf.

  This I2C slave address is enforced by the device itself. You can choose three other alternative addresses, but this involves soldering pads.

11.5.2 Declare the gamepad device

As a child node to the `mcu_i2c0` bus, now declare an I2C device for the gamepad, following the above rules.

You will also need to set the `clock-frequency` property of the bus to `400000`, as this is the best operating frequency for the gamepad device according to its datasheet.

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/bus@4000000/i2c@4900000/joystick@50
```

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 gamepad /tmp/dts
```

Once your new Device Tree seems correct, commit your changes.

12 Basic I2C driver

12.1 Objectives

Here is what we are going to achieve in this lab:

- Create a basic I2C driver for the gamepad device
- Make sure that the driver is bound to our device

12.2 Create I2C driver

Following examples shared in the lectures or other ones found in the kernel sources, let's create an I2C driver for the gamepad.

To do this, go to `nfsroot/home/root/gamepad/` and add code to `gamepad.c`:

- Define `probe()` and `remove()` functions with the prototype expected by the `struct i2c_driver` structure. For the moment, just fill these functions with calls to `pr_alert()`, to make sure that the functions are called when the driver is bound to a device or unbound from one.
- Define supported devices
- Create this `struct i2c_driver` structure
- Register an I2C driver with this structure.

12.3 Driver loading tests

Compile your driver, load it, unload it, and make sure that you receive the messages confirming that `probe()` and `remove()` were called.

If that's not the case, there must be a mismatch between the `compatible` string in the Device Tree and the one registered by the driver. It can also be an incorrect driver registration.

When the driver is loaded, list the contents of `/sys/bus/i2c/drivers/gamepad/`. You should see a link to your device. Conversely, if you go to `/sys/bus/i2c/devices/`, you should recognized your gamepad thanks to its `50` address. In the corresponding subdirectory, you will find a link back to your driver.

Driver and device(s) are indeed bound to each other.

By the way, you can run `i2cdetect -r 5` again and see the change.

12.4 Polishing code

Don't forget your good habits, and check up the style of your code:

```
~/kernel-labs/src/linux/scripts/checkpatch.pl -f gamepad.c
```

13 I2C Communication

13.1 Objectives

In this lab, we are going to expand the `probe()` function by:

- Using raw I2C commands to initialize the device
- Using raw I2C commands to access the states of the joystick and its buttons.

Actually, a full driver already exists in the Linux kernel for our gamepad device: drivers/input/joystick/adafruit-seesaw.c.

However, it is a bit more complicated than necessary, and from a learning perspective, it's always better to build our own, piece by piece, in a very progressive way, so that each step is well understood.

Therefore, we will reuse a few defines from the original code, but the code we will create will be simpler. However, when your driver is complete, you can then compare yours with the official one, and find room for improvement, in your code but hopefully in the official code too!

13.2 Gamepad initialization

First, remove the debug messages from the previous lab, from `probe()` and `remove()`.

13.2.1 How to interact with the device

It's good to give you the general picture about how the device works, and how you can write to it and read from it.

The device is typically controlled through commands at specific offets, whether the command is a control instruction or something that reads specific data from the device. In both cases, you first have to write the 16 bit offset of the command on the I2C bus. This can easily be done via the `i2c_master_send()` command.

One constraint is that the gamepad apparently works in Big Endian mode (most significant bytes first) while ARM systems are Little Endian by default. As a consequence, the offset has to be converted to Big Endian first before being passed to the device.

Then:

- For control commands, you will write one or more bytes on the I2C bus. Such bytes, if more than one, also have to be converted to Big Endian.
- For read commands, you will read a given number of bytes on the I2C bus, corresponding to the type of data you're supposed to get. The data you receive is Big Endian too, so you will also have to convert it back to the Little Endian format that your CPU uses.

We will implement read and write routines to make it easier to write our driver.

