UE B4 - Embedded Linux

Embedded Linux Project

Thomas Boulanger Jules Farnault
Martin raynaud

March 2, 2025

Contents

Introduction

The goal of this TPs are to compile a custom linux kernel and create drivers to interfaces devices with a simulated ARM board.

QEMU is utilised to emulate an ARM computer in conjunction with an AXDL345 accelerometer connected via an I2C interface.
The initial phase of this tutorial pertains to the compilation and simulation of a Linux kernel.
Subsequent phases entail the creation of a driver and the establishment of an interface with the user space.
The final phase involves the incorporation of interrupt handling.

1 | TP - 1 : Compiling and simulating a linux kernel (Thomas Boulanger)

The goal of this TP is to compile a linux kernel for a ARM vexpress board, then add busybox utils and finaly launch everything with the bootloader U-Boot. As the versions provided in the TP are 4 years old, the lastest stable version of the softwares will be used instead. Compiled binaries can be found on Github

1.1 - Configuration

- Computer OS : Archlinux 6.13.3
- Kernel Image : Linux 6.14.0 rc1

1.2 - Linux Kernel

First, the last version of linux (6.14.0) is downloaded from the git repository :

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

The kernel can then be configured with the following command :

```
$ make O=build vexpress_defconfig
$ make O=build menuconfig
```

In the menuconfig, the "Support for uevent helper" option is enable in Device Drivers, Generic Driver Options.

The Linux kernel can then be compiled with the following command :

```
$ make O=build -j$nproc
```

The `-j$nproc` enable a multi-core compilation for a faster result. After 10min, the compilation fished with the following lines :

```
  LD      vmlinux
  NM      System.map
  SORTTAB vmlinux
  OBJCOPY arch/arm/boot/Image
  Kernel: arch/arm/boot/Image is ready
  LDS     arch/arm/boot/compressed/vmlinux.lds
  AS      arch/arm/boot/compressed/head.o
  GZIP    arch/arm/boot/compressed/piggy_data
  CC      arch/arm/boot/compressed/misc.o
  CC      arch/arm/boot/compressed/decompress.o
  CC      arch/arm/boot/compressed/string.o
  AS      arch/arm/boot/compressed/hyp-stub.o
  CC      arch/arm/boot/compressed/fdt_rw.o
  CC      arch/arm/boot/compressed/fdt_ro.o
  CC      arch/arm/boot/compressed/fdt_wip.o
  CC      arch/arm/boot/compressed/fdt.o
  CC      arch/arm/boot/compressed/fdt_check_mem_start.o
  AS      arch/arm/boot/compressed/lib1funcs.o
  AS      arch/arm/boot/compressed/ashldi3.o
  AS      arch/arm/boot/compressed/bswapsdi2.o
  AS      arch/arm/boot/compressed/piggy.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
make[1]: Leaving directory '~/M2_SETI/B4/TP/linux/build'
```

Once compiled, the kernel can be started with Qemu :

```
$ qemu-system-arm -machine vexpress-a9 -nographic -kernel \
linux/build/arch/arm/boot/zImage -dtb \
linux/build/arch/arm/boot/dts/arm/vexpress-v2p-ca9.dtb
```

The starting of the kernel can be then observed which endup with a kernel panic due to the missing initramfs image :

```
Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
CPU: 0 UID: 0 PID: 1 Comm: swapper/0 Not tainted 6.14.0-rc1 #2
Hardware name: ARM-Versatile Express
Call trace:
 unwind_backtrace from show_stack+0x10/0x14
 show_stack from dump_stack_lvl+0x50/0x64
 dump_stack_lvl from panic+0x110/0x364
 panic from mount_root_generic+0x1e8/0x2b4
 mount_root_generic from mount_root+0x22c/0x248
 mount_root from prepare_namespace+0x200/0x250
 prepare_namespace from kernel_init+0x1c/0x12c
 kernel_init from ret_from_fork+0x14/0x28
Exception stack(0x88825fb0 to 0x88825ff8)
5fa0:                                     00000000 00000000 00000000 00000000
5fc0: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
5fe0: 00000000 00000000 00000000 00000000 00000013 00000000
---[ end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0) ]---
```

To test the kernel, a very simple init function is written and compiled :

```
$ cd $TPROOT
$ mkdir initramfs_simple
$ cd initramfs_simple
$ vim init.c
```

The following code is written :

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

int main(int argc, char *argv[])
{
  printf("Hello world!\n");
  sleep(10);
}
```

And then compiled and a initramfs image is created

```
$ arm-linux-gnueabihf-gcc -static init.c -o init
$ echo init | cpio -o -H newc | gzip > test.cpio.gz
7873 blocks
```

The Qemu simulation can then be started

```
$ cd ..
$ qemu-system-arm -machine vexpress-a9 -nographic -kernel \
linux/build/arch/arm/boot/zImage \
-dtb linux/build/arch/arm/boot/dts/arm/vexpress-v2p-ca9.dtb \
-initrd initramfs_simple/test.cpio.gz
```

As it can be seen below, the kernel start as expected and then panic as the init function should never return :

