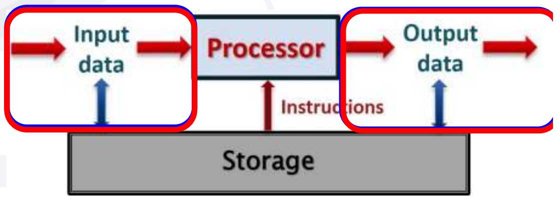
ENS1161 Computer Fundamentals
Module 8
I/O software and modes

Moving forward..

- Last module:
  - Basics of I/O interfacing
  - Types of I/O devices and the operation
- Focus of this module:
  - I/O Software and communication
  - I/O modes



Module Objectives

On completion of this module, students should be able to:

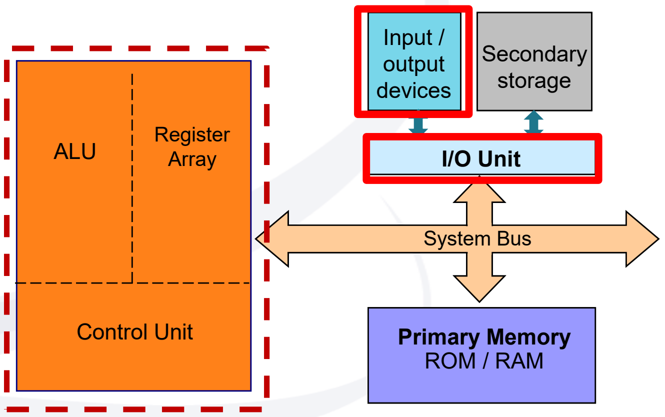
- Explain the basic principles of I/O interfacing and the role of I/O software and protocols.
- List and describe the roles of the different levels of I/O interfacing software, particularly the BIOS
- List and describe the principles of operation of I/O modes.
- Explain what is I/O mapping and describe the 2 main I/O mapping methods.
- Describe the principles of operation of a PC interrupt system.

Introduction

- **Module Scope**
  - I/O Interface software
  - Role of operating system, BIOS and device drivers
  - I/O modes
    - Focus on interrupts
  - Memory mapped vs isolated I/O
  - Examples of I/O interfacing

Basic Components of a Computer *(recap – Module 2)*

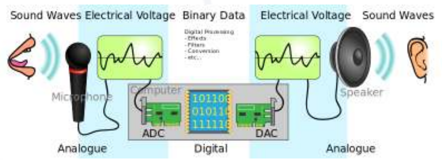
- Every computer contains the same basic components:
  - Arithmetic logic unit (ALU)
  - Register array
  - Control unit
  - Memory
  - Input/Output (I/O) unit
  - System Bus



'Processor'

I/O Devices *(recap – Module 7)*

- There are a wide variety of input and output devices
  - Keyboard, mouse, microphone, scanner, display, printers, displays, speakers, etc.
- Most input devices translate a variety of analogue physical inputs to digital form for use by a processor
  - E.g. key presses, movements of mouse, voice commands, images
- Output devices translate binary data into a variety of physical output forms
  - E.g. text, images, sound, etc.
- Some devices are both input and output devices
  - transfer binary data to and from processor
  - E.g. secondary storage, network devices



Milesjpool, 2016
https://commons.wikimedia.org/wiki/File:CPT-Sound-ADC-DAC.svg

I/O Interfacing *(recap – Module 7)*

- *Interfacing*: connecting devices together so that they can share information
- An interface includes:
  - the physical connection (the hardware)
    - physical dimensions, pinning, voltages
      - *Broadly covered in Module 7*
  - a set of rules or procedures governing the transfer of information over that connection (the software/algorithm)
    - I/O modes, protocols, data format, software interface, etc.
      - ***Main focus of this module***

I/O Communication *(recap – Module 7)*

- The processor must communicate with peripheral I/O devices
- Information transferred:
  - ***Data***
    - usually encoded in some suitable coding system
      - e.g. ASCII, PDL, scan codes, etc.
  - ***Control*** information
    - commands from the processor
    - requests for service from peripheral devices
    - control codes from the processor
    - status codes from I/O devices

I/O Communication Terminology

- **I/O (input/output)**
  - The transmission of data from one device to another,
  - The source is referred to as the ***sender*** and the destination the ***receiver***
  - Information flow not just in this one direction
  - Control information will flow in both directions for ***handshaking***
- **Handshaking**
  - is used to coordinate the I/O process
    - e.g. a DAV (*data available*) signal sent to inform receiver that data is available
    - Receiver accepts the data
    - Receiver sends a DACK (*data acknowledge*) signal to inform sender that data was received OK

