

Memory-Mapped Input-Output

CIS*2030
Lab Number 8

Name: _____

Mark: _____/100

Overview

The term *Input-Output (I/O)* is used to describe the transfer of data between the processor and external (hardware) devices. One of the most common I/O strategies that processors use to communicate with external devices is called *memory-mapped I/O*. To this point in the course, we have naturally assumed that an address is always associated with a particular location in memory (or RAM). However, in the case of memory-mapped I/O, a set of addresses in the processor's address space is set aside, and then assigned to registers inside the I/O devices. In other words, the registers inside the external I/O devices get their own "memory addresses" just like the locations in RAM get their own "memory addresses". The primary advantage of memory mapped I/O is that accessing registers inside external devices is the same as accessing locations in memory, at least from the programmer's perspective.

As discussed in class, actually performing I/O operations typically requires performing read and write operations on the various registers associated with the I/O device. Depending on the capabilities of the interface, communicating with the device may either done using a programmed I/O strategy or an interrupt-driven I/O strategy. The former requires the processor to interrogate the device until the device is ready to participate in the I/O operation. The latter frees up the processor by requiring the I/O device to contact the processor when it is ready to participate in the I/O operation.

Objectives

Upon completion of this lab you will:

- Understand memory-mapped I/O from a programmer's perspective
- Understand how to communicate with simple I/O devices using either a programmed I/O or interrupt-driven I/O strategy

Evaluation

The only graded activity in this lab is the final programming exercise. No marks are assigned to any of the prior question. Nevertheless, it is crucial that you complete all of the exercises, as these will provide you with the requisite knowledge and experience to complete the final programming exercise.

Preparation

Prior to starting the lab, you should review your course notes, and perform the following reading assignments from your textbook (if you have not already done so):

- All of Chapter 7 (Note: This chapter describes both the software and hardware sides of input-output. Our concern is with the software side of things, not the hardware.)

Introduction

As discussed in class, the 68000 employs a memory-mapped I/O strategy to communicate with input-output devices external to the processor. This means that each external device is allocated one (or more) addresses in the processor's address space, and the programmer communicates with the device by performing read and write operations on these addresses.

In the case of the Easy68K, the simulator supports four simple I/O devices: (1) an eight digit 7-segment display, (2) a bank of eight Light-Emitting Diodes (LEDs), (3) a bank of eight toggle switches, and (4) a bank of eight push-button switches.

To see these devices, first start the Easy68K simulator. Then select "Hardware" from the View pull-down menu. You should see a hardware window similar to the one shown in Fig. 1 on the next page.

The four I/O devices are located in the upper half of the hardware window, starting with the 7-segment display and ending with the bank of eight push-button switches. To the immediate right of each I/O device is an **Address** field. This field shows the current memory-mapped address of the device. For example, the current memory-mapped address of the bank of eight LEDs is 0x00E00010. In practice, each of the four I/O devices can be mapped to any valid 68000 address by manually entering the address in the Address field. If the address entered conflicts with the address of another device, the color of the address will change to **red**. The memory map in Fig. 1 provides a visual illustration of the current address space, showing the memory-mapped address (or addresses) associated with each I/O device.