13.2.2 Routine to write a register

As all operations start by writing a register to the bus, let's add a function that does exactly this. Since this is the first time, and since you may not know how to convert to Big Endian, we give you this ready made file to add to your code (please copy and paste it!):

```
static int gamepad_write_register(struct i2c_client *client, u16 reg)
{
        int ret;
        char reg_buf[2];

        put_unaligned_be16(reg, reg_buf);
        ret = i2c_master_send(client, reg_buf, 2);
        return ret;
}
```

A few notes:

- `client` is the structure pointer received by `probe()`.
- `i2c_master_send()` expects a `char *` pointer, not a numerical offset. We're using `put_unaligned_be16()` function to convert the `reg` value to the `char *reg_buf` pointer, and take care of the conversion to Big Endian.
- Of course, it's very important to check the return value of `i2c_master_send()`. That's why we return it's return value, so that the calling function can check it.

Add this function to your code, and find the missing includes allowing to compile your code.

13.2.3 Routine to write a single byte

To give you a last example, here's a routine that writes a single byte at the specified register offset:

```
static int gamepad_write_u8(struct i2c_client *client, u16 reg, u8 value)
{

        int ret;
        char buf[1] = {value};

        ret = gamepad_write_register(client, reg);

        if (ret != 2)
                return ret < 0 ? ret : -EIO;

        ret = i2c_master_send(client, buf, 1);

        if (ret != 1)
                return ret < 0 ? ret : -EIO;

        return 0;
}
```

At you can see, it's important to check the return values of the I2C communication commands. The data transfer is only successful if the return value equals to the number of bytes to transmit. A negative error value indicates a complete failure, such as unavailability of the hardware or a permission issue. A non matching positive value indicates a partial transfer, but is still usually considered as an error.

The above function returns `0` in case of success. Otherwise, that's a failure that must be handled by the calling code.

Add this function to your code.

Similarly, create two extra functions with the below prototypes:

```
static int gamepad_write_u32(struct i2c_client *client, u16 reg, u32 value);
/* Writes 4 bytes at the spefied register offset */
/* Includes conversion of the value to be32 */

static int gamepad_read(struct i2c_client *client, u16 reg, void *buf, int count);
/* Reads count bytes from the specified offset into the specified buffer */
/* Returns raw big-endian data, to be converted outside of the function */
```

Write and compile these functions. We are then ready to use them to initialize the device and access device data.

13.2.4 Add defines

Let's add these definitions from the official driver:

```
#define SEESAW_BUTTON_A                 0x05
#define SEESAW_BUTTON_B                 0x01
#define SEESAW_BUTTON_X                 0x06
#define SEESAW_BUTTON_Y                 0x02
#define SEESAW_BUTTON_START             0x10
#define SEESAW_BUTTON_SELECT            0x00

#define SEESAW_GPIO_DIRCLR_BULK         0x0103
#define SEESAW_GPIO_BULK                0x0104
#define SEESAW_GPIO_BULK_SET            0x0105
#define SEESAW_GPIO_PULLENSET           0x010b

#define SEESAW_STATUS_SWRST             0x7f00

#define SEESAW_ANALOG_X                 0x0e
#define SEESAW_ANALOG_Y                 0x0f
#define SEESAW_ADC_BASE                 0x0900
#define SEESAW_ADC_OFFSET               0x07

#define SEESAW_JOYSTICK_MAX_AXIS        1023
#define SEESAW_JOYSTICK_FUZZ            20
#define SEESAW_JOYSTICK_FLAT            20

static const u32 SEESAW_BUTTON_MASK =
        BIT(SEESAW_BUTTON_A) | BIT(SEESAW_BUTTON_B) | BIT(SEESAW_BUTTON_X) |
        BIT(SEESAW_BUTTON_Y) | BIT(SEESAW_BUTTON_START) |
        BIT(SEESAW_BUTTON_SELECT);
```

Note: the two bytes in `SEESAW_STATUS_SWRST` were swapped. Not sure why this was necessary, as the code in the official driver looks good too.

13.2.5 Device initialization code

To initialize the device, add the below operations:

- Reset the device, by writing `0xff` at `SEESAW_STATUS_SWRST`
- Leave time for the device to reset: `usleep_range(10000, 15000);`
- Set pin mode to input and enable pull-up resistors:
  - Write `SEESAW_BUTTON_MASK` (32 bit) at `SEESAW_GPIO_DIRCLR_BULK`
  - Write `SEESAW_BUTTON_MASK` (32 bit) at `SEESAW_GPIO_PULLENSET`
  - Write `SEESAW_BUTTON_MASK` (32 bit) at `SEESAW_GPIO_BULK_SET`

Of course, don't forget to check the return values. Make sure that your code compile well, and load and remove your module to make sure everything works as planned.