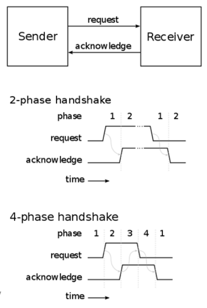


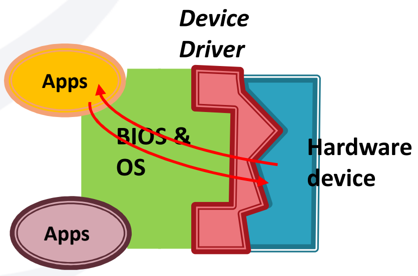
Image: Fragapanagos, 2017
https://commons.wikimedia.org/wiki/File:2_and_4_phase_handshakes.svg

I/O interfacing software / protocols

- Anything that the processor does is based on programs
  - The instructions being executed
- Includes interfacing with I/O devices
  - Even reading a keyboard, moving a mouse cursor
- I/O interfacing software is required:
  - To implement the communication protocols between processor and device
  - To control the transfer of information
- The programs to 'service' these devices are usually quite small and need to be fast
  - Called *procedures*, *routines*, *functions*

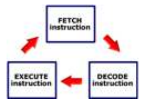
Levels of I/O interfacing software

- **Operating System**
  - Software that controls the operations of the whole system
    - *Covered in more detail in Module 9*
- **BIOS *(Basic Input/Output System)***
  - collection of routines to interact with I/O devices at a low level
    - Normally stored as firmware
- **Device Drivers**
  - Device specific data and routines
    - Often created by the manufacturers
    - Know how the particular device works and what it understands
- ***Devices***
  - Devices are normally combination of hardware and software
  - Most have their own processor *(controller)* and programs *(firmware)*



BIOS (Basic Input Output System)

- Stored on ROM on motherboard (firmware)
- Main functions:
  - **Boot-up Sequence**
    - Initial power-on / reset sequence of a computer
      - Processors will power-on with a specific value in the *Program Counter* (PC)
      - The circuitry is designed such that the first instruction loaded is from the BIOS
    - Power On Self-Test (POST)
      - Initial code sequence that tests if basic components are working correctly (based on configuration)
    - Looking for and loading operating system (OS)
  - **BIOS drivers / routines**
    - The low-level drivers and access routines for basic I/O components for:
      - The transfer of data between the CPU and peripheral devices
      - Interrupt handling for hardware interrupts



BIOS (Basic Input Output System)

- BIOS configuration software
  - Common additional function
  - Software in the ROM that allows base configuration of the system hardware
    - E.g. type of drives, drive to find OS, bus and memory configuration
    - May also have simple diagnostic tests
  - Settings are often stored in low power RAM
    - RAM connected to battery to keep information
    - In older systems, in chip with *real-time clock* (RTC)
    - The low power RAM is based on CMOS technology
      - CMOS = Complementary Metal Oxide Semiconductor
    - Reason why it is often called the *'CMOS Setup'*

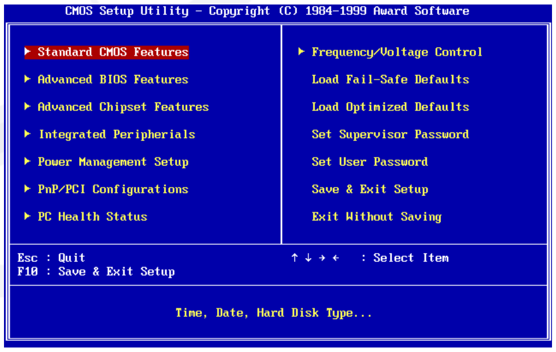
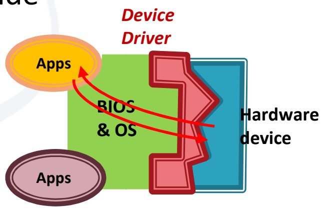

```
CMOS Setup Utility - Copyright (C) 1984-1999 Award Software

▸ Standard CMOS Features          ▸ Frequency/Voltage Control
▸ Advanced BIOS Features            Load Fail-Safe Defaults
▸ Advanced Chipset Features         Load Optimized Defaults
▸ Integrated Peripherals            Set Supervisor Password
▸ Power Management Setup            Set User Password
▸ PnP/PCI Configurations            Save & Exit Setup
▸ PC Health Status                  Exit Without Saving

Esc : Quit                         ↑ ↓ → ←   : Select Item
F10 : Save & Exit Setup

                Time, Date, Hard Disk Type...
```