```
Freeing unused kernel image (initmem) memory: 1024K
input: AT Raw Set 2 keyboard as
  ↪ /devices/platform/bus@40000000/bus@40000000:motherboard-bus@40000000/
  ↪ bus@40000000:motherboard-bus@40000000:iofpga@7,00000000/10006000.kmi/
  ↪ serio0/input/input2
drm-clcd-pl111 10020000.clcd: DVI muxed to daughterboard 1 (core tile)
  ↪ CLCD
Run /init as init process
drm-clcd-pl111 10020000.clcd: initializing Versatile Express PL111
Hello world!
Kernel panic - not syncing: Attempted to kill init! exitcode=0x00000000
CPU: 0 UID: 0 PID: 1 Comm: init Not tainted 6.14.0-rc1 #2
Hardware name: ARM-Versatile Express
Call trace:
 unwind_backtrace from show_stack+0x10/0x14
 show_stack from dump_stack_lvl+0x50/0x64
 dump_stack_lvl from panic+0x110/0x364
 panic from do_exit+0xa14/0xae4
 do_exit from do_group_exit+0x34/0x98
 do_group_exit from pid_child_should_wake+0x0/0x60
---[ end Kernel panic - not syncing: Attempted to kill init!
  ↪ exitcode=0x00000000 ]---
```

1.3 - Busybox

To create a better initramfs, busybox will be used as it provides various UNIX utilities for embedded linux. The last stable version (BusyBox 1.36.1) is first downloaded and extracted

```
$ cd $TPROOT
$ wget https://busybox.net/downloads/busybox-1.36.1.tar.bz2
$ tar xjf busybox-1.36.1.tar.bz2
$ cd busybox-1.36.1
```

Once extracted, busybox needs to be configured. First, the command `make defconfig` is run to create a default `.config` file, then `make menuconfig` is run to configure busybox as wanted. However, running `make menuconfig` raise the follwoing error :

```
make menuconfig
  HOSTLD  scripts/kconfig/mconf
  HOSTCC  scripts/kconfig/lxdialog/checklist.o
  HOSTCC  scripts/kconfig/lxdialog/inputbox.o
  HOSTCC  scripts/kconfig/lxdialog/lxdialog.o
  HOSTCC  scripts/kconfig/lxdialog/menubox.o
  HOSTCC  scripts/kconfig/lxdialog/msgbox.o
  HOSTCC  scripts/kconfig/lxdialog/textbox.o
  HOSTCC  scripts/kconfig/lxdialog/util.o
  HOSTCC  scripts/kconfig/lxdialog/yesno.o
  HOSTLD  scripts/kconfig/lxdialog/lxdialog
 *** Unable to find the ncurses libraries or the
 *** required header files.
 *** 'make menuconfig' requires the ncurses libraries.
 ***
 *** Install ncurses (ncurses-devel) and try again.
 ***
make[2]: *** [~/M2_SETI/B4/TP/busybox-1.36.1/scripts/kconfig/lxdialog/Makefile:15: scripts/kconfig/lxdialog/dochecklxdialog] Error 1
make[1]: *** [~/M2_SETI/B4/TP/busybox-1.36.1/scripts/kconfig/Makefile:14: menuconfig] Error 2
make: *** [Makefile:444: menuconfig] Error 2
```

When running the `make menueconfig` on Archlinux, an error occurs with unrecognized `ncurses` header even if it is installed correctly and in the PATH, this is a known issue. To fix it, the file `scripts/kconfig/lxdialog/check-lxdialog.sh` needs to be modified : on line 49 `#include CURSES_LOC` should be changed to `#include <ncurses.h>` and on line 50 `main() {}` should be changed to `int main() {}`. Once changed, busybox works as expected.

In the menuconfig, 2 parameters needs to be changed, first, Build Static Binary needs to be enable in the settings option, then TC option needs to be disable in the network configuration due to incompatibility with the last linux version.

Busybox can then be compiled with make, the end of the compilation looks like this :

```
  AR      util-linux/volume_id/lib.a
  LINK    busybox_unstripped
Static linking against glibc, can't use --gc-sections
Trying libraries: crypt m resolv rt
 Library crypt is not needed, excluding it
 Library m is needed, can't exclude it (yet)
 Library resolv is needed, can't exclude it (yet)
 Library rt is not needed, excluding it
 Library m is needed, can't exclude it (yet)
 Library resolv is needed, can't exclude it (yet)
Final link with: m resolv
  DOC     busybox.pod
  DOC     BusyBox.txt
  DOC     busybox.1
  DOC     BusyBox.html
```

Once compiled, busybox is installed with sudo make install which end as the following :

```
  ./_install//usr/sbin/ubirename -> ../../bin/busybox
  ./_install//usr/sbin/ubirmvol -> ../../bin/busybox
  ./_install//usr/sbin/ubirsvol -> ../../bin/busybox
  ./_install//usr/sbin/ubiupdatevol -> ../../bin/busybox
  ./_install//usr/sbin/udhcpd -> ../../bin/busybox


--------------------------------------------------
You will probably need to make your busybox binary
setuid root to ensure all configured applets will
work properly.
--------------------------------------------------
```

Once installed, a initramfs image can be created with busybox with the following commands

```
$ cd $TPROOT
$ mkdir initramfs_busybox
$ cd initramfs_busybox
$ fakeroot /bin/bash
$ cp -r ../busybox-1.34.1/_install/* .
$ rm linuxrc
$ ln -s bin/sh init
```

```
$ mkdir proc sys tmp root var mnt dev etc
$ sudo mknod dev/console c 5 1
$ sudo mknod dev/tty1 c 4 1
$ sudo mknod dev/tty2 c 4 2
$ sudo mknod dev/tty3 c 4 3
$ sudo mknod dev/tty4 c 4 4
$ cd etc
$ cp ../../busybox-1.34.1/examples/inittab .
$ mkdir init.d
$ cd init.d
$ cat <<EOF > rcS
#!/bin/sh
mount -a
mkdir -p /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
mkdir -p /var/lock
ifconfig lo 127.0.0.1
EOF
$ chmod 755 rcS
$ cd ..
$ cat <<EOF > fstab
proc    /proc   proc    defaults    0   0
tmpfs   /tmp    tmpfs   defaults    0   0
sysfs   /sys    sysfs   defaults    0   0
tmpfs   /dev    tmpfs   defaults    0   0
EOF
$ cd ..
$ find . | cpio -o -H newc -R +0:+0 | gzip > initramfs.gz
```

Now that a initramfs image is created, the Qemu simulation can be started with the following command :

