# 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 1 below.

Table 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[[1]](#footnote-1), 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 1 below shows the flowchart for the handler.

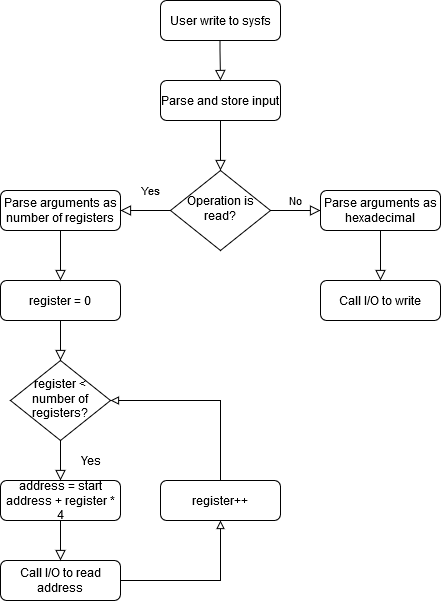


Figure Flowchart for sysfs store handler

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 1, 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 2).

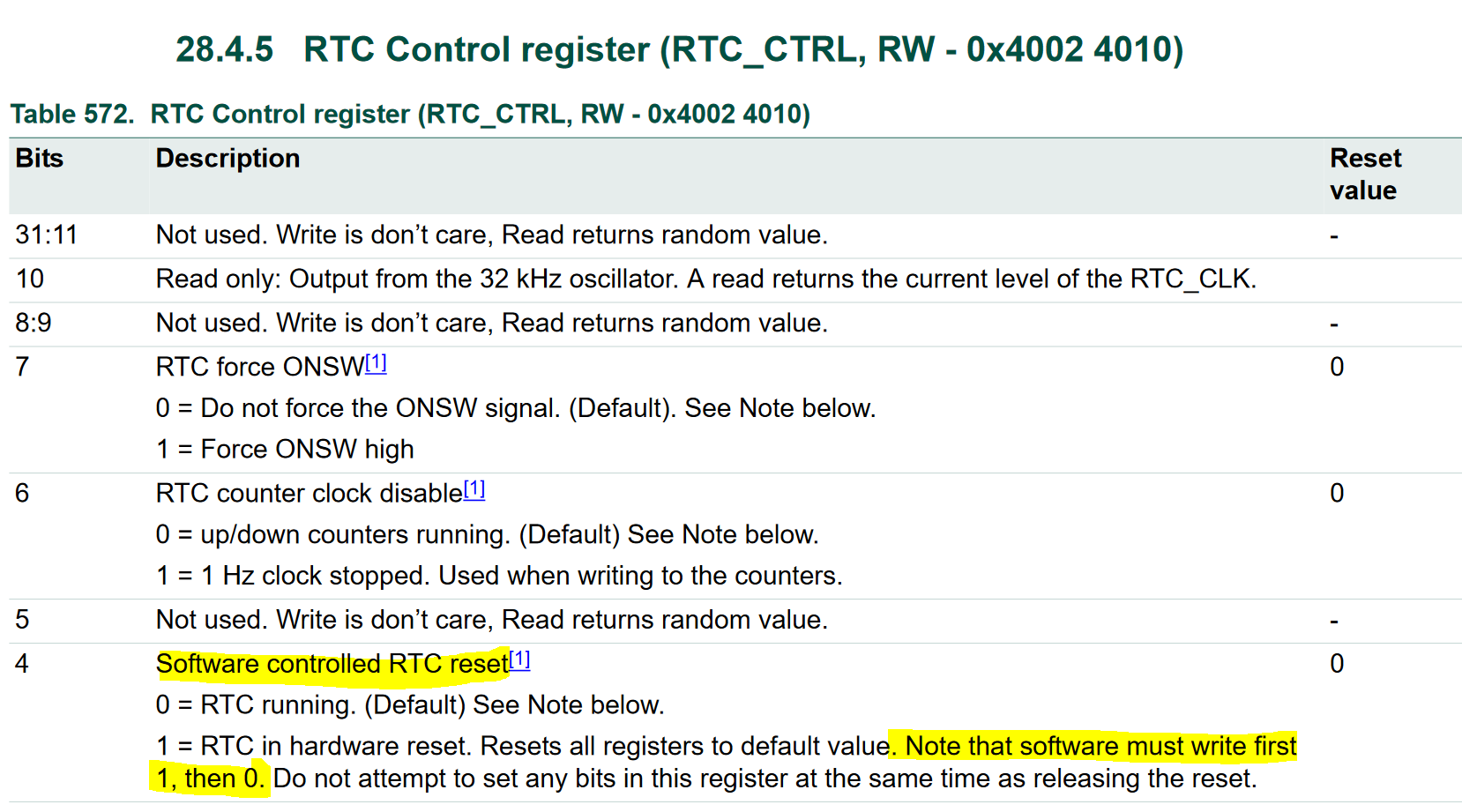


Figure RTC Control Register from the Datasheet

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 3)

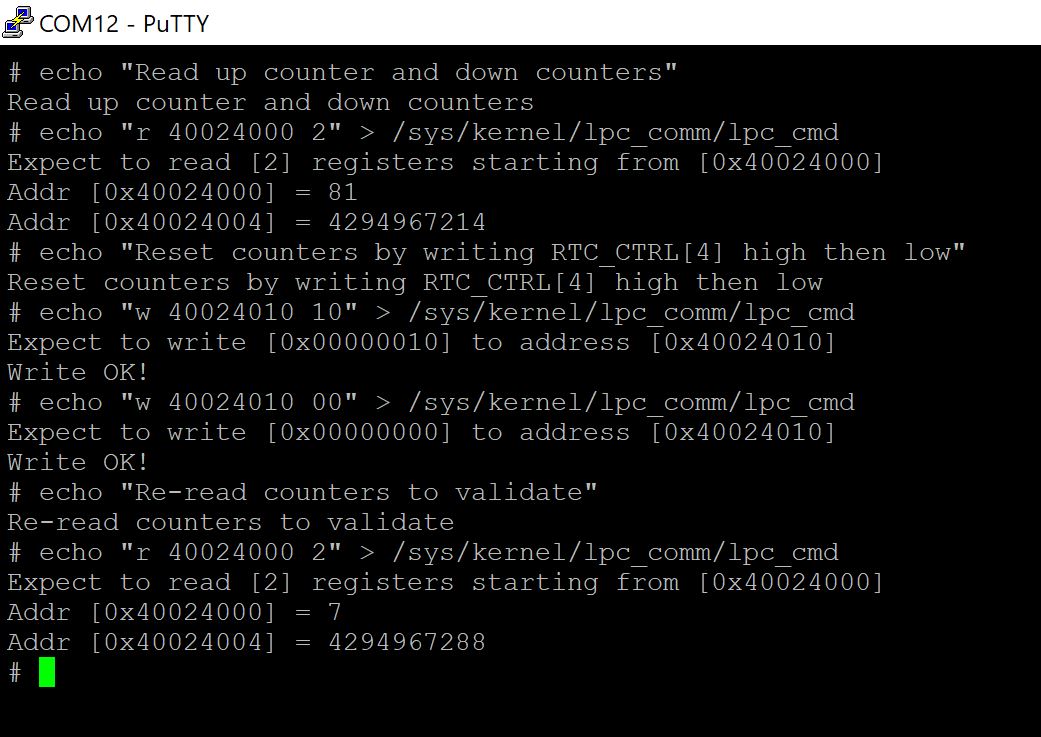


Figure Reset counter via kernel module

1. Section “Software-mapped I/O Memory”, from <https://www.oreilly.com/library/view/linux-device-drivers/0596000081/ch08s04.html> [↑](#footnote-ref-1)