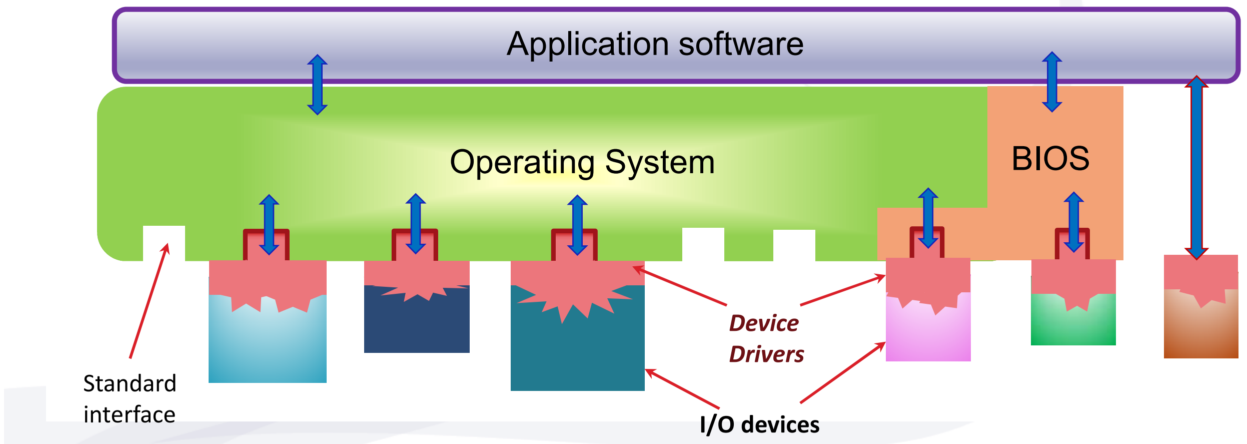
Image: Public domain
https://commons.wikimedia.org/wiki/
File:Award_BIOS_setup_utility.png

I/O Interfacing Software

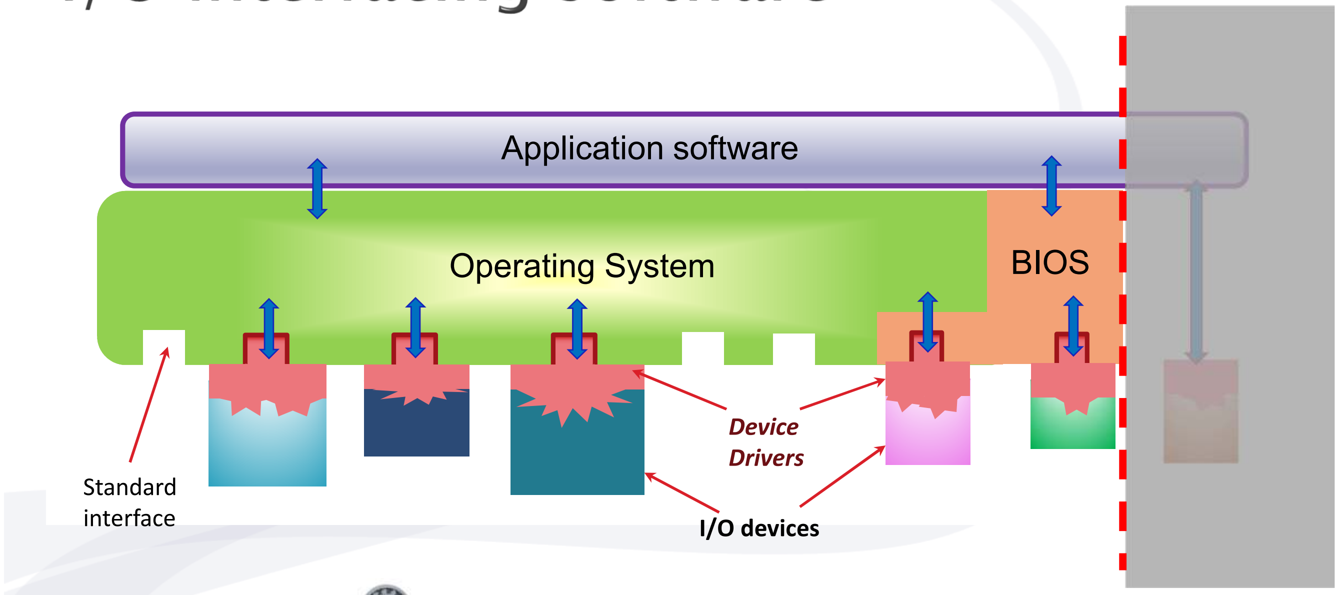
- **Device Drivers**
  - Specific subroutines or interrupt service routines
  - Provide the software 'face' of the interface
    - programmers need not know the physical details of the interface
    - only need to know how to invoke the functions of the software driver
  - Commonly included as part of standard interfaces in computer systems
  - Allows one company to provide hardware and another to provide software
    - No need for detailed communication between them
    - Only agreement upon a standard to use



I/O interfacing software

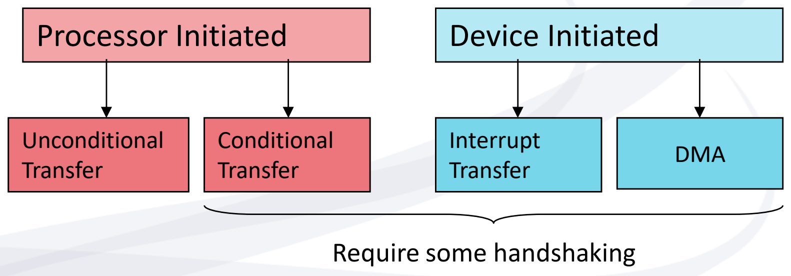


I/O interfacing software



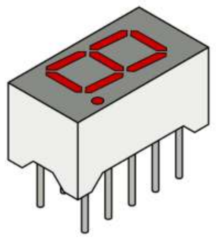
I/O Modes

- There are 4 main ways (*modes*) I/O transfers can be initiated and controlled in a computer system:



