|  |
| --- |
| Fontys Hogescholen |
| ES6: Peek And Poke |
| Research Document |
|  |
| **Group: Valeri / Thimo / Thanh** |
|  |

|  |
| --- |
|  |

Contents

[Part 1 2](#_Toc34766984)

[Find Up Counter Address 2](#_Toc34766985)

[Reading from computer 2](#_Toc34766986)

[Reading from LPC3250 3](#_Toc34766987)

[Part 2 6](#_Toc34766988)

[I/O module 7](#_Toc34766989)

[Sysfs module 7](#_Toc34766990)

[Main module 9](#_Toc34766994)

[Testing 9](#_Toc34766995)

# Part 1

To complete Part 1, there are 3 main activities: finding out the Up Counter address, reading it from the computer, reading it from the LPC3250.

### Find Up Counter Address

The Up Counter of the Real Time Clock is located at address 0x4002 4000 with 32-bit data (it is a 4-byte register). This information is known via the datasheet (see Figure 1).

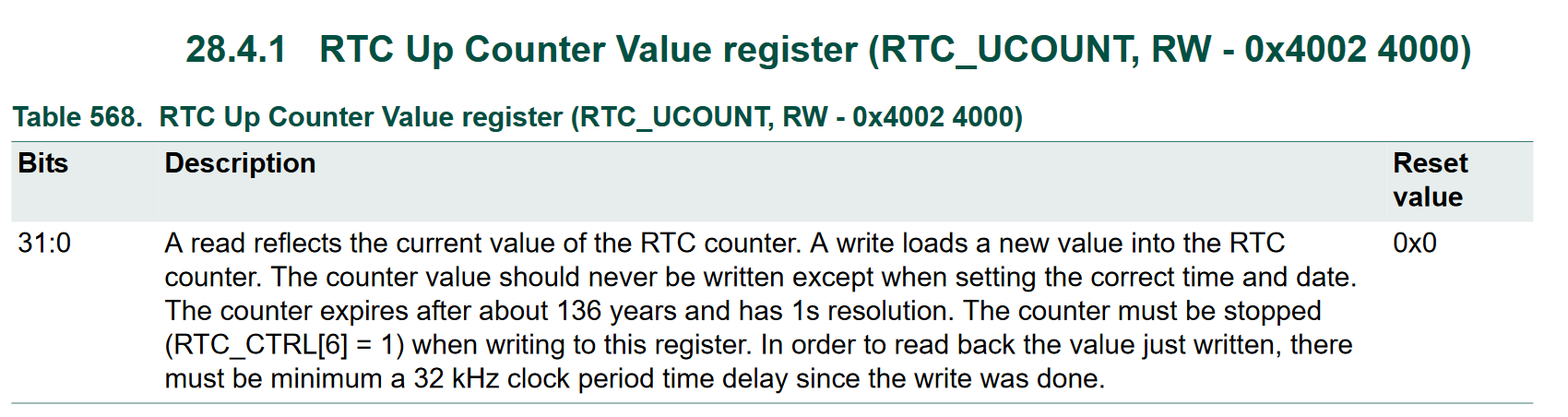


Figure Description of the Up Counter of the Real Time Clock from datasheet[[1]](#footnote-1)

### Reading from computer

The user space program is a simple program which declares a pointer to an integer (example below)

int\* addr=0x40024000;

The program later attempts to read, or deference, the pointer (with some printing for debugging purposes). The pointer here means that the program is expecting to read an integer back from address 0x4002 4000. On 32-bit machines, the size of integers is typically 4 bytes, or 32-bit, which matches with the data expected from the register (see section above).

