Computer Interfaces

David Vernon
Computer Interfaces

- The goal of this course is to introduce hardware and software design techniques and issues for interfacing computers and peripheral devices
Computer Interfaces

◆ Course Contents

- Digital data communication standards – serial communications
  - Devices
  - RS232
  - RS422
  - Handshaking
  - Implementation of RS232 on a PC

- Universal Serial Bus (USB)
  - USB standards
  - Types and elements of USB transfers
  - Development procedure for USB applications
Computer Interfaces

Course Contents

- Parallel Communications
  - General Purpose Interface Bus (GPIB)
  - GPIB signals and lines
  - Handshaking and interface management
  - Implementation of a GPIB on a PC
Computer Interfaces

Course Contents

- Digital and Analogue Interfacing
  - Digital Interfacing
    - Digital I/O ports
    - Interfacing external signals to digital I/O ports
    - Optical isolation
  - Analogue Interfacing
    - Revision of A/D and D/A conversion techniques
    - Multiplexing
    - Analogue I/O cards
    - Data acquisition and control using a PC
Computer Interfaces

- Course Texts:

  *PC Interfacing, Communications and Windows Programming*, William Buchanan, Addison Wesley

  *Microcomputer Interfacing and Applications*, M. A. Mustafa, Newnes
Computer Interfaces

- Course Loading: 0.5 Module

1 Lecture each week  
(weeks 1, 2, 3, ...)

1 Tutorial every second week  
(weeks 3, 5, 7, 10, 12, 14)

1 Lab demonstration every fourth week  
(weeks 4, 11, 15)

1 Assignment, due on the last day of week 15 (22\textsuperscript{nd} Dec.)
Computer Interfaces

Motivation
Computer Interfaces
Computer Interfaces

◆ The need for computer interfacing
  - Advanced control applications need flexible processing power, i.e. computers
  - Control data has to be input and output
    - Input from sensors (speed, acceleration, temperature, ..)
    - Output to actuators (motors, switches, valves, ..)
  - Examples:
    - Robotics
    - Industrial process control
  - Advantages of using computers for Data Acquisition & Control
    - High speed
    - Programming flexibility (compared with hard-wired logic)
    - Mass storage of data
    - Data analysis and visualization
    - Low cost (relatively)
Computer Interfaces

Issues:

- Input vs output (direction)
  - Communication between devices
  - Data acquisition
  - Device control
- Digital vs Analogue data (modality)
  - Signal levels (may be a need for signal conditioning)
  - Analogue to digital conversion (ADC)
  - Digital to analogue conversion (DAC)
- Serial vs parallel (mechanism)
  - Speed, distance, number of required lines, standard I/F
- Polled vs interrupt driven (mechanism)
  - Simplicity, processor efficiency
Computer Interfaces

Microprocessor Architecture
Microprocessor Architecture

History

- Early ’70s: Intel 4004
  - 4 bit of data (nibble)
  - 2000 transistors
  - 46 instructions
  - 4kB program code
  - 1kB data

- 1974 - : 8008, 8080, 8085
  - 8008: 14-bit address space (16kB memory)
  - 8080: 16-bit address space (64kB memory)
Microprocessor Architecture

History

- Early 80s: 8086 – Major revolution in processing power
  - 16-bit data bus
  - 20 bit address bus (1MB memory)
  - PC & PC-XT
  - 8088: Multiplexed 8-bit data bus
  - 80286: enhance version (PC AT)
Microprocessor Architecture

History

- 1985: Intel 80386
  - 32 bit data bus
  - 32-bit address bus (4GB memory)
- 1986: Intel 80486
  - Memory cache
  - On-chip maths co-processor
Microprocessor Architecture

History

1990s: Pentium (P-5)
- 64-bit superscalar architecture
- MISD instruction pipeline (can execute more than one instruction at a time)
- 64-bit databus
- 32-bit address bus

2000s: Pentium II (P-6)
- Up to 4 processors on the same bus
- Single bit error detection and correction on the data-bus
- Multiple bit error detection
Microprocessor Architecture

8088

- 8288 bus controller generates signals based on the 8088 status lines S0, S1, S2:
  - Main control signals:
    - IOR (I/O Read): processor is reading from the content of the address which is on the I/O bus
    - IOW (I/O Write): processor is writing the contents of the data bus to the address on the I/O bus
    - MEMR (memory read): processor is reading from the contents of the address which is on the address bus
    - MEMW (memory write): processor is writing the contents of the data bus to the address which is on the address bus
    - INTA (interrupt acknowledge): used by the processor to acknowledge an interrupt (sent via the 8259 interrupt controller)
Microprocessor Architecture

◆ 8088
  ❏ Two address spaces:
    ▪ Memory space
    ▪ I/O space
  ❏ Registers
    ▪ 4 general purpose registers
      AX Accumulator (all I/O operations & some arithmetic)
      BX base register (can be used as an address register)
      CX count register (e.g. loops)
      DX data register (some I/O and when multiplying & dividing)
    ▪ 4 addressing registers
      SI source index (extended addressing commands)
      DI destination index (used in some addressing modes)
      BP base pointer
      SP stack pointer
Microprocessor Architecture

8088

Registers

- Status registers
  - IP instruction pointer (contains address of next instruction to be executed)
  - F flag register (with bits set depending on conditions or results of operations, e.g. is the result negative)

- Segment registers (16 bits – 64kB)
  - CS register: code segment; defines the memory location where the code/instructions are stored
  - DS register: data segment; defined where the data from the program will be stored
  - SS register: stack segment; location of the stack
  - ES register: extra segment

- All addresses are defined with respect to the segment registers
- 8086 has a segmented memory architecture
Microprocessor Architecture
Microprocessor Architecture

◆ Memory segmentation

- 8086 has a 20-bit address space (1MB)
- But can only directly address 64kB (16-bit address space)
- I.e. it addresses the 1MB in chunks or segments
- A segmented memory address location is identified with a segment and an offset address (this is the logical address)

\[
\text{segment:offset} \quad 16 \text{ bits : 16 bits}
\]

- The actual physical address is calculated by shifting the segment address 4 bits to the left and adding the offset!
- What’s the actual physical address in the following example?
Microprocessor Architecture

Segment (2F84):
Offset (0532):
Actual address:

0010 1111 1000 0100 0000
0000 0101 0011 0010
0010 1111 1101 0111 0010
Microprocessor Architecture

- Memory segmentation
  - In C the address 1234:9876h is specified as 0x12349876
  - Near and far pointers
    - A near pointer is a 16-bit pointer (64kB of data)
    - A far pointer is a 20-bit pointer (1MB)

```c
char far *ptr; /* declare a far pointer */
ptr = (char far *) 0x1234567; /* initialize a far pointer */
```
Memory mapped I/O vs. Isolated I/O

- Memory mapped I/O
  - Devices are mapped into the physical memory space
  - Given real addresses on the address bus