Unconditional I/O Transfer

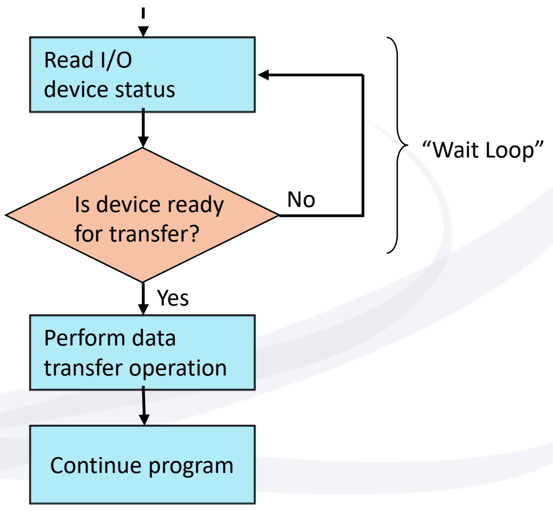
- Basic assumption: the input or output devices are ***always*** ready to send data to, or accept data from, the processor
  - no control signals need to be exchanged
  - data are simply written to the output device or read from the input device whenever desired
- E.g. a simple segmented LED display
  - LED display is connected to an output port
  - any data bytes sent from the processor to the port will then immediately appear on the bank of LEDs



Conditional (Polled) I/O

- Processor checks if the peripheral device is ready for communication before the actual transfer takes place
  - Transfer can be to or from the processor
- Called ***polled I/O***
  - Because the device must be polled continually
  - Polling: check if device ready for data / has data
- Requires handshaking
  - the control signals needed for handshaking will depend on the device involved

Processor Initiated Conditional (Polled) I/O


Read I/O device status
Is device ready for transfer?
No
Yes
"Wait Loop"
Perform data transfer operation
Continue program

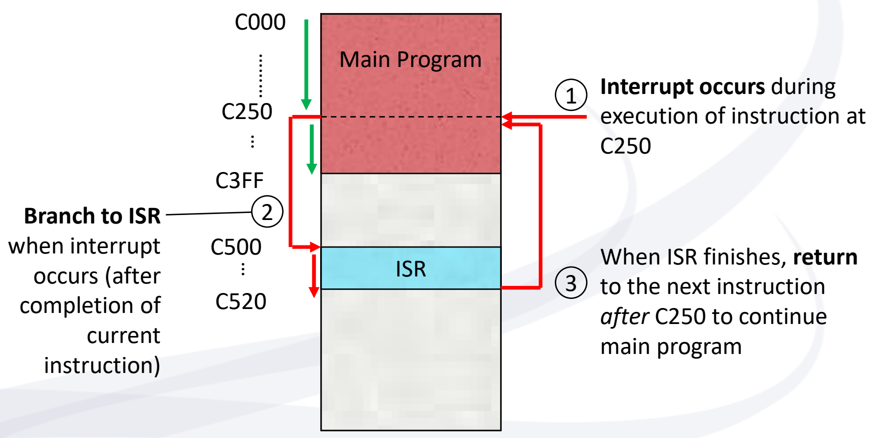
Processor Initiated Conditional (Polled) I/O

- Major disadvantage of polled I/O transfers
  - the processor has to ***wait*** for the I/O device
  - processor wastes time reading and testing the status of the I/O device
    - especially for slow I/O devices
- Processor could be performing other tasks while waiting
  - Unless the processor is dedicated to this task
- *What better method?*

Device Initiated I/O – Interrupts

- More efficient use of the processors time
  - does not have to repeatedly check if device is ready for transfer
- Processor is free to perform other tasks
- When the I/O device is ready:
  - Device sends a signal to one of the processor's interrupt inputs
  - Processor will suspend execution of its current program
    - after completing current instruction
  - Processor runs a special ***interrupt service routine*** (ISR)
    - process (program) containing instructions to *service* the interrupting device
      - E.g. transfer data to or from the device
  - Processor will resume execution of the interrupted program once ISR completed

Device Initiated I/O – Interrupts

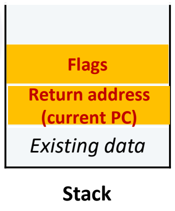

C000
C250
C3FF
C500
C520
Main Program
ISR
1 **Interrupt occurs** during execution of instruction at C250
2 **Branch to ISR** when interrupt occurs (after completion of current instruction)
3 When ISR finishes, **return** to the next instruction *after* C250 to continue main program

