ES6

Peek & Poke

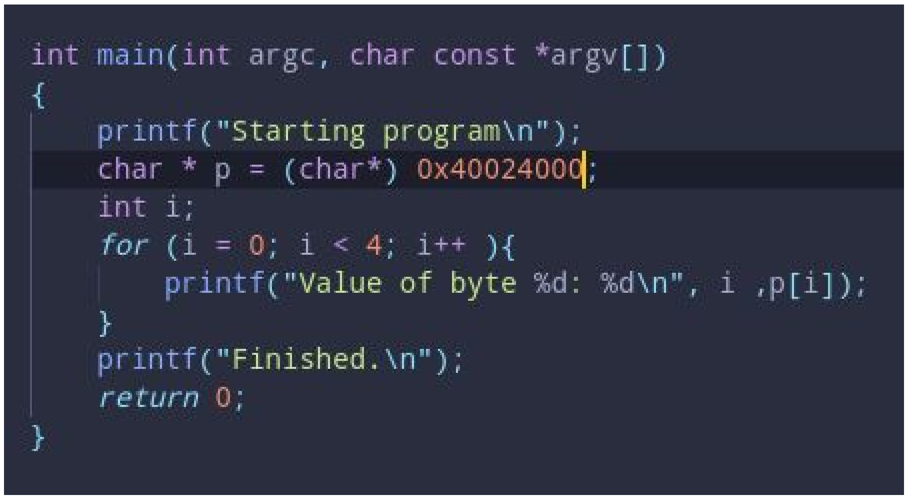
Rick van Melis & Simon Lit

15-03-2019

Part 1 - Reading the RTC

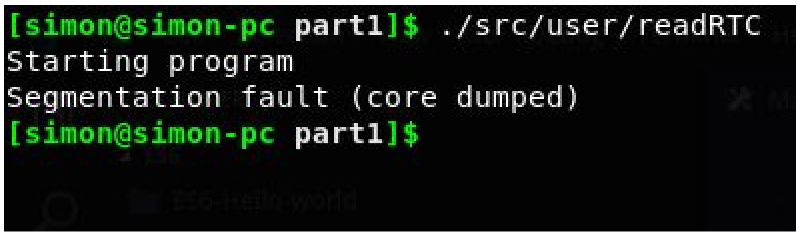
Our findings

Code below is the program used for testing if the RTC clock can be read on an embedded device and to see what happens when running it on Ubuntu.



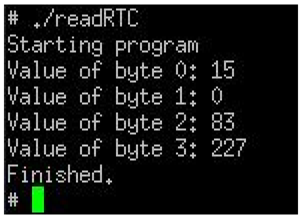
```c
int main(int argc, char const *argv[])
{
    printf("Starting program\n");
    char * p = (char*) 0x40024000;
    int i;
    for (i = 0; i < 4; i++ ){
        printf("Value of byte %d: %d\n", i ,p[i]);
    }
    printf("Finished.\n");
    return 0;
}
```

Trying out the program on our ubuntu results in to the following:



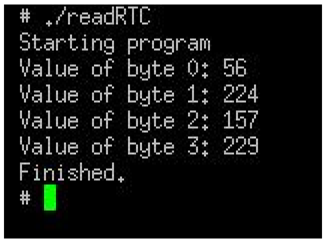
```
[simon@simon-pc part1]$ ./src/user/readRTC
Starting program
Segmentation fault (core dumped)
[simon@simon-pc part1]$
```

Trying it out on our embedded LPC3250 the following happens:



```
# ./readRTC
Starting program
Value of byte 0: 15
Value of byte 1: 0
Value of byte 2: 83
Value of byte 3: 227
Finished.
#
```

We are trying to read from address 0x4002 4000 which is according to the datasheet of the LPC32x the address that matches with the “RTC Up Counter Value register” which when you read it reflects the current value of the real time clock. When reading it twice, or even on another LPC3250 board is gives back the same values. Clearly an indication that we are not reading the value of the clock. (the value is upped once every second, so getting the same number is not a option)

As experiment we tried, entering memory address 0x 4003 c008 which is the address where you can read the value of the “watchdog timer counter register” we get the following result. 

```
# ./readRTC
Starting program
Value of byte 0: 56
Value of byte 1: 224
Value of byte 2: 157
Value of byte 3: 229
Finished.
# 
```

On ubuntu we still of course have an “segmentation fault”.