13.3 Reading device data

It's now time to check our ability to read the state of the joystick and buttons.

For the moment, we will do it by printing values in the `probe()` function which is absolutely not a proper way to read device data, as this function is only called once when a device is bound to our driver. However, that will be good enough so far.

Add the below operations at the end of your `probe()` function:

- Read button status data into a `u32` variable at offset `SEESAW_GPIO_BULK`, convert to the CPU byte order (`be32_to_cpu()`) and get the state of each button by using a bitmask, for example for the A button:

  ```
  pr_alert("Button A: %d\n", (buttons_state & BIT(SEESAW_BUTTON_A)) ? 0 : 1);
  ```

- Read the X-axis value into a `u16` variable at offset `SEESAW_ADC_BASE | (SEESAW_ADC_OFFSET + SEESAW_ANALOG_X)`. After endianness conversion, you will also have to invert the value, so that left is the minimum, and right is the maximum. Do this by substracting the obtained value from the maximum possible value (`SEESAW_JOYSTICK_MAX_AXIS`).
- Read the Y-axis from a similar operation, though you won't have to invert the value.

To have enough time to try multiple states, you could read the values in a long enough `for` loop, containing a call to `usleep_range(100000, 150000);` to leave enough time between readings.

Compile your code, fix issues, and test that the code correctly reads the states of your device. Don't hesitate to ask your instructor for help whenever you get stuck, or whenever you have questions about the code.

Last but not least, watch your style with `checkpatch.pl` and fix the reported issues (except the absence of Device Tree bindings).

The next step is to make the state of the device available to user-space through standard interfaces. Stay tuned.

14 Input Interface

14.1 Goals

In this lab, we will:

- Expose device data through a propose user-space interface, the *input* subsystem
- Allocate and manage a device specific data structure, keeping references to the framwork (input) and bus (i2c) data structures.

14.2 Input and joystick interface support

To use the input interface for a device, the kernel needs to be compiled with `CONFIG_INPUT_EVDEV`, which should already be enabled. You also need to enable `CONFIG_INPUT_JOYDEV`. This last setting will only be needed at the end of the lab, when we make tests with games that rely on Linux' joystick interface.

Update your kernel and reboot your board.

14.3 Allocate and register an input interface

First, comment out the code that was printing the states of the device. We will reuse it soon.

Then, declare a pointer to an `struct input_dev` structure in your `probe()` routine.

Allocate this structure with `devm_input_allocate_device()` and register the input interface with `input_register_device()`.

Don't forget to check the return values as these operations could fail.

You should also think about the `remove()` routine. Fortunately, thanks to allocating the structure with a `devm_` function, it is automatically garbage collected with the device structure is destroyed. Furthermore, in the input subsystem, even the `struct input_dev` structure is automatically unregistered. So, there's nothing to add in `remove()`.

Now, compile and test your code.

Don't be surprised if you get with information message:

```
input: Unspecified device as /devices/platform/bus@f0000/bus@f0000:bus@4000000/
4900000.i2c/i2c-5/5-0050/input/input2
```

That's because we haven't filled some of the fields of the `struct input_dev` structure yet.

14.4 Add missing `struct input_dev` fields

Let's initialize the `struct input_dev` structure before registering it:

```
input->name = "Adafruit Mini I2C Gamepad";
input->id.bustype = BUS_I2C;

set_bit(EV_KEY, input->evbit);      // Mandatory to get key events
set_bit(BTN_SOUTH, input->keybit);
set_bit(BTN_WEST, input->keybit);
set_bit(BTN_NORTH, input->keybit);
set_bit(BTN_EAST, input->keybit);
set_bit(BTN_START, input->keybit);
set_bit(BTN_SELECT, input->keybit);

set_bit(ABS_X, input->absbit);
set_bit(ABS_Y, input->absbit);
input_set_abs_params(input, ABS_X, 0, SEESAW_JOYSTICK_MAX_AXIS,
  SEESAW_JOYSTICK_FUZZ, SEESAW_JOYSTICK_FLAT);
input_set_abs_params(input, ABS_Y, 0, SEESAW_JOYSTICK_MAX_AXIS,
  SEESAW_JOYSTICK_FUZZ, SEESAW_JOYSTICK_FLAT);
```

You will find details about the `input_set_abs_params()` function in `input/input-programming` in the kernel documentation.