Device Initiated I/O – Interrupts

- Interrupt operation is very useful
- Especially need to interface with several ***asynchronous*** I/O devices
  - cannot predict when such devices will be ready to send or receive information
  - so programmed unconditional I/O transfers cannot be used
- Number of considerations that must be dealt with

Device Initiated I/O – Interrupts

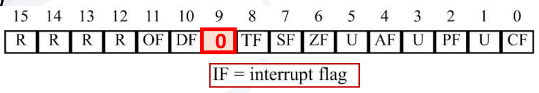
- **Return Address**
  - Where does processor return to after executing the ISR?
    - An interrupt can occur at any time
    - No way of knowing which instruction processor will be processing at the time
  - Solution: Automatically save the program counter before branching to the ISR
    - Intel x86 family processors
      - *Push* PC and Flags register contents onto the stack
  - *Return from interrupt* instruction is executed at the end
    - retrieves these values back off the stack (*pop*)
  - Similar to *subroutine call*
    - *Refer to Module 5 : stack operation, subroutines*



Device Initiated I/O – Interrupts

- **Disabling Interrupts**
  - What if an interrupt occurs while the processor is executing code that requires continuous processing?
    - e.g. in a timing loop or already communicating with another I/O device
  - Method for disabling interrupt operation under program control
    - usually via an interrupt mask (or enable) flag (I)
  - The processor will ignore any standard interrupt requests if
    - mask flag is set high (*block*) or enable flag low (*off*)
    - E.g. CLI instruction on x86
  - Called ***maskable interrupt request*** signals
    - e.g. INTR on the x86

8088 flags register

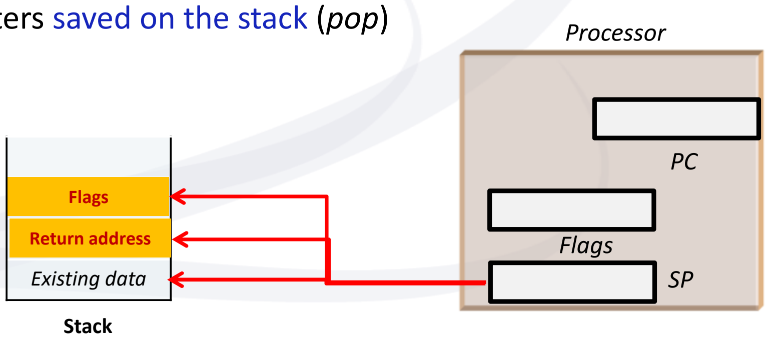


Device Initiated I/O – Interrupts

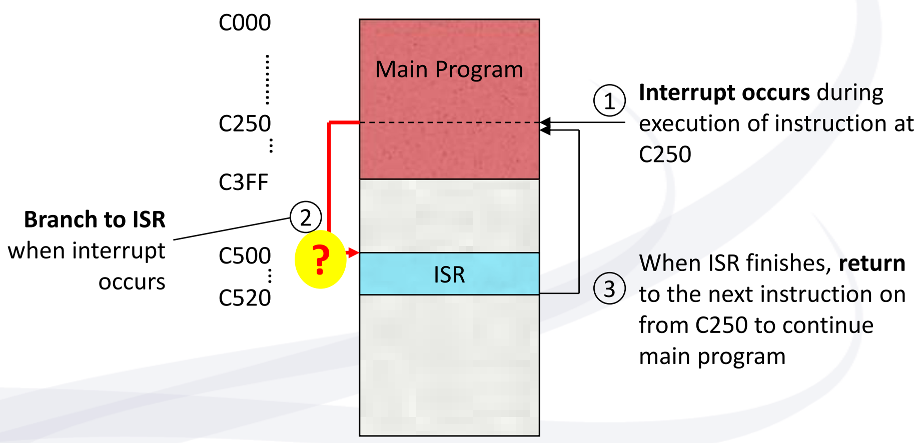
- **Types of Interrupt Inputs**
  - Many processors have 2 types of interrupt
    - ***maskable*** (e.g. $\overline{\text{INTR}}$ – Intel x86)
    - ***nonmaskable*** (e.g. $\overline{\text{NMI}}$ – Intel x86)
  - Maskable interrupt
    - ignored if the interrupt is masked
    - e.g. CLI clears I flag (x86)
  - Non-maskable interrupt signal
    - not affected by this flag
    - the processor will always respond to it (cannot be disabled)
    - has priority over the maskable interrupt input

Device Initiated I/O – Interrupts

- Return from Interrupt instruction
  - the final instruction in the ISR (different to normal return)
  - will return the processor to the program it was executing at the time the interrupt occurred
  - will restore the registers saved on the stack (*pop*)



Device Initiated I/O – Interrupts


C000
C250
C3FF
C500
C520
Main Program
ISR
?
1 **Interrupt occurs** during execution of instruction at C250
2 **Branch to ISR** when interrupt occurs
3 When ISR finishes, **return** to the next instruction on from C250 to continue main program