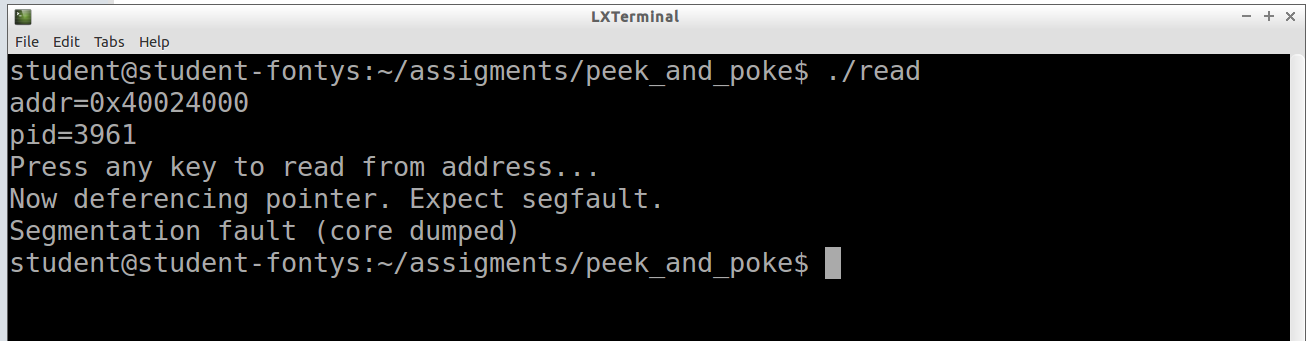
Running this on Linux VM results in a segmentation fault when trying to dereference the pointer (see Figure 2). 

Figure Reading address on Linux VM

As can be seen from Figure 2, the PID of the process which runs the “read” program is printed and the read occurs only after user input (“Press any key to read from address”). This is extra information later used to investigate the segfault.

A segmentation fault is an access violation to memory regions. To prove that the address 0x40024000 is indeed not accessible for the “read” program, the memory map of the program is shown using /proc/[pid]/maps[[2]](#footnote-2)

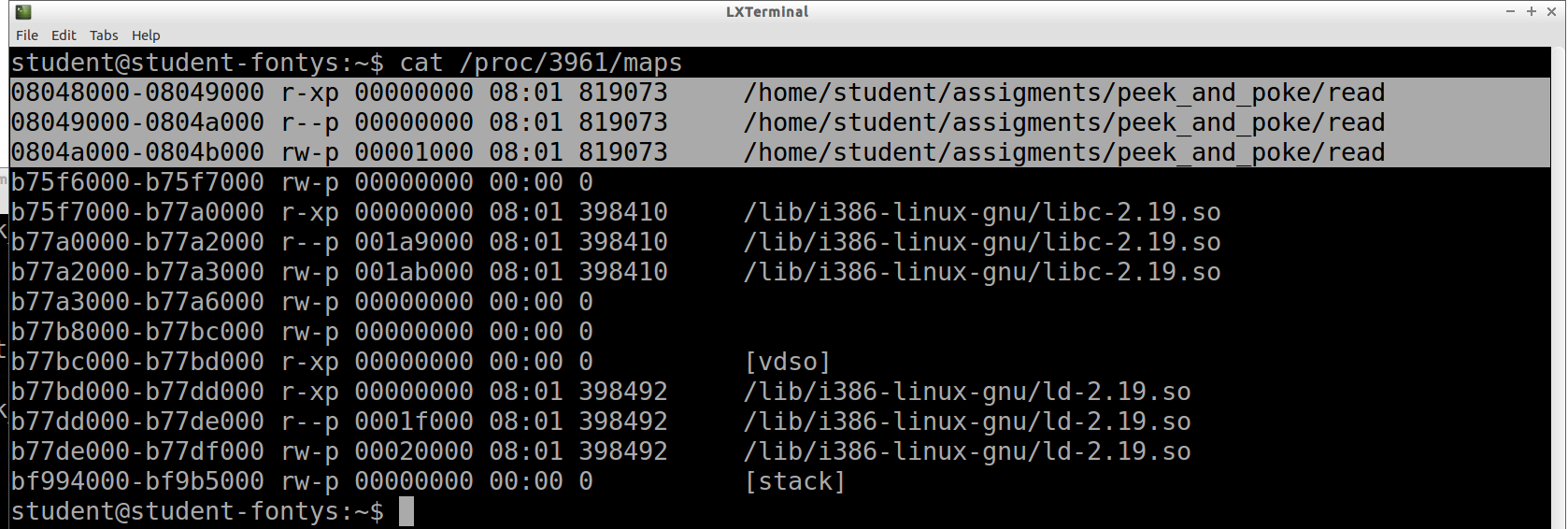


Figure Memory map of "read" program

The highlighted section in Figure 3 is the memory region allowed for the “read” program. Clearly 0x40024000 does not belong in this section and therefore explains the segmentation fault.