This is caused because we are trying to read a physical memory address from the user space. Basically it's quite senseless to try and directly read a physical address from the user space, as the system takes our input as a virtual address and maps it to the belonging physical address which can be a totally different address. Since we do not know the mapping, we cannot say what we are reading. This is the case for the embedded Arm board. However on ubuntu the MMU does not allow us to read a register that does not belong to us. Why we can read some of these on our Arm board we will explain further on.

The behavior we have encounter is caused by the memory management unit or in short MMU. It’s job is to perform the translation of virtual memory addresses to physical addresses, memory protection, cache control, bus arbitration and sometimes bank switching.

On Ubuntu the memory protection functionality of the MMU prevents us from reading an physical address that does not belong to us. (as it should)

Then the question is raised, why are we allowed to access the memory on our embedded board. Should it not be protected by the MMU just like in Ubuntu?

Yes it should but it isn’t. There are multiple theories to why this is happening that we came up with although we can’t verify any of them to be true.

Options

One option is if the kernel was compiled with `CONFIG_DEVKEM=y` and there are no LSMs (linux security models) that are blocking access and `CAP_SYS_RAWIO` is set (in other words root has all capabilities and these can be granted to other processes) can it be sufficient enough to write to `/dev/kmem` which is how once upon a time module loading worked. `/dev/kmem` is a character device file that is an image of the main memory of the computer. It may be used to examine and patch the system. Unlike `/dev/mem` it gives access to the kernel virtual memory.

Another option is that the peripheral register are memory locations that are mapped to specific addresses in the processor address space thus them being accessible via software. In other words they are are mapped at physical addresses and not virtual ones, so even in operating systems with memory protection they are accessible from within the OS kernel. Although testing this out by accessing different peripherals on the board does not always give us the same result as with the RTC and the Watchdog timer. So we can conclude this is not the case.

Last but not least there is the option that the addresses have been remapped and these we need have different protection levels that the rest of the memory. There are multiple levels of protection in the form of `PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE` this can be achieved by using the **`mmap`** function. We did notice in a device called `/dev/shm` which is shared memory. Although there could have been tampered with the device mappings, software error or even an hardware bug.

Conclusion

The conclusion is for this part that we just don’t know what exactly it is with this board. It might be an option that we haven’t explored or came up with. But with did find a few good contenders.

Part 2 - Creating a kernel module

There are multiple options for creating a kernel module, we started with creating an `/dev` driver but ended up with building a `/sys` kernel module. This code is attached with to this document.

What is *sysfs*

There is a need to provide information related to each process, to the user space, which can then be used by programs such as ps. Originally *procfs* was created for this but because of how easy it is to add new directories and files to the */proc* file system it became cluttered with lots of non-process related information. This is why *sysfs* has been developed in the linux 2.5 cycle. Sysfs is a RAM based file system, designed to export the kernel data structures and their attributes from the kernel to the user space.

Each folder in the *sysfs* directory are represented by kernel object also know as *kobjects*. By using `kobject_create_and_add(example, NULL)` we can create a directory where we will add all our attributes. An attribute is simplistically speaking a file, which follows the *one item per file* rule. Which means that only one value can be put in it or read from it. This is one of the advantages of *sysfs* as it forces the developer to create cleaner interfaces.

An attribute can be created with:

```
struct kobj_attribute example_info_attr = __ATTR(example_info, 0666, example_show,
example_store);
```

And needs to be added to the attribute list, important is that the list is NULL terminated

```
struct attribute *example_attrs[] = {
&example_attr.attr,
NULL,
};
```

After this the attributes have to be given to the attribute group as follows:

```
struct attribute_group example_attr_group = {
.attrs = example_attrs,
};

int sysfs_create_group(struct kobject *kobj, const struct attribute_group *grp);
```

And finally we can call the *sysfs* API call that will create all our files (attributes) like show in the above code snippet.