```
qemu-system-arm -machine vexpress-a9 -nographic -kernel \
linux/build/arch/arm/boot/zImage \
-dtb linux/build/arch/arm/boot/dts/arm/vexpress-v2p-ca9.dtb \
-initrd initramfs_busybox/initramfs.gz
```

As expected, linux boot in a shell :

```
Run /init as init process
input: AT Raw Set 2 keyboard as
↪ /devices/platform/bus@40000000/bus@40000000:motherboard-bus@40000000/
↪ bus@40000000:motherboard-bus@40000000:iofpga@7,00000000/10006000.kmi/
↪ serio0/input/input2
```

```
drm-clcd-pl111 10020000.clcd: DVI muxed to daughterboard 1 (core tile)
 ↪ CLCD
drm-clcd-pl111 10020000.clcd: initializing Versatile Express PL111

Please press Enter to activate this console.
~ #
```

1.4 - U-Boot

Now that a working kernal and initramfs have been created, Das U-boot will be used as a bootloader to load the kernel and the image in ram. First, the most recent stable release of U-Boot (2025.01) is downloaded, extracted and compiled :

```
$ wget https://ftp.denx.de/pub/u-boot/u-boot-2025.01.tar.bz2
$ tar xjf u-boot-2025.01.tar.bz2
$ cd u-boot-2025.01
$ make distclean
$ make vexpress_ca9x4_defconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  YACC    scripts/kconfig/zconf.tab.c
  LEX     scripts/kconfig/zconf.lex.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
#
# configuration written to .config
#
$ make
scripts/kconfig/conf  --syncconfig Kconfig
  UPD     include/config.h
  CFG     u-boot.cfg
[...]
  CC      net/net-common.o
  AR      net/built-in.o
  LDS     u-boot.lds
  LD      u-boot
  OBJCOPY u-boot.srec
  OBJCOPY u-boot-nodtb.bin
  COPY    u-boot.bin
  SYM     u-boot.sym
===================== WARNING ======================
CONFIG_OF_EMBED is enabled. This option should only
be used for debugging purposes. Please use
CONFIG_OF_SEPARATE for boards in mainline.
See doc/develop/devicetree/control.rst for more info.
```

```
========================================================
  OFCHK   .config

```

To test the compilation, Qemu can be started with the following command :

```
$ qemu-system-arm -machine vexpress-a9 -nographic -kernel
 ↪ u-boot-2025.01/u-boot

U-Boot 2025.01 (Feb 11 2025 - 22:30:13 +0100)

DRAM:  128 MiB
WARNING: Caches not enabled
Core:  23 devices, 11 uclasses, devicetree: embed
Flash: 128 MiB
MMC:   mmci@5000: 0
Loading Environment from Flash... *** Warning - bad CRC, using default
 ↪ environment

In:    uart@9000
Out:   uart@9000
Err:   uart@9000
Net:   eth0: ethernet@3,02000000
Hit any key to stop autoboot:  0
=>
```

A simulated SD card will be created to store both U-boot, the linux kernel and the initramfs image :

```
$ cd $TPROOT
$ mkdir sdcard
$ cd sdcard
$ dd if=/dev/zero of=sd bs=1M count=64
$ parted sd
(parted) mklabel msdos
(parted) mkpart primary fat32 1MiB 100%
(parted) print
Modèle :  (file)
Disque : 67,1MB
Taille des secteurs (logiques/physiques) : 512B/512B
Table de partitions : msdos
Drapeaux de disque :

Numéro  Début   Fin     Taille  Type     Système de fichiers  Drapeaux
 1      1049kB  67,1MB  66,1MB  primary  fat32                lba
```

```

(parted) quit
$ sudo losetup -f --show -P sd
/dev/loop0
$ sudo mkfs.fat -F 32 /dev/loop0p1
$ mkdir mnt
$ sudo mount /dev/loop0p1 mnt
$ sudo cp ../linux/build/arch/arm/boot/zImage mnt
$ sudo cp ../linux/build/arch/arm/boot/dts/arm/vexpress-v2p-ca9.dtb mnt
$ sudo cp ../initramfs_busybox/initramfs.gz mnt
$ cd mnt
$ sudo ../../u-boot-2025.01/tools/mkimage -A arm -O linux -T ramdisk -d initramfs.gz uinitramfs
$ cd ..
$ sudo umount mnt
$ sudo losetup -d /dev/loop0
```

Finally, U-boot can be started from the simulated SD card and linux can be loaded and started from U-boot :

```
qemu-system-arm -machine vexpress-a9 -nographic -kernel \
u-boot-2025.01/u-boot -sd sdcard/sd
WARNING: Image format was not specified for 'sdcard/sd' and probing guessed raw.
         Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
         Specify the 'raw' format explicitly to remove the restrictions.


