Computation II - 5EIB0
Lab 5: Creating a Linux driver v1.5

Diederik Markus   Wouter Schoenmakers   Patrick Wijnings

In this lab you will create a Linux driver for the counter from Lab 1. Using this driver you are going to enable the counter and read its output while running Linux on the board.

Contents

1 Booting Up Linux

Task 1. Set the boot jumper to the SD position (Fig. 1). It is located in the top right corner of the Pynq.

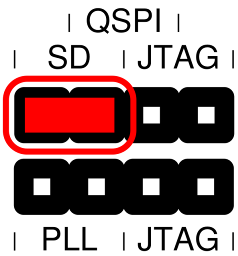

Figure 1: Correct position of boot jumper.

Task 2. Establish a network connection between the virtual machine and the Pynq. Follow the instructions given in Sec. 2.1–2.4 in the *Pynq Basics Tutorial* (OnCourse).

After following the instructions you should be able to log in on the Pynq using the command `ssh student@pynq.local` in the virtual machine. Username and password are both **student**.

2 The Driver

Writing to the counter from Linux is a bit more complex than it was previously. In Lab 1 you were able to write directly to the counter using pointers because your code was the only thing running. Instead, Linux has multiple processes running at the same time, each with their own virtual memory space. This is done so that applications cannot spy on private information (such as bank accounts or passwords) of other applications. As a consequence, using a normal Linux program it is impossible to access a specific part of the real (physical) memory space. For this you need a driver.

A driver template is given in `counter_driver.c` (OnCourse). You will only need to write the `counter_read` and `counter_write` functions, the rest is already there.

Task 3. Create a folder `driver` in the Pynq shared folder (`/shared/ on pynq` in the File Manager of the VM (Fig. 2)). Download the following files from OnCourse and copy them into this folder:

- `counter_driver.c` (driver template)
- `pl.dtsi` (overlay definition)
- `pl.txt` (overlay description)
- `makefile_driver` (makefile)
  Change the name of this file to `Makefile` *with a capital M and no extension.*

These files are needed for building and loading your driver. `Makefile` is used to compile the driver such that it can be loaded into the kernel. `pl.dtsi` and `pl.txt` are an overlay that explains Linux where the counter is located in the FPGA (i.e. its physical memory address) as well as which driver to load.

The job of a driver is to convey data and instruction from a normal process (called the *user*) to the hardware and back. In the case of your counter you want to use the driver to read the `count` value from the counter and send it back to the user that prints it to the terminal.

The driver and the user communicate through a special file called the device file. For our counter, this file will be located in `/dev/tue_counter`. When the user writes to this file, the Linux kernel calls the `counter_write` function in the driver. When the user *reads* from this file, the `counter_read` function in the driver is called.

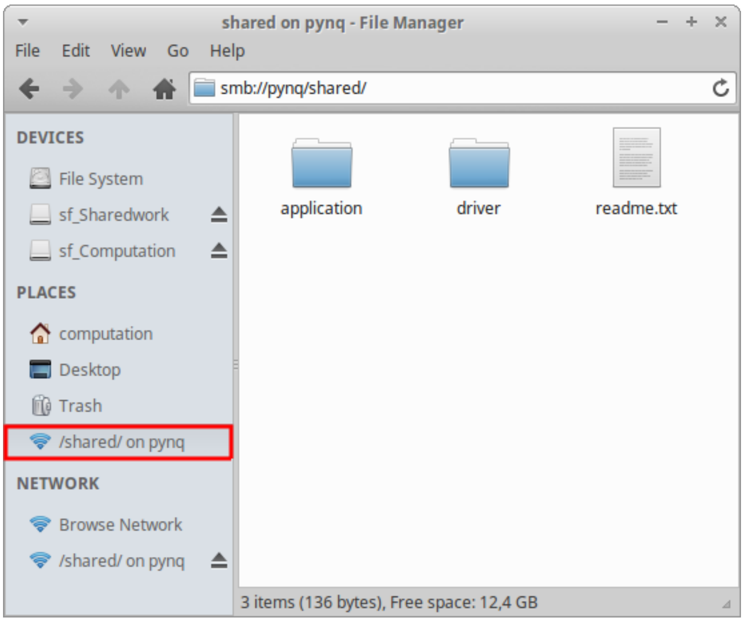

Figure 2: The shared folder of the PYNQ board.

Task 4. Finish the `counter_write` function in `counter_driver.c` on line 96. It should write the data that it receives to the counter registers (16 bytes, 4 for each register). Check whether the `size` and `offset` inputs are valid, then copy the values from the user using `copy_from_user(..)` (`https://www.fsl.cs.sunysb.edu/kernel-api/re257.html`). Follow the TODO comments in the code and consult the two notes below.

Task 5. Finish the `counter_read` function in `counter_driver.c` on line 78. It should read the values from the counter registers to the user buffer (16 bytes, 4 for each register). Check whether the `size` and `offset` inputs are valid, then copy the values to the user using `copy_to_user(..)` (`https://www.fsl.cs.sunysb.edu/kernel-api/re256.html`). Again, follow the TODO comments in the code and consult the two notes below.

Note: A driver does not have access to the C standard library. This means functions such as `printf` and `malloc` cannot be used. You can still print messages to the terminal with `printk(KERN_DEBUG "...");`. Use the `dmesg` command in the Pynq terminal to read these messages.