- Isolated I/O
  - Devices are mapped into a special isolated memory space
  - Accessed via ports which act as a buffer between the processor and the device
  - Accessed using the IN and OUT instructions (there are C equivalents)
  - Isolated I/O uses 16-bit addressing from 0000h to FFFFh
Microprocessor Architecture
Microprocessor Architecture
Microprocessor Architecture

◆ 80x86 bus controller signals
  □ R/W
    ▪ Low when data is being written
    ▪ High when data is being read
  □ M/IO
    o Low when selecting isolated memory
    o High when selecting normal memory space
Microprocessor Architecture

Typical PC Memory Map

- Extended memory
- Extended memory
- Video graphics
  - Text display
- Application programs
  - (640 KB)
- Interrupt vectors
  - BIOS

- FFFFFFFFh (4 GB)
- FFFFFFFFh (16 MB)
- FFFFFFFFh (1 MB)
- F0009FFFFh (640 KB)
- 00000600h
- 00000000h
## Typical isolated I/O Memory Map

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000h-01Fh</td>
<td>DMA controller</td>
</tr>
<tr>
<td>020h-021h</td>
<td>Programmable interrupt controller (PIC)</td>
</tr>
<tr>
<td>040h-05Fh</td>
<td>Counter/Timer</td>
</tr>
<tr>
<td>060h-07Fh</td>
<td>Digital I/O</td>
</tr>
<tr>
<td>080h-09Fh</td>
<td>DMA controller</td>
</tr>
<tr>
<td>0A0h-0BFh</td>
<td>NMI Reset</td>
</tr>
<tr>
<td>0C0h-0DFh</td>
<td>DMA controller</td>
</tr>
<tr>
<td>0E0h-0FFh</td>
<td>Math co-processor</td>
</tr>
<tr>
<td>170h-178h</td>
<td>Hard disk (secondary IDE drive or CD-ROM drive)</td>
</tr>
<tr>
<td>1F0h-1F8h</td>
<td>Hard disk (primary IDE drive)</td>
</tr>
<tr>
<td>200h-20Fh</td>
<td>Game I/O adapter</td>
</tr>
<tr>
<td>210h-217h</td>
<td>Expansion unit</td>
</tr>
<tr>
<td>278h-27Fh</td>
<td>Second parallel port (LPT2:)</td>
</tr>
<tr>
<td>2F8h-2FFh</td>
<td>Second serial port (COM2:)</td>
</tr>
<tr>
<td>300h-31Fh</td>
<td>Prototype card</td>
</tr>
<tr>
<td>378h-37Fh</td>
<td>Primary parallel port (LPT1:)</td>
</tr>
<tr>
<td>380h-3AFh</td>
<td>SDLC interface</td>
</tr>
<tr>
<td>3A0h-3AFh</td>
<td>Primary binary synchronous port</td>
</tr>
<tr>
<td>3B0h-3BFh</td>
<td>Graphics adapter</td>
</tr>
<tr>
<td>3C0h-3DFh</td>
<td>Graphics adapter</td>
</tr>
<tr>
<td>3F0h-3F7h</td>
<td>Floppy disk controller</td>
</tr>
<tr>
<td>3F8h-3FFh</td>
<td>Primary serial port (COM1:)</td>
</tr>
</tbody>
</table>
Microprocessor Architecture

◆ Isolated I/O
  □ Inputting a byte from an I/O port
    ▪ Borland C

      unsigned char value;
      value = inportb(PORTADDRESS);

      Prototyped in  dos.h

    ▪ Microsoft C++
      unsigned char value;
      value = _inp(PORTADDRESS);

      Prototyped in  conio.h
Microprocessor Architecture

◆ Isolated I/O
  □ Inputting a word from an I/O port
    ▪ Borland C

    unsigned int value;
    value = inport(PORTADDRESS);

    Prototyped in dos.h

    ▪ Microsoft C++
    unsigned int value;
    value = _inpw(PORTADDRESS);

    Prototyped in conio.h
Microprocessor Architecture

◆ Isolated I/O

- Outputting a byte from an I/O port
  - Borland C
    
    ```c
    unsigned char value;
    outportb(PORTADDRESS, value);
    ```
    
    Prototyped in dos.h

  - Microsoft C++
    
    ```c
    unsigned char value;
    _outp(PORTADDRESS, value);
    ```
    
    Prototyped in conio.h
Microprocessor Architecture

- **Isolated I/O**
  - Outputting a word from an I/O port
    - **Borland C**
      
      ```c
      unsigned int value;
      outport(PORTADDRESS, value);
      
      Prototyped in dos.h
      ```
    
    - **Microsoft C++**
      
      ```c
      unsigned int value;
      _outw(PORTADDRESS, value);
      
      Prototyped in conio.h
      ```
Serial Interfaces: RS-232

- Serial data transmission is used for digital communication between:
  - Sensors and computers
  - Computers and computers
  - Computers and peripheral devices (printer, stylus, mouse, ..)
  - One of the most widely used communication techniques to interface external equipment.

- Serial communication protocol: 1 bit at a time, sequentially.

- Parallel transmission: 1 word at a time (i.e. n bits in parallel).

- Advantages of serial transmission: very simple wiring.

- Transmission Characteristics of RS-232:
  - Maximum distance of 20 metres
  - Maximum bit rate 19600 bps

- Alternative serial communication standards:
  - RS-422 (up to 10 Mbps over distance of 1.2km)
  - USB-1 & USB-2
  - IEEE 1394 (Firewire / iLink)
Serial Interfaces: RS-232

◆ Electrical characteristics

- Logic 1: -3V to -25V; typically -12V
- Logic 0: +3V to +25V; typically +12V
- Any signal in the range -3V to +3V has an indeterminate logical state
- Quiescent or inactive state is -12V (i.e. logic 1)
Serial Interfaces: RS-232

◆ Connectors

- DB25S is a 25 pin connector with full RS-232 functionality
- The computer socket has a female outer casing with male connecting pins
- The terminating cable connector has a male outer casing with female connecting pins
Serial Interfaces: RS-232

1. Protective ground
2. Transmit Data
3. Receive Data
4. Request to Send
5. Clear to Send
6. Data Set Ready
7. Signal Ground
8. Carrier Detect
9. Positive DC Test
10. Negative DC Test
11. Secondary Carrier Detect
12. Secondary Clear to Send
13. Secondary Transmit Data
14. Transmit Clock DCE
15. Secondary Receive Data
16. Receiver Clock
17. Receiver dabit clock
18. Secondary Request to Send
19. Data Terminal Ready
20. Signal Quality Detector
21. Ring Indicator
22. Data Signal Rate Select
23. Transmit Clock
24. Busy
Serial Interfaces: RS-232

<table>
<thead>
<tr>
<th>Pin</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Tx</td>
</tr>
<tr>
<td>3</td>
<td>Rx</td>
</tr>
<tr>
<td>4</td>
<td>RTS</td>
</tr>
<tr>
<td>5</td>
<td>CTS</td>
</tr>
<tr>
<td>6</td>
<td>DSR</td>
</tr>
<tr>
<td>7</td>
<td>GND</td>
</tr>
<tr>
<td>20</td>
<td>DTR</td>
</tr>
</tbody>
</table>
Serial Interfaces: RS-232