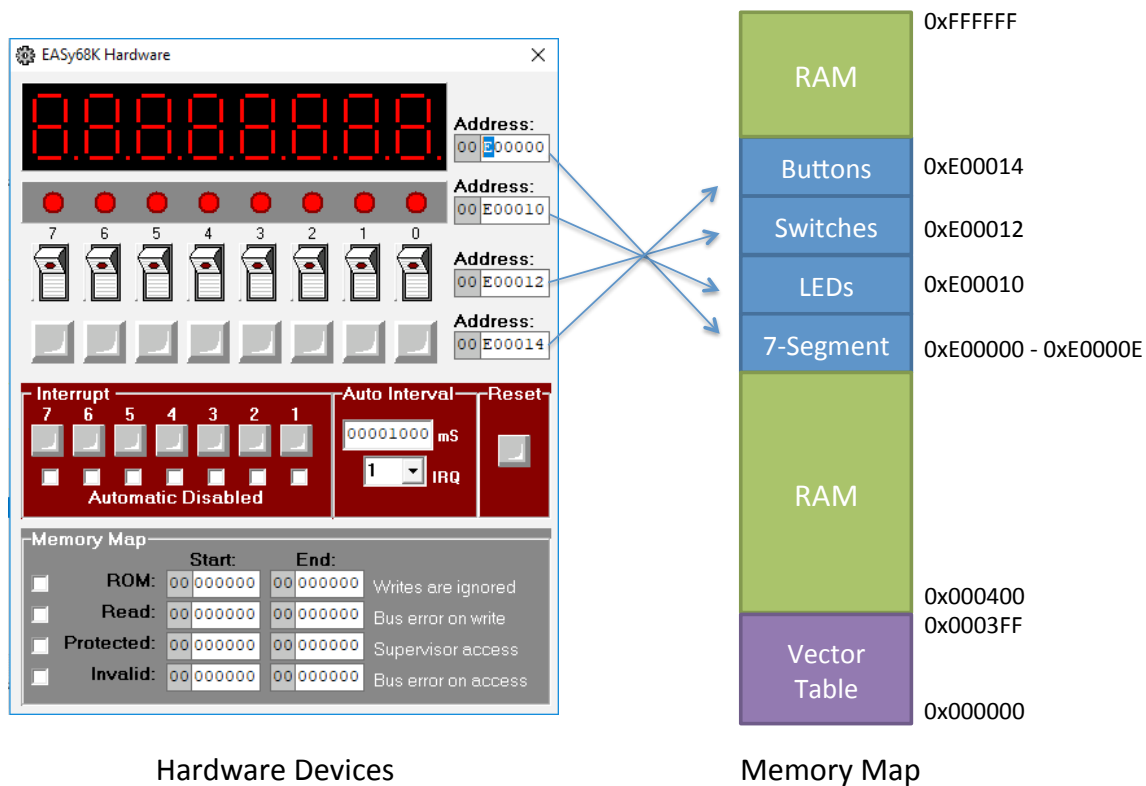


Figure 1: Hardware window and memory-mapped addresses of I/O devices.

The Easy68K enables a program to interact with the simulated hardware devices through use of TRAP #15 and task number 32 (stored in D0). For example, rather than select “Hardware” from the view menu, the following code can be used to automatically display the hardware window:

```
MOVE.L #32,D0      ;task 32
CLR.B D1           ;display hardware window
TRAP #15           ;system call
```

Similarly, the following code can be used to determine the (base) address of the 7-segment display:

```
MOVE.L #32,D0      ;task 32
MOVE.B #1,D1       ;return address of 7-segment display in D1.L
TRAP #15           ;system call
```

Sim68K Environment

TRAP #15 is used for I/O. Put the task number in D0.

Task	
30	Clear the Cycle Counter
31	Return the Cycle Counter in D1.L. Zero is returned if the cycle count exceeds 32 bits.
32	<div>Hardware/Simulator D1.B = 00, Display hardware window D1.B = 01, Return address of 7-segment display in D1.L D1.B = 02, Return address of LEDs in D1.L D1.B = 03, Return address of toggle switches in D1.L D1.B = 04, Return Sim68K version number in D1.L Version 3.9.10 is returned as 0003090A D1.B = 05, Enable exception processing. Exceptions will be directed to the appropriate 68000 exception vector. This has the same effect as checking Enable Exceptions in the Options menu. D1.B = 06, Set Auto IRQ D2.B = 00, disable all Auto IRQs or Bit 7 = 0, disable individual IRQ Bit 7 = 1, enable individual IRQ Bits 6-0, IRQ number 1 through 7 D3.L, Auto Interval in milliseconds, 1000 = 1 second D1.B = 07, Return address of push button switches in D1.L</div>

Figure 2: Interacting with hardware simulator through software.

Next, you will work through a series of programming exercises to understand how to communicate with some of the previous hardware devices.

Part 1: Writing to LEDs

An LED is like most other primitive semi-conductor devices in the sense that it operates like a switch. Under the right conditions, the switch closes, current flows, and the LED turns “on” and lights up. Otherwise, the LED remains “off” and does not light up.

Figure 3 illustrates how the process works on the Easy68K. The eight LEDs are mapped to the byte at memory location 0xE00010. Each bit in the byte controls one of the eight LEDs. In particular, bit_i controls LED_i. To turn LED_i on (red), bit_i must be 1.