Note that the `flat` argument is quite important here. It turns out that the values from the X and Y coordinates are not well centered when the joystick is idle, and a high value compensates for this, reporting slightly offset values as centered ones anyway. This avoids biases that can move the cursor left or right otherwise.

Recompile your code and you will see that `Unspecified device` has been replaced by `Adafruit Mini I2C Gamepad`.

14.5 Implement polling function

14.5.1 Register polling function

As through the I2C interface, we don't have access to interrupts to know when the state of the device has changed, we need to implement a polling function that will periodically read the state of the device and report it to the input subsystem.

So, create a `gamepad_poll` function, using the prototype expected by the `input_setup_polling()` function.

Associate it to the `struct input_dev` structure in the `probe()` routine using `input_setup_polling()`.

You also need to define the polling intervals, not too quick (the device may not be given enough time), not too slow (your device may not be polled often enough, which could degrade gameplay):

```
#define SEESAW_GAMEPAD_POLL_INTERVAL_MS 16
#define SEESAW_GAMEPAD_POLL_MIN         8
#define SEESAW_GAMEPAD_POLL_MAX         32

input_set_poll_interval(input, SEESAW_GAMEPAD_POLL_INTERVAL_MS);
input_set_max_poll_interval(input, SEESAW_GAMEPAD_POLL_MAX);
input_set_min_poll_interval(input, SEESAW_GAMEPAD_POLL_MIN);
```

14.5.2 Need for pointers between structures

Then, from this function, we need to access the I2C client structure to read the device data. However, this function will be called from the context of the input subsystem, not knowing anything about the specific bus the device is connected to.

That's why, in the `probe()` routine, we need to keep a reference to the I2C `struct i2c_client` pointer.

Fortunately, with all framework structures like `struct input_dev`, there's a way to add a device specific pointer. In the case of the input subsystem, we can use:

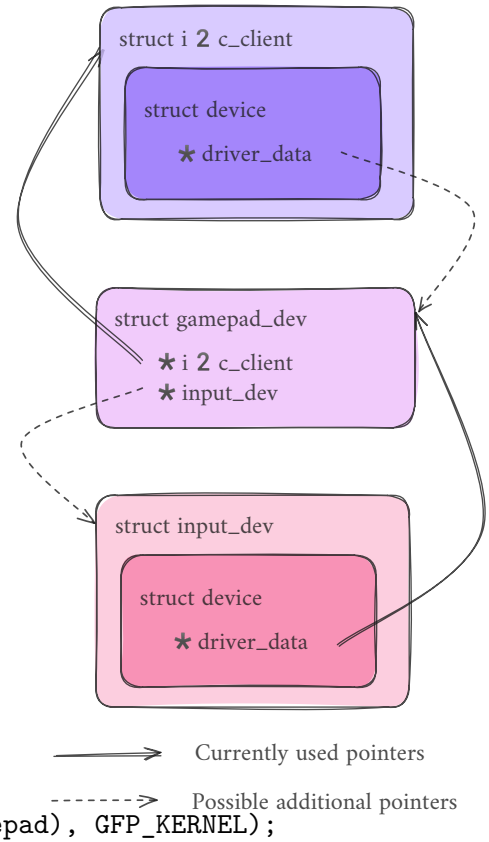
```
input_set_drvdata(input, data);
```

We could directly store a link to the `struct i2c_client` pointer, which would be easy to access in the polling routine. However, what if we have more similar device specific data to access too? The best way, that many drivers use, is to allocate a per-device structure. Here's the type that what we propose in our case:

```
struct gamepad_dev {
        struct i2c_client *i2c_client;
};
```

Later, we will be able to add more fields according to our needs.

In the `probe()` routine, allocate an instance of such a structure for each device:




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

We will cover the kernel memory allocation topic very soon. Note that we are using a `devm_` function again, to make it easier to free this resource at device unbinding time.