Now a file named *example_info* will be created in the */sys/kernel/example_dev* directory. Whenever that file is read, the *example_show* function will be called and whenever the file is written the, *example_store function* will be called. (is has to be defined of course)

What is the difference between /Dev vs /Sys

The /sys filesystem (sysfs) contains files that provide information about devices: whether it's powered on, the vendor name and model, what bus the device is plugged into, etc. It's of interest to applications that manage devices.

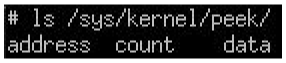
The /dev filesystem contains files that allow programs to access the devices themselves: write data to a serial port, read a hard disk, etc. It's of interest to applications that access devices.

A metaphor is that /sys provides access to the packaging, while /dev provides access to the content of the box.

Source: https://unix.stackexchange.com/questions/176215/difference-between-dev-and-sys

Our solution

We have implemented a driver that lists 3 files in /sys/kernel/peek namely:
address, count and data.

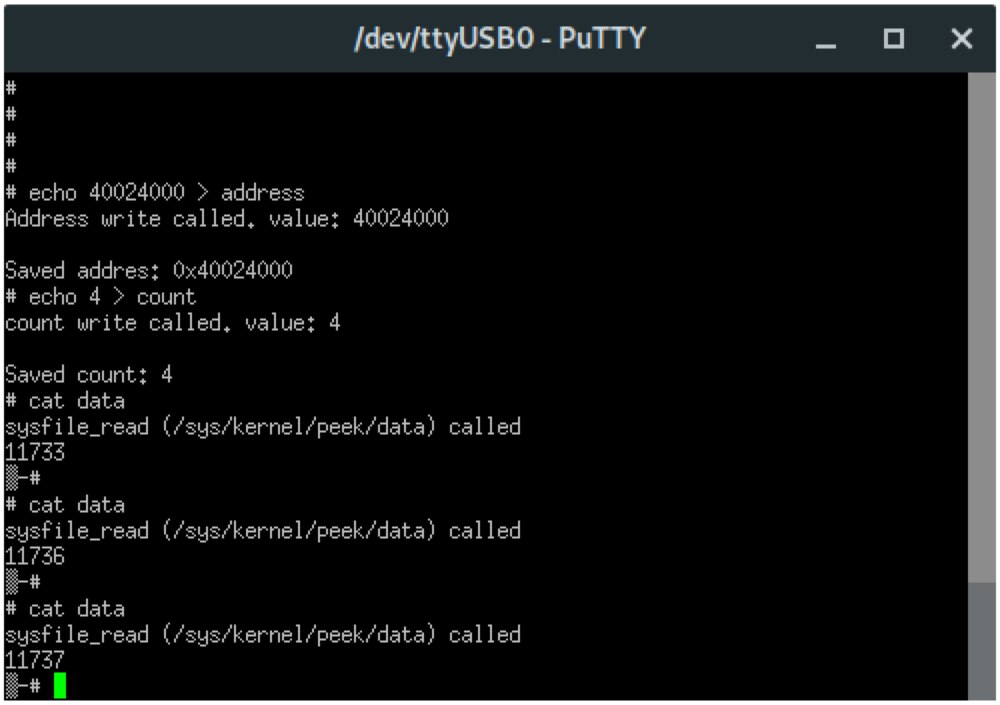


```
# ls /sys/kernel/peek/
address  count    data
```

As the names suggest, address can be used to read and write a register address.
Count can be used to set the number of bytes that should be read from the set address.
Finally data can be used to read the set register for x bytes, or write for the amount of bytes that are sent. We have chosen to implement separate files instead of a single file since this enables us to skip the use of string parsing, something which is frowned upon for /sys files. Furthermore this more straightforward, accessible and in line with common practice.

