

Assignment 4

CIS*2030: Structure and Application of Microcomputers

Due: **11:59pm, November 28, 2024**

Neatly print your name, student identification number, and lab information below:

Name:

ID:

Section:

The goals of this final assignment are to solidify your understanding of interrupts, as covered in class, and to provide hands-on experience with interrupts in the Easy68K environment. The questions in this assignment are designed similarly to those in the lab exercises.

Instructions

- (1) You must work alone and not discuss solutions with others (artificial or human). All submitted solutions and code **must be your original work**, and they will be **subject to verification for authenticity and originality**.
- (2) Review your answers 1 or 2 days after completing the assignment. Taking a few minutes to do this can dramatically improve your understanding of key concepts and increase your retention. Also, it makes it much easier for you to catch any errors.

Submission

- (1) Print the PDF and fill it in by hand or fill in an electronic copy with a PDF editor (Acrobat Reader, GoodNotes, etc.). Submissions that do not include this document or in which the coversheet has not been filled in **will receive a grade of zero**.
- (2) If you have completed work by hand, scan the individual pages (including the cover sheet) and **convert them into a single PDF**. Do not upload individual pages. If you do not have a scanner, take pictures of each page and use a program like Adobe Acrobat Scan to create a single PDF. Ensure the pictures are clear and well-lit.
- (3) Document all programs according to the *Coding Specifications* section of the document "2030 Lab Specifications" available on CourseLink.
- (4) Submit the source (.X68), executable (.S68) and listing (.L68) files for all programming problems.

Upload all files (PDF, source, executable, and listing) as a single submission to the Dropbox labeled "Assignment 4" on CourseLink. Only your final submission will be graded.

The course is officially over when this Dropbox closes. Therefore, **late assignments will not be accepted for any reason**. Therefore, it is your responsibility to start early and submit your assignment before the deadline.

Question 1

As discussed in class, considerable amounts of time may be wasted during the polling process, especially if multiple (possibly inactive) devices must be polled. Interrupt-driven I/O seeks to overcome this limitation. With interrupt-driven I/O, the processor continues to run normal application tasks. However, when an external device wishes to communicate and exchange data with the processor, the external device generates an interrupt request signal in hardware. If this signal is acknowledged by the processor, an appropriate interrupt-service routine is run automatically by the processor's exception-processing mechanism to transfer the data.

The Easy68K supports seven auto-vectorized interrupts – one for each interrupt priority level. Table 1 shows the vector numbers (25-31) and the vector addresses (0x064 – 0x7C) for each auto-vector. The address of the interrupt-service routine (ISR) for an auto-vector interrupts must be placed into the vector table in order to be executed when an interrupt occurs.

Table 1: Location of auto-vector interrupts in vector table.

Vector Number (decimal)	Vector Address (hexadecimal)	Interrupt Level
25	064	1
26	068	2
27	06C	3
28	070	4
29	074	5
30	078	6
31	07C	7

Step 1

Download the sample program called **interrupt.X68** from the course website.

Step 2

Assemble the program, then read through it carefully. Notice that main function consists of a loop that continuously displays the message 'Executing in main() function' in the terminal window.

```
*
* Main () function continuously displays text message on screen

                LEA      MSG,A1          ;point to message
                MOVE.B   #13,D0          ;task 13 - display string on console
MAIN            TRAP     #15
                BRA      MAIN

MSG             DC.B     'Executing in main() function',0
```

The program also defines a level-2 auto-vector interrupt service routine that causes the message `Executing level-2 interrupt service routine` to be displayed on the terminal whenever a level-2 interrupt is generated.

```
ISR2      ORI.W    #$0700,SR           ;set priority level to 7
          MOVEM.L  A1/D0,-(A7)         ;save working registers
          ANDI.W   #$FAFF,SR          ;set priority level to 2
          LEA      MSG2,A1            ;point to message
          MOVE.B   #13,D0              ;display message on console
          TRAP     #15                 ;system call
          MOVEM.L  (A7)+,A1/D0         ;restore working registers
          RTE
```