U-Boot 2025.01 (Feb 11 2025 - 22:30:13 +0100)

DRAM:  128 MiB
WARNING: Caches not enabled
Core:  23 devices, 11 uclasses, devicetree: embed
Flash: 128 MiB
MMC:   mmci@5000: 0
Loading Environment from Flash... *** Warning - bad CRC, using default environment

In:    uart@9000
Out:   uart@9000
Err:   uart@9000
Net:   eth0: ethernet@3,02000000
Hit any key to stop autoboot:  0
=> fatload mmc 0:1 0x62000000 zImage
5918880 bytes read in 5006 ms (1.1 MiB/s)
=> fatload mmc 0:1 0x63000000 vexpress-v2p-ca9.dtb
```

```
14329 bytes read in 29 ms (482.4 KiB/s)
=> fatload mmc 0:1 0x63100000 uinitramfs
1835460 bytes read in 1508 ms (1.2 MiB/s)
=> bootz 0x62000000 0x63100000 0x63000000
Kernel image @ 0x62000000 [ 0x000000 - 0x5a50a0 ]
## Loading init Ramdisk from Legacy Image at 63100000 ...
   Image Name:
   Image Type:   ARM Linux RAMDisk Image (gzip compressed)
   Data Size:    1835396 Bytes = 1.8 MiB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 63000000
   Booting using the fdt blob at 0x63000000
Working FDT set to 63000000
   Loading Ramdisk to 66957000, end 66b17184 ... OK
   Loading Device Tree to 66950000, end 669567f8 ... OK
Working FDT set to 66950000

Starting kernel ...

Booting Linux on physical CPU 0x0
[...]
Run /init as init process
input: ImExPS/2 Generic Explorer Mouse as
 ↪ /devices/platform/bus@40000000/bus@40000000:motherboard-bus@40000000/
 ↪ bus@40000000:motherboard-bus@40000000:iofpga@7,00000000/10007000.kmi/
 ↪ serio1/input/input2
drm-clcd-pl111 10020000.clcd: DVI muxed to daughterboard 1 (core tile)
 ↪ CLCD
drm-clcd-pl111 10020000.clcd: initializing Versatile Express PL111
/etc/init.d/rcS: line 5: can't create /proc/sys/kernel/hotplug:
 ↪ nonexistent directory

Please press Enter to activate this console.
~ #
```

During this TP, a fully functional kernel, initramfs with utils and U-boot bootloader have been setup, compiled and simulated with Qemu.

2 | TP - 2 : Creating a custom driver for a adxl345 accelerometer (Jules Farnault)

The goal of this part is to write a driver for a I2C peripheral. QEMU simulates an ADXL345 accelerometer which is connected via I2C. A driver skeleton given in the course is used as a starting point.

2.1 - Configuration

- Computer OS : Ubuntu 24.04
- Kernel Image : Linux 6.14.0 rc1+

2.2 - ADXL345 accelerometer installation

To begin with, we need to add the _probe function and the _remove print function to check that the functions are run:

```
static int adxl345_probe(struct i2c_client *client)
{
    pr_info("ADXL345 is connect\n");
    return 0;
}

static void adxl345_remove(struct i2c_client *client)
{
    pr_info("ADXL345 is disconnect\n");
}
```

A return 0 is removed as well as const struct i2c_device_id *id from the _probe function to adapt it to our kernel version.

```
# mount -t 9p -o trans=virtio mnt /mnt -oversion=9p2000.L,msize=10240
# insmod /mnt/adxl345.ko
adxl345: loading out-of-tree module taints kernel.
```

It has been observed that the connection is not displayed. This is due to the incorrect simulation of the ADXL345 accelerometer, which is not defined as existing. In order to simulate it, the following code must be added to the dts file : (`$TPROOT/linux/arch/arm/boot/dts/arm/vexpress-v2p-ca9.dts`)

```
&v2m_i2c_dvi {
	adxl345: adxl345@53 {
		compatible = "qemu,adxl345";
		reg = <0x53>;
		interrupt-parent = <&gic>;
		interrupts = <0 50 4>;
	};
};
```

Compilation is launch with `make O=build -j$nproc`.

QEMU is launch with the following command :

```
./qemu-system-arm -machine vexpress-a9 -nographic -kernel
  linux/build/arch/arm/boot/zImage -dtb
  linux/build/arch/arm/boot/dts/arm/vexpress-v2p-ca9.dtb -initrd
  rootfs.cpio.gz -fsdev
  local,path=pilote\_i2c,security\_model=mapped,id=mnt -device
  virtio-9p-device,fsdev=mnt,mount\_tag=mnt
```

The device is mount :

```
# insmod /mnt/adxl345.ko
adxl345: loading out-of-tree module taints kernel.
ADXL345 is connect
# rmmod adxl345
ADXL345 is disconnect
```

The _probe and _remove functions are launched and printed correctly. The ADXL345 is therefore accessible to QEMU.

2.3 - Read and write in accelerometer register

The ADXL345 documentation give the adresse of each register :

```
#define ADXL345_REG_DEVID 0x00
#define ADXL345_REG_BW_RATE 0x2C
#define ADXL345_REG_POWER_CTL 0x2D
#define ADXL345_REG_INT_ENABLE 0x2E
#define ADXL345_REG_DATA_FORMAT 0x31
#define ADXL345_REG_FIFO_CTL 0x38
```

After, a function for read or write a register in I2C is written.