How tested our solution

The full cycle of using the driver to read the RTC up count is shown in the figure below.
Keep in mind that no string formatting is used, the raw characters are returned.
This is why we see the ‘strange’ characters in the terminal.
For clarity and debugging we use printk, to show the values as a number.

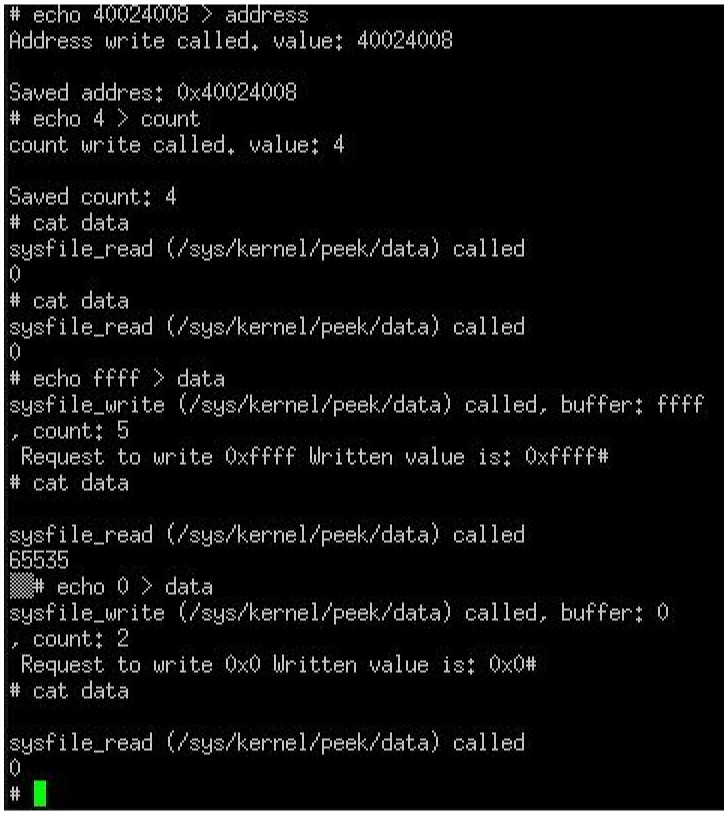

/dev/ttyUSB0 - PuTTY

```
#
#
#
#
# echo 40024000 > address
Address write called, value: 40024000

Saved addres: 0x40024000
# echo 4 > count
count write called, value: 4

Saved count: 4
# cat data
sysfile_read (/sys/kernel/peek/data) called
11733
▒-#
# cat data
sysfile_read (/sys/kernel/peek/data) called
11736
▒-#
# cat data
sysfile_read (/sys/kernel/peek/data) called
11737
▒-#
```

See next page

To verify write functionality we used RTC_MATCH0 at 0x40024008

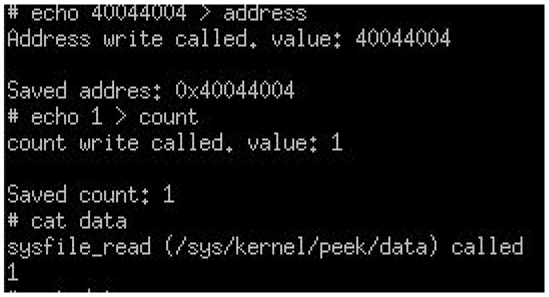
In the picture we can see that the register starts out empty. We then set it to0xffff, check the value, set it back to 0x0000 again and we verifiy that the register is empty again.



```
# echo 40024008 > address
Address write called, value: 40024008

Saved addres: 0x40024008
# echo 4 > count
count write called, value: 4

Saved count: 4
# cat data
sysfile_read (/sys/kernel/peek/data) called
0
# cat data
sysfile_read (/sys/kernel/peek/data) called
0
# echo ffff > data
sysfile_write (/sys/kernel/peek/data) called, buffer: ffff
, count: 5
 Request to write 0xffff Written value is: 0xffff#
# cat data

sysfile_read (/sys/kernel/peek/data) called
65535
▒# echo 0 > data
sysfile_write (/sys/kernel/peek/data) called, buffer: 0
, count: 2
 Request to write 0x0 Written value is: 0x0#
# cat data

sysfile_read (/sys/kernel/peek/data) called
0
# 
```

Finally to test our software we used the standard timers.