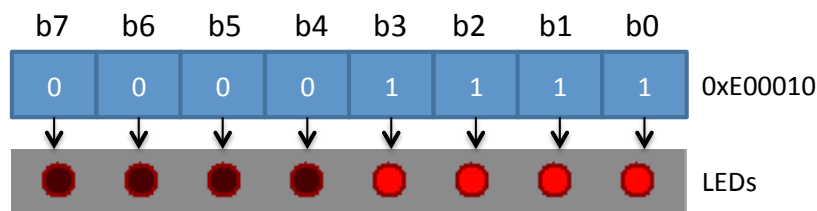


Figure 3: Interface for bank of eight LEDs.

Step 1

Download the sample program called **Lab8a.X68** from the course website.

Step 2

Assemble, and then execute the program using the Easy68K simulator. You should see an output similar to the one in Fig. 3.

Programming Task

Write a program for the previous LED interface that inputs a character from the keyboard and displays the ASCII code for the character on the LEDs. Put the program in a loop and terminate when 'q' is detected.

- TRAP #15 with task number 5 should be used to read a single ASCII character from the keyboard into D1.B (see Easy68K help)
- Save your program in a file named **Lab8b.X68**.

Part 2: Reading from Toggle Switches

The next interface that you will consider is simple *input* from the bank of eight toggle switches. Figure 4 illustrates how the process works on the Easy68K. The bank of eight toggle switches is mapped to the byte at memory location 0xE00012. Each bit in the byte is affected by one of the eight switches. In particular, bit_i is affected by switch_i. For any switch that is closed (or down) the corresponding bit is set to logic 1. Otherwise, the corresponding bit is set to logic 0.

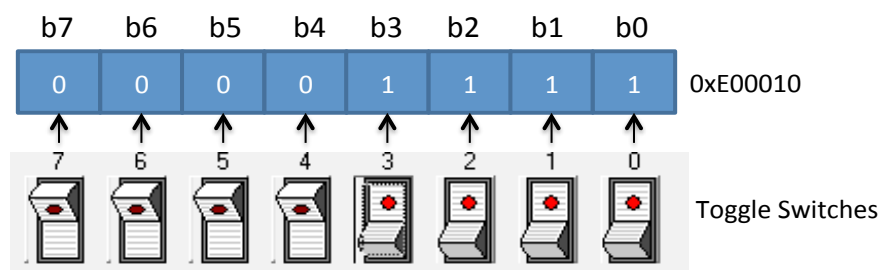


Figure 4: Interface for bank of eight toggle switches.

Step 1

Download the sample program called **Lab8c.X68** from the course website.

Step 2

Assemble the program. Set a breakpoint at the `MOVE.B SWITCHES(A1), D1` instruction on line 24, and then execute the program. Once the program breaks, use your mouse to toggle some of the switches. Now execute the instruction on line 24 to read the state of the switches. Examine the contents of `D1` to determine which bits are set to 1 and which bits are set to 0 based on the settings of the switches.

Programming Task

Write a program to simulate a *wire* connection between the bank of eight LEDs and the bank of eight toggle switches (see Fig. 5). The program should *continuously* read the switches and turn on/off the corresponding LED for any switch that is closed/open. Save your program in a file called **Lab8d.X68**.

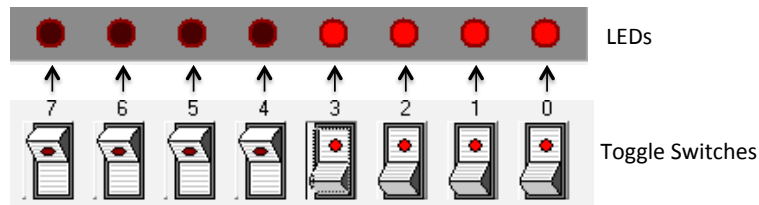


Figure 5: Software simulation of wire connection between switch/LED interfaces.

Part 3: Reading from the Push Buttons

The push buttons are another *input* interface. However, there are two differences between the push buttons and the previous toggle switches. First, unlike the toggle switches, push buttons write a logic 0 to the corresponding bit in memory when the button is pressed, and write a logic 1 when the button is released. Second, a push button only writes a logic 0 to the corresponding bit in memory while the button is being pressed. Once the button is released, it continually writes a logic 1 to the corresponding bit in memory, as illustrated in Fig. 6.