### Reading from LPC3250

The same program was run on the LPC3250. However, the read does not segfault. (see Figure 4)

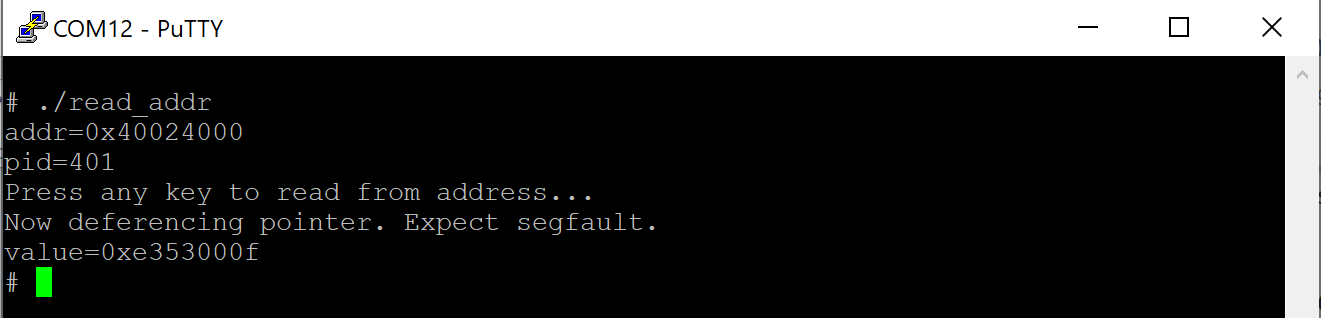


Figure Reading from address in LPC

The guess as to why LPC3250 did not report a segfault is: there is an MMU present and active on the LPC3250. The onboard MMU did report a translation fault when attempting to read from 0x40024000; the MMU then returns a data abort to the CPU core; the CPU handles this data abort by restoring the register to its old value (before read/write occurs) and no further handling is done.

The following sections will document the thought process and research done to reach the conclusion above.

##### Is an MMU active?

This section solves the question of whether an MMU is present and active. This is the first question to come to mind as an MMU should have reacted upon the invalid memory accesses.

**Investigation**

The first issue is whether an MMU is present; it is. This is known through the datasheet (see Figure 5[[3]](#footnote-3)):

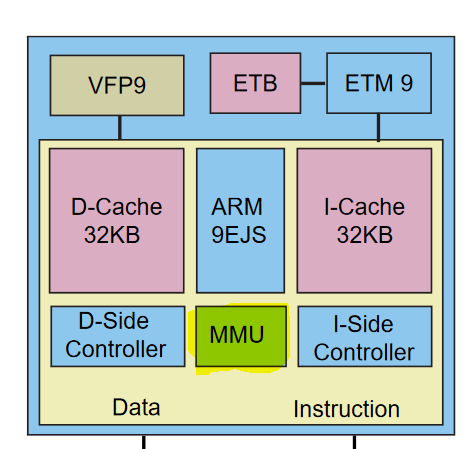


Figure Snippet of the Block Diagram of LPC3250

The second issue is whether the MMU is active. The datasheet lets us know that the processor is ARM9EJS. There is a [Technical Reference Manual](http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0198e/) provided by ARM for this generation of processor. (From here onwards references to “reference manual” or “technical reference” is talking about this manual.)

The technical reference mentions the enabling/disabling of MMU, which is done via writing to ARM control register. The technical reference also includes an assembly code on how to read the control register.

To know whether MMU is enabled or disabled, a kernel module is written to read the control register[[4]](#footnote-4). Figure 6 below shows the output from said kernel module.