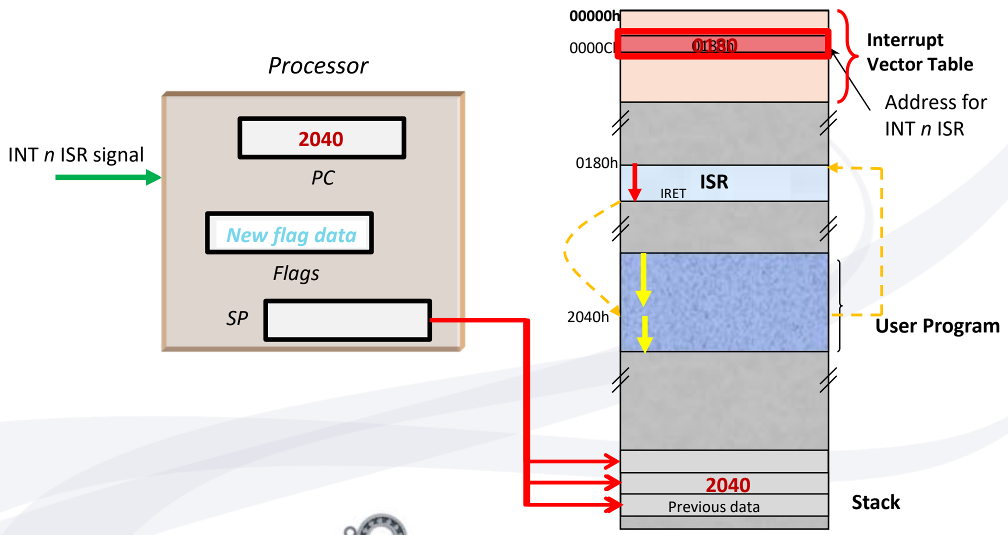
Device Initiated I/O – Interrupts

- **Interrupt Vectors**
  - How does the processor know address of the ISR for an interrupt?
  - The processor will obtain an ***interrupt vector*** from a fixed location in memory
    - Pointer to (*address of*) first instruction of the ISR
  - Interrupt vector gets loaded into the program counter (PC)
  - Interrupt vectors normally located in a fixed location in memory – the **Interrupt Vector Table** (IVT)
    - 'Address book' for ISRs

Interrupt Vector Table

Address of ISR 1
Address of ISR 2
Address of ISR 3
…..
Address of ISR *n*

Device Initiated I/O – Interrupts



- User program runs
- Interrupt signal received
- Current instruction completed then program stopped
- PC contents pushed to stack
- Flags register contents pushed onto stack
- Processor reads IVT entry for interrupt
- ISR address loaded to PC
- ISR runs (may change flags)
- Interrupt Return reached
- Flags register contents popped from stack
- Saved PC value popped from stack
- User program continues from where it was interrupted (instruction fetch)

I/O modes – device ↔ memory

- So far only considered data transfer operations between I/O devices and the processor
- Also many situations where a large amount of data needs to be quickly transferred between an I/O device and memory
  - E.g. the transfer of large blocks of data between a disk drive and RAM
- 2 basic ways to transfer large blocks of data between RAM and peripheral I/O devices:
  - ***programmed transfer*** or
  - ***direct memory access*** (DMA)

Programmed Transfer

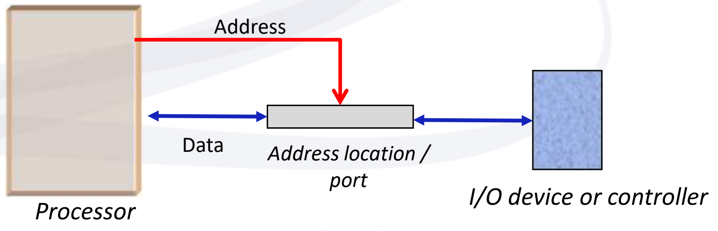
- Transfer is carried out under program control
- Processor executes a sequence of instructions that transfer data between RAM and I/O ***through*** the processor
  - Data must first be loaded into the processor
    - usually one of the registers
  - Data then written to its destination from processor
- Inefficient for large blocks of data
  - the data must be first read into the processor and then written out
- Can be speeded up if data could be transferred ***directly*** between the peripheral device and RAM

Direct Memory Access I/O Transfer

- Data flows directly between I/O devices and memory without involving the processor at all
- Transfer is controlled by a DMA controller (*DMAC*)
  - temporarily takes control of the address and control buses to carry out the transfer
  - supplies the RAM with appropriate addresses and R/$\overline{\text{W}}$ signals
  - exchanges handshaking signals with peripheral devices as required
- processor supplies the DMAC with parameters for each data transfer
  - processor then free to continue other operations
  - some form of *bus mastering* is required to co-ordinate the sharing of the system bus between the DMAC and the processor

I/O communication interface

- Processors communicate using the system bus
  - Instructions set addresses on the address bus
  - Then read from or write to data bus
  - Only way for processor send / receive I/O data and handshaking signals
- Devices are 'mapped' to certain addresses – called ports
  - Devices are connected (wired) in such a way that writing to / reading from a particular address is actually writing / reading the device