So, let's add the pointers to the `probe()` routine:

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

Note that, similarly, as many device drivers to, we could also keep a reference to the input framework pointer, by adding a `struct input_dev` member to `struct gamepad_dev` and then store a pointer to the per-device structure:

```
i2c_set_clientdata(client, gamepad);
```

However, that won't be necessary in our driver, because the `remove()` routine doesn't need to access the `struct input_dev` structure to unregister and free it, because that's already done automatically.

14.5.3 Polling function code

Back to the polling function, we can now:

- Define a `gamepad` pointer retrieved from `input_get_drvdata()`

- Define a `client` pointer to `gamepad->i2c_client`

Then, move the commented out code that was reading the state of the buttons and joystick into this function.

You can then pass the states of the buttons and joystick using the `input_report_key()` and `input_report_abs()` functions.

Don't forget to call `input_sync()` at the end of the polling routine.

14.6 Cleaning up

Run `checkpatch.pl` and clean up your style mistakes. It's good to be warned about your old habits 😉.

14.7 Testing

14.7.1 Testing with `evtest`

When the final module is loaded, run the `evtest` command on your target. It's a great way to make sure all the input events are properly recognized.

14.7.2 Testing with games

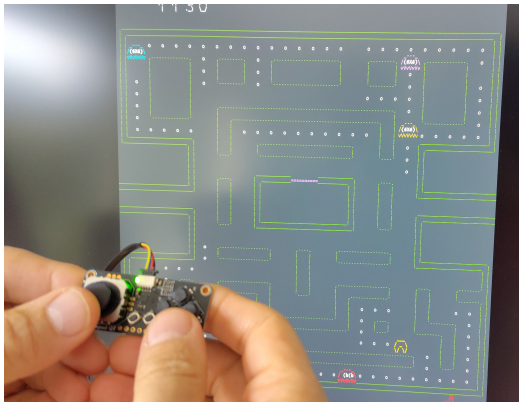
The first game you can play with your gamepad is `tetris`. It has already been added to your root filesystem (thanks to our Yocto course).

Then, you can also play `myman`, a Pacman clone, which we also compiled during our Yocto course. You can play it on the console, but it's monochrome.

It's actually much better to play it through SSH. Just SSH to your board:

```
ssh root@172.20.0.2
```

And then start `myman`.



15 Accessing I/O memory and ports

15.1 Goals

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.

15.2 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.

15.3 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 (https://openbeagle.org/beagleplay/beagleplay/-/blob/main/BeaglePlay_sch.pdf) 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, 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";
};
```

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 = "rootcommit,serial";
};

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

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.

15.4 Operate a platform device driver

Go to the `~/kernel-labs/nfsroot/home/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[1].

15.5 Create a device private structure

In the same way as in the gamepad 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;
...
```

[1] 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!

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

15.6 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);
```

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.

15.7 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.

15.7.1 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.

15.7.2 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);
```

15.7.3 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);
```

Declare `baud_divisor` and `uartclk` as `unsigned int`, and `clk` as `struct clk *`.

After the `return 0` statement, also add this error recovery code:

```
disable_runtime_pm:
        pm_runtime_disable(&pdev->dev);
        return ret;
```

Indeed, before you return with an error code, you have to remove any allocation or revert any special mode that you set before. By making the code `goto` the right label, you can undo a series of actions, depending on how far you went before failing.

15.7.4 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);
```

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.

15.8 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 `tio` instance on your new serial port (not the serial console):

```
tio /dev/ttyACM1
```

Load your module on the target. You should see the corresponding character in the new `tio` 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).

15.9 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.

16 Output-only misc driver

16.1 Objectives

During this lab, you will implement the write part of a *misc* driver

- Write a simple misc driver, allowing to write data to the serial ports of your Beaglebone.
- Write simple `struct 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.

16.2 Misc driver registration

In the same way we added an input interface to our Gamepad 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 `struct 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 Gamepad driver, we have to add such a structure to our device specific private data structure:

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

i Here, we include the entire structure, not just a pointer. The benefit of this approach is that everything is allocated at once.

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-%llx", 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 `struct 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.

16.3 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 `tio` 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.

One thing you need to add to your code, even if it has no obvious impact, is to increase the value of the offset passed to the `write()` operation. If the driver doesn't do this, nobody else will, and the offset in the open file won't increase.

