ES 6 - HW assignment

Write your first kernel …..

Johri van Eerd
Mark Ehrhart

Johri van Eerd
Mark Ehrhart
T61 15-3-2016

Assignment 1 HW:

Write a user space program to read the RTC, describe your findings:

First we tried a simple C++ program that uses the " /dev/rtc/" directory. As expected this worked like charm, although this was not the goal of the assignment.

To read the RTC from user space we thought we had to use the CMOS register. Some google results later we arrived at the following page for the CMOS RTC addresses : http://stanislavs.org/helppc/cmos_ram.html and using this site we got some program examples: http://kernelx.weebly.com/cmos.html although this website was using DOS defined variables. Using the Unix MAN pages we found out that " outportb == outb " for linux users.
After trying this for a long time and consulting the tutor and co-students it turned out that reading the CMOS is not the way to go, although we did learn some interesting facts, now we know we should read the registers and write them into a variable and try to print them.

To test this we wrote a user space program that will declare and assign an **int**. We will print the memory address of this **int** and try to access this memory address from a different user space program. The MMU (memory management unit) should not allow us to do this and give us a segmentation fault.
For example our first user space program created and reserved an **int** on the following memory address : 0xbfa835cc, when we tried to read from this memory address or write on this address we received the segmentation fault we were expecting.
To confirm that it was not a program or code error, we tried reserving that same **int** on the memory and then altering it within the same user space program. This was allowed because we were only accessing our own segment of memory.

We tried this as well on the LCP3250, this had the same results as before. We could read the variable that we assigned in our program, but when trying to read some of the reserved addresses on the board's memory chip we encountered a segmentation fault.

Now we wanted to try the same test with an address of a hardware register, for this we used the RTC UP counter address that was found on page 569 in the LCP data sheet. When we tried reading from this address on our laptop we got another segmentation fault this is because we have no idea if this hardware address is used or accessible or even points to the same register that we expect to find at that address.
On the LCP3250 we were able to read the address successfully, but at first we got a negative value when reading the register. Then we made the variable an **unsigned long long** and were able to read the value successfully.

Now for the reason that we are able to read this RTC register without the MMU interfering: We think a logical explanation could be that the RTC is an entity that

functions on its own, and so it is not influenced by the MMU. It has it owns power supply (0.9volt) and own SRAM (32bytes).
Partly confirming this thought is page 5 in the datasheet:

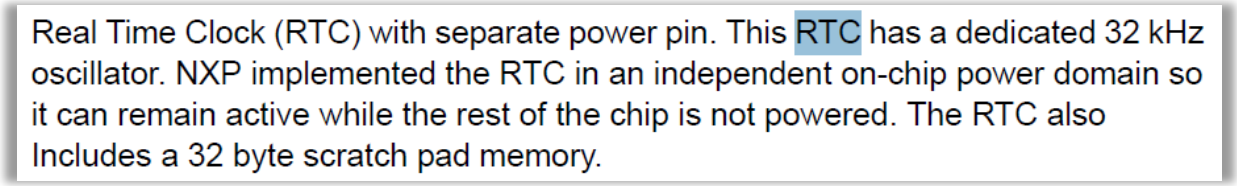
Real Time Clock (RTC) with separate power pin. This RTC has a dedicated 32 kHz oscillator. NXP implemented the RTC in an independent on-chip power domain so it can remain active while the rest of the chip is not powered. The RTC also Includes a 32 byte scratch pad memory.

Image 1: page 5 LPC3250, RTC information

After this we managed to at least verify that we can reach the memory but we still can't read what is in it. The address that we now dereferenced was already mapped in the system page table so it was already linked to a physical address somehow. However as we are still operating from user space the value we request is most likely the mapping from virtual memory to the physical memory. So we requested the virtual address 0x40024000 which is in the standard user space range but it not yet claimed by another process.
A possible explanation for why this would work on the LPC3250 but not on one of our laptops, is that the MMU of an ARM processor (found on the LPC3250) and the MMU of an Intel processor (found on our systems) behave differently when it comes to user space reserved address pools. This could result in the LPC3250 process variant accessing a memory address not currently in use by another process so it was allowed to read it. On our laptop, a lot more processes are running in the background so the possibility of finding a "free" memory address is a lot smaller, if the Intel MMU would even allow us to do so on our system.

Assignment 2 : Write a kernel module that can read and write the hardware registers

For this we started with reading the LKMPG chapter 6, so first we tried the helloWorld example. Then we created and tested the write/read example.

Using the code provided by LKMPG (http://wiki.tldp.org/lkmpg/en/2_6#ch6) we started out trying to understand what was happening in this specific kernel module. Important for us was to truly understand the code that allows us to send a message to the kernel module. When we were doing this we wanted to implement logic for our kernel protocol.

The first challenge was interpreting the incoming **char[]** that is supplied by sysfs when calling 'echo' on a kernel module. We tried using sscanf & sprintf to write the array into a physical memory address. This did not work correctly because the Linux kernel modules do not have access to all the libraries we are normally use to in user space. So we had to find a replacement that was available in the kernel modules. We ended up settling with simple_strtol() with that library function we were also able to have variable lengths of our supplied parameters/ user input. The kernel module protocol we defined can be found below.

The next challenge we encountered was finding an appropriate way to read the physical addresses, luckily our co-student Menno Sijben informed us that we could use ioread32() method for this, that is included in the io.h library. This method allows us to supply a virtual address as an **int** . Although we could not access this address in user space now in kernel space we were able to read and write to such addresses.

While experimenting with the sysfs_store() method we also discovered some interesting behavior that has to do with the return value. Namely that if the return value is not equal to the **count** parameter supplied by sysfs then sysfs will try to call the store method again with the remainder of the message that was allegedly not handled.

Kernel Protocol :

Our kernel module can read physical memory and write to physical memory. For using our kernel module it is important to follow the protocol. First some examples of calling our kernel.

Examples:

```
echo "r 7 0x40024000" > /sys/kernel/hwReadWrite/result
```
Read 7 physical memory registers *starting at 0x4002400*

```
echo "r 3 0x56fe9382" > /sys/kernel/hwReadWrite/result
```
read 3 physical memory registers *starting at 0x56fe9382*

```
echo "w 0x56fe9382 abc" > /sys/kernel/hwReadWrite/result
```
write 0xabc to physical memory *at address 0x56fe9382*

```
echo "w 0x12345678 678" > /sys/kernel/hwReadWrite/result
```
write 0x678 to physical memory *at address 0x12345678*

Johri van Eerd
Mark Ehrhart
T61 15-3-2016