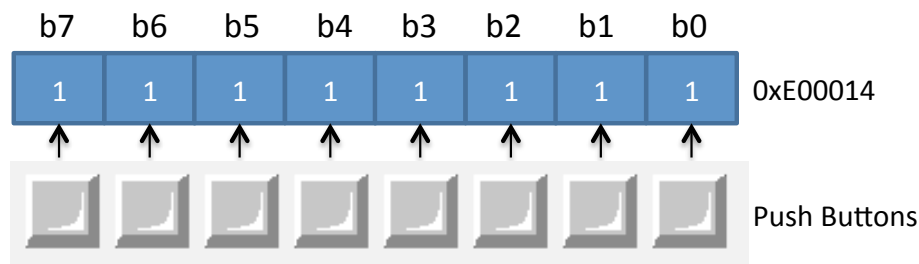


Figure 6: Push-button interface.

Programming Task

Write a program to simulate a *wire* connection between the bank of eight LEDs and the bank of eight push buttons (see Fig. 6). The program should *continuously* read the push buttons and turn on/off the corresponding LED for any switch that is pressed/released. Save your program in a file called **Lab8e.X68**.

Part 4: Writing to the 7-Segment Display

One of the most common *output* interfaces used in commodity electronic devices is the 7-segment display. This is because the display can be used to output various values, including the decimal digits 0 through 9 and the hexadecimal digits A through F. Shown pictorially in Fig. 7, a 7-segment display consists of 7 segments, labeled *a*, *b*, *c*, *d*, *e*, and *f*. Each of these segments can be selected and turned on/off to display a numeric value, as shown in Fig. 7b.

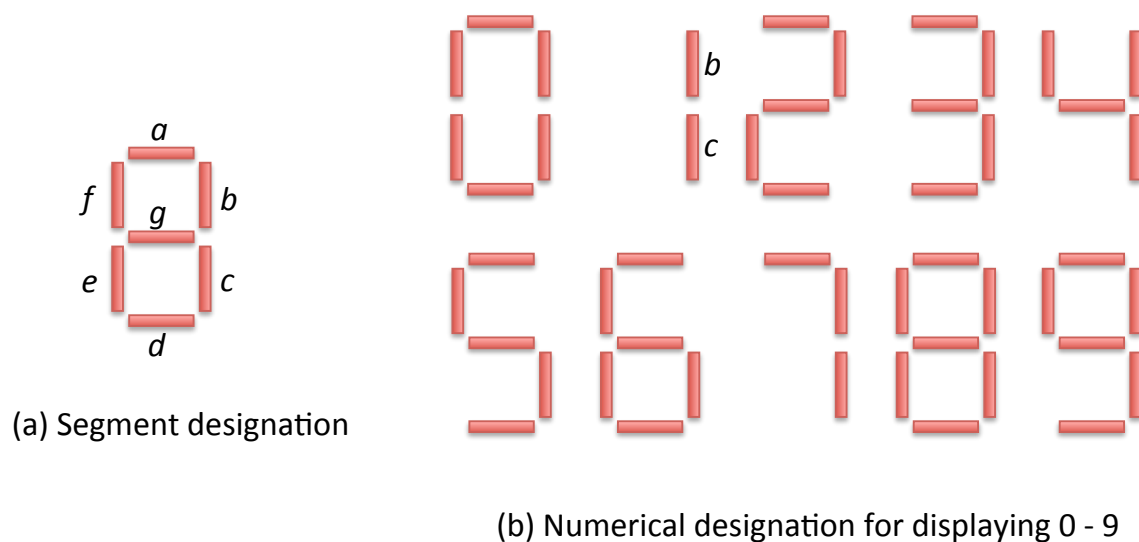


Figure 7: 7-segment display segment designation.

As shown in Fig. 7b, to display the number 1 on the 7-segment display it is necessary to select and turn on segments *b* and *c*. Figure 8 shows how this can be accomplished. The first thing to observe is that the Easy68K supports an 8-digit 7-segment display, where each 7-segment display can be used to display a digit. Each of the eight 7-segment displays has its own memory-mapped address: the first 7-segment display is located at address 0xE00000, the second at 0xE00002, the third at 0xE00004, and so on. (Notice that the 7-segment displays are located at consecutive *even* addresses.) The bytes at each even address are used to write to the 7-segment displays. For example, to display the number 1 on digit 1 – the first 7-segment display on the left in Fig. 8 – the binary code 00000110₂ must be written to address 0xE00000 in order to select and turn on segments *b* and *c*. Notice that the most-significant

bit in the previous byte is set to 0. The most-significant bit is used to display a decimal sign if set to 1. However, we are not interested in using this feature in the current example, so it is set to 0.

