I/O Concepts

I/O Devices

- **Input/output devices** allow the computer to communicate with the world around it
- A computer receives information from an **input device**
  - A keyboard sends information from the user to the computer

- A computer provides information to an **output device**
  - A monitor displays information to the user

- Many devices are capable of both input & output

Basic I/O Devices

- General-purpose I/O (GPIO) pins
  - The most basic form of output is a pin that the processor can set to 1 or 0 (voltage levels)
    - Turn a light ON or OFF
  - The most basic form of input is a pin that the processor can be read to determine its voltage level (1 or 0)
    - Check if a button is pressed or not
- Analog-to-digital converter (input device)
  - Sample an analog signal and represent it as binary data
- Digital-to-analog converter (output device)
  - Convert digital (binary) data into an analog signal

Complex I/O Devices

- Most devices operate as both input and output
  - Network interface
    - Transfer data to and from remote locations
  - Bluetooth radio
    - Use 2-way radio link to interact with Bluetooth devices
  - USB interface
    - The USB interface connects your computer to USB devices
    - All USB devices communicate in both directions, but some have a dominant function
      - Input: USB mouse
      - Output: USB speakers, printer
      - Input/Output: USB flash drive
  - Hard drive
    - Load programs, read and write data

I/O Concepts and Terminology

- Asynchronous I/O and handshaking
- Unconditional vs. conditional I/O
- Isolated vs memory-mapped I/O
- Polled vs. interrupt-driven I/O

I/O Synchronization

- Processors run at a rate controlled by the clock
- But, most I/O activity occurs at unpredictable times unrelated to the processor clock
  - Example: Someone typing on a keyboard
- This is **asynchronous** I/O
  - Asynchronous means "not related to the clock"
- For the processor to operate with asynchronous I/O devices, there must be a way for the processor to check if the device needs service
  - Does the keyboard have a character ready?
  - Does the keyboard know that we just read it?
  - This exchange of information is called **handshaking**

☆ I/O activity is slower than the processor, not related to the clock.

☆ The interchange of information between processor and I/O device

Unconditional vs Conditional I/O

- Some simple I/O devices are always ready to accept or supply new data
  - Example: An output pin driving an indicator light
  - This is **unconditional I/O**
- For most I/O devices, we need to check if they are ready before transferring data
  - Example: Has the display finished processing the last thing we sent it?
  - This is **conditional I/O**

Isolated vs Memory-Mapped I/O

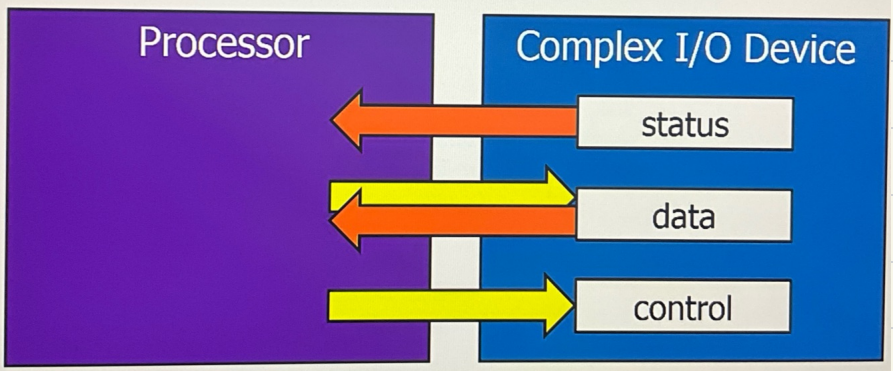
- A processor communicates with an I/O device by reading/writing registers in the I/O interface
- Some processors use special instructions and a different address space to do this
  - This is **isolated I/O** → ☆ Different regions of space for I/O Device
- Most processor use their normal load/store instructions to access I/O devices
  - To the processor, the I/O device registers appear to be normal memory locations… but they are not!
  - We set aside some locations in the memory space, and add logic so that loads/stores to these addresses access the I/O device registers instead of memory
  - This is **memory-mapped I/O** → ☆ Normal memory locations, but add logic that diverts to I/O Device

Polled vs Interrupt-Driven I/O

- **Polling** is the process of repeatedly checking to see when something becomes ready (checks the I/O Device)
  - For example: we poll the keyboard to see if it has a new character; the answer is "no" until one is typed!
- Processors also contain hardware that allows an I/O device to **interrupt** the processor → (Process is doing its task, by the time I/O Device is ready, it jump to the I/O instructions)
  - The device interrupts the processor
  - The processor stops executing the program, and automatically runs code to service the device
    - ☆ The **interrupt service routine** is a special form of subroutine that is invoked automatically by the hardware
  - When it finishes, it resumes execution of the program

I/O Device Interface

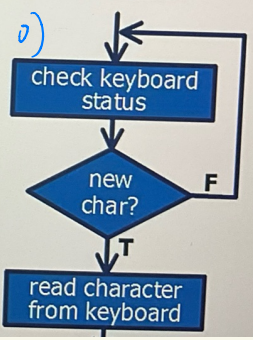
- Programmers do not need to know how an I/O device works internally – just how to use it!
- The I/O device interface is a set of registers
  - **Status** registers: processor reads to check device state
    - A keyboard sets a status register bit to indicate data is ready
  - **Data** registers: used for the exchange of data
    - When a key is typed, a keyboard places the corresponding ASCII code in a data register for the processor to read
  - **Control** registers: processor writes to change how the I/O device operates
    - A keyboard may have a backlight that the processor can turn off to save power when the computer is idle
    - The processor may change the operating speed of a serial communications port