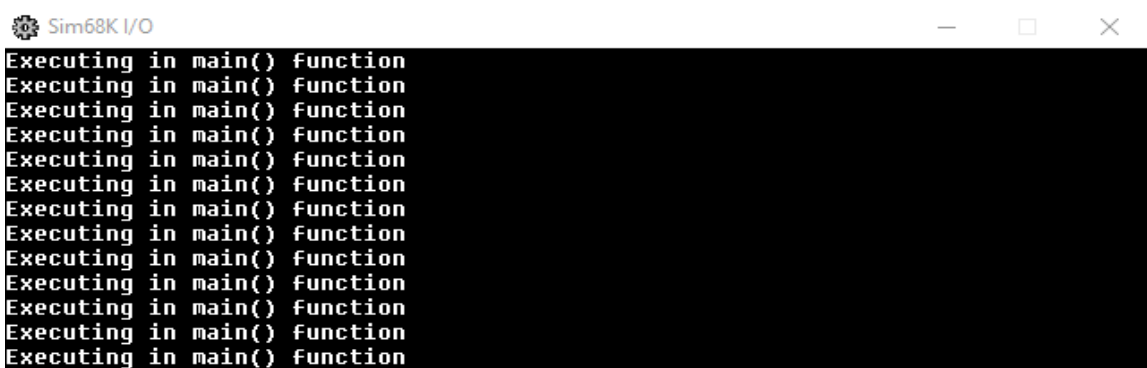
Notice that the ISR begins by setting the priority mask to 7, thus (temporarily) disabling all lower-level interrupts. With all lower interrupts disabled, the ISR saves its working registers on the system stack, thus making itself transparent to the (interrupted) main code. Once saved, the ISR returns the interrupt mask to level 2. Notice that at the end of the ISR, a RTE instruction is used to return from the ISR back to the main function.

In order for the ISR to execute when a level-2 interrupt is generated, it is necessary that the address of the ISR be stored in the proper location in the vector table. This is accomplished using the following instruction:

```
*
* Place address of the level-2 ISR into the vector table
          MOVE.L   #ISR2,$68
```

Step 3

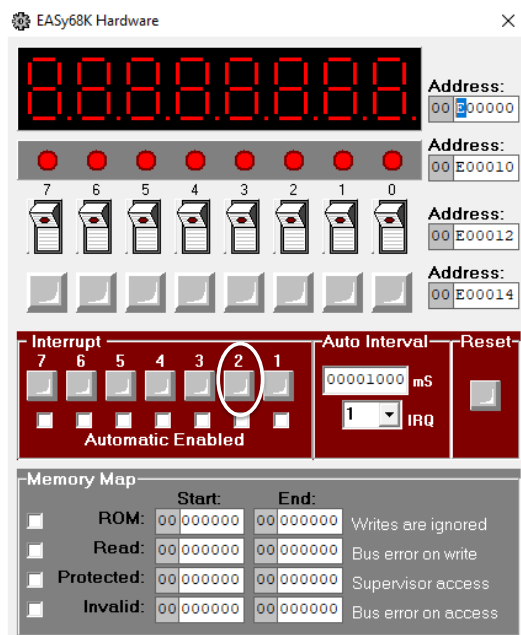
Run the program on the Easy68K simulator. Initially, you will see the following output being generated on the terminal:

A screenshot of a terminal window titled "Sim68K I/O". The window has a black background with white text. The text consists of a vertical list of 12 identical lines: "Executing in main() function". The lines are stacked vertically, filling most of the window's content area. The window's title bar shows standard macOS window controls (red, yellow, green buttons) and the text "Sim68K I/O".

The output shows that the program is executing the infinite loop in the main function.

Step 4

Now, trigger a level-2 interrupt. To do this, proceed to the Hardware Window and use the mouse to press the Interrupt button labeled 2 (circled in the figure on the next page).



There is no need to hold the button down. A level-2 interrupt will be generated as soon as the button is pressed. Press the button and observe the output that appears on the I/O terminal. Each time the button is pressed, a level-2 interrupt is generated. This causes the main function to be “interrupted” and temporarily suspended, while the level-2 ISR services the interrupt. Execution of the level-2 ISR causes the message `Executing level-2 interrupt service routine` to be displayed on the terminal window. Finally, the RTE instruction causes control to be returned to the suspended main function, and the main function begins to run again just as if the interrupt had not occurred.

Note: Depending on the speed of your computer, the message printed by the level-2 ISR may pass by so quickly that it is almost impossible to read. Therefore, a simple delay loop is added to the ISR, where a NOP instruction is repeatedly executed a large number of times, to slow things down.

