

There have been several questions regarding lab 5, and I have tried to write this document to help explain any ambiguities that may exist in the lab. If there are any additional questions that I have not covered, please do not hesitate to let me know.

Nathan

Prelab Question #1

First of all, with respect to prelab question one, you will have to deal with two separate interrupt controllers in this lab. One is the main interrupt controller that is attached to the external interrupt pin of the core (the System Interface Unit (SIU) interrupt controller) and the other is the Communication Processor Interrupt Controller (CPIC)¹.

Prelab Question #2

This question is asking you to write code that will help you determine which interrupt was triggered. It should be closely related to the block diagram you draw in question #1.

Prelab Question #3

As a note, the fact that your ISR is counting is somewhat redundant. The RTC has a count register and it can be read at any time. You may even find it helpful to make sure that your value is equal to the value in the RTC register².

You will be using all of the functions for serial communication that you had in lab4, but they have been modified a bit. The `ser_getchar_nb` and `ser_putchar_nb` functions no longer take care of the CISR for you. Also, the `ser_init` function no longer takes care of initializing the CPIC for you.

Most of your handler code for your serial handler should be the ISR code that you wrote in lab 4. In addition to dealing with the interrupts, you will have to make modifications to implement the three new commands that you will be required to implement.

Prelab Question #4

Your initialization routines should all be separate functions that are called in the main section of your code. You will be given your normal initialization routine called `ser_init`. This function initializes the FADS motherboard and the serial communications respectively. You will have to write functions for initializing the real-time counter, the timers, and the interrupt controllers. I recommend that you write individual functions for each of the initialization routines that you must write. It will come in handy when you are debugging your programs.

An important note is that you must call `ser_init` first when you start to initialize everything.

You do not have to deal with any of the registers that are part of the SMC.³

Read Section 16.4.3, it has a good explanation of initializing the timers. The ordering of steps 6 and 7 is not necessarily important.

The order for setting up the CPIC should be:

- 1) Set up the CICR,
- 2) Clear all In-Service and pending bits in CIPR and CISR (Remember that to clear bits, you have to write a 1 to them)
- 3) Set up the masks in the CIMR

¹ The CPIC is discussed in section 16.15 of the manual, and there is helpful information in that section for doing this question. The CPIC is part of the Communication Processor (CPM).

² The description of the RTC can be found in section 12.7 of the manual. The description of the timers is in section 16.4 of the manual.

³ The SMC or Serial Management Controller is basically the on-chip UART, though it is far more complex than the simulator UART. You are using SMC1, so when you deal with registers in the CPIC, deal with the ones related to SMC1.

Prelab Question #5

Hint: read the section in the lab report that discusses the IMMR

Prelab Question #6

Alright, this question had wording problems. I am going to rewrite it and separate the parts. Hopefully this helps out. After experimentation, it turns out that 1kHz is a better blink rate, so you should adjust your calculations to achieve that blink rate.

- a) The input to the timers is the system clock, which is currently set at 24MHz. Since you are asked to achieve a duty cycle at 1% accuracy of the 1kHz signal (1ms period), you will need to have 10 μ s accuracy (1% of 1ms). What are valid values for the PS and the ICLK fields in the TMRx reg? You may use any values for these two fields that will allow you to achieve this resolution⁴⁵.
- b) To make your block diagram simpler, I'll basically describe what it looks like and you can draw it. The input to the timer stage is the 24MHz system clock, then it is conditionally divided by 16 depending on the value in TMRx[ICLK]. It is then further divided by the value in the prescaler. Then it finally goes into the counter itself and counts up to the reference value found in TRRx. When TCNx is equal to TRRx, an interrupt occurs.
- c) Given the values that you chose for your ICLK and PS fields of your timer, what values will you put in the TRRx register to achieve your 0-100% duty cycle range?

Other Notes:

Turning on and Turning off Timer interrupts

In addition to turning off interrupts when you enter the command LED 0, you should turn off interrupts when you enter the command LED 100. (I.e. just turn the led on and stop the timer.)

Priorities

There have been questions regarding prioritizing your interrupts. You can pick priorities for the CPM and the Real-time counter. I recommend four and zero respectively. Note that the bit fields for the respective units require different values to indicate the interrupt level. The RTCIRQ field uses a one-hot mechanism to determine the interrupt level for the RTC⁶. For the CPIC, the IRL field of the CICR is just a simple binary number that indicates the interrupt level.

As you should have noticed, the CPIC is a separate interrupt controller. The priorities of this interrupt controller are more or less fixed, so you do not have to worry about the priority of the serial relative to the timer⁷.

CIVR/IACK

There seems to be some confusion related to determining the interrupt vector that was triggered when you receive an interrupt in the CPIC. To determine the vector, you need to set the IACK bit by writing to the CIVR (doing this will cause the CPIC to set the vector field in the CIVR) and then read the vector out of the CIVR. See table 16-46 to determine what each vector means.

When you set the IACK bit in the CIVR, it will automatically clear the CIPR (pending) bit and set the CISR (in-service) bit corresponding to the interrupt it is reporting. Setting the CISR bit automatically prevents the CPIC from reporting equal- or lower-priority interrupts to the SIU (read section 16.15.2.3), nicely supporting nested interrupts among the CPM interrupt sources. You must clear the CISR bit before

⁴ Remember that the TRR register has 16 bits of resolution.

⁵ Hint: I recommend that you determine PS and ICLK values that will not require you to multiply to determine the value that you must put in the TRR reg.

⁶ One hot means that each bit corresponds to an interrupt level, and only one can be set at a time. If you want to use Interrupt Request Level 0, you would set bit 0 (set the field to 10000000). If you wanted to use request level 1, you would set the field to 01000000, etc.

⁷ Incidentally, you can look at table 16-46 to see the interrupt priorities. 1F is highest and 0 is lowest. You can note that the Timer has higher priority than the serial controller does.

you return from your interrupt handler or these interrupts will be permanently suppressed. Read section 16.15.5.4 carefully to determine how to clear CISR bits.

The SIU interrupt controller is very different; it has no in-service register and no IACK bit, and it does not automatically mask equal- and lower-priority interrupts. If you want to nest interrupts at the SIU level, you can do these things in software, but for this lab you don't need to (see below).

Linking

Link with adsinit.o located at eecs373\lab5\adsinit.o. The functions have changed slightly since lab 4, so your program will not work if you link with the lab 4 version of adsinit.o.

Nesting Interrupts

To make your lab simpler, you will only be required to allow nested interrupts within your serial handling ISR. The timer and RTC interrupt handlers do not need to re-enable interrupts. One thing that may help you get your nested interrupts correct is to remember that you must either mask the interrupt you are servicing or clear the interrupt condition (i.e. make the device stop asserting its interrupt) before you re-enable interrupts. If you do not do one of these, the device you are servicing will immediately trigger another interrupt because the machine will think that the interrupt has not been handled. In general, you should also mask any lower-priority interrupts (if the controller doesn't do this for you: the CPIC does (via the CISR) but the SIU controller doesn't). Since there are no lower-priority interrupts than the serial handler you don't need to worry about that for this lab.

Note that the way to clear the interrupt condition varies among different devices.