LC-3 ID

LC-3 Keyboard Input

- Registers:
  - **K**eyboard **S**tatus **R**egister (**KBSR**) address = 0xFE00
    - KBSR[15] is 1 if a new character is available, 0 otherwise (MSB)
  - **K**eyboard **D**ata **R**egister (**KBDR**) address = 0xFE02
    - KBDR[7:0] contains the ASCII code of the last character typed
    - Reading KBDR automatically resets bit 15 of the KBSR (set to 0)
- Usage:
  - To check if a character is available, read KBSR and test if bit 15 is 1
    - Is the value negative?
  - When a character is available, read it from KBDR



(Data register never be empty, it always contain the most recent value, only meaningful when status register indicates there is a new character (1))

Keyboard Input Programming

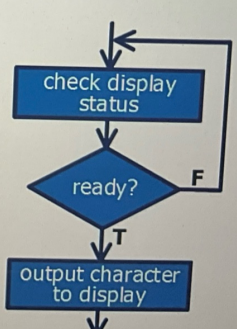
```
; getchar
; Gets (waits for) a character from the keyboard
; Assumes: Nothing
; Returns: R0 = character
getchar
    ST    R1, getchar_R1        ; context save
    ST    R2, getchar_R2
    LD    R1, IO_BASE           ; get I/O base address
getchar_wait
    LDR   R2, R1, #0            ; read KBSR (check keyboard status)
    BRzp  getchar_wait          ; wait until keyboard has data
    LDR   R0, R1, #2            ; read char from KBDR (read char)
getchar_exit
    LD    R2, getchar_R2        ; context restore
    LD    R1, getchar_R1
    RET

getchar_R1    .BLKW  1
getchar_R2    .BLKW  1
IO_BASE       .FILL  xFE00      ; start of memory-mapped I/O
```

(KBSR)

LC-3 Character Display Output

- Registers:
  - Display Status Register (DSR) address = 0xFE04
    - DSR[15] is 1 if the display is ready, 0 otherwise
  - Display Data Register (DDR) address = 0xFE06
    - The character (ASCII code) written to the DDR will be displayed on the screen
- Usage:
  - To check if the display is ready to accept a character, read DSR and check if bit 15 is 1
  - When the display is ready, write the character to DDR



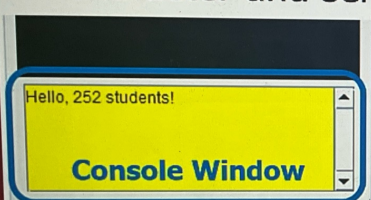
Display Output Programming

```
; putchar
; Prints a character to the console output
; Assumes: R0 = character to print
; Returns: Nothing
putchar
    ST    R1, putchar_R1      ; context save
    ST    R2, putchar_R2
    LD    R1, IO_BASE         ; get I/O base address
putchar_wait
    LDR   R2, R1, #4          ; read DSR (check display status)
    BRzp  putchar_wait        ; wait until display is ready
    STR   R0, R1, #6          ; write char to DDR (output char)
putchar_exit
    LD    R2, putchar_R2      ; context restore
    LD    R1, putchar_R1
    RET

putchar_R1    .BLKW  1
putchar_R2    .BLKW  1
IO_BASE       .FILL  xFE00    ; start of memory-mapped I/O
```

Working With I/O In PennSim

- Test program repeatedly gets a new keyboard character and sends that character to the display



☐	x01FF	x0000	.FILL x0000
☐	x0200 TEST	x480E	JSR GETCHAR
☐	x0201	x4801	JSR PUTCHAR
☑	x0202	x0FFD	BR TEST
☐	x0203 PUTCHAR	x3208	ST R1, PUTCHAR_R1
☐	x0204	x3408	ST R2, PUTCHAR_R2
☐	x0205	x2208	LD R1, IO_BASE
☐	x0206 PUTCHAR_WAIT	x6444	LDR R2, R1, #4
☐	x0207	x060A	BRzp GETCHAR_WAIT

- Characters are displayed in the console window
- The console window must be selected to provide keyboard input to simulated LC-3 processor
  - It turns yellow when it is selected!

LDI/STI versus LD,LDR / LD,STR

- The book uses LDI and STI for its examples; this video instead uses LD+LDR and LD+STR
  - LDI is equivalent to LD followed by LDR
  - STI is equivalent to LD followed by STR
- Using LDI in a polling loop means two memory reads for each iteration instead of one
  - Remember – memory accesses are actually very slow…
  - Why read memory repeatedly to get the status register address?
    - We only read the IO_BASE address once, then use LDR/STR instructions with the appropriate offsets
- Most modern processors do not have LDI/STI
  - Performance issues outweigh any benefits
  - Included because the idea of indirect access is important