Step 5

Stop the program from running. Now, set a breakpoint at line 23 – the first instruction (`LEA MSG, A1`) in the main function. Next, set a breakpoint at line 34 – the first instruction (`ORI.W #0700, SR`) in the level-2 interrupt service routine.

Now run the program until it breaks at line 23. Examine the interrupt mask in the status register (i.e., bits 10, 9, and 8). What is the value of the interrupt mask (in binary)?

Interrupt mask: _____

What does the previous setting mean with regards to which external interrupts the processor will respond to?

Continue the execution of the program. Now, trigger a level-2 interrupt by using the mouse to press the Interrupt button labeled 2 in the hardware window. Once the breakpoint on line 34 is reached, examine the interrupt mask in the status register. What is the value of the interrupt mask (in binary)?

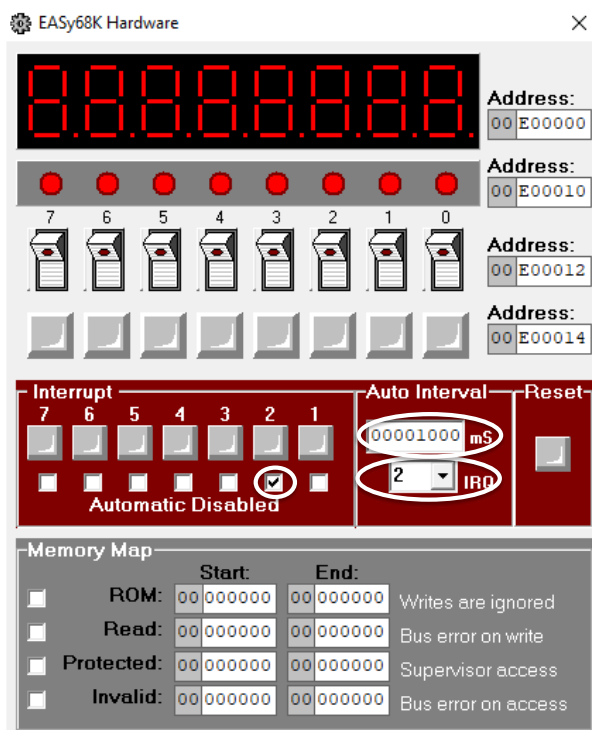
Interrupt mask: _____

There are a couple of important things to note:

- First, the level-2 interrupt is acknowledged by the processor because the interrupt mask is set to zero while the main function is executing. As discussed in class, an interrupt mask of zero means that all interrupt levels (1-7) will be acknowledged, thus all interrupts can be used to get the attention of the processor.
- Second, as part of the processor's built-in exception-processing mechanism, once the processor acknowledges an interrupt, the level of that interrupt is automatically written to the interrupt mask. This is done to ensure that the processor will only respond to higher priority interrupts. Interrupts at the same or lower levels will not be acknowledged until the current level interrupt is serviced, and the interrupt mask returned to its previous (lower) value.

Step 6

In step 4 above, you saw how the interrupt push buttons can be used to manually simulate an interrupt request at levels 1-7. The timer on the Easy68K can also be used to automatically create interrupts. As shown below, this can be done by selecting the desired interrupt in the "Auto Interval" drop-down menu, and then entering the interval in milliseconds. Selecting the checkbox under the interrupt push button enables auto interrupts for the corresponding interrupt.



To illustrate the previous features, start the previous program running, and then configure the hardware window to generate level-2 interrupt requests every 1000 milliseconds. (See the circles in the figure above.) Observe the output on the I/O terminal. Every second, a level-2 interrupt is automatically generated by the timer, causing the level-2 ISR to be executed.

Question 2

To better understand interrupt-driven I/O you will write a simple program involving all 7 interrupt levels. Your task is to write 7 interrupt service routines, one for each of the seven interrupt levels. Each interrupt service routine should be written to conform to the following requirements:

- Each interrupt service routine should be labeled as ISR_n , where n is the level of the interrupt; that is, $n = 1, 2, 3, 4, 5, 6$, or 7 .
- Upon entry, each service routine should indicate that it has started executing by outputting the message "IRQn – Starts", where n is the interrupt level.
- The previous message should be preceded by $n-1$ tabs, so that each message appears in its own column. (Note: the ASCII value of tab is 9.)
- After displaying the previous message, the service routine should execute a simple delay loop. The loop should perform a total of $0x1000000$ iterations, and on each iteration a NOP instruction should be executed.
- After the delay loop exits, the service routine should output the message "IRQn – Ends", where n is the interrupt level.
- The previous message should be preceded by $n-1$ tabs, so that each message appears in its own column.

