Introduction

Objectives

The objective of this tutorial is to write the first part of a driver for an I2C device:

Driver skeleton
Registration to the I2C system
Declaration of supported devices
Initialization of the device when it is detected
Reset the device when it is removed
and test it to make sure everything works. The next labs will aim to complete this driver (communication with the user space, interrupts, etc.).

Pre-requisites (Make sure it is ready before/in lab)

Environment of Practical session

You must have kept from the previous TP :

The modified version of QEMU
The compiled sources of the kernel
The initial memory image
The cross compiler

Driver Skeleton

Code

Below is the skeleton of a driver for an I2C device.

Note that the names of functions and structures start with foo. By convention, we replace foo with the name of the device (e.g. for a driver for the ADXL345 device, we will name the functions adxl345_probe, adxl345_remove, etc. and the structures adxl345_idtable, adxl345_of_match, etc.)

The content and role of each part of this skeleton has normally been seen in class.

```
#include <linux/init.h>
```

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/of.h>
#include <linux/i2c.h>

static int foo_probe(struct i2c_client *client,
                     const struct i2c_device_id *id)
{
    /* ... */
}

static int foo_remove(struct i2c_client *client)
{
    /* ... */
}

/* The following list allows the association between a device and its driver
   driver in the case of a static initialization without using
   device tree.

   Each entry contains a string used to make the association
   association and an integer that can be used by the driver to
   driver to perform different treatments depending on the physical
   the physical device detected (case of a driver that can manage
   different device models).*/
static struct i2c_device_id foo_idtable[] = {
    { "foo", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, foo_idtable);

#ifdef CONFIG_OF
/* If device tree support is available, the following list
   allows to make the association using the device tree.

   Each entry contains a structure of type of_device_id. The field
   compatible field is a string that is used to make the association
   with the compatible fields in the device tree. The data field is
   a void* pointer that can be used by the driver to perform different
   perform different treatments depending on the physical device detected.
   device detected.*/
static const struct of_device_id foo_of_match[] = {
    { .compatible = "vendor,foo",
      .data = NULL },
    {}
};
```

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/of.h>
#include <linux/i2c.h>

static int foo_probe(struct i2c_client *client,
                     const struct i2c_device_id *id)
{
    /* ... */
}

static int foo_remove(struct i2c_client *client)
{
    /* ... */
}

/* The following list allows the association between a device and its driver
   driver in the case of a static initialization without using
   device tree.

   Each entry contains a string used to make the association
   association and an integer that can be used by the driver to
   driver to perform different treatments depending on the physical
   the physical device detected (case of a driver that can manage
   different device models).*/
static struct i2c_device_id foo_idtable[] = {
    { "foo", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, foo_idtable);

#ifdef CONFIG_OF
/* If device tree support is available, the following list
   allows to make the association using the device tree.

   Each entry contains a structure of type of_device_id. The field
   compatible field is a string that is used to make the association
   with the compatible fields in the device tree. The data field is
   a void* pointer that can be used by the driver to perform different
   perform different treatments depending on the physical device detected.
   device detected.*/
static const struct of_device_id foo_of_match[] = {
    { .compatible = "vendor,foo",
      .data = NULL },
    {}
};
```

```
MODULE_DEVICE_TABLE(of, foo_of_match);
#endif

static struct i2c_driver foo_driver = {
    .driver = {
        /* The name field must correspond to the name of the module
           and must not contain spaces. */
        .name   = "foo",
        .of_match_table = of_match_ptr(foo_of_match),
    },

    .id_table       = foo_idtable,
    .probe          = foo_probe,
    .remove         = foo_remove,
};

module_i2c_driver(foo_driver);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Foo driver");
MODULE_AUTHOR("Me");
```

Makefile

As a reminder (see the tutorial on writing the first module), here is the Makefile needed to compile the module (assuming that the source file is named foo.c):

```
ifneq ($(KERNELRELEASE),)
# kbuild part of makefile
obj-m  := foo.o

else
# normal makefile
KDIR ?= /lib/modules/`uname -r`/build

default:
    $(MAKE) -C $(KDIR) M=$$PWD
endif
```

Warning! The line $(MAKE)-C... which indicates the command to be executed to build the default target must be indented with a tab and not with one or more spaces.

The ADXL345 acceleromoter

The TP board that we should have used is equipped with a 3-axis accelerometer (ADXL345 from Analog Devices) connected to the I2C0 controller via an I2C bus. The accelerometer also has an interrupt output, connected directly to one of the FPGA inputs.