```
static int read_register_i2c(struct i2c_client *client, char
    reg_address, char *value)
{
    char buffer[1];
    int error=0;
    buffer[0] = reg_address;
    error = i2c_master_send(client, buffer, 1);

    if (error < 0){
        pr_info("[ERROR] reading register (send) %x\n", reg_address);
        return error;
    }

    error = i2c_master_recv(client, value, 1);

    if (error < 0){
        pr_info("[ERROR] reading register (receive) %x\n", reg_address);
        return error;
    }
    return 0;
}

static int write_register_i2c(struct i2c_client *client, char
    reg_address, char value)
{
    char buffer[2];
    int error;
    buffer[0] = reg_address;
    buffer[1] = value;
    error = i2c_master_send(client, buffer, 2);

    // Gestion erreur
    if (error < 0){
```

```c
        pr_info("[ERROR] writing register %x\n", reg_address);
        return error;
    }
    return 0;
}
```

ADXL345 documentation give the value to write in each register to achieve the desired behavior. For example, to have an output data rate of 100Hz, 0x0A is written in the BW_RATE register.

```c
static int adxl345_probe(struct i2c_client *client)
{
    char id;
    int error=0;

    pr_info("ADXL345 is connect\n");

    error = read_register_i2c(client, ADXL345_REG_DEVID, &id);


    pr_info("ADXL345 DEVID: %x\n", id);

    write_register_i2c(client, ADXL345_REG_BW_RATE, 0x0A);
    write_register_i2c(client, ADXL345_REG_INT_ENABLE, 0x00);
    write_register_i2c(client, ADXL345_REG_DATA_FORMAT, 0x00);
    write_register_i2c(client, ADXL345_REG_FIFO_CTL, 0x00);
    write_register_i2c(client, ADXL345_REG_POWER_CTL, 0x08);

    return 0;
}

static void adxl345_remove(struct i2c_client *client)
{
    write_register_i2c(client, ADXL345_REG_POWER_CTL, 0x04);
    pr_info("ADXL345 is disconnect\n");
}
```

The results are as follows :

```
# insmod /mnt/adxl345.ko
adxl345: loading out-of-tree module taints kernel.
ADXL345 is connect
ADXL345 DEVID: e5
# rmmod adxl345
ADXL345 is disconnect
```

ADXL345 is connected and is DEVID is E5. Register is modified without problem. To finish, the accelerometer is disconnected.

3 | TP - 3 : Interface with the user space (Martin Raynaud)

3.1 - Configuration of the previous TPs

- Computer OS : Ubuntu 22.04
- Kernel Image : Linux 5.10.19

Trying to compile the file adxl345.c created in TP2 was not working because of the different version of the kernel linux used. It was the linux 6.14.0 rc1+ for Jules contrary to linux 5.10.19 for Martin. The errors returned while trying to compile were :

```
/home/coolman/seti_b4_tp/pilote_i2c/adxl345.c:124:23: erreur:
↪ initialization from incompatible pointer type
↪ [-Werror=incompatible-pointer-types]
     .probe          = adxl345_probe,
                       ^~~~~~~~~~~~~


/home/coolman/seti_b4_tp/pilote_i2c/adxl345.c:125:23: erreur:
↪ initialization from incompatible pointer type
↪ [-Werror=incompatible-pointer-types]
     .remove         = adxl345_remove,
                       ^~~~~~~~~~~~~~
```

They were two changes made to the adxl345.c file regarding the probe and remove functions :

- `static int adxl345_probe(struct i2c_client *client, const struct i2c_device_id *id)` : the second argument of the function was added.
- `static int adxl345_remove(struct i2c_client *client)` : the function now returns an int.

3.2 - First Step - probe and remove functions

First, a struct adxl345_device is declared containing a struct miscdevice and an int axis which will contain the axis selected by the application.

```
struct adxl345_device
{
    struct miscdevice misc_dev;
    int axis;
};
```

A structure named file_operations is also created which groups the callback functions provided by our device driver : open, read and unlocked_ioctl.

```
static const struct file_operations adxl345_fops = {
    .owner = THIS_MODULE,
    .open = adxl345_open,
    .read = adxl345_read,
    .unlocked_ioctl = adxl345_ioctl,
};
```

The following code is added to the probe function to register with the misc framework properly. All the potential errors caused by the functions called are handled.

```
// Creation of an instance adxl345_dev of the struct adxl345_device
struct adxl345_device *adxl345_dev;
char *name;

// Dynamically allocate memory for an instance of the struct
    adxl345_device
adxl345_dev = kmalloc(sizeof(struct adxl345_device), GFP_KERNEL);
if (!adxl345_dev) {
    pr_info("[ERROR] kmalloc failed\n");
    return -ENOMEM;
}

// Association between the instance of i2c_client and the adxl345_dev
i2c_set_clientdata(client, adxl345_dev);

// Creation of the field name of the struct miscdevice
name = kasprintf(GFP_KERNEL, "adxl345-%d", number_of_device);
if (!name) {
    pr_info("[ERROR] kasprintf name failed\n");
```

```
    kfree(adxl345_dev);
    return -ENOMEM;
}
number_of_device++;

// Fill of the structure misc_dev contained in the structure adxl345_dev
adxl345_dev->misc_dev.minor = MISC_DYNAMIC_MINOR;
adxl345_dev->misc_dev.name = name;
adxl345_dev->misc_dev.parent = &client->dev;
adxl345_dev->misc_dev.fops = &adxl345_fops;

// Register with the misc framework
if (misc_register(&adxl345_dev->misc_dev)) {
    pr_info("[ERROR] misc_register failed\n");
    kfree(name);
    kfree(adxl345_dev);
    return -ENOMEM;
}
```

3.3 - Second Step - read function and application main.c

3.3.1 - read function

```
static ssize_t adxl345_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
    // Initialization of the adxl_dev and the client
    short value;
    pr_info("Read function call\n");
    struct adxl345_device *adxl345_dev = container_of(file->private_data, struct adxl345_device, misc_dev);
    struct i2c_client *client = container_of(adxl345_dev->misc_dev.parent, struct i2c_client, dev);
    char data[2];
    int error = 0;