The figure below shows example output for the program, when the user presses the push buttons to generate interrupts in the following order: 1, 2, 3, 4, 5, 6 and 7.

```

Sim68K I/O
IRQ1 - Starts
  IRQ2 - Starts
    IRQ3 - Starts
      IRQ4 - Starts
        IRQ5 - Starts
          IRQ6 - Starts
            IRQ7 - Starts
              IRQ7 - Ends
                IRQ6 - Ends
                  IRQ5 - Ends
                    IRQ4 - Ends
                      IRQ3 - Ends
                        IRQ2 - Ends
                          IRQ1 - Ends

```

This example shows how a higher-priority interrupt is always able to interrupt a lower-priority input. The figure below shows example output for the when the user presses the push buttons to generate interrupts in the following order: 7, 6, 5, 4, 3, 2, 1, and 0.

```

Sim68K I/O
              IRQ7 - Starts
              IRQ7 - Ends
            IRQ6 - Starts
            IRQ6 - Ends
          IRQ5 - Starts
          IRQ5 - Ends
        IRQ4 - Starts
        IRQ4 - Ends
      IRQ3 - Starts
      IRQ3 - Ends
    IRQ2 - Starts
    IRQ2 - Ends
  IRQ1 - Starts
  IRQ1 - Ends

```

This example shows how a lower-priority interrupt cannot interrupt a higher-priority interrupt.

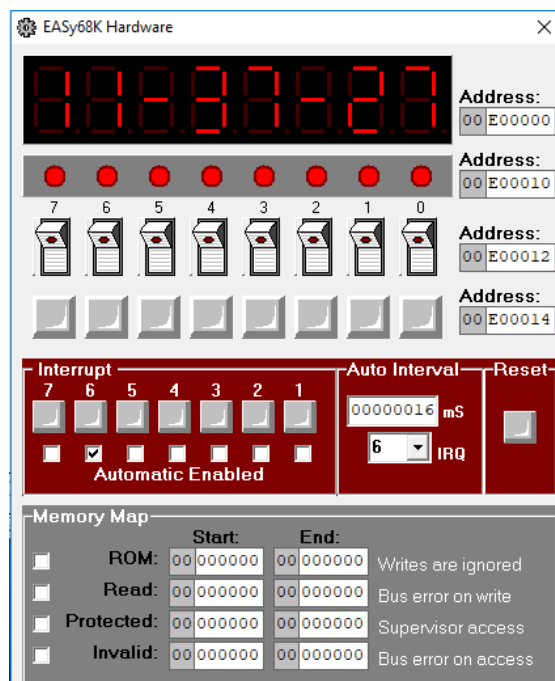
Save your program in a file called **levels.X68**.

Question 3

Implement an auto-vector interrupt-driven 24-hour digital clock. In completing this problem you will learn:

- To use the bank of eight 7-segment displays available in the hardware window,
- To use global memory locations to share information between a main function, polling loop, and interrupt-driven service routine,
- To use a timer to generate interrupts at a fixed rate, and,
- To implement multiple interrupt-service routines for performing different tasks in the same application.

The figure below shows the hardware window with the 24-hour clock running. From left-to-right, the bank of eight 7-segment displays shows time in the following format: *HH-MM-SS*. In this case, the clock indicates 11 hours, 37 minutes, and 27 seconds. Within the code that implements the 24-hour clock, three global memory locations are used for the hours (HH), minutes (MM) and seconds (SS). These values are accessible to all parts of the program, including the main function, the interrupt-service routine, and the polling routine used to implement the clock.



When implementing the 24-hour clock first design and test the modules below. Once each module is functioning correctly, they can be integrated into a single program.

1. Write a level-6 interrupt service routine, called ISR6. (The interrupt interface available through the hardware window, will be used to continuously generate an auto-vectored level 6 interrupt at 16 milliseconds intervals; that is, at 1/60 second intervals.) At each interrupt, the level 6 interrupt-service routine should decrement a counter (that is initially set to 60) and check to see if the counter is zero. If the value of the counter is not zero, the counter should be decremented by 1, and exit waiting for the next interrupt to arrive. If the counter is zero, this means that a total of 60, 1/60 second time intervals have elapsed. Therefore, the service routine should reset the counter to 60, and then update, in order, the second, minute, and hour values. When the hours reach 24, or the minutes and seconds reach 60, the values should be reset to zero.