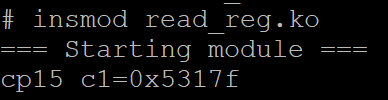


Figure Investigation into ARM9 registers

As shown in Figure 6, the value of the control register (CP15 c1) is 0x5317f. This translates to the bit mapping in Table 1 below. The first row shows the corresponding binary of 0x5317f (zero-padded to match 32-bit size); the second row shows the bit name. The relevant bits are highlighted and decoded in Table 2.

Table Bit mapping of control register

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Reserved | | | | | | | | | | | | | | | | L4 | RR | V | I | Reserved | | R | S | B | Reserved | | | | C | A | M |

Table Decoding bit value from control register

|  |  |  |  |
| --- | --- | --- | --- |
| Bit name | Description | Value | Effects |
| I bit | ICache enable/disable  0 = disabled; 1 = enabled | 1 | Instruction fetches are mapped from virtual addresses to physical addresses |
| C bit | DCache enable/disable  0 = disabled; 1 = enabled | 1 | Data accesses are mapped from virtual addresses to physical addresses |
| M bit | MMU enable/disable  0 = disabled; 1 = enabled | 1 | MMU enabled |

**Conclusion**

There is an MMU present and enabled. Data accesses and instruction fetches are remapped from Virtual Addresses to Physical Addresses.

##### What does the MMU do when given the Up Counter address?

The address given in the datasheet of the Up Counter is a physical address. When declared in a user space program or a kernel module, the address is still so far a virtual address (no mapping done in Part 1 of the assignment). The question arises is: how does the MMU behave upon receiving the Up Counter address?

**Investigation**

Further look into the reference manual reveals 2 more relevant registers of the ARM processor:

* Fault Status Register c5: contains the status of the last fault reported by the MMU
* Fault Address Register c6: contains the Modified Virtual Address of the last invalid data access[[5]](#footnote-5)

There is mention of a “Modified Virtual Address”. According to the manual, “addresses issued by the ARM9EJ-S core **in the range 0 to 32MB** are translated”[[6]](#footnote-6) to a modified virtual address. 0x40024000 is well above the 32MB range, so it is safe to say that the address will not be modified.

A kernel module[[7]](#footnote-7) is written to: read from the up counter address (without remapping) then read the control registers. The goal is to see whether or not the MMU registers a fault while trying to read the up counter address.

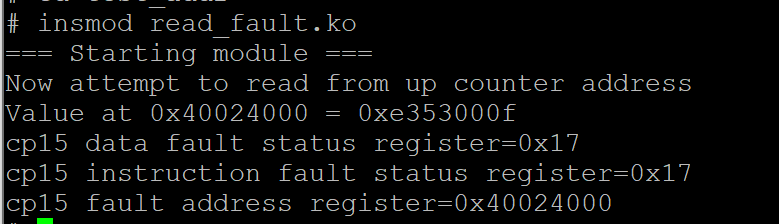


Figure Fault registers upon attempting an invalid read

As can be seen from Figure 7, the fault address register stores the invalid virtual address 0x40024000. Because the Fault Address Register is updated, an invalid data access happened – this is called a Data Abort. This Data Abort is returned to the CPU core.

The Data Fault Status Register (relevant here because it was a data abort) indicates that the fault occurred is a translation fault[[8]](#footnote-8). This is expected as the address given to the MMU is a virtual address not mapped to any physical address.

So a translation fault is reported by the MMU upon data access. It is now up to the CPU how to handle the abort. According to [this ARM article](https://developer.arm.com/docs/dui0471/l/handling-processor-exceptions/data-abort-handler) about data abort handler: if an abort happens to be a single load/store to a register (which in this case, it is), and “if the abort takes place on an ARM9™ or later processor, **the address is restored by the processor to the value it had before the instruction started. No code is required to undo the change.**”

Perhaps it is this handling behaviour – where the processor automatically restores the old value of the register upon a data abort – that let the up counter address be read without causing a segfault on LPC3250.

**Conclusion**

The MMU reports a translation fault and returns a data abort to the CPU core. The CPU core handles this abort by restoring the register to its old value (before the read/write operation) and no further error handling is done.