Specifically timer control register T0TCR at 0x40044004

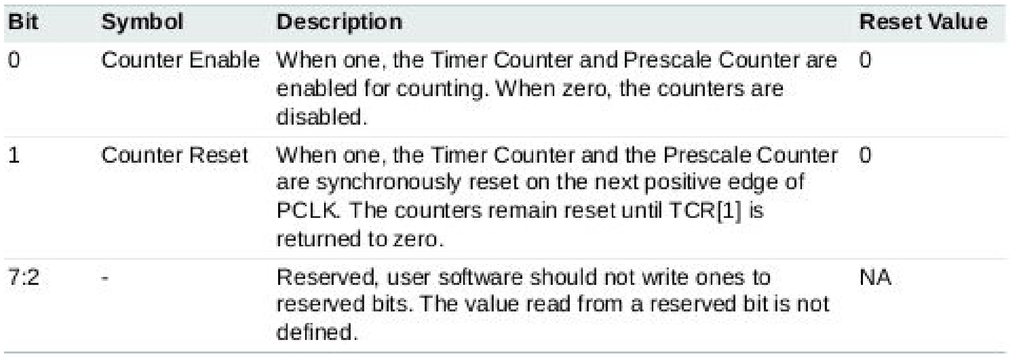


```
# echo 40044004 > address
Address write called, value: 40044004

Saved addres: 0x40044004
# echo 1 > count
count write called, value: 1

Saved count: 1
# cat data
sysfile_read (/sys/kernel/peek/data) called
1
```

When reading, we get the number 1. (printed as unsigned long)

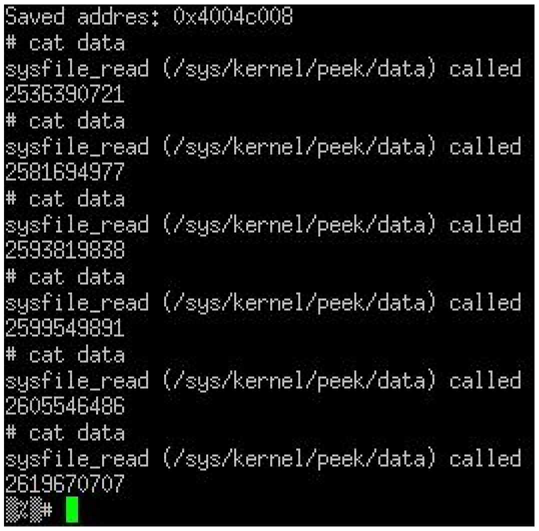
Converted to binary this is 0000 0001

According to the documentation this means the counter is enabled.



Bit	Symbol	Description	Reset Value
0	Counter Enable	When one, the Timer Counter and Prescale Counter are enabled for counting. When zero, the counters are disabled.	0
1	Counter Reset	When one, the Timer Counter and the Prescale Counter are synchronously reset on the next positive edge of PCLK. The counters remain reset until TCR[1] is returned to zero.	0
7:2	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

We further verified this by reading the timer counter register T0TC at 0x4004C008, as you can see it is incremented in between calls.



```
Saved addres: 0x4004c008
# cat data
sysfile_read (/sys/kernel/peek/data) called
2536390721
# cat data
sysfile_read (/sys/kernel/peek/data) called
2581694977
# cat data
sysfile_read (/sys/kernel/peek/data) called
2593819838
# cat data
sysfile_read (/sys/kernel/peek/data) called
2599549891
# cat data
sysfile_read (/sys/kernel/peek/data) called
2605546486
# cat data
sysfile_read (/sys/kernel/peek/data) called
2619670707
# 
```