Remember that interrupt-service routines are not subroutines. You will need to make sure that code is included in your program to place the address of the level 6 service routine into the vector address for the level 6 auto-vector in the vector table.

Also, when testing the service routine, you can initially avoid the complexities of using the 7-segment display interface by simply printing the time (HH:MM:SS) as text in the terminal I/O window using TRAP #15 (and task numbers 3 and 6).

2. Write a polling loop to display the current time on the 7-segment interface, as illustrated in Fig. 10. This loop should run continuously. On each iteration it should read the values of the global three variables HH, MM, and SS. It should then convert each of these (binary) integer values into two binary-coded decimal digits (each between 0 and 9) for displaying on the 7-segment display. For example, if the current number of minutes (MM) is 37_{10} , the first digit will be 7_{10} and the second digit 3_{10} . The 7-segment pattern for displaying each digit should be determined, and then written to the appropriate location in the 7-segment interface for displaying the time. The time format should match that in Fig. 10; that is, HH-MM-SS, including the “-” used to separate the hours, minutes, and seconds.
3. Combine the previous two modules into a single program. The program should automatically display the hardware window. When the program is first run, the 7-segment display should be blank (i.e., all of the segments are “off”) waiting for the first interrupt to be generated. To generate this interrupt, the following tasks should be performed by the user in the order below:
 - a. From the pull-down window in the Auto-Interval interface, select IRQ 6.
 - b. Set the timer delay to 16 milliseconds, again in the Auto-Interval interface.
 - c. Check the checkbox under the push button for interrupt-level 6.

The clock should now start running. (Note: You may wish to have the clock start from the following state: 00-00-00. However, in practice, the clock can start from any legal state depending on how you initialize the global variables HH, MM, and SS in your program.)

4. The final feature you must add to your program is a way to reset the clock to the state 00-00-00 at any point in time. One way to do this is to implement a level 7 interrupt-service routine to clear the global variables HH, MM, and SS. To generate a level-7 interrupt, the user need only press push-button 7.

A Final Consideration

As things stand, the interrupt interface provides an auto-vectored level 6 interrupt at 1/60 second intervals. However, if you compare the time displayed on the 7-segment interface with wall-clock time, you will likely find that they do not agree; that is, you may find that your software clock is running slower (or faster) compared to wall-clock time.

In general, this is not an uncommon problem with computers. The clocks on multiple computers can easily differ due to a myriad of issues including accuracy, stability, jitter, resolution, and monotonicity. There can even be a difference between the system clock and the hardware clock in the same computer. For example, the C function `adjtime()` can be used to adjust the value of the system clock by up to 2 hours in order to synchronize it with the computer's hardware clock.

In the case of the Easy68K, the built-in software timer cannot be trusted to be accurate due to vagaries in the execution time of code before, after, and during a timing event, like a level-6 interrupt.

In the case of your 24-hour clock, one way to attempt to address the difference between wall-clock time and the time displayed by the simulator is to use a calibration factor to adjust the initial value of the counter in the service routine. To do this, perform the following tasks:

1. Reset the 24-hour clock to 00-00-00, and let it run for 3 minutes of wall-clock time. (You can track wall-clock time using your phone, for example.) After exactly 3 minutes of wall-clock time, record the time that appears on the 7-segment display in seconds below:

Simulator Time = _____ (seconds)

2. Now compute the following ratio:

$$Calibration = \frac{Simulator\ Time(s)}{Wall\ Clock\ Time\ (s)} = \frac{Simulator\ Time(s)}{180}$$

3. The previous ratio will either be greater than or equal to one, or less than or equal to one. In the former case, you will want to proportionally increase the value of the counter used in the service routine. In the latter case, you will want to proportionally decrease the value of the counter in the service route. You can do this as follows:

$$Counter\ (new) = Counter\ (old) \times Calibration$$

where *Counter (old)* has the value 60.

4. In your program, change the old counter value to the new counter value. (Remember to change this value throughout your code. You don't want to leave the old counter value (i.e., 60) anywhere in your code.)

Now try running your code. The time displayed on the 7-segment display should be more closely synchronized to wall-clock time.

Save your program in a file called **clock.X68**.