- **Connectors**
  - DB9S is a 9 pin connector with reduced RS-232 functionality
  - The computer socket has a female outer casing with male connecting pins
  - The terminating cable connector has a male outer casing with female connecting pins
## Serial Interfaces: RS-232

### RS-232 Serial Communications

<table>
<thead>
<tr>
<th>DB9 MALE DTE</th>
<th>1</th>
<th>CD   (in)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>RXD (in)</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>TXD (out)</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>DTR (out)</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>SG</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>DSR (in)</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>RQS (out)</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>CTS (in)</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>R1</td>
</tr>
</tbody>
</table>
Serial Interfaces: RS-232

<table>
<thead>
<tr>
<th>Pin</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Rx</td>
</tr>
<tr>
<td>3</td>
<td>Tx</td>
</tr>
<tr>
<td>4</td>
<td>DTR</td>
</tr>
<tr>
<td>5</td>
<td>GND</td>
</tr>
<tr>
<td>6</td>
<td>DSR</td>
</tr>
<tr>
<td>7</td>
<td>RTS</td>
</tr>
<tr>
<td>8</td>
<td>CTS</td>
</tr>
</tbody>
</table>
Serial Interfaces: RS-232

◆ Connectors

- Most PCs use either a 9-pin connector for the primary (COM1:) serial port and a 25-pin for a secondary serial port (COM2:), or they use two 9-pin connectors.
- Note: the 25-pin parallel port (LPT1:) is a 25-pin female connector on the PC and male on the cable.
- The serial connector is male on the PC and female on the cable.
- 25-to-9 in adaptors are available.
Serial Interfaces: RS-232

- **Frame format**
  - RS-232 uses asynchronous communications
  - Start-stop format
  - Each character is transmitted one at a time
    - Delay between each character: inactive time
    - Inactive time: -12V logic level high
    - Each character is framed by a start bit (0), 7 data bits, 1 parity bit, 2 stop bits
    - ASCII coding is used
    - Parity is either even, odd, none
      - Parity bit is set to make the total number of either even or odd (or not checked)
Serial Interfaces: RS-232

ASCII character

Start bit

Parity bit

Stop bit(s)

‘A’ (100 0001)

0 1 0 0 0 0 0 0 1 1 1 1 1

0 b₀ b₁ b₂ b₃ b₄ b₅ b₆ P S₀ S₁
Serial Interfaces: RS-232

◆ Frame format

- Both the transmitter and the receiver need to be set to the same bit-time interval (baud rate)
- Since RS-232 is asynchronous, clock rates don’t have to be exactly synchronized
- There is an overhead in using asynchronous communication: the additional start and stop bits
- The advantages is that it makes communication very simple
Serial Interfaces: RS-232

Example

ASCII coding, even parity, 2 stop bits:

1111101000001011000001111111111111100000111111
110001100111101010011111111111

{inactive}11111 {start bit} 0 {'A'}100001 {parity bit} 0
{stop bits} 11 {start bit}0 {'p'}000111 {parity bit} 1
{stop bits}11 {inactive}11111111 {start bit}0
{'p'}0000111 {parity bit} 1 {stop bits}11 {inactive}11
{start bit}0 {'L'}0011001 {parity bit} 1 {stop bits}11

Message is ‘AppL’
Serial Interfaces: RS-232

- Parity
  - Simple form of error coding
  - A parity bit is added to transmitted data to make the number of 1s (in the data) either even (even parity) or odd (odd parity)
  - A single parity bit can only detect an odd number of errors (why?)
Serial Interfaces: RS-232

- **Baud Rate**
  - 11 bits required to send a single character (10 if one stop bits are used)
  - Bit rate (bits/sec): actual rate at which bits are transmitted
  - Baud rate: rate at which the signalling elements, used to represent bits, are transmitted
    - Since one signalling element encodes one bit, both rates are identical
    - However, baud rates will differ from bit rates in modem communication
  - Time period of each bit = 1/baud rate seconds
Serial Interfaces: RS-232

◆ Types of equipment

- **DTE** Data Terminal Equipment
  - Originally applied to CRT terminals or other input devices
  - Today, DTE mainly applies to a computer

- **DCE** Data Communication Equipment
  - Originally applied to modems or similar communications equipment
  - Still applies today

- A modem is a device that converts a digital signal (e.g. from an RS232 interface) to an analogue signal for transmission over a traditional telephone line (MODEM: MOdulator-DEModulator)
## Serial Interfaces: RS-232

<table>
<thead>
<tr>
<th>Pin</th>
<th>Name</th>
<th>Abbrev</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Frame Ground</td>
<td>FG</td>
<td>This ground normally connects the outer sheath of the cable to the earth ground</td>
</tr>
<tr>
<td>2</td>
<td>Transmit Data</td>
<td>TD</td>
<td>Data is sent from the DTE (computer or terminal) to a DCE via TD</td>
</tr>
<tr>
<td>3</td>
<td>Receive Data</td>
<td>RD</td>
<td>Data is sent from the DCE to a DTE via RD</td>
</tr>
<tr>
<td>4</td>
<td>Request to Send</td>
<td>RTS</td>
<td>DTE sets this active when it is ready to transmit data</td>
</tr>
<tr>
<td>5</td>
<td>Clear to Send</td>
<td>CTS</td>
<td>DCE sets this active to inform the DTE that it is ready to receive data</td>
</tr>
<tr>
<td>6</td>
<td>Data Set Ready</td>
<td>DSR</td>
<td>Signals that the DCE is ready to communicate</td>
</tr>
<tr>
<td>7</td>
<td>Signal Ground</td>
<td>SG</td>
<td>All signals are referenced to the signal ground</td>
</tr>
<tr>
<td>20</td>
<td>Data Terminal Ready</td>
<td>DTR</td>
<td>Signals that the DTE is ready to communicate. RTS and CTS have no effect if DTR is not asserted and connected to DSR</td>
</tr>
</tbody>
</table>
Serial Interfaces: RS-232

- Communication between two nodes
  - Handshaking
    - Hardware handshaking
      - RTS
      - CTS
      - DTR
      - DSR
    - Software handshaking
      - Sending special control characters X-OFF, X-On (ctrl-S, ctrl-Q)
    - No handshaking
      - If no handshaking is used then the receiver must be able to read the received characters before the transmitter sends more (otherwise the input buffer will be overwritten)
Serial Interfaces: RS-232

- Simple No Handshaking Connections
  - note the loop-backs
Simple No Handshaking Connections

- Advantage: no handshaking so fewer wires
- Disadvantage:
  - all handshaking has to be done by software (hence, more complex software)
  - The lack of the DTR means neither device knows whether the other device is powered up and ready for data transfer
- Typically used when the devices can operate at much faster speeds than the communication channel
Serial Interfaces: RS-232

- Software handshaking
  (Note that the software must recognize the X-ON and X-OFF characters and take the appropriate action)
Hardware handshaking

