IN1805II project — Input/Output

Phase V of your emulator will consist of an input/output mechanism and your final report. For the report, refer to Section A.3 of the project manual. This handout deals with the input/output part of the assignment.

Unlike for example x86-based chips on which you can use an interrupt poll, the 6502 has no special instructions for input/output. For the processor to perform these actions, it must do one of two things.

The first option is continually poll if I/O devices are ready, by checking certain memory locations. Typically, communication lines with I/O devices (connected via interface cards) were mapped to specific parts of the memory. The other option is to utilize interrupts.

Fixed interval polling can lead to significant resource wastage, especially when there are numerous I/O devices. Interrupts are a better solution, but the original 6502 featured them only in hardware, and devices were always custom made. Reasons for this are that every byte cost money: if obscure communication schemes could make a byte obsolete, it was worth the effort.

Another option is that hardware can be used to trigger actions. For instance, when an ASCII output chip would see a \0 (null byte) character coming along, signalling the end of its input, it could start the conversion and output immediately. As assembly does not allow for convenient memory value triggers, we will not be able to precisely replicate such behaviour. This leaves us with 3 main ways to emulate I/O in your software emulator:

- Create a separate thread for each I/O device that can cause an interrupt to the emulator and checks certain memory locations;
- Perform I/O with all devices when a special instruction is encountered;
- Use fixed interval polling.

Independent of the method used, a memory map is needed which specifies what data should be put where in which format (ever tried executing a text document by renaming the extension or by setting its executable flag?). The 6502 reference tells us that I/O memory is commonly assigned to the address space (40:00) to (7F:FF), being 16 KB of memory. The following list shows the decomposition of this space as we will be using it.

Graphical output	
(40:00)-(6A:29)	120x90x8 bit color graphical bitmap
(6A:30)-(6F:EF)	reserved for future use (high definition displays?)
(6F:F0)-(6F:FC)	reserved for additional graphical interface control flags
(6F:FD)	x resolution
(6F:FE)	y resolution
(6F:FF)	number of colors
Text output and input	
(70:00)-(77:CF)	80x25 terminal screen, 1 byte per character, no unicode
(77:D0)-(77:FF)	reserved for terminal flags/keymaps
(78:00)-(78:FF)	printf string, char or int, terminated by \0
(79:00)-(79:FF)	scanf string, terminated by \0 (the result of the scanf call is written to (78:00)-(78:FF))
Auxiliary (network?) input and output	
(7A:00)-(7C:7F)	auxiliary in
(7C:80)-(7E:FF)	auxiliary out
General	
(7F:00)-(7F:FE)	reserved for future I/O expansion
(7F:FF)	I/O mode: 1 = printf, 2 = scanf, 4 = screen output, 16 = auxiliary in, 32 = auxiliary out

This means, for example, that to print the “Hello, world” string, one would set byte (7F:FF) to 1, fill the bytes (78:00) to (78:0D) with the characters ’H’ through ’!’ and a null byte, and perform the I/O by one of the means described above. When byte (7F:FF) is 0 again, indicating that the I/O has been done, we can do the next input or output call. To show larger texts, we would fill (70:00) to (77:C7), set (7F:FF) to 4, and perform I/O.

Abide to this scheme if you want the assistants to be able to offer some help. The minimal I/O implementation for the project uses the second method as described above: performing I/O when a special instruction is encountered. We will be “misusing” the BRK instruction to perform I/O. Your implementation should be able to at least read a string or number using scanf(), process this input, and show it on the screen using printf(). Note that this would be the bare minimum needed for a user to communicate with a computer.

If you want to do more advanced I/O, you could use your framebuffer (in Linux, memory associated with your screen output) by means of the fbset shell command and devices like `/dev/fb0`, in order to create a graphical I/O environment. It is left to the reader to find out how this works. Several libraries are available on the internet as well to facilitate easier access to the framebuffer. Note that graphical output is not by definition supported by the assistants due to the complexities associated with this approach.