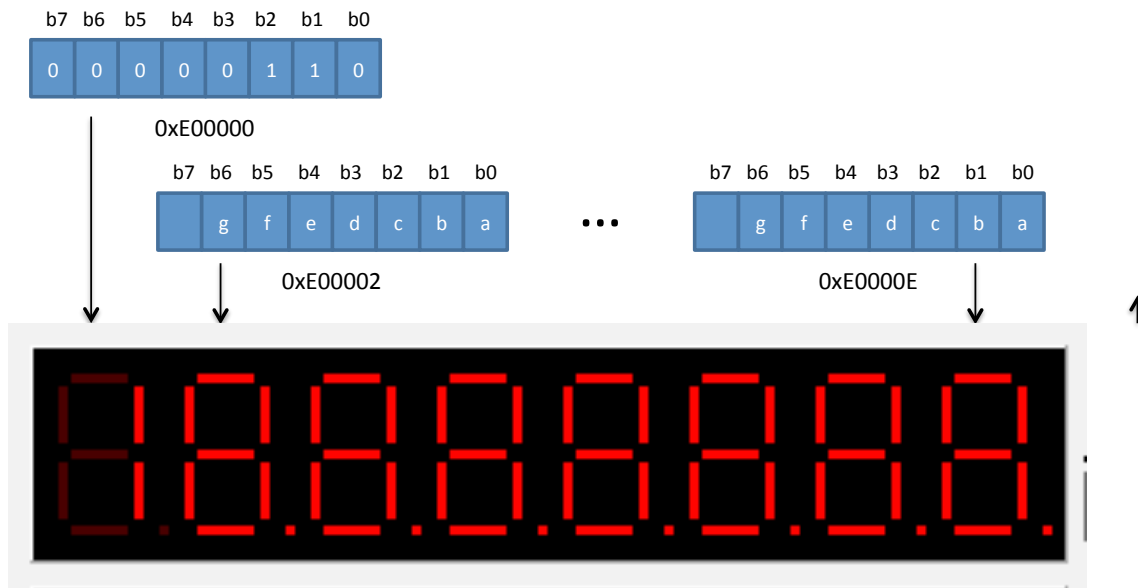


Figure 8: Writing to the 8-digit 7-segment display interface.

Step 1

Download the sample program called **Lab8f.X68** from the course website.

Step 2

Assemble, and then run the program. You should see the numbers 0 through 7 displayed on digits 0 through 7, as shown below:



Programming Task

Write a program that *continuously* reads a hexadecimal code in the range of 0 to F from the toggle switches and displays the equivalent 7-segment pattern on the right-most display (digit 7). Save your program in a file called **Lab8g.X68**.

Now that you know how to communicate with the previous input-output (I/O) interfaces, you will use the interfaces to gain experience with the following input-output strategies: Programmed I/O and Interrupt-Driven I/O. (Note: Easy68K does not provide a direct-memory access (DMA) interface.)

Part 5: Programmed I/O

As discussed in class, the processor communicates with external devices using either programmed I/O, interrupt-driven I/O or direct-memory access methods. However, among these three, *programmed I/O* is by far the simplest. With this method, when the processor wants to communicate with an external device, it executes a *polling loop* in software. The purpose of the polling loop is to continuously interrogate the external device until it is ready to participate in the I/O operation. Once ready, the processor proceeds with the data transfer. If more than one device is involved, the devices are typically polled in a serial fashion. The polling process is illustrated in Fig. 9.

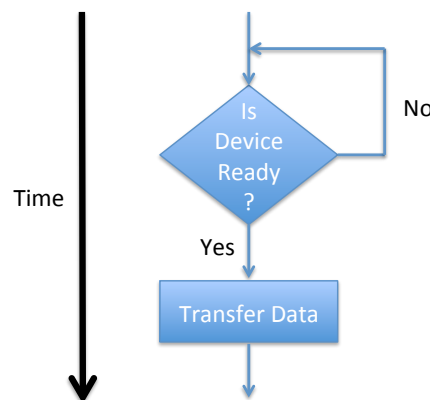


Figure 9: Polling process.

As explained in class, determining if an external device is ready to communicate typically involves testing a bit in a status register. For example, if the bit is set to 0, the device is not ready to participate in the operation, but if the bit is set to 1, the device is ready to participate in the operation. None of the Easy68K interfaces are sophisticated enough to require a status register (or any other registers for that matter). However, as you will see, it is possible to simulate a “ready bit” using the push-button interface.