- To transmit:
  - Assert Transmitter RTS (high)
  - Wait until Transmitter CTS high
  - When receiver reads from its buffer (buffer empty), Receiver RTS goes high
  - Transmitter CTS, goes high
  - Transmitter sends data
Serial Interfaces: RS-232

- DTR and DSR are responsible for establishing the connection
- RTS and CTS are responsible for the data transfer
- Without an active DTR signal, the RTS and CTS signals have no effect
- Deactivating DTR or DSR breaks the connection
Serial Interfaces: RS-232

- Hardware handshaking
Serial Interfaces: RS-232

- Hardware handshaking
Serial Interfaces: RS-232

- DTE to DCE connections for handshaking
Serial Interfaces: RS-232

- Typical System Connections
  - Loop-back connections are used to test hardware (cf. program 15.1 in text)
  - Null modem connections are used for communication (cf. programs send and receive; 15.2 & 15.3 in text)
Serial Interfaces: RS-232

Types of error in serial communications

- Framing error: if the receiver has detected an invalid stop bit then the received serial character does not fit into the frame that the setup data format and the setup baud rate define. Thus the receiver has detected a framing error.

- Break error: if the reception line is at a logical low for a longer time than usual then the receiver assumes that the connection to the transmitter is broken. The transmitter usually drives the line to a logical high level as long as no data is being transferred.

- Overrun error: if data is arriving in the receiver faster than it is read from the receiver buffer register by the CPU, then a later received byte may overwrite the older data not yet read from the buffer.

- Parity error
Serial Devices: UART

- Universal Asynchronous Receiver Transmitter (8250)
  - UART transmits and receives characters using RS-232
  - 40 pin IC
    - Connection to the microprocessor is via D0-D7
    - Microprocessor **write**: asserts DOSTR and DOSTR (high & low, respectively)
    - Microprocessor **read**: asserts DISTR and DISTR (high & low, respectively)
Serial Devices: UART
Serial Devices: UART

- Universal Asynchronous Receiver Transmitter (8250)
  - Seven registers, selected using address lines A0, A1, A2
Serial Devices: UART

◆ Universal Asynchronous Receiver Transmitter (8250)

- Main input RS-232 handshaking lines:
  - RI: Ring Indicate
  - DSR: Data Set Ready
  - CTS: Clear To Send

- Main output RS-232 handshaking lines:
  - RTS: Ready to Send
  - DTR: Data Terminal Ready

- Serial Output
  - SOUT

- Serial Input
  - SIN
Serial Devices: UART

- **Universal Asynchronous Receiver Transmitter (8250)**
  - **Clock input:**
    - XTAL1: connect to a crystal to control the internal clock oscillator (typically 1.8432 MHz)
    - XTAL2: clock oscillator (typically 1.8432 MHz)
    - BAUDOT: Baud rate
      - (clock frequency / 16N; typically 9600 Baud)
  - **Hardware interrupts:**
    - INT
  - **Reset**
    - RESET: active low
Serial Devices: UART

◆ Programming RS-232
  ❑ Main registers used in RS-232:
    ▪ Line Control Register (LCR)
    ▪ Line Status Register (LSR)
    ▪ Transmit and Receive Buffer (TD/RD)
  ❑ Base Address
    ▪ primary port (COM1:) 3F8h
    ▪ Secondary port (COM1:) 2F8h
    ▪ Standard PC can support up to four COM ports
    ▪ The base addresses are set in the BIOS memory; address of each port stored at address locations:
      0040 : 0000 (COM1:)
      0040 : 0002 (COM2:)
      0040 : 0004 (COM3:)
      0040 : 0006 (COM4:)

Copyright © 2007 David Vernon (www.vernon.eu)
Serial Devices: UART

```c
#include <stdio.h>
#include <conio.h>
int main(void){
    int far *ptr; /* 20-bit pointer */
    ptr = (int far *) 0x0400000; /* 0040:0000 */

    clrscr();
    printf("COM1: %04x\n", *ptr);
    printf("COM2: %04x\n", *(ptr+1));
    printf("COM3: %04x\n", *(ptr+2));
    printf("COM4: %04x\n", *(ptr+3));
    return(0);
}
```
Serial Devices: UART

- Programming RS-232
  - Line Status Register (LSR)
    - Determines the status of the transmitter and receiver buffers
    - Read-only; all bits set by hardware

- Status bit S6 should be checked to see if the output buffer (TD/RD) is empty – i.e. data has been sent – before writing to the buffer
Serial Devices: UART

- Character to be transmitted
- TX buffer
- LSR
  - Test b₆ to determine if TX buffer empty
- Receiver
- RX buffer
  - Test b₀ to determine if RX buffer is full
  - Character received
Serial Devices: UART

- Programming RS-232
  - Line Control Register (LCR)
    - Read/Write register
    - Sets up communications parameters
      - Number of bits per character
      - Parity
      - Number of stop bits
      - Baud rate
Serial Devices: UART

Line Control Register

- **Break**: 0 - Normal output, 1 - Send a break
- **Stop bit**: 0 - No stop bit, 1 - Stop bit
- **Set bits per word**: 00 - 5 bits, 01 - 6 bits, 10 - 7 bits, 11 - 8 bits
- **Parity bit**: 0 - No parity, 1 - Parity
- **Parity type**: 0 - Even parity, 1 - Odd parity

Register address discriminator
Programming RS-232

- Line Control Register (LCR)
  - Baud rate
    - C7 set high to access the Baud rate divider
    - (C7 set low to access the TR/RX buffer)
    - Set baud rate by loading 16-bit divisor N
      LSB (least significant byte) -> TR/RX buffer address
      MSB (most significant byte) -> TR/RX buffer address + 1
    - Value loaded depends on crystal frequency connected to the IC

\[
\text{Baud rate} = \frac{\text{Clock frequency}}{16 \times N}
\]

e.g., for 1.8432 MHz crystal frequency, 9600 baud,
\[
N = \frac{1.8432 \times 10^6}{9600 \times 16} = 12 \text{ (000Ch)}
\]
Serial Devices: UART

- Write to TX/RD buffer
- Read from TX/RD buffer

TX buffer

RX buffer

3F8h

TD

RD

Copyright © 2007 David Vernon (www.vernon.eu)
/* send.c */

#define TXDATA 0x3F8
#define LSR 0x3FD
#define LCR 0x3FB

#include<stdio.h>
#include <conio.h> /* included for getch */
#include <dos.h> /* included for inputb and outputb */

void setup_serial(void);
void send_character(int ch);

int main(void)
{    int ch;
    puts("Transmitter program. Please enter text (Cntl-D to end)\n");
    setup_serial();
    do
    {    ch=getch();
        send_character(ch);
    } while (ch!=4);
    return(0);
}
Serial Interfaces: RS-232

```c
void setup_serial(void)
{
    /* set up bit 7 to a 1 to set Register address bit */
    outportb( LCR, 0x80);

    /* load TxRegister with 12, crystal frequency is 1.8432MHz */
    outportb(TXDATA,0x0C);
    outportb(TXDATA+1,0x00);

    /* Bit pattern loaded is 00001010b, from msb to lsb these are:
    /* Access TD/RD buffer, normal output, no stick bit
    /* even parity, parity on, 1 stop bit, 7 data bits */
    outportb(LCR, 0x0A);
}

void send_character(int ch)
{
    char status;

    /*repeat until Tx buffer empty ie bit 6 set*/
    do {
        status = inportb(LSR) & 0x40;
    } while (status!=0x40);

    outportb(TXDATA,(char) ch);
}
```
Serial Interfaces: RS-232