# Part 2

The assignment is to read and write to a hardware address using sysfs. Roughly, this breaks down to three C modules as listed in Table 3 below.

Table 3 Modules and responsibilities

|  |  |
| --- | --- |
| Name | Responsibility |
| Main | Implement initialize/cleanup module |
| Sysfs | Handle user input via sysfs |
| I/O | Handle direct access to memory |

The dependencies are: main depends on sysfs, sysfs on I/O, I/O has no dependencies. Because of this dependency, the modules are implemented bottom-up. The following sections document the implementation for each module.

## I/O module

The responsibility of this module is to perform read and write on a “hardware register”. Because a register is 4 bytes, the I/O module should handle read and write to 4 bytes of memory.

The user input will be a physical address. To do operations on this address, the I/O module must arrange a mapping from the given physical address to a virtual one.

From the book Linux Device Drivers[[9]](#footnote-9), it is known that:

* The function ioremap (and iounmap) can be used for mapping (and unmapping) from physical to a virtual address
* The address returned by ioremap should not be referenced directly, but rather goes through a read/write function offered by the kernel.

A simple kernel module was written to test this direct memory access (i.e map the hard-coded up counter address to a virtual one, read and output to the screen). After this test was OK (the counter does count up after every read), the next module is written.

## Sysfs module

The responsibility of this module is to handle user input via sysfs; this includes configuring the sysfs itself, correctly parsing input and according to the input, call I/O module.

### User input

From the assignment, the only user interaction is writing to a sysfs file and at which point, a read and write to memory occurs. Therefore, a sysfs file is made with no show handler and only a store handler.

A struct is declared to store the user input. The struct has three members:

* Operation: a character to indicate read or write
* Address: an unsigned integer to save the physical address
* Data: the user argument stored as a string (because a read operation interprets this as a number of register, whereas write interprets it as a hexadecimal value, the parsing is left up to the operation-specific function).

### Sysfs store handler

Most of the logic happens inside the store handler (i.e. when the user writes a command to sysfs). Figure 8 below shows the flowchart for the handler.

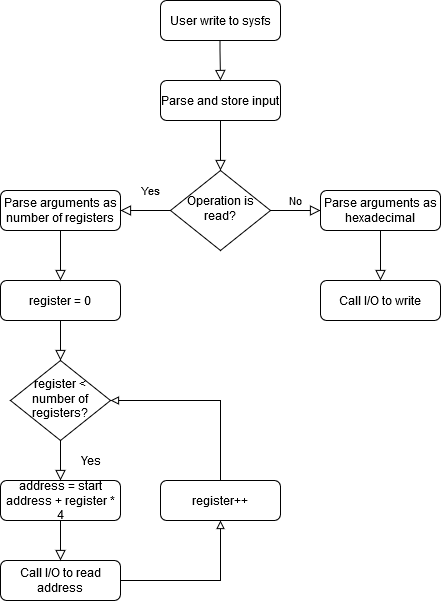


Figure 8 Flowchart for sysfs store handler

The flowchart is quite straightforward. The only thing of note is the read operation where user wants to read multiple registers; this operation requires some pre-processing of the physical address before sending it to the I/O module. As can be seen in Figure 8, the loop starts with register set to 0 and loop until register is equal to the number of registers the user sent. The register variable is the register count; the start address is the address the user sent; because one register is four bytes, the address offset is (register \* 4) bytes. The address the user sent (start address) added by the offset is the correct address to send to the I/O module.

### Testing

A simple kernel module was written to test this module. This module initializes the sysfs and print out debug information to verify that the input was correctly parsed and the correct functions were called to do read/write. After this test was OK, the main module was written.

## Main module

The main module simply calls the sysfs module during its initialize and cleanup functions.

## Testing

The final kernel module is tested as follows. The requirement is to read and write to any hardware any register.

From the datasheet, there is a RTC Control Register where user can do a reset of the counters (see Figure 9).

