Input and Output (I/O)

CIS*2030
Fall 2024

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

Name:

ID:

Section:

Overview

As covered in class, there are various methods like serial I/O, parallel I/O, and polling employed for communication between processors and peripheral devices within computer systems. *Serial* interfaces transmit data sequentially one bit after another (for instance, through a bus system), whereas *parallel* interfaces dispatch several bits concurrently. In contrast to these hardware-based strategies, *polling* is a software protocol wherein the central processing unit (CPU) persistently verifies if any peripheral device requires its service; this method is compatible with both serial and parallel communication modes. This lab provides practical exposure to these three distinct communication strategies.

Objectives

During this lab you will:

- Learn to interface with memory mapped I/O devices using the Easy68K hardware window,
- develop subroutines for character I/O via an asynchronous serial interface based on the 68681 DUART chip, and
- implement simple subroutines for parallel I/O to interact with LEDs, switches, and push buttons.

Preparation

Prior to attending your lab section, you must complete the following reading assignments from your textbook:

- Chapters 7 and 8

- Sections 9.1-9.3

Please note that chapters 7 and 8 of your textbook offer specific details on the 68KMB implementation. While you don't need to know these details intimately, they provide useful background information. The essential information was covered in the lecture, so make sure to review your lecture notes on these topics. Remember, you won't have time to do the reading assignments and complete the lab questions during the lab session. Therefore, it's crucial to come prepared!

Evaluation

This lab has 5 questions all of which contain multiple parts. Questions *with* an asterisk '*' beside them can be completed prior to the lab. However, all other questions must be performed during your scheduled lab session.

Question	1*	2*	3*	4	5	Total
Max points	5	5	5	10	25	50
Your Points						

All marking is performed in real time *during* the lab. Therefore, when you complete a question raise your hand to let the lab instructor know that you are prepared to be marked. Marks will be given based on the written answer you provide and/or your ability to orally answer related questions asked by the lab instructor. The key to obtaining full marks is to demonstrate comprehension.

Introduction

The 68000 processor uses a memory-mapped I/O strategy to communicate with external input-output devices, where each device is assigned one or more addresses in the processor's address space, allowing the programmer to interact with the device through read and write operations.

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 start the Easy68K simulator and select "Hardware" from the View pull-down menu. You should see a hardware window like the one shown in Fig. 1.