    // Read data from the accelerometer based on the selected axis
    if (adxl345_dev->axis == 1) {
        error = read_register_i2c(client, ADXL345_REG_DATA_X0, &data[0]);
        if (error) return error;
```

```
        error = read_register_i2c(client, ADXL345_REG_DATA_X1, &data[1]);
        if (error) return error;
    } else if (adxl345_dev->axis == 2) {
        error = read_register_i2c(client, ADXL345_REG_DATA_Y0, &data[0]);
        if (error) return error;
        error = read_register_i2c(client, ADXL345_REG_DATA_Y1, &data[1]);
        if (error) return error;
    } else if (adxl345_dev->axis == 3) {
        error = read_register_i2c(client, ADXL345_REG_DATA_Z0, &data[0]);
        if (error) return error;
        error = read_register_i2c(client, ADXL345_REG_DATA_Z1, &data[1]);
        if (error) return error;
    } else {
        pr_info("[ERROR] Invalid axis\n");
        return -EINVAL;
    }

    // Combine the data into a single value
    value = (data[1] << 8) | data[0];

    // Copy the value to user space
    if (copy_to_user(buf, &value, sizeof(value))) {
        pr_info("[ERROR] copy_to_user\n");
        return -EFAULT;
    }

    return sizeof(value);
}

```

The read function makes the link between the user space and the driver. The user wants to be able to obtain a value of a specific axis of the accelerometer. To do that, a field of the structure adxl_device named axis is created to store the axis selected by the user. The acceleration value has a length of 2 bytes which are recovered one byte after the other from the accelerometer.

3.3.2 - The application

```c
int main()
{
    // Initialization
    char axis;
    int selected_axis;

    // Request the axis to read
    printf("Choose an axis (x, y, z): ");
    scanf("%c", &axis);

    if (axis == 'x' || axis == 'X')
        selected_axis = IOCTL_SET_AXIS_X;
    else if (axis == 'y' || axis == 'Y')
        selected_axis = IOCTL_SET_AXIS_Y;
    else if (axis == 'z' || axis == 'Z')
        selected_axis = IOCTL_SET_AXIS_Z;
    else
    {
        printf("Failed to read an invalid axis\n");
        return -1;
    }

    // Request the number of samples to read
    unsigned int samples;
    printf("Number of samples: ");
    scanf("%u", &samples);

    // Open the device file using a standard system call
    int file = open(FILE_NAME, O_RDONLY);
    if (file < 0)
    {
        printf("[ERROR] Error opening file\n");
        return 1;
    }

    // Set the axis using ioctl
    if (ioctl(file, selected_axis) < 0)
    {
        printf("[ERROR] Failed to set axis\n");
        close(file);
        return -1;
    }

    // Create a buffer to hold the samples
    short buffer[samples];
```

```
    ssize_t bytes_read;

    for (unsigned int i = 0; i < samples; i++)
    {
        printf("Sample %u\n", i+1);
        // Read from the device the bytes corresponding to one sample
        bytes_read = read(file, &buffer[i], sizeof(long));
        if (bytes_read < 0)
        {
            printf("[ERROR] Failed to read from the device\n");
            close(file);
            return -1;
        }

        printf("Result = %hd\n\n", buffer[i]);
        usleep(10000); // Sleep for 10 milliseconds
    }

    // Close the device file
    close(file);

    return 0;
}
```

The user selects an axis and the number of samples he wants. The application will them retrieve the values using the system call read. The bash code below shows an example of the output of the application.

```
Welcome to Buildroot
buildroot login: root
# random: fast init done
mount -t 9p -o trans=virtio mnt /mnt -oversion=9p2000.L,msize=10240
# insmod /mnt/adxl345.ko
adxl345: loading out-of-tree module taints kernel.
ADXL345 is connect
ADXL345 DEVID: E5
# cd /mnt
# ./main
Choose an axis (x, y, z): y
Number of samples: 5
Sample 1
Read function call
Result = 1

Sample 2
Read function call
Result = -5
```

```

Sample 3
Read function call
Result = -5

Sample 4
Read function call
Result = 9

Sample 5
Read function call
Result = 9
```

The application correctly prints 5 samples of the axis y of the accelerometer. Since the application is sleeping 10 ms between each read function call, the values are not synchronized perfectly.
Using interruptions afterwards will ensure that the values are retrieved correctly.

4 | TP - 4 : Interrupt Handling with ADXL345 (Thomas Boulanger et al.)

The goal of this TP is to use interruptions and FIFO to read the data from the ADXL345. Source code and compiled binaries can be found on Github.

4.1 - Configuration

- Computer OS : Archlinux 6.13.3
- Kernel Image : Linux 6.14.0 rc1+

4.2 - ADXL345 configuration

First, the ADXL345 needs to be configured in order to activate the FIFO (register `FIFO_CTL`) and configure the `INT_ENABLE` register. To do so, the `adxl345_probe` have been slightly modified :

```
write_register_i2c(client, ADXL345_REG_BW_RATE, 0x0A);
write_register_i2c(client, ADXL345_REG_INT_ENABLE, (1 << 1)); // Activate Watermark
write_register_i2c(client, ADXL345_REG_DATA_FORMAT, 0x00);
write_register_i2c(client, ADXL345_REG_FIFO_CTL, (1 << 7) + 20); // config FIFO with IRQ at 20 items
write_register_i2c(client, ADXL345_REG_POWER_CTL, 0x08);
```

With this configuration, the ADXL345 will store the data in its internal FIFO and trigger an interrupt when the FIFO reach 20 elements.

The structure of the `adxl345_device` is also changed to add the fifo :