If you are not able to do this test in person, the special version of QEMU that you got during the previous test contains a model (a simulation) of this accelerometer.

This model behaves very closely to the physical accelerometer, except for a few details:

The recovered data do not come from a real measurement of the acceleration
The sampling frequency is fixed (the BW_RATE register is ignored)
The data format indicated in the DATA_FORMAT configuration register is ignored (the data is always represented on 16 bits in 2's complement)
Only one interrupt output is modeled (the INT_MAP register is ignored)
Tap, Double Tap and Freefall functions are not modeled
The only interrupt source is the FIFO watermark
The Trigger mode of the FIFO is not implemented
Here is how the data are generated. At each measurement, a counter c (initialized to 0) is incremented. The values produced on the different axes are :

- if `c % 2 == 0`, `X = c * 4` then `X = -c * 4`
- if `c % 2 == 0`, `Y = c * 4 + 1` then `Y = -(c * 4 + 1)`
- if `c % 2 == 0`, `Z = c * 4 + 2` then `Z = -(c * 4 + 2)`

The first samples are therefore : `(0,1,2), (-4,-5,-6), (8,9,10)`...

This information will allow you to verify, in the following exercises, that you are recovering all the sampled data without losing any.

You will gradually write a device driver for this simulated accelerometer.

Start by taking a quick look at the documentation for the real component:
Documentation of ADXL345

Provided Device Tree:

This ADXL345 device is declared in the tree of devices that you recovered during the previous tutorial.

Here is an interesting extract:

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

Do not consider for the moment the lines starting with interrupt.

&v2m_i2c_dvi points to the I2C controller to which the accelerometer is virtually connected. The accelerometer is placed at address 0x53 on this bus in QEMU.

Work to Do!!
First Step (Assignment):

1. Create a working directory for your first module:

```
$ cd $TPROOT
$ mkdir pilote_i2c
$ cd pilote_i2c
```

2. Copy the above skeleton (naming it adxl345.c) and the Makefile

3. Adapt the provided skeleton to :

   -It respects the conventions (replace foo by the name of the device in the names of functions and structures, put correct values in the MODULE_xxx macros...)
   -That it associates well with the device declared in the device tree (hint: look at the compatible strings)
   -That it displays a message when a supported device is detected
   -That it displays a message when a supported device is removed
   Also adapt the provided Makefile so that the module can compile.

4. Compile your module

```
$ make CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm KDIR=../linux-5.10.19/build/
```

Your module is now in the adxl345.ko file.

5. Start QEMU

```
$ cd ..
$ ./qemu-system-arm -nographic -machine vexpress-a9 -kernel linux-
5.10.19/build/arch/arm/boot/zImage \
-dtb linux-5.10.19/build/arch/arm/boot/dts/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
```

6. After logging in as root, mount the share with the host machine by typing, in the console emulated by QEMU :

```
$ mount -t 9p -o trans=virtio mnt /mnt -oversion=9p2000.L,msize=10240
```

If this command is executed correctly, you will then have access, in the /mnt directory of the machine emulated by QEMU, to the contents of the driver_i2c directory of your machine (and thus to the module that you have just compiled).

7. Load your module (from QEMU):

   ```
   $ insmod /mnt/adxl345.ko
   ```

8. Check that your module has the expected behavior (display your device discovery message)

9. Unload your module (rmmod adxl345) and check that it has the expected behavior (display your device removal message)

10. You can stop QEMU (ctrl-a c then quit)

Second Step:

For the moment your driver does not do much. In this second step we will try to read the DEVID register of the accelerometer. This register contains a fixed value (0xe5). If we get this value, it will show that we are able to exchange data with this device using the I2C API of the kernel.

Assignment:
Modify the probe function of your module to retrieve the value of this register from the physical device, and display it.

Hint: you will need to do a write followed by an I2C read. The write (i2c_master_send) and read (i2c_master_recv) functions have been detailed in class.

To recompile and reload your module, refer to the previous step.

Third Step (Assignment):

We will now modify our driver so that it correctly configures the physical device.

In the probe function, configure the different registers of the accelerometer. We will use the following configuration, although some values are not implemented in the simulation (see the accelerometer documentation provided to you):

- Output data rate: 100 Hz (output data rate, BW_RATE register)
- All interrupts disabled (INT_ENABLE register)
- Default data format (DATA_FORMAT register)
- FIFO bypass (bypass mode, FIFO_CTL register)
- Measurement mode activated (POWER_CTL register)

In the remove function, switch off the accelerometer:

- Standby mode (POWER_CTL register)