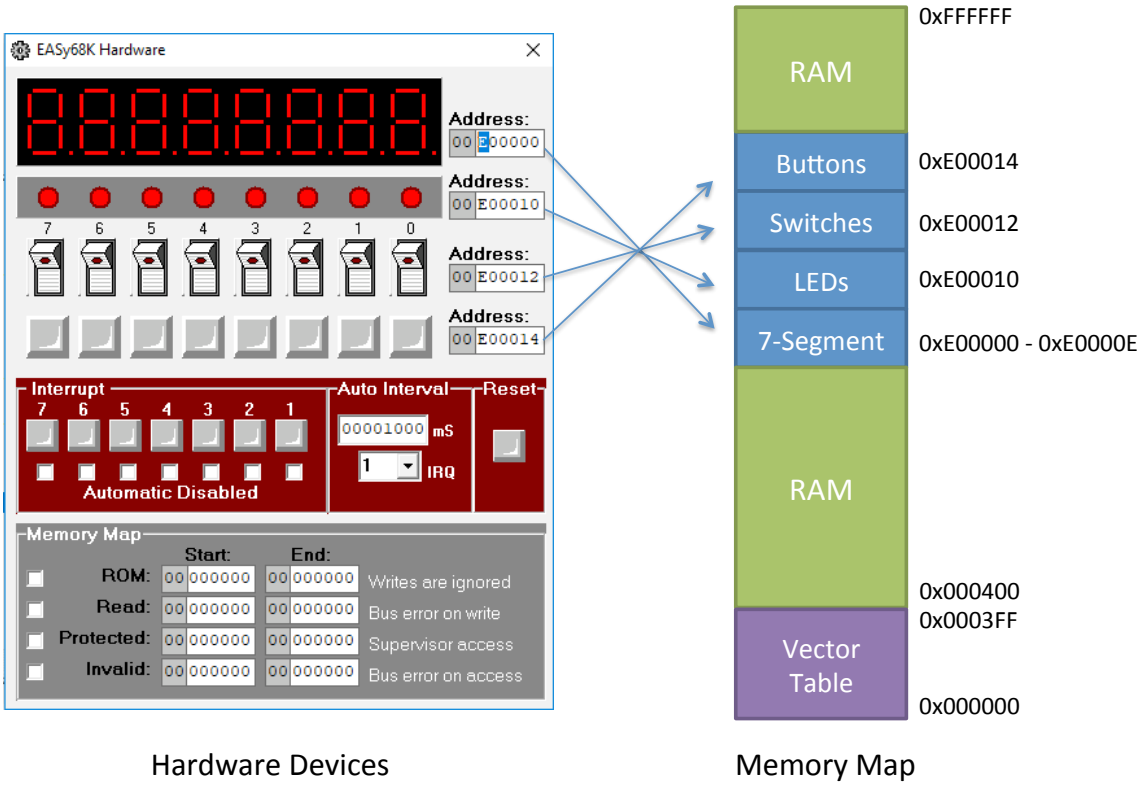

Figure 1: Hardware window and memory-mapped addresses of I/O devices.

The hardware window displays the four I/O devices and their current memory-mapped addresses. For example, the bank of eight LEDs is mapped to 0x00E00010. You can manually change these addresses, but if a conflict occurs, the address color will turn red. The memory map visually illustrates the address space and the addresses associated with each I/O device.

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
```

Also, TRAP instructions can be used to determine the start (base) address of a particular device. For example, when executed the code below returns the 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
```

By changing the value of the parameter passed to the TRAP handler in data register D1, the address of all the I/O devices can be determined as illustrated in Fig. 2.

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	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

Figure 2: Interacting with hardware simulator through software.

Question 1: Writing to LEDs

We have seen LEDs before in an earlier lab. Nevertheless, recall that 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 red. Otherwise, the switch remains open, the LED remains "off" and does not light up and remains black.

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. The bank of 8 LEDs acts as a parallel interface, meaning each LED can be controlled independently and simultaneously because each has its own dedicated line or bit for control.

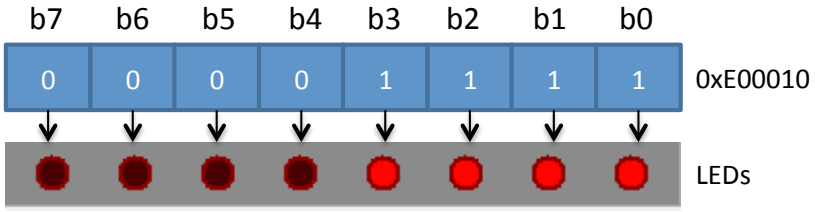

Figure 3: Parallel interface for bank of eight LEDs.

Download the sample program called **LEDs.X68** from the course website and load it into the Easy68K environment. Assemble and run the program. You should see an output like that in Fig. 3.

Now, write a program for the previous parallel 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 **LED_ASCII.X68**.

Question 2: Reading from Toggle Switches

This next interface is also parallel – but consists of 8 toggle switches each of which can be used as a 1-bit input device. 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. This arrangement allows for simultaneous, parallel input across all switches.

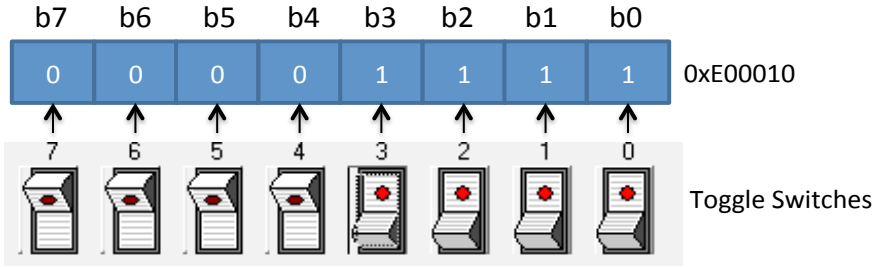

Figure 4: Parallel Interface for bank of eight toggle switches.

Download the sample program called **toggle.X68** from the course website and open it in the Easy68K environment. Assemble the program. Set a breakpoint at the `MOVE.B TOGGLE(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.

Now, 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 **wire.X68**.