```c
struct fifo_element {
    char data[6];
};

struct adxl345_device {
    DECLARE_KFIFO(samples_fifo, struct fifo_element, 32);
    struct miscdevice misc_dev;
    int axis;
};
```

4.3 - Bottom Half function adxl345_irq

A bottom half function for the interruption is then declared :

```c
irqreturn_t adxl345_irq(int irq, void *dev_id){
    struct fifo_element fifo;
    int samples_cnt;
    struct adxl345_device *adxl345_dev;
    struct i2c_client *client;
    char buffer[6];
    int bytes_read;
    int i;
    char reg_fifo_status;
    char reg_data_x0;

    adxl345_dev = (struct adxl345_device *)dev_id;
    client = (struct i2c_client *)to_i2c_client(adxl345_dev->misc_dev.parent);

    reg_fifo_status = ADXL345_REG_FIFO_STATUS;
    reg_data_x0 = ADXL345_REG_DATA_X0;

    samples_cnt = i2c_master_send(client, &reg_fifo_status, 1) & 0x3F;

    for (i = 0; i < samples_cnt; i++) {
        bytes_read = 0;
        i2c_master_send(client, &reg_data_x0, 1);
        while (bytes_read < 6)
            bytes_read += i2c_master_recv(client, buffer + bytes_read, 6 - bytes_read);
        memcpy(fifo.data, buffer, 6);
        kfifo_put(&adxl345_dev->samples_fifo, fifo);
    }

```

```
    return IRQ_HANDLED;
}
```

This function ask the ADXL345 the number of samples and then read the samples and add them the kernel device fifo.

The adxl345_probe is also changed to initiate the FIFO and the interruption, the following lines are added at the end of the prod function :

```
    INIT_KFIFO(adxl345_dev->samples_fifo);

    pr_info("FIFO_INIT_SUCESSFUL\n");

    if (devm_request_threaded_irq(&client->dev, client->irq, NULL,
     adxl345_irq, IRQF_ONESHOT, name, adxl345_dev)) {
        dev_err(&client->dev, "Failed to request IRQ %d\n",
         client->irq);
        return error;
    }

    pr_info("IRQ_INIT_SUCESSFUL\n");

    return 0;
```

4.4 - Modification to adxl345_read

The adxl345_read function now needs to be changed to read the kernel device fifo and give the data to the user. To do so, the following adxl345_read function was written :

```
static ssize_t adxl345_read(struct file *file, char __user *buf, size_t
 count, loff_t *ppos) {
    struct adxl345_device *adxl345_dev;
    struct i2c_client *client;
    // pr_info("Read function call\n");
    adxl345_dev = container_of(file->private_data, struct
     adxl345_device, misc_dev);
    client = container_of(adxl345_dev->misc_dev.parent, struct
     i2c_client, dev);

    struct fifo_element fifo;

    if (kfifo_get(&adxl345_dev->samples_fifo, &fifo)) {
        // char *buffer = fifo.data;
```

```c
        // pr_info("Result = X%hd Y%hd Z%hd\n\n", (short)(buffer[0] + (buffer[1] << 8)),(short)(buffer[2] + (buffer[3] << 8)), (short)(buffer[4] + (buffer[5] << 8)));

        if (copy_to_user(buf, &fifo.data, sizeof(fifo.data))) {
            pr_info("[ERROR] copy_to_user\n");
            return -EFAULT;
        }
        return sizeof(fifo.data);
    } else {
        pr_info("The FIFO is empty :( \n");
        return 0;
    }
}
```

4.5 - Modification to the user interface

To accommodate the change to the read function, the user interface now reads all the data until the fifo is empty every 500ms, the implementation can be seen below :

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <errno.h>

#define FILE_NAME "/dev/adxl345-0"

int main() {
    static volatile unsigned long long i;

    int file = open(FILE_NAME, O_RDONLY);
    if (file < 0) {
        perror("Error opening file");
        return 1;
    }

    char buffer[6];
    ssize_t bytes_read;

    for(;;){
        printf("\nSample %llu\n", ++i );
        do{
            bytes_read = read(file, buffer, sizeof(buffer));
```

```
            if (bytes_read !=0) printf("Result = X: %hd Y: %hd Z:
             ↪  %hd\n",(short)(buffer[0] + (buffer[1] <<
             ↪  8)),(short)(buffer[2] + (buffer[3] <<
             ↪  8)),(short)(buffer[4] + (buffer[5] << 8)));
        }while(bytes_read != 0);

        usleep(500000);
    }

    close(file);
    return 0;
}
```

4.6 - Results

The ADXL345 module and the user interfeace are then compiled

```
make CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm KDIR=../linux/build/
arm-linux-gnueabihf-gcc -Wall -static -o main main.c
```

And then tested :

```
# ./main

Sample 1
The FIFO is empty :(

Sample 2
Result = X: 0 Y: 1 Z: 2
Result = X: -4 Y: -5 Z: -6
Result = X: 8 Y: 9 Z: 10
Result = X: -12 Y: -13 Z: -14
Result = X: 16 Y: 17 Z: 18
The FIFO is empty :(

Sample 3
Result = X: -20 Y: -21 Z: -22
Result = X: 24 Y: 25 Z: 26
Result = X: -28 Y: -29 Z: -30
Result = X: 32 Y: 33 Z: 34
Result = X: -36 Y: -37 Z: -38
Result = X: 40 Y: 41 Z: 42
Result = X: -44 Y: -45 Z: -46
```

```
Result = X: 48 Y: 49 Z: 50
Result = X: -52 Y: -53 Z: -54
Result = X: 56 Y: 57 Z: 58
Result = X: -60 Y: -61 Z: -62
The FIFO is empty :(

Sample 4
Result = X: 64 Y: 65 Z: 66
Result = X: -68 Y: -69 Z: -70
Result = X: 72 Y: 73 Z: 74
Result = X: -76 Y: -77 Z: -78
Result = X: 80 Y: 81 Z: 82
Result = X: -84 Y: -85 Z: -86
Result = X: 88 Y: 89 Z: 90
Result = X: -92 Y: -93 Z: -94
Result = X: 96 Y: 97 Z: 98
Result = X: -100 Y: -101 Z: -102
The FIFO is empty :(
```

As it can be seen, everything works as expected.