/* receive.c */
#define TXDATA 0x3F8
#define LSR 0x3FD
#define LCR 0x3FB
#include <stdio.h>
#include <conio.h>  /* included for getch */
#include <dos.h>    /* included for inputb and outputb */

void setup_serial(void);
int get_character(void);

int main(void)
{
    int inchar;
    setup_serial();
    do
    {
        inchar=get_character();
        putchar(inchar);
        putchar(inchar);
    } while (inchar!=4);
    return(0);
}
Serial Interfaces: RS-232

void setup_serial(void)
{
    /* set up bit 7 to a 1 to set Register address bit */
    outportb(LCR, 0x80);

    /* load TxRegister with 12, crystal frequency is 1.8432MHz */
    outportb(TXDATA, 0x0C);
    outportb(TXDATA+1, 0x00);
    /* Bit pattern loaded is 00001010b, from msb to lsb these are: */
    /* Access TD/RD buffer, normal output, no stick bit */
    /* even parity, parity on, 1 stop bit, 7 data bits */
    outportb(LCR, 0x0A);
}

int get_character(void)
{
    int status;
    /* Repeat until bit 1 in LSR is set */
    do {
        status = inportb(LSR) & 0x01;
    } while (status!=0x01);

    return((int)inportb(TXDATA));
}
Serial Interfaces: RS-422

- Signal levels 0 to +5V
- Uses differential amplifiers at both transmitting and receiving ends to achieve high noise immunity
- Up to 10 Mbps over distance of 1.2km
Motivation for the development of the USB

- Cost, configuration, and attachment of many peripheral devices to a PC creates problems
- Traditional interfacing (legacy systems)
  - Peripherals mapped onto the I/O space
    - Although the I/O space has 64k of addressable locations (16 bits), legacy ISA bus PCs only decode 10 bit, leaving only 1k of I/O space
    - I/O address conflicts are common
  - Assigned a specific IRQ (Interrupt Request line)
    - But there is a limited number of IRQs available (16)
    - Some of these are dedicated to specific devices
    - Some are shared (but not at the same time)
  - Sometimes assigned a DMA (Direct Memory Access) channel

(Based in part on a summary written by Craig Peacock, www.beyondlogic.org)
Universal Serial Bus - USB 2.0

Motivation for the development of the USB

- Traditional interfacing (legacy systems)
  - Physical limitation:
    - Serial and parallel interfaces only support a single device each
    - Only solution is to add additional (expensive) expansion cards
  - Usability
    - Too many connectors and/or cables
    - System must be shut down to attach most peripheral
    - System must be restarted to install/load software
Universal Serial Bus - USB 2.0

USB Design Goals

- Single connector type for any peripheral
- Ability to attach many peripherals
- Ease system resource conflicts
- Hot plug support (hot-swapable)
- Automatic detection and configuration (plug-and-play)
  - Dynamically loadable and unloadable drivers
    - Plug in device
    - Host detects device
    - Host interrogates device
    - Host loads appropriate driver
- Low-cost for both PC and peripheral
- Enhanced performance
- Support for new peripheral designs
- Support for legacy hardware and software
- Low-power implementation
Universal Serial Bus - USB 2.0

- 650 pages in the definition of the USB 2.0 specification (see www.usb.org)

- USB 1.1 supported two speeds:
  - Full speed 12 Mbit/s
  - Low speed 1.5 Mbits/s
    - Less susceptible to EMI (electromagnetic interference)
    - Can use cheaper components

- USB 2.0 supports 480 Mbits/s
  - High speed
  - Competes with IEEE 1394 (Firewire) serial bus
Universal Serial Bus - USB 2.0

- **USB is host controlled**
  - Only one host per bus
  - Host is responsible for
    - undertaking all transactions
    - Scheduling bandwidth
    - Data can be sent by one of several transaction methods using a token-based protocol

- **USB uses a tiered star topology**
  - Devices/peripherals can’t be daisy-chained
  - Need a hub
    - Some peripherals have in-built hubs (e.g. keyboards)
    - You can plug other devices into these
  - Up to 127 devices can be connected to any one USB at any one given time
Universal Serial Bus - USB 2.0

- USB host controller specifications (USB 1.1)
  - UHCI (Universal Host Controller Interface)
    - Developed by Intel
    - More work is done in software
    - Hence cheaper hardware
  - OHCE (Open Host Controller Interface)
    - Developed by Compaq, Microsoft, and National Semiconductor
    - More work done in hardware
    - Hence simpler software!
  - USB 2.0
    - EHCI (Enhanced Host Controller Interface)
Universal Serial Bus - USB 2.0

USB Transfer modes

- Control
- Interrupt
- Bulk
- Isochronous

- Allows a device to reserve a defined amount of bandwidth with guaranteed latency
- Ideal for audio or video
- Avoids dropped frames due to congestion
Universal Serial Bus - USB 2.0

◆ Connectors

- Upstream connection to the host
- Downstream connection to a device
- Not interchangeable
- Type A plugs (and sockets) always face upstream to the host
- Type B plugs (and sockets) always face downstream to the peripheral device
- USB 2.0 includes mini-USB B connectors

Copyright © 2007 David Vernon (www.vernon.eu)
Universal Serial Bus - USB 2.0

- Connectors

<table>
<thead>
<tr>
<th>Pin Number</th>
<th>Cable Colour</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Red</td>
<td>$V_{BUS}$ (5 volts)</td>
</tr>
<tr>
<td>2</td>
<td>White</td>
<td>D-</td>
</tr>
<tr>
<td>3</td>
<td>Green</td>
<td>D+</td>
</tr>
<tr>
<td>4</td>
<td>Black</td>
<td>Ground</td>
</tr>
</tbody>
</table>
Universal Serial Bus - USB 2.0

- **Electrical Characteristics**
  - USB uses a differential transmission pair for data
  - Differential ‘1’ is transmitted by
    - Pulling D+ over 2.8V
    - Pulling D- under 0.3V
  - Differential ‘0’ is transmitted by
    - Pulling D- over 2.8V
    - Pulling D+ under 0.3V
  - The polarity of the signal is inverted depending on the speed of the bus
    - Use ‘J’ and ‘K’ to signify logic levels
    - ‘J’ is differential 0 in low speed
    - ‘J’ is differential 1 in high speed
Universal Serial Bus - USB 2.0