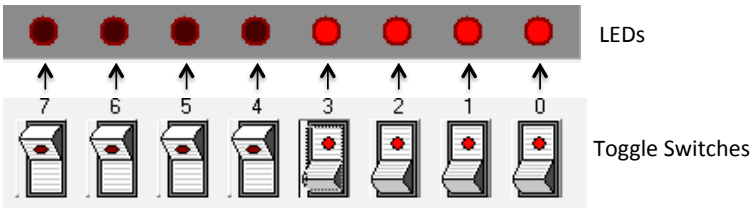

Figure 5: Software simulation of wire connection between switch/LED interfaces.

Question 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.

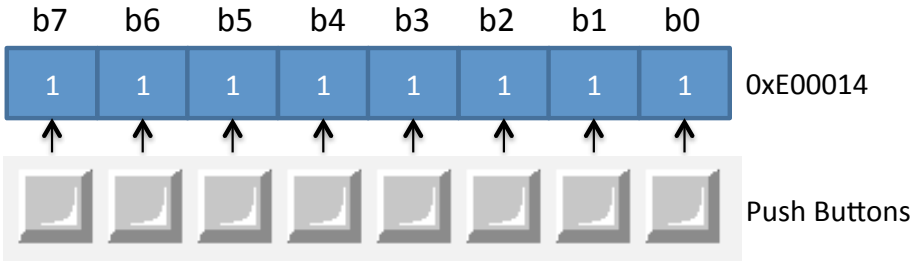

Figure 6: Push-button interface.

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 button that is pressed/released. Save your program in a file called **button.X68**.

Question 4: Polling

When the processor wants to communicate with an external device, the simplest way that this can be done is to execute 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. 7.

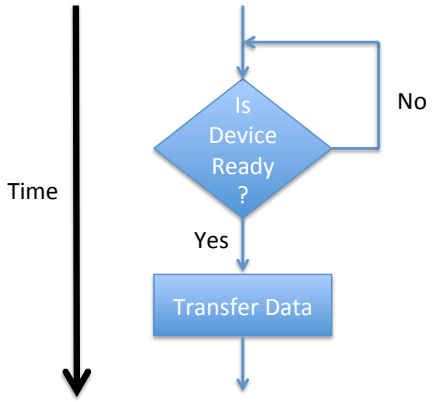

Figure 7: Polling process.

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.

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 **polling.X68**.

Question 5: Serial I/O Interface

By far, the simplest and cheapest way to connect a computer to a dumb terminal is using an asynchronous serial interface. This approach is used in your textbook to connect the 68000-based single-board computer (68KMB) to a host computer that runs software that allows it to function as a dumb terminal for the 68KMB. Figure 8 provides a simplified illustration of this configuration.

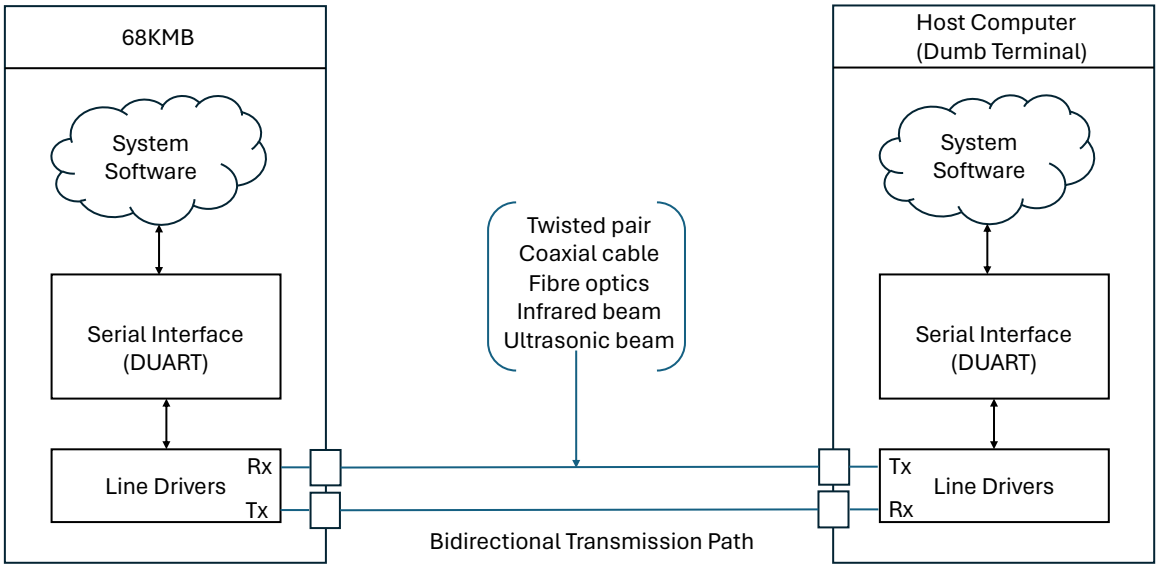

Figure 8: Functional units used in the serial datalink presented in your textbook.