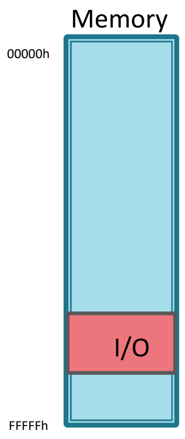


I/O Mapping

- 2 main methods used to interface I/O devices to processors
- Memory-mapped I/O
  - the same instructions that are used to access memory are also used to access the I/O devices
  - I/O port is treated the same as any other memory location
- I/O-mapped I/O
  - specific instructions are used to transfer data between the processor's accumulator and the I/O device
- Both techniques are commonly used

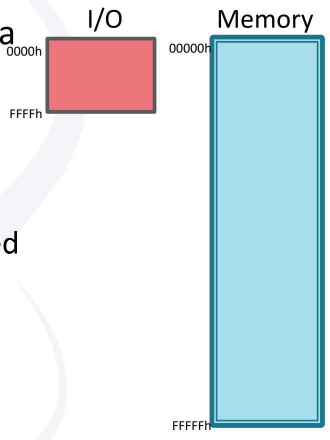
Memory Mapped I/O

- A memory mapped I/O device is treated as a standard memory location in a processors addressable space
- Same instructions used as in memory access
- Main advantages:
  - no additional instructions are needed
  - no additional control circuitry is needed
    - reduces the complexity of the decoding circuitry
- Main disadvantage:
  - part of addressable space is used as the I/O space
    - reduces the amount of memory available to applications



Isolated or I/O Mapped I/O

- The I/O locations are ***isolated*** from the memory system in a separate I/O address space
- I/O devices are accessed through separate ***port addresses***
- The main advantage of I/O mapped I/O :
  - memory can be expanded to the full size address range allowed
- A disadvantage:
  - data transferred between I/O and the processor must be accessed using specific instructions
  - separate control signals used to access the I/O space
    - E.g. $\overline{\text{IOR}}$ and $\overline{\text{IOW}}$ to read from or write to an I/O device respectively (in *x86*)



Interfacing on PCs

- **I/O Ports**
  - 8088/86 processors able to transfer data to and from I/O ports
    - to send data out from the processor or to input data into it
    - 2 specific instructions for port-based data transfers
      - IN and OUT
    - IN instruction receives data from a specified port into the accumulator (AL or AX)
    - OUT instruction sends data from the accumulator to a specified port
    - Original IBM PC systems: only 8-bit I/O port *addresses*
      - 256 separate addresses, 00h – FFh
    - Newer machines : extended to 16-bit addresses
      - allows to 65,536 different port addresses, 0000h – FFFFh

Interfacing on PCs

- **Some Basic Ports on a legacy PC**
  - Programmable Interrupt Controller, ports 20 – 23h

20h	Command Register
21h	Mask Register

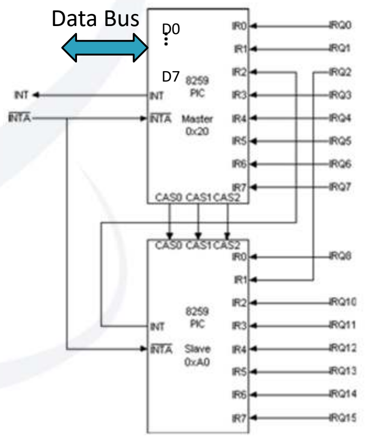
  - Programmable Peripheral Interface, ports 60 – 63

60h	In/Out or bidirectional Port A (PA)
61h	In/Out Port B (PB)
62h	In/Out or bit set/reset Port C (PC)
63h	Control word register

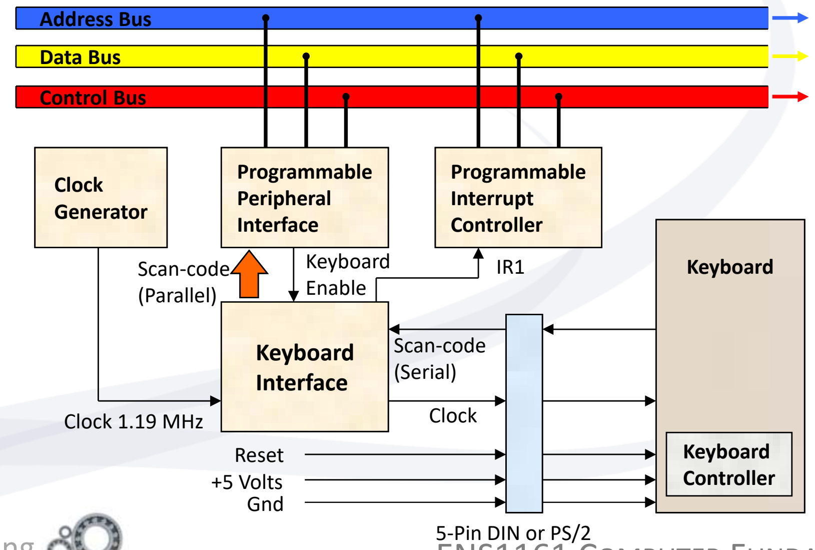
  - Direct Memory Access Controller, ports 80h – 83h
  - Others: for disk drives, keyboard, video, printers, serial devices, etc.