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[1].

16.4 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.

16.5 Ioctl operations

We would like to maintain a count of the number of characters written through the serial port. In order to do this, add a counter member to the per-device 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 `home/root/serial/` NFS shared directory. They assume that `SERIAL_RESET_COUNTER` is ioctl operation `0` and that `SERIAL_GET_COUNTER` is ioctl operation `1`.

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

Compile them:

```
clang --target=aarch64-linux-gnu -static -o serial-get-counter serial-get-counter.c
clang --target=aarch64-linux-gnu -static -o serial-reset-counter serial-reset-counter.c
```

The new executables are then ready to run on your target. As they are static, they don't depend on the C library installed on the target. They take as argument the path to the device file corresponding to your UART.

17 Sleeping and handling interrupts

17.1 Goals

The goals are to 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.

17.2 Setup

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

17.3 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 `tio` 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.

17.4 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.

17.5 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`[1].

Last but not least, in the interrupt handler, after storing the received characters in the circular buffer,

[1] 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.

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 `tio` 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 `tio` don't appear on the screen. This happens because they are not echoed back by the target.

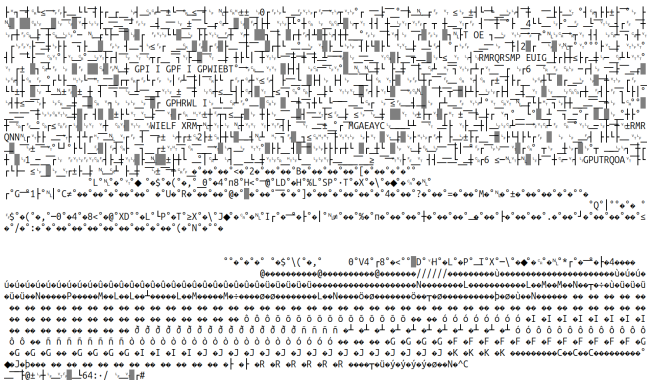


Figure 17.1: Fun debugging the read operation!

18 Concurrency prevention and locking

18.1 Goals

During this lab, we will expose an issue in the driver, due to the lack of protection against concurrent access to shared resources.

We will fix the issue by adding locking to our code.

18.2 Concurrency issue

Let's observe an issue with multiple processes accessing the serial port at the same time, which is a perfectly legal situation.

First, reset the write counter:

```
# ./serial-reset-counter /dev/serial-2850000
Counter reset
# ./serial-get-counter /dev/serial-2850000
Counter value: 0
```

Now, let's write a 100,000 byte file into the serial line:

```
# cat les-miserables-100k.txt > /dev/serial-2850000
# ./serial-get-counter /dev/serial-2850000
Counter value: 100000
```

The counter confirms that we tranferred exactly 100,000 characters.

However, let's run a program that runs two instances of this command in parallel:

```
# ./serial-reset-counter /dev/serial-2850000
Counter reset
# ./torture-serial
(wait for about 20 seconds until there is no more activity on tio on UART5)
# ./serial-get-counter /dev/serial-2850000
Counter value: 169575
```

We should have transferred 200,000 characters, but we lost 30,425. That's more than 15% of the total 😱

18.3 Why does this happen?

This issue must happen because we are calling the same kernel code from multiple processes running concurrently on the system.

For example, one process making a `write()` system call can be spending time in the loop waiting for the UART to be ready to transmit a character.

If another process is running the same code in parallel, the two processes could end up sending a character at the exact same time or trying to increase the write counter simultaneously. This can lead to non deterministic results!

This kind of issue was less likely to happen in the past, because:

- Single core systems were more frequent in the past (like the Beaglebone Black). Only one piece of code was running, and kernel concurrency issues could only happen because of kernel preemption.
- Kernel preemption was disabled by default in most systems, being compiled with `CONFIG_PREEMPT_NONE`. That meant that system calls were never preempted, and therefore executed to their completion before giving way to other system calls. However, that's no longer the case of our kernel today, which was compiled with `CONFIG_PREEMPT`, which makes all kernel code preemptible, except for critical sections like holding spinlocks or executing an interrupt handler.