Two-way communication is required between the 68KMB and the dumb terminal because information from the keyboard is transmitted (from the dumb terminal) to the 68KMB and information from the 68KMB is transmitted to the (dumb terminal's) screen. The heart of the data link is the box labeled *serial interface* which translates the data between the form in which it is stored within the 68KMB (parallel) and the form in which it is transmitted over the data link (serial). This conversion of data between parallel and serial form is carried out by a 68681 Dual Universal Asynchronous Receiver Transmitter (DUART). The line drivers translate the signals processed by the DUART into a suitable form of sending

over the transmission path. The transmission path is normally a twisted pair of wires but could be fiber optics or even infrared links.

As explained in class, each character is transmitted in the following sequence:

- a start bit (low)
- 7 or 8 data bits (LSB first)
- a parity bit (optional)
- one or more stop bits (high)

This type of communication is called *asynchronous* since the receiver must resynchronize itself with each new character. Usually, 7 data bits are used since this is the size of ASCII data. Figure 9 shows the ASCII code for the character *a* framed by a start bit, an odd parity, and a stop bit. (Only the data bits and the parity bit are counted when computing parity.) The reciprocal of the transmission time for each bit is called the baud rate. For the 68KMB, the baud rate is 9600, so the period of transmission for each bit is 104 μs.)

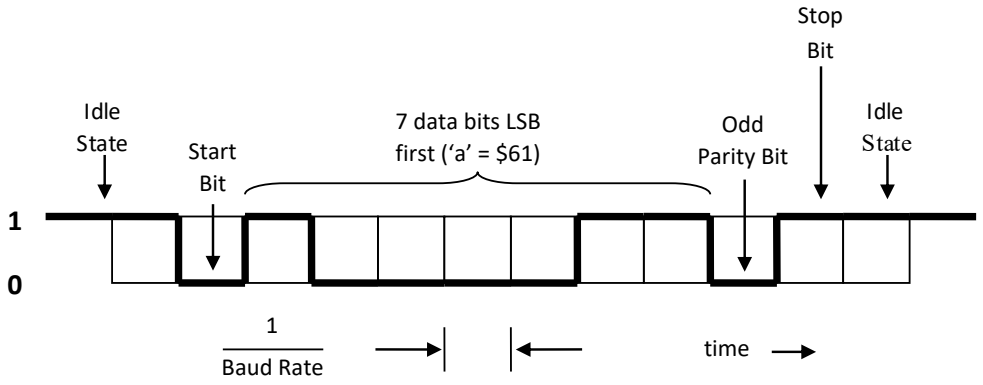

Figure 9: Format of asynchronous serial data.

Figure 10 provides a simplified view of the interface between the 68000 processor and the DUART, along with a visual description of the main registers used to program the serial interface. The 68681 features a double-buffered transmitter and a quadruple-buffered receiver. During transmission, one character can wait in the Transmit Buffer while the previous character is being transmitted from the shift register. For reception, a three-character FIFO (first in, first out) buffer holds characters waiting to be read by software, while the next character is clocked into the shift register.