PC Interrupt System

- **The 8259 Programmable Interrupt Controller**
  - **Enables Multiple Interrupts**
    - Only one standard and one NMI interrupt input for x86 processors
    - Capability expanded using the 8259 programmable interrupt controller (PIC)
    - PIC adds eight vectored, priority encoded interrupts
    - By using additional slave devices, can accept up to 64 interrupt requests



Example: Legacy PC Keyboard Interface

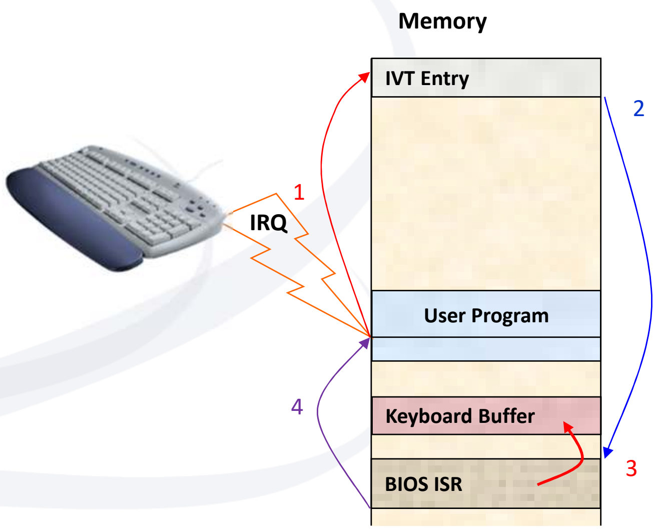


Example: Legacy PC Keyboard Interface

- The keyboard contains its own controller with an embedded program that:
  - detects key-presses of various types
  - generates appropriate key scan codes
  - transmits scan codes serially to the motherboard
- Motherboard has a similar controller
  - Receives the scan codes in serial form
    - Includes handshaking with keyboard controller
  - Sends the scan code in parallel to PPI and sends interrupt signal to PIC (IR1)
- Interrupt triggers a BIOS ISR that:
  - fetches the scan code from port A of the PPI
  - converts it to ASCII (if possible)
  - stores both values in the keyboard buffer (BIOS temporary space)
- Applications retrieve ASCII / scan code using another BIOS function

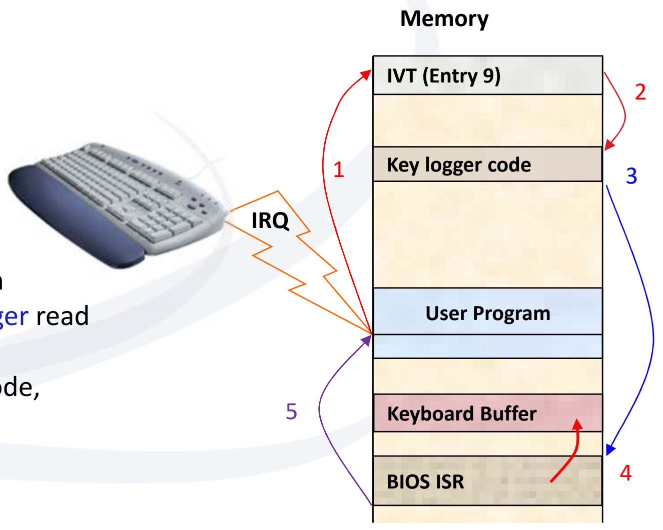
Example: Legacy PC Keyboard Interface

1. Keyboard input interrupt stops program
2. Interrupt Vector table entry for keyboard interrupt read to Program Counter
3. BIOS ISR runs
   - reads the scan code
   - Find ASCII character
   - Stores scan code and ASCII char in keyboard buffer
4. User program continues



Interrupt Driven Code Execution

- **Interrupt Hooking**
  - E.g. key logger
  - Copies IVT entry and puts jump to BIOS ISR as last command
  - Puts its own address in IVT



1. Keyboard input interrupt stops program
2. Interrupt Vector table entry for key logger read to Program Counter
3. Key logger program runs: stores scan code, ends with jump to BIOS ISR
4. BIOS ISR runs as normal
5. User program continues

Module Objectives

On completion of this module, students should be able to:

- Explain the basic principles of I/O interfacing and the role of I/O software and protocols.
- List and describe the roles of the different levels of I/O interfacing software, particularly the BIOS
- List and describe the principles of operation of I/O modes.
- Explain what is I/O mapping and describe the 2 main I/O mapping methods.
- Describe the principles of operation of a PC interrupt system.