- Electrical Characteristics
  - USB also uses single ended outputs (on D+, D-, or both)
  - For example:
    - Single Ended Zero (SE0)
    - Used to signify a device reset if held for more than 10mS
    - Generated by holding both D- and D+ low (<0.3V)
  - If you are building a USB interface / device, you can’t get away with just sampling the differential output
Universal Serial Bus - USB 2.0

- **Speed Identification**
  - A USB device must indicate its speed by pulling either the D+ or D- line high to 3.3V (full speed and low speed respectively)

![Diagram of USB Speed Identification](image-url)
Universal Serial Bus - USB 2.0

- **Speed Identification**
  - High speed devices start by connecting as full speed
  - Once attached, it does a high speed chirp (signal with a time-varying increase in frequency) during reset
  - A high speed connection is then established if the hub supports it

- **Presence of a device on the bus**
  - These pull-up resistors are also used by the host or hub to detected the presence of a device connected to its port
  - Without a pull-up resistor, USB assumes there is nothing connected to the bus
  - Some devices have the resistor built into its silicon, which can be turned on and off under firmware control; others require an external resistor
Universal Serial Bus - USB 2.0

Speed Identification

- A USB 2.0 compliant device (downstream) is not required to support high-speed mode
  - Allows cheaper devices to be produced if speed isn’t critical
  - Same is true for low speed USB 1.1 devices: don’t have to support full speed

- High speed device MUST NOT support low speed mode
  - Only support full speed (to establish connection) and high speed (if successfully negotiated with the host)

- A USB 2.0 compliant downstream facing device (hub or host) must support all three modes
  - High speed 480 Mbits/s
  - Full speed 12 Mbits/s
  - Low speed 1.5 Mbits/s
Universal Serial Bus - USB 2.0

Power Characteristics

- One big advantage of USB is that devices can be powered by the bus (bus-powered devices)
- But there are restrictions
  - USB devices specify power consumption in 2mA units in the configuration descriptor (later)
  - A device cannot increase its power consumption over that specified during enumeration (set-up phase) EVEN IF IT LOSES EXTERNAL POWER
  - 3 classes of USB power functions
    - Low-power bus powered functions
    - High-power bus powered functions
    - Self-powered functions
Universal Serial Bus - USB 2.0

◆ Power Characteristics

□ Low-power bus powered functions
  ▪ Devices draw all its power from the $V_{BUS}$
  ▪ Cannot draw more than one unit load (100mA)
  ▪ Must be designed to work with $4.40V \leq V_{BUS} \leq 5.25V$

□ High-power bus powered functions
  ▪ Device draws all its power from the $V_{BUS}$
  ▪ Cannot draw more than one unit load (100mA) until it has been configured
  ▪ Then it can draw 5 unit loads maximum, provided it requested it in its descriptor

□ Self-powered functions
  ▪ May draw up to 1 unit load from the bus and derive the rest from an external source
  ▪ The 1 unit load allows detection and enumeration without external supply (but not operation)
Universal Serial Bus - USB 2.0

◆ Suspend Mode

- Suspend mode is mandatory on all devices
- Suspend mode imposes further constraints on power
- The maximum suspend current is proportional to the unit load
  - 1 unit load device: maximum suspend current is 500μA
  - This is not a lot!
  - The pull-up resistor (at the device, for detection) and pull-down resistor (at the host) will draw 200μA (3.3A / 16.5kΩ)
  - 5V to 3.3V regulators to allow 3.3V devices to work will draw more

Copyright © 2007 David Vernon (www.vernon.eu)
Universal Serial Bus - USB 2.0

◆ Suspend Mode

- A USB device will enter suspend when there is no activity on the bus for greater than 3.0ms
- It then has a further 7ms to shutdown the device and draw no more than the suspend current
- To stay connected, the device must still provide power to the speed selection pull-up resistor during suspend
- USB can send ‘start of frame’ packets sent periodically to prevent an idle (no data) bus entering suspend
  - High speed bus: frames sent every 124 $\mu$s
  - Full speed bus: frames sent every 1ms
  - Low speed bus: EOP (end of packet) every 1 ms
- A suspended device will resume operation when it receives any non-idle signalling.
Universal Serial Bus - USB 2.0

USB Protocols - Overview

- RS-232 serial interface does not define the format of the data being sent
- USB is made up of several layers of protocols
  - ICs normally take care of the lower layer
- Each USB transaction consists of
  - Token Packet (header defining what will follow)
  - Optional Data Packet
  - Status Packet (to acknowledge transaction & provide error correction)
- Remember: USB is host-centric – the host initiates all transactions
Host generates token packet:
- Specifies what is to follow, and
- Whether the data transaction is a read or a write

Host (usually) then sends a data packet
Followed by a handshaking packet (check for errors)

*Note: data is transmitted LSB first*
Universal Serial Bus - USB 2.0

USB Packet Structure

- Each USB packet comprises the following fields
  - Sync
    - 8 bits long
    - used to synchronize the clock of the receiver with the clock of the transmitter
  - PID (Packet ID)
    - identifies the type of packet being sent
    - 4 bits (but complemented and repeated to give 8 bits)
  - ADDR
    - Address of device to which packet is being sent
    - 7 bits (127 devices)
    - Address 0 is not valid
      - any device which is not yet assigned an address must respond to packets sent to address zero
Universal Serial Bus - USB 2.0

**USB Packet Structure**

- Each USB packet comprises the following fields
  - **ENDP**
    - End point field
    - 4 bits (16 end-points)
  - **CRC**
    - Cyclic Redundancy Check
    - Performed on all the data within the packet payload
    - Token packets have a 5 bit CRC
    - Data packets have a 16 bit CRC
  - **EOP**
    - End of packet
    - Signalled by a Single Ended Zero (SE0) for approx. 2 bit times followed by a ‘J’ for 1 bit time
## Universal Serial Bus - USB 2.0

### USB Packet Structure – PID Values

<table>
<thead>
<tr>
<th>Group</th>
<th>PID Value</th>
<th>Packet Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Token</td>
<td>0001</td>
<td>OUT Token</td>
</tr>
<tr>
<td></td>
<td>1001</td>
<td>IN Token</td>
</tr>
<tr>
<td></td>
<td>0101</td>
<td>SOF Token</td>
</tr>
<tr>
<td></td>
<td>1101</td>
<td>SETUP Token</td>
</tr>
<tr>
<td>Data</td>
<td>0011</td>
<td>DATA0</td>
</tr>
<tr>
<td></td>
<td>1011</td>
<td>DATA1</td>
</tr>
<tr>
<td></td>
<td>0111</td>
<td>DATA2</td>
</tr>
<tr>
<td></td>
<td>1111</td>
<td>MDATA</td>
</tr>
<tr>
<td>Handshake</td>
<td>0010</td>
<td>ACK Handshake</td>
</tr>
<tr>
<td></td>
<td>1010</td>
<td>NAK Handshake</td>
</tr>
<tr>
<td></td>
<td>1110</td>
<td>STALL Handshake</td>
</tr>
<tr>
<td></td>
<td>0110</td>
<td>NYET (No Response Yet)</td>
</tr>
<tr>
<td>Special</td>
<td>1100</td>
<td>PREAMBLE</td>
</tr>
<tr>
<td></td>
<td>1100</td>
<td>ERR</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>Split</td>
</tr>
<tr>
<td></td>
<td>0100</td>
<td>Ping</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PID0</th>
<th>PID1</th>
<th>PID2</th>
<th>PID3</th>
<th>nPID0</th>
<th>nPID1</th>
<th>nPID2</th>
<th>nPID3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Universal Serial Bus - USB 2.0