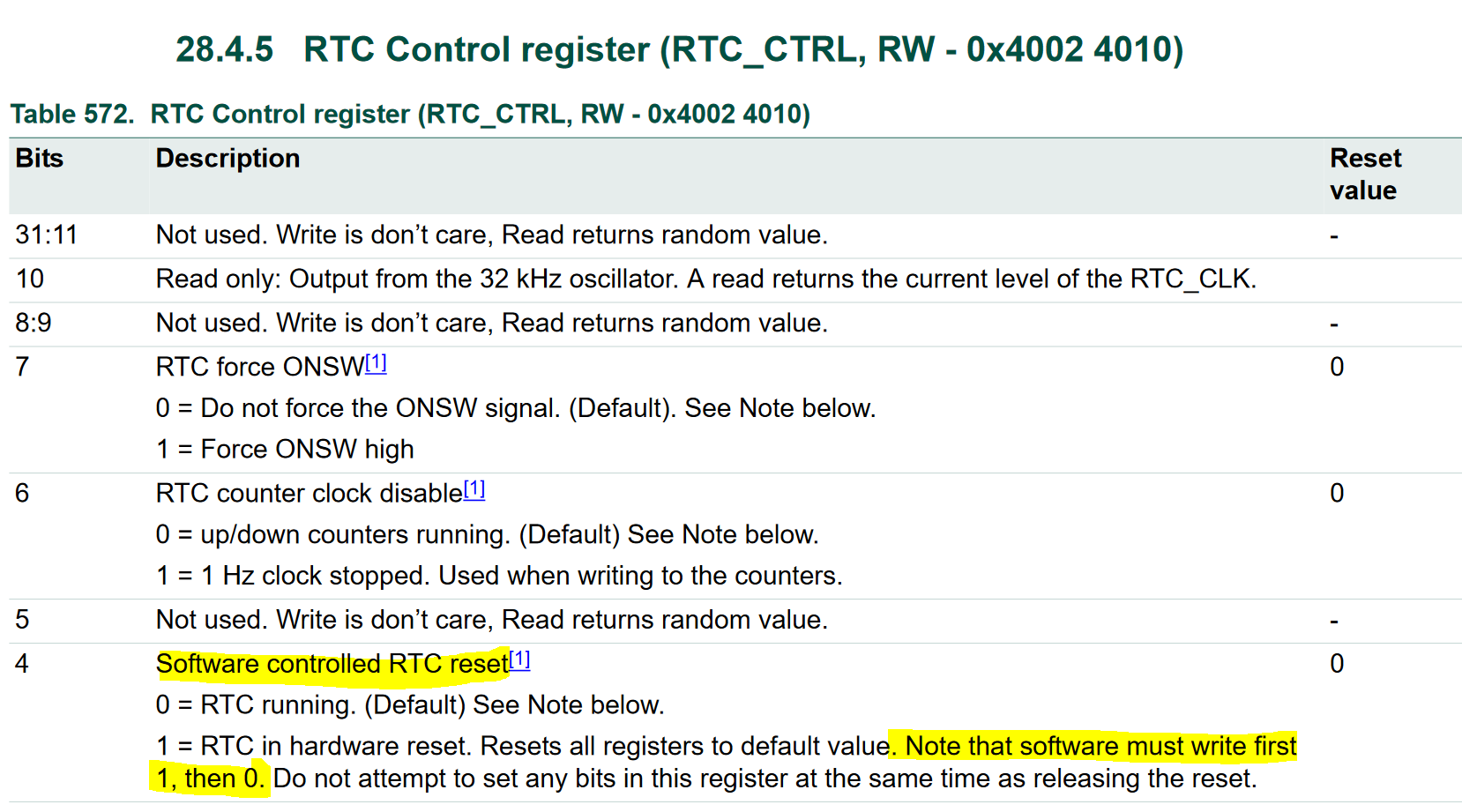


Figure 9 RTC Control Register from the Datasheet

Bit 4 of the RTC Control Register is suitable for the testing purposes. A test is then to:

* Read from the Up Counter
* Write 1 and then 0 to Bit 4 of RTC Control Register (to perform a software-controlled reset)
* Read from the Up Counter again

At this point, if the module works, the Up Counter should be reset. This test was OK (see Figure 10)