Note: Both functions, `counter_write` and `counter_read`, take the same four arguments:

- `file` represents the device file. **You do not need to do anything with this argument.**
- `buffer` is a pointer to a buffer in the virtual memory space of the user. Hence, you can not read from/write to this buffer directly. Instead, use `copy_from_user(..)` (`https://www.fsl.cs.sunysb.edu/kernel-api/re257.html`) and `copy_to_user(..)` (`https://www.fsl.cs.sunysb.edu/kernel-api/re256.html`) to copy the memory into the memory space of the kernel.
- `size` is the number of bytes that the user wants to read or write (i.e. the length of `buffer`).
- `offset` **points** to the offset (in bytes) relative to the base address (i.e. the first register of the counter) from which the user wants to read or write.

The functions must return the number of bytes that were copied, or an error code if it failed. If one or more arguments were invalid, use the error code `-EINVAL`. If `copy_to_user` or `copy_from_user` could not copy all bytes successfully, return `-EFAULT`. Both `copy_to_user` and `copy_from_user` return the number of bytes that were **not** copied successfully. In the template code a struct containing a pointer to the counter registers is provided in the first two lines of both `counter_write` and `counter_read`. It is the pointer called `reg` in the `counter_private` struct. Thus, when you need the address of the first register, use `&priv->reg[0]`. Each element of `reg` is 32 bits wide.

Task 6. In the terminal *of the Pynq board* (see Task 2), browse to `shared/driver`. Build the driver by first running `sudo make` and then `sudo make install`. The sudo password is **student**. If everything went correctly you should now have a file called `counter_driver.ko` and a file called `pl.dtbo`.

3 Installing the Driver

Note: Every time you reboot the Pynq board you need to redo all the tasks in this section. They are undone by Linux each time it boots up.

You now have a driver for the counter. However, before you can run it you need to program the FPGA with the counter bitstream, as well as install your driver.

The FPGA can be programmed through Vivado while Linux is running; you do not have to shutdown Linux to do this. However, you do need to tell Linux that the FPGA is going to change. You can do so by loading a new *overlay file*.

Task 7. Unload the default overlay file with the command `sudo overlay unload buffer_stream`. Then, **while you keep the board running**, open the Hardware Manager in Vivado and connect the board using `Open Target` → `Auto Connect`. Click on `Program device` and select the bitstream for this lab that is provided on OnCourse.

Note: If the connection fails, do not reboot the board but click on `localhost` → `Close server` instead. You can then try to open the target and program the FPGA again. Keep trying until you successfully programmed the board. It is normal that the LEDs turn off during this step. However, when programming has finished, the DONE LED should be on. **Do not change any jumpers on the board**.

The next step is to tell Linux about the new hardware and your driver. The overlay `pl.dtsi` that you copied to the Pynq in Task 3 should already have been compiled when you ran the `make` commands for your driver in Task 6.

Task 8. Browse to `/home/student/shared/driver` on the Pynq. Use `sudo overlay load pl.dtbo` to load the overlay and `sudo insmod counter_driver.ko` to load the driver. If everything went correctly, the device file `/dev/tue-counter` should now exist. You can also check the debug messages generated by the driver using `dmesg`. The LEDs should not start blinking yet, since the counter is still disabled (i.e. the `enable` input is low).

Hint: `lsmod` list all loaded modules (including your driver if it is loaded) and `rmmod` can be used to unload a module.

Note: The functions `counter_write` and `counter_read` are called only when you read from/write to the device file. You should not yet see any debug messages from these functions in this section.

4 The User Application

The user program is a normal C program with a main function. It writes to/reads from the device file to interact with the counter. There is a template available on OnCourse.

Task 9. In `/shared/ on pynq` (where you also created the `driver` folder), create a folder called `user`. Download the following files from OnCourse and copy them into this folder:

- `counter.c` (application template)
- `makefile_user` (makefile)
  Change the name of this file to `Makefile`.

You can read from/write to the device file just like you would for a normal file, e.g. using the C standard library functions `read` and `write`. However, the size of all reads and writes should match with what the driver expects, i.e. 16 bytes (see Task 4 and 5).

The registers of the counter are connected as follows:

- `slv_reg0` → reset (lowest bit)
- `slv_reg1` → enable (lowest bit)
- `slv_reg2` → counter (lowest 4 bits)
- `slv_reg3` → unconnected

The registers are each 4 bytes long which means that if you write to the first 4 bytes of the file you write to `slv_reg0`, if you write to the next 4 bytes, you write to `slv_reg1`, etc. To reset the counter, you should set both the *enable* and *reset* bit high. Then, you should write to the device file again to set *reset* low again. This will enable the counter.

Task 10. In `counter.c` finish the main function (`int main(void)`) such that it first enables the counter and than continuously keeps printing the output. You can use `sleep(..)` to limit the amount of messages in the terminal. Use the functions `read` and `write` to read and write to the device file (see notes below). Compile your program using `make` and then run it using `sudo ./counter`. If your code works correctly, then the LEDs should start counting up.

Note: `ssize_t result = read(fd_counter, buffer, length)`

- `fd_counter`: An ID number associated with the file you opened using `open`.
- `buffer`: A buffer to which the bytes that you want to read should be written. Should be the address of an uint32_t array.
- `length`: The amount of bytes that you want to read.
- `result`: The number of bytes that were read. If it fails to read it returns an error code that is always less than 0.

Note: `ssize_t result = write(fd_counter, buffer, length)`

- `fd_counter`: An ID number associated with the file you opened using `open`.
- `buffer`: A buffer containing the data that you want to write. Should be the address of an uint32_t array.
- `length`: The amount of bytes that you want to write.
- `result`: The number of bytes that were written. If it fails to write it returns an error code that is always less than 0.

Hint: If you edited your driver you can reload it with the following steps:

1. `sudo rmmod counter_driver`
2. `sudo make`
3. `sudo make install`
4. `sudo insmod counter_driver.ko`

This only works if you did not shutdown the board.

Hint: You can use Ctrl + C to terminate a running program and `dmesg` shows the printk messages from the driver.