USB Packet Types
- USB has four packet types
  - Token packets (type of transaction)
  - Data Packets (payload / information)
  - Handshake Packets (ack & error correction)
  - Start of Frame packets (flag start of a new frame)
Universal Serial Bus - USB 2.0

USB Packet Types

- Token packets
  - 3 types of token packets
  - **In**: informs the USB device that host wishes to read info.
  - **Out**: informs the USB device that host wishes to send info.
  - **Setup**: used to begin control transfers

Packet format:

<table>
<thead>
<tr>
<th>Sync</th>
<th>PID</th>
<th>ADDR</th>
<th>ENDP</th>
<th>CRC5</th>
<th>EOP</th>
</tr>
</thead>
</table>

Copyright © 2007 David Vernon (www.vernon.eu)
Universal Serial Bus - USB 2.0

USB Packet Types

- Data packets
  - 2 types of data packets
  - Each can transmit 0-1023 bytes of data
  - Data0
  - Data1
  - Packet format:

<table>
<thead>
<tr>
<th>Sync</th>
<th>PID</th>
<th>Data</th>
<th>CRC16</th>
<th>EOP</th>
</tr>
</thead>
</table>

Copyright © 2007 David Vernon (www.vernon.eu)
Universal Serial Bus - USB 2.0

♦ USB Packet Types

- Handshake packets
  - 3 types of handshake packets
  - **ACK** Acknowledgement that the packet has been successfully received
  - **NAK** Reports that the device can neither send nor receive data
  - **STALL** The device needs intervention from the host

- Packet format:
Universal Serial Bus - USB 2.0

- **USB Packet Types**
  - **Start of Frame packets**
    - Consists of an 11-bit number & is sent by the host every 1mS +/- 500nS
    - Packet format:

```
| Sync | PID | Frame Number | CRC5 | EOP |
```
Universal Serial Bus - USB 2.0

USB Functions

- USB functions are USB devices / peripherals (e.g. printers, disk drives, cameras, scanners, modems, ...)
- Most functions have a series of buffers, typically 8 bytes long
- Each buffer will belong to an endpoint
  - EP0 IN
  - EP0 OUT
  - EP1 IN
  - ...

Copyright © 2007 David Vernon (www.vernon.eu)
Universal Serial Bus - USB 2.0
Universal Serial Bus - USB 2.0

USB Functions – Conceptual Example

- Host sends a device descriptor request
- Function hardware reads the setup packet
- Function determines from the address field whether the packet is addressed to it
- If it is, the function copies the payload of the following data packet to the appropriate endpoint buffer (given by the value in the endpoint field of the setup token)
- Function then sends a handshake packet to acknowledge the reception of the byte
- Function generates an internal interrupt for the appropriate endpoint, flagging that it has received a packet
- Typically, this is all done in hardware
- The software interrupt handler now reads the endpoint buffer, interprets it, and initiates the appropriate action
Universal Serial Bus - USB 2.0

◆ Endpoints
  □ Sources or sinks of data
  □ Occur at the end of the communications channel at the USB function
  □ The endpoint OUT is where the host sends data to (on the function). E.g. **EP1 OUT**
  □ The endpoint IN is where the function places data when it is to be sent to the host, e.g. **EP1 IN**
  □ NB: the function cannot simply write to the bus – the bus is controlled by the host and it must send an IN packet to enable the transfer of data from **EP2 IN** to the host
Universal Serial Bus - USB 2.0

◆ Pipes

- A pipe is a logical connection between the host and the endpoint(s)
- Client software transfers data through pipes
- Pipes have a set of parameters associated with them
  - How much bandwidth is allocated
  - What transfer type it uses
    - Control
    - Bulk
    - Isochronous
    - Interrupt
  - Direction of data flow
  - Maximum packet/buffer size
- Default pipe is bidirectional, between **EP0 IN** and **EP0 OUT** with control transfer type
Universal Serial Bus - USB 2.0

◆ Pipes

☐ USB defines two types of pipes
  ☐ Stream Pipes
    ▪ No defined format
    ▪ Can send any type of data down a stream pipe
    ▪ Data flows sequentially
    ▪ Data has a predefined direction (in or out)
    ▪ Support bulk, isochronous, interrupt transfer types
    ▪ Can be controlled by either host or function / device

☐ Message Pipes
  ▪ Defined format
  ▪ Host controlled
  ▪ Initiated by a request sent from the host
  ▪ Data is then sent in the desired direction (depends on the request)
  ▪ Supports only control transfers
Universal Serial Bus - USB 2.0

- Endpoint / Transfer Types
  - Control Transfers
  - Interrupt Transfers
  - Isochronous Transfers
  - Bulk Transfers
Universal Serial Bus - USB 2.0

Control Transfers

- Used for command and status operations
- E.g. to set up a USB device (enumeration)
- Initiated by the host
- Packet length
  - 8 bytes – low speed devices
  - 64 bytes – full speed devices
  - 8, 16, 32, or 64 bytes - high speed devices
- A Control Transfer can have up to three stages
  - Setup Stage
  - Data Stage
  - Status Stage
Control Transfers

Setup Stage

- Comprises 3 packets
  - Setup token
    - contains address and endpoint number
  - Data packet
    - always has PID of type Data0
    - Contains a setup packet with the type of request
- Handshake
  - Acknowledge successful receipt or indicate and error
  - If the function successfully receives the setup data (CRC and PID etc are okay), it responds with ACK
  - Otherwise, it ignores the data and doesn’t send a handshake
  - Functions cannot issue STALL or NAK in response to a setup packet
Universal Serial Bus - USB 2.0

- Control Transfers
  - Setup Stage

![Diagram of control transfers including setup, data, and handshake packets with error handling]

Copyright © 2007 David Vernon (www.vernon.eu)
Universal Serial Bus - USB 2.0

Control Transfers

- Data Stage (optional)
  - The amount of data to be sent in this stage has been specified in the previous setup stage (in the setup packet)
  - If the amount of data exceeds the maximum packet size, data is sent in multiple transfers (each of the maximum size except the last one)
  - Two scenarios
    - IN direction
    - OUT direction
Control Transfers

- Data Stage (optional)
  - IN token issued when the host is ready to receive control data
    - If the function receives the IN token with an error, it ignores the packet
    - If the function receives the token correctly, it responds with either
      - DATA packet (with the control data to be sent), or
      - STALL packet (endpoint has had an error), or
      - NAK packet (endpoint if working but has no data to send yet)
Control Transfers