Programming Task

To better understand programmed I/O you will write a simple program involving the following three interfaces:

- Toggle switches
- LEDs
- Push buttons

The purpose of the program is to read an 8-bit value from the toggle switches and then display the value on the LEDs when the left-most push button is pressed. The program should conform to the following requirements:

- Initially, all eight LEDs are to be off.
- The user will enter a value on the toggle switches, but this value should not, yet, be displayed on the LEDs. The value should only be displayed with the left-most push button is pressed.
- A polling loop is required to determine when the left-most push button has been pressed. Only when the left-most push-button is pressed, should the value entered on the toggle switches be output to the LEDs. (In this context, the push button is acting like a ready bit in a status register indicating that the output operation can now be performed.)
- The program should run in a continuous loop, allowing the user to repeatedly enter and display values.

Remember to properly comment your code. Save your program in a file called **Lab8h.X68**.

Part 6: Interrupt-Driven I/O

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-vectored 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 **Lab8i.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    #$0700,SR       ;set priority level to 7
                MOVEM.L  A1/D0,-(A7)     ;save working registers
                ANDI.W    #$F9FF,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
```

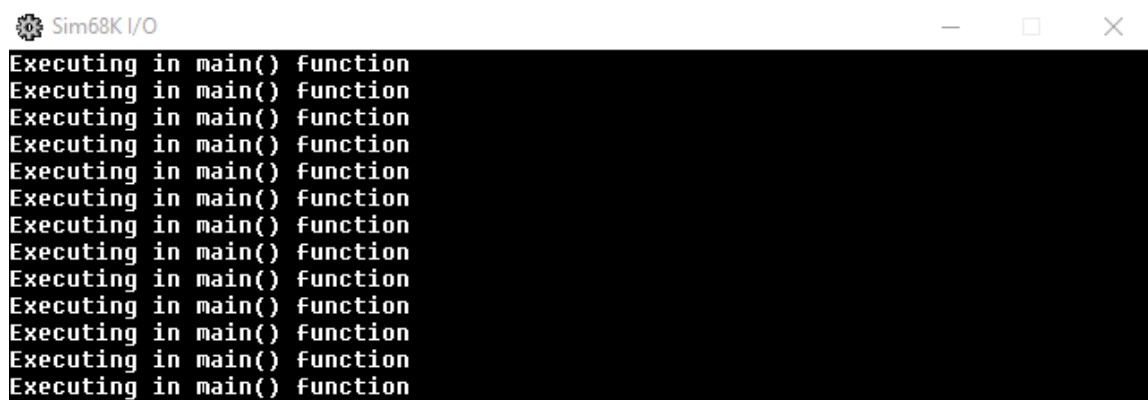
Notice that the ISR begins by setting the priority mask to 7, thus (temporarily) disabling all lower-level interrupts. With all lower interrupts disabled, the ISR saves its working registers on the system stack, thus making itself transparent to the (interrupted) main code. Once saved, the ISR returns 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

