Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Fall Term 2018

SYSTEMS PROGRAMMING AND COMPUTER ARCHITECTURE
Assignment 13: Writing a Device Driver

Assigned on: **19th Dec 2018**
Due by: **Open-ended**

1 Introduction

In this assignment, you have been asked to write a driver for new hardware (the `MPFC` device), which generates important data for human consumption.

The device uses a single descriptor ring and DMA to transfer data to main memory on the host, and interrupts to signal conditions such as the arrival of a new item of data, or running out of receive buffers. It also has several hardware registers, used for control and status information as well as setting up DMA descriptors.

The rate at which the device generates data is somewhat unpredictable. Sometimes, it will generate a burst of data items very quickly. However, there can also be lengthy pauses between items arriving.

Your driver should:

a) Initialize the device and the required data structures in memory.

b) Start the device, causing it to start sending data to your driver using DMA.

c) Receive the data and examine it.

d) Use interrupts to avoid "spinning" waiting for the device - in other words, your driver should go to sleep if there is no data to receive, and wake up when the device has more data.

e) Keep running indefinitely - in other words, when you receive buffers of data from the device, you should return them to the device when you are done with the data.

2 Device details

The device communicates with the driver using a *descriptor ring*, one interrupt, and four registers.

The descriptor structure is defined in the `device.h` file with a name `mpfc_desc`. The comments in the definition code should provide more information about the role of each member of the structure.

Following is the list of registers and small description.

a) **base register**: This register holds the address of the buffer ring of the descriptors.

b) **size register**: This register holds the size of the buffer ring of the descriptors.

c) **control register**: This register provides control over starting/stopping the device and configuring the interrupts generated by the device.

d) **status register**: This register allows finding out the current state of the device like:

(a) Is device running or stopped?
(b) Are the pending interrupts?
(c) Is it out of descriptors?

You can also write into this register to clear the error and interrupt bits.

The `device.h` file declares functions for reading and writing these device registers.

3 Implementation Tips

This section will give you some hints which will help you in solving the assignment.

3.1 Step-1: Setup Interrupt handling

In this step, you will write your own interrupt handler and register it.

This device can generate an interrupt on two conditions:

a) When new data generated by the device is ready for the consumption.

b) When device is out of buffers to generate more data.

These interrupt notifications avoid expensive polling when waiting for new data and also provide alternate error handling paths. The driver can enable or disable these interrupts by modifying the *control register*, and can read the current interrupt values by reading the *status register*.

You can write your interrupt handler by extending the `my_irq` function in the `main.c` file. There are comments in the function explaining what needs to be done.

Also, you may want to look at the description of `os_wakeup, os_sleep` functions in the `device.h`, as you will need to use these to avoid expensive polling.

Use the function `os_register_irq` to register your interrupt handler. This function tells the system which function to call when it wants to send an interrupt notification.

You need to register the interrupt handler only once as part of the driver initialization, but you can maintain control over interrupt generation by writing to the *control register*.

3.2 Step-2: Create and Register the descriptor ring

In this step, you should create a *descriptor ring* and inform the device about it.

For completing this assignment, you need to understand the descriptor structure `mpfc_desc` which is used for communication between the device and your device driver. You can find it in the `device.h` file. The **descriptor ring** is essentially a list of these descriptors. You should use the `base` and the `size` registers to inform the device about where your descriptor ring is in memory and how big it is. For this exercise we ignore the difference between virtual and physical memory.

You can use *malloc* and friends for creating the descriptors and actual buffers.

3.3 Step-3: Start the device

By now all the device initialization is complete, so the device can be started. This can be done by writing a proper value into the `control` register. After this point, your driver should go into a loop handling new data generated by the device and consuming it.

3.4 Step-4: Consume the data

This step will be performed indefinitely in a loop. In each iteration, the driver will try and consume one more line of generated data and return the buffer to the device to be reused for further data generation.

The flag `owned` in the descriptor plays an important role here.

You should also handle the case where the device has not generated any new data.

Also, note that there is a flag in the descriptor which the device uses to indicate that it had to discard some data, for example if the buffer associated with the descriptor is not large enough.

4 Provided code

We provide you several files:

device.h This header file specifies the format of the DMA descriptors used by the device, and contains declarations of functions to read and write the device registers. It also contains comments that explain what the formats of the registers are. Finally, it also gives an interface to the "operating system" to put the driver to sleep, wake it up, and register for interrupts.

libmpfcdevice.a This is a library which contains the emulator for the device. You'll link your driver with this library, which implements the functions in `device.h`. The provided objects requires that your machine architecture is x86-64.

main.c This is the outline for your driver program, and this is what you need to worry about! You'll modify and extend this code. This code is linked with the library and with POSIX thread library which is needed by the device emulator.

You should be able to compile your code by just running the `make` command, and you can run it by running `make run` command. As the program runs in infinite loop, you should use `CTRL+C` to stop it.

5 The final output

If your device driver works correctly, then it will be able to produce some meaningful output as a reward for your hard work :-)

Hand In Instructions

You can submit your code by committing into your SVN repository under new directory with name `assignment13`.