5 | TP - 5 : Concurrence (Jules Franault et al.)

The goal of this TP is to fix concurrency issues that where present in the driver and user interface. Source code and compiled binaries can be found on Github.

5.1 - Configuration

- Computer OS : Ubuntu 24.04
- Kernel Image : Linux 6.14.0 rc1+

5.2 - Variable `number_of_device`

`number_of_device` is a global variable that allows to define the device name. However, it is possible that several devices are connected at the same time. So, we create several different device names.

Thus, the variable is initialized in atomic. And to increment it, an atomic function is used. This function returns the new number, so a -1 is added to the value.

```
atomic_t number_of_device = ATOMIC_INIT(0);
val = atomic_inc_return(&number_of_device)-1;
name = kasprintf(GFP_KERNEL, "adxl345-%d", val);
```

5.3 - Kfifo

`kfifo_get` is not thread-safe, so a mutex is added. A mutex is created with the function `static DEFINE_MUTEX(mutex_kfifo)` It is lock with `mutex_lock(&mutex_kfifo);` and unlock with `mutex_unlock(&mutex_kfifo);`.

```c
static DEFINE_MUTEX(mutex_kfifo);

static ssize_t adxl345_read(struct file *file, char __user *buf, size_t
    count, loff_t *ppos) {
    struct adxl345_device *adxl345_dev;
    struct i2c_client *client;
    adxl345_dev = container_of(file->private_data, struct
        adxl345_device, misc_dev);
    client = container_of(adxl345_dev->misc_dev.parent, struct
        i2c_client, dev);

    struct fifo_element fifo;
    mutex_lock(&mutex_kfifo);
    if (kfifo_get(&adxl345_dev->samples_fifo, &fifo)) {
        mutex_unlock(&mutex_kfifo);
        if (copy_to_user(buf, &fifo.data, sizeof(fifo.data))) {
            pr_info("[ERROR] copy_to_user\n");
            return -EFAULT;
        }
        return sizeof(fifo.data);
    } else {
        mutex_unlock(&mutex_kfifo);
        pr_info("The FIFO is empty :( \n");
        return 0;
    }
}
```

5.4 - I2C

The same problem exist, so another mutex for read and write in I2C is added.

```c
static DEFINE_MUTEX(mutex_i2c);

irqreturn_t adxl345_irq(int irq, void *dev_id){
    struct fifo_element fifo;
    int samples_cnt;
    struct adxl345_device *adxl345_dev;
    struct i2c_client *client;
    char buffer[6];
    int bytes_read;
    int i;
    char reg_fifo_status;
    char reg_data_x0;

```

```c
    adxl345_dev = (struct adxl345_device *)dev_id;
    client = (struct i2c_client *)to_i2c_client(adxl345_dev->misc_dev.parent);

    reg_fifo_status = ADXL345_REG_FIFO_STATUS;
    reg_data_x0 = ADXL345_REG_DATA_X0;

    mutex_lock(&mutex_i2c);
    samples_cnt = i2c_master_send(client, &reg_fifo_status, 1) & 0x3F;
    mutex_unlock(&mutex_i2c);

    for (i = 0; i < samples_cnt; i++) {
        bytes_read = 0;
        mutex_lock(&mutex_i2c);
        i2c_master_send(client, &reg_data_x0, 1);
        while (bytes_read < 6)
            bytes_read += i2c_master_recv(client, buffer + bytes_read, 6 - bytes_read);
        mutex_unlock(&mutex_i2c);
        memcpy(fifo.data, buffer, 6);
        kfifo_put(&adxl345_dev->samples_fifo, fifo);
    }

    return IRQ_HANDLED;
}

static int read_register_i2c(struct i2c_client *client, char reg_address, char *value) {
    char buffer[1];
    int error = 0;
    buffer[0] = reg_address;
    mutex_lock(&mutex_i2c);
    error = i2c_master_send(client, buffer, 1);

    if (error < 0) {
        pr_info("[ERROR] reading register (send) %x\n", reg_address);
        return error;
    }

    error = i2c_master_recv(client, value, 1);
    mutex_unlock(&mutex_i2c);

    if (error < 0) {
        pr_info("[ERROR] reading register (receive) %x\n", reg_address);
        return error;
    }
    return 0;
}
```

Conclusion

During these TPs, we where able to :

- Compile a linux kernel from source
- Simulate it with Qemu
- Add an initramfs to boot
- Setup a bootloader (U-boot)
- Modify the device tree to add a ADXL345
- Write a simple driver to communicate with the I2C device
- Add a user interface to interact with the ADXL345 from the userspace
- Use interruption and FIFO to asynchronously transfer the data
- Write a safe driver with the atomicity and mutex ensuring the lack of race conditions and deadlocks

Overall, it was an interesting project where we leaned a lot about the linux kernel, how to properly use it, write driver to interact between the hardware and a userspace.