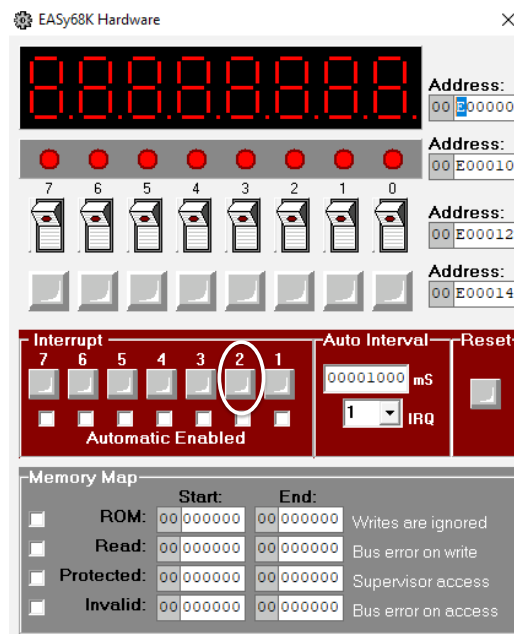
Run the program on the Easy68K simulator. Initially, you will see the following output being generated on the terminal:

A screenshot of a terminal window titled "Sim68K I/O". The window has a black background with white text. The text consists of 13 identical lines stacked vertically, each reading "Executing in main() function". The window has standard macOS-style window controls (a gear icon, a minus sign, a square, and a cross) in the top right corner.

This 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 below):



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)?

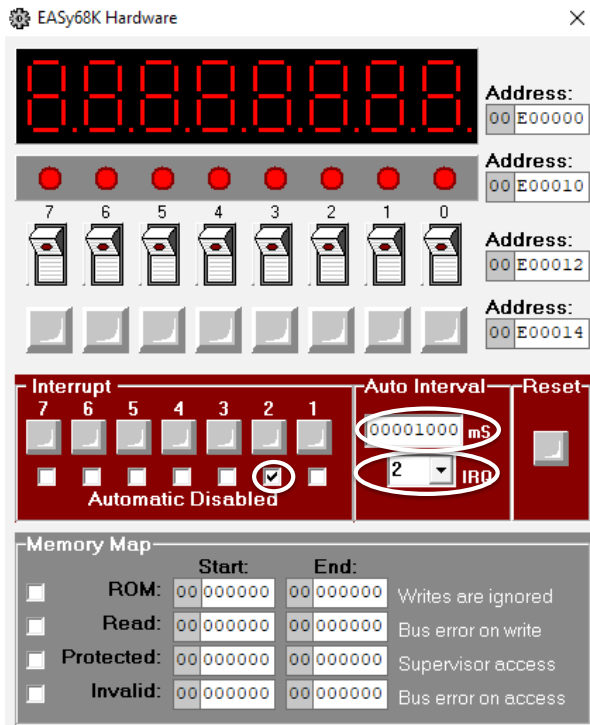
Interrupt mask: _____

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.

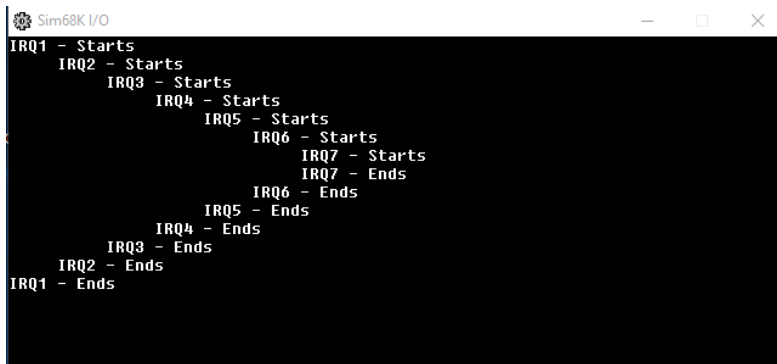
Programming Task

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 “ IRQ_n – 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 pervious message should be proceeded 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.

Remember to properly comment your code. Save your program in a file called **Lab8j.X68**.

Part 7: Final Programming Exercise

In this final programming exercise, you will implement an auto-vector interrupt-driven 24-hour digital clock. In completing this exercise, 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.

Figure 10 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.

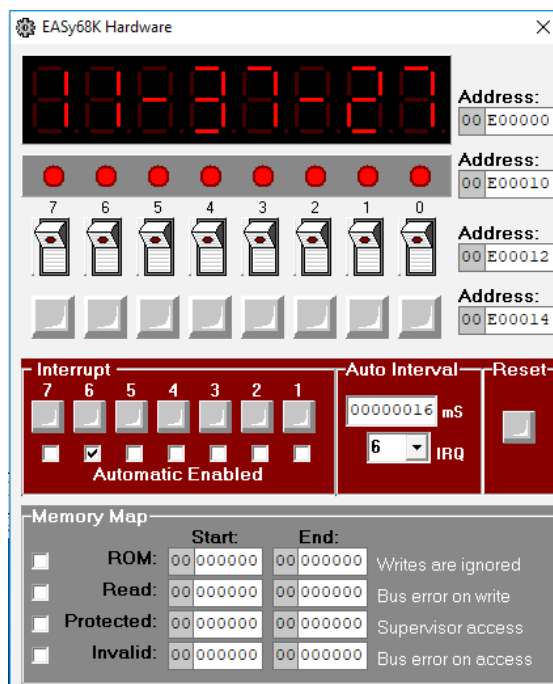


Figure 10: 24-hour clock.

Programming Tasks

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 (HH) 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.)

Reset Feature

A useful feature that you may wish to 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.

Name your program CLOCK.X68