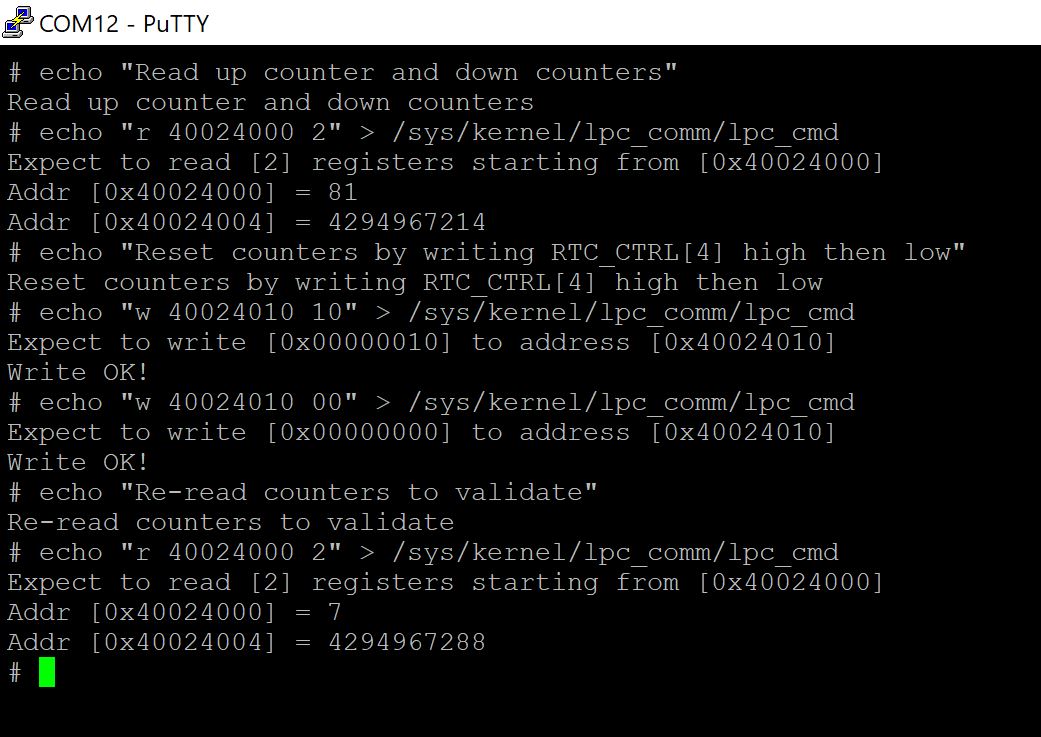


Figure 10 Reset counter via kernel module

1. Taken from p.570 of the datasheet (DataSheet-UM10326.pdf) [↑](#footnote-ref-1)
2. /proc/[pid]/maps is “a file containing the currently mapped memory regions and their access permissions”. See: <http://man7.org/linux/man-pages/man5/proc.5.html> [↑](#footnote-ref-2)
3. Taken from p.29 of the datasheet (DataSheet-UM10326.pdf) [↑](#footnote-ref-3)
4. The source code was submitted as well (kernel\_read\_cp15.c). This was written with the help of this article: <https://stackoverflow.com/questions/49051641/how-read-coprocessor-registers-in-arm-architecture> [↑](#footnote-ref-4)
5. Properly phrased, this register updates only when a Data Abort occurred (and not a Prefetch Abort). A Prefetch Abort is an abort when the processor tries to execute an instruction fetched from an illegal address; a Data Abort is an abort when the processor tries an invalid data access. [↑](#footnote-ref-5)
6. See FCSE PID Register section [↑](#footnote-ref-6)
7. The source code was submitted as well (kernel\_read\_mmu\_fault.c) [↑](#footnote-ref-7)
8. The value read from the register (0x17) was converted to binary and the bits were decoded by reading the Reference Manual. The process was not documented here, for brevity. [↑](#footnote-ref-8)
9. Section “Software-mapped I/O Memory”, from <https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch08s04.html> [↑](#footnote-ref-9)