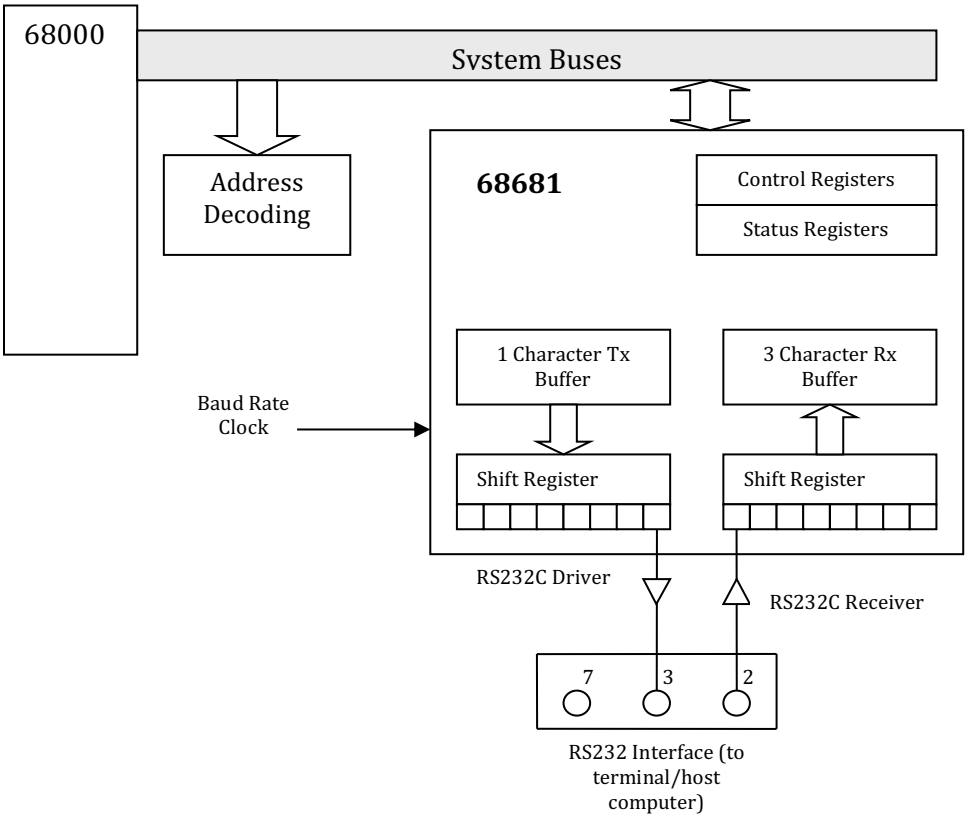

Figure 10: Simplified interface between CPU and DUART.

As discussed in class, address decoding allows access to the 68681's registers through odd-byte addresses ranging from $00C001 to $00C01F. Before using the DUART for transmitting or receiving characters, it must first be configured. Once configured and activated, the transmit and receive buffers can be used to send and receive characters using a simple polling strategy.

Although students no longer have access to the 68KMB hardware board, we have developed a program called **duart.X68** that simulates a serial datalink between programs running in Easy68K and a dumb terminal This program is designed to provide a practical understanding of how data transmission occurs in this specific hardware setup. By using duart.X68, you can visually observe the process of sending or receiving characters, once the DUART has been correctly configured for operation.

Start by downloading duart.X68 and loading it into the 68K environment. Carefully read through the program code. Notice that the code provides a series of labels corresponding to the base address of the DUART (i.e., $00A001), along with the offsets of the various registers (e.g., MR1A, MR2A, etc.) from the base. As explained in class, by assigning each register to a label, we can efficiently access a register using address register indirect addressing with offset. To access a register, we first need to initialize address register `A0`

with the base of the address register. We can then use the label as the offset. For example, to store the value 12_{10} in the CRA register, we can write the following code:

```
MOVEA.L    #DUART,A0
MOVE.B     #12,CRA(A0)
```

The code in duart.X68 also contains a label for the address of the subroutine `printASCII`. This subroutine is part of the simulator code. When called, it will display the ASCII value of the character stored in D0.B on the console. You will require this subroutine when writing the code described next.

Your task is to write code for the PC side of the interface. This code will read a null-terminated string of characters sent one at a time from the 68KMB and display these on the PC's console. Your program will contain two main parts. The first part requires writing a sequence of instructions to initialize the DUART's channel A to operate at 4800 baud with 1 start bit, 1 stop bit, 7 data bits, and even parity. Don't forget to enable the transmitter and receiver.

Once the DUART on the PC side is ready to receive characters from the 68KMB, you can move to the second part of your code. Here, you will write a sequence of instructions to implement a polling loop. The loop will read a string of characters from the 68KMB sent one at a time. On each iteration of the polling loop, you will need to examine the status register. This will let you determine if a character is waiting to be read in the receive buffer. You will need to repeat this step until a character arrives from the 68KMB.

Once a character arrives, you should read it from the receive buffer. Then, use the `printASCII` subroutine described above to display the character on the PC's console. Your program should terminate when the null-character is received and should not attempt to print the character.

Submit

You are required to submit your files online. Each file will be uploaded separately; that is, do not zip your files. In the Week 11 folder on CourseLink you will find a Dropbox into which you can submit the following:

- The source (.X68) files,
- the executable files,
- the listing (.L68) files, and
- the log (.txt) files.