18.4 Making the write counter atomic

The problem with the write counter is an interesting opportunity to experiment with atomic types.

So, edit the per-device structure to have an atomic counter instead of a regular integer:

```
...
        atomic_t counter;
...
```

You will also have to modify the `write()` operation to use a dedicated function to increment the counter.

Similarly, you will also have to modify the way the `ioctl()` operation accesses and modifies the counter. The compiler will remind you if you forget.

Rebuild the module and run the tests again. The counter issue should be gone now.

18.5 Issue with the number of transmitted bytes

The write counter is now correct, but what about the actual number of transmitted bytes. If two processes do try to transmit a byte at the same time, one of them may succeed, but the second one won't know that the device needs time to be ready again. Characters are likely to be lost this way.

Let's double check the actual number of transmitted bytes, as received on the PC side. To do it, first temporarily comment out the code that transmits an additional `\r` after each `\n`.

After updating the module, quit `tio` on UART5 (for example), and restart it with the `-l` option that will duplicate all the received bytes into a log file:

```
tio -l /dev/ttyACM1
```

On the board side, send 100,000 characters to the device:

```
# cat les-miserables-100k.txt > /dev/serial-2850000
```

Once the copy is over, exit `tio` and check the size of the log file:

```
wc -c tio_ttyACM1_2025-06-18T12:10:43.log
100000 tio_ttyACM1_2025-06-18T12:10:43.log
```

So, 100,000 characters were received as expected.

Now, let's repeat all this, but this time with the `torture-serial` program:

```
./torture-serial
```

Once the experiment is over, let's check the results:

```
wc -c tio_ttyACM1_2025-06-18T12:13:07.log
100095 tio_ttyACM1_2025-06-18T12:13:07.log
```

Ouch, we are very far from the expected 200,000!

What we anticipated turned out to be true: when two processes are trying to send a character over the serial line, only one succeeds, and the second one thinks that it also successfully sent one (because the condition was true for it too), but the line wasn't ready any more.

Let's try to avoid this issue by adding locking to our code:

- Add a `struct mutex` member to your per-device structure
- Protect the code section that calls `serial_write_char()` with this mutex.
- This protection doesn't need to include the counter, as its atomicity is already guaranteed.
- Don't forget to initialize the mutex in the `probe()` routine!

Then, update your module and run the torture tests again. You should get:

```
wc -c tio_ttyACM1_2025-06-18T12:25:31.log
200000 tio_ttyACM1_2025-06-18T12:25:31.log
```

Yoohoo, the number of received characters matches.

You can now restore the `\r` code, but keep it in the section protected by the mutex.

Don't be too excited about locking though. Locking degrades performance, and you should apply it to sections that are are small as possible. For example, you could have protected the whole `write()` operation, but the performance degradation would have been worse.

19 Kernel debugging

19.1 Goals

In this lab, we are going to practise with kernel debugging mechanisms, going on working on the code of our serial driver.

19.2 Make debug messages more explicit

In the code of your serial driver, replace all the `pr_` debug functions by `dev_` ones, so that they tell what device they relate to.

In particular, add back a message to the `probe()` and `remove()` functions, to see how the results look like, without having to wait for an error to happen!

19.3 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 `CONFIG_DYNAMIC_DEBUG`: this will allow you to see debugging messages.

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

Once this is done, in U-Boot, add `loglevel=8` to the kernel command linei 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 (if necessary). 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.

19.4 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.

Tip: You can use the `debugfs_create_atomic_t()` function to expose the `atomic_t` counter that your driver manages.

Tip: In the `remove()` routine, you can use the `debugfs_remove_recursive()` function to remove the custom `debugfs` subdirectory and its contents.

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.

20 Kernel crash analysis

20.1 Goals

In this lab, we will investigate a kernel crash in a broken driver.

20.2 Setup

Go to the `~/kernel-labs/nfsroot/home/root/crash/` directory.

Also make sure your kernel was compiled with `CONFIG_DEBUG_INFO`. This makes it possible to see source code in disassembled kernel code, and as no impact on the runtime kernel, as the final kernel is stripped.

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

20.3 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 the crash happens, 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.

20.4 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 `clang -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:

```
llvm-objdump -S file.o > file.S
```

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.