- Data Stage (optional)
  - **OUT** token issued when the host needs to send the device a control data packet
  - Followed by a **DATA** packet with the control data as a payload
    - If the function receives either the **OUT** token or the **DATA** packet with an error, it ignores the packet
    - Otherwise, it issues one of three responses
      - **ACK** (if the function receives the data correctly - successful transfer to the endpoint buffer)
      - **NAK** (if the buffer was not empty e.g. still processing previous packet)
      - **STALL** (endpoint has had an error)
Universal Serial Bus - USB 2.0

- Control Transfers
  - Data Stage (optional)
Universal Serial Bus - USB 2.0

Control Transfers

- Status Stage
  - Reports the status of the overall request
  - Varies depending on the direction of transfer
  - Status reporting is always performed by the function
  - **IN**
    - If the host sent **IN** token during the data stage to receive data, the host must acknowledge successful receipt of the data
    - Done by host sending an **OUT** token followed by a zero length data packet
    - Function can now report status
      - **ACK** (success; ready for another command)
      - **STALL** (error)
      - **NAK** (still processing)
Universal Serial Bus - USB 2.0

◆ Control Transfers
  ◼ Status Stage
    ▪ IN

---

Copyright © 2007 David Vernon (www.vernon.eu)
Universal Serial Bus - USB 2.0

Control Transfers

- Status Stage
  - **OUT**
    - If the host sent **OUT** token(s) during the data stage to transmit data, the function must acknowledge successful receipt of the data
    - Done by host sending an **IN** token
    - Function responds:
      - **DATA0** (success; ready for another command)
      - **STALL** (error)
      - **NAK** (still processing; host has to retry status stage later)
    - Host completes the stage by sending an **ACK**
Universal Serial Bus - USB 2.0

Control Transfers

- Status Stage
  - OUT

Diagram:

- IN
- Normal Completion
- Error
  - NAK
  - STALL
- DATA0 Zero Length
- ACK
Universal Serial Bus - USB 2.0

Control Transfer Example

Host wants to request a device descriptor during enumeration

Host sends **Setup** token (⇒ next packet is a setup packet)
- Address field will hold the address of the device / function
- Endpoint should be zero (⇒ default pipe)

Host sends **DATA0** packet
- 8 byte payload (containing device descriptor request; ch. 9 USB Specification)

Device send **ACK** (or ignores it, if there is an error)

This is the first USB transaction (setup stage): the device now decodes the 8 bytes received, determines it is a device descriptor request, & then attempt to send the descriptor. This is the next transaction (next slide)
# Universal Serial Bus - USB 2.0

<table>
<thead>
<tr>
<th>Step</th>
<th>Sync</th>
<th>PID</th>
<th>ADDR</th>
<th>ENDP</th>
<th>CRC5</th>
<th>EOP</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Setup Token</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Address &amp; Endpoint Number</td>
</tr>
<tr>
<td>2. Data0 Packet</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Device Descriptor Request</td>
</tr>
<tr>
<td>3. Ack Handshake</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Device Ack. Setup Packet</td>
</tr>
</tbody>
</table>

Copyright © 2007 David Vernon (www.vernon.eu)
Universal Serial Bus - USB 2.0

Control Transfer Example

- Assume the maximum payload size is 8 bytes
- Host now send IN token (tell device it can now send data)
- Device descriptor is sent in chunks of 8 bytes:
  - Device sends DATA1 packet
  - Host sends ACK handshake
  - Host sends IN token
  - Device sends DATA0 packet
  - Host sends ACK handshake
  - Host sends IN token
  - Device sends DATA1 packet (last 8 bytes of descriptor)
  - Host send ACK handshake

- This is the second USB transaction (data stage)
# Universal Serial Bus - USB 2.0

<table>
<thead>
<tr>
<th>Step</th>
<th>Token Type</th>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>In Token</td>
<td>Sync</td>
<td>Address &amp; Endpoint Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PID</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADDR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENDP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CRC5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EOP</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Data1 Packet</td>
<td>Sync</td>
<td>First 8 bytes of Device Descriptor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PID</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CRC16</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EOP</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Ack Handshake</td>
<td>Sync</td>
<td>Host Acknowledges Packet</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PID</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EOP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>Token Type</th>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>In Token</td>
<td>Sync</td>
<td>Address &amp; Endpoint Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PID</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADDR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENDP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CRC5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EOP</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Data0 Packet</td>
<td>Sync</td>
<td>Second 8 bytes of Device Descriptor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PID</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CRC16</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EOP</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Ack Handshake</td>
<td>Sync</td>
<td>Host Acknowledges Packet</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PID</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EOP</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step</th>
<th>Token Type</th>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>In Token</td>
<td>Sync</td>
<td>Address &amp; Endpoint Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PID</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADDR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ENDP</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CRC5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EOP</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Data1/0 Packet</td>
<td>Sync</td>
<td>Last 8 bytes of Device Descriptor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PID</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data0/1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CRC16</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EOP</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Ack Handshake</td>
<td>Sync</td>
<td>Host Acknowledges Packet</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PID</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>EOP</td>
<td></td>
</tr>
</tbody>
</table>
Control Transfer Example

- Host sends **OUT** Token
- Host sends a zero length **DATA1** packet (overall transaction is successful)
- Device sends **ACK** handshake

- This is the third USB transaction (status stage)
### Universal Serial Bus - USB 2.0

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Out Token</td>
<td>Sync, PID, ADDR, ENDP, CRC5, EOP</td>
</tr>
<tr>
<td>2.</td>
<td>Data1 Packet</td>
<td>Sync, PID, Data1, CRC16, EOP</td>
</tr>
<tr>
<td>3.</td>
<td>Ack Handshake</td>
<td>Sync, PID, EOP</td>
</tr>
</tbody>
</table>
Universal Serial Bus - USB 2.0

◆ **Interrupt Transfers**

  - Under USB, if a device requires the attention of the host, it must wait until the host polls it before it can report it needs attention!
  - **Characteristics of Interrupt Transfers**
    - Guaranteed latency
    - Stream pipe – unidirectional
    - Error detection and next period retry
  - Typically non-periodic small device-initiated communication requiring bounded latency
  - **Maximum data payload**
    - 8 bytes – low speed devices
    - 64 bytes – full speed devices
    - 1024 bytes - high speed devices
Universal Serial Bus - USB 2.0

◆ Interrupt Transfers
  ❑ Interrupt IN Transaction
    ▪ Host will periodically poll the interrupt endpoint
    ▪ Polling rate is specified in the endpoint descriptor
    ▪ Host sends an IN token, the function respond with a DATA packet, and the host acknowledges
  ❑ Interrupt OUT Transaction
    ▪ When the host wants to send the device interrupt data
    ▪ Sends OUT token, followed by a data packet, and device acknowledges
Universal Serial Bus - USB 2.0

Token Packet | Data Packet | Handshake Packet
---|---|---
IN | DATA x | ACK
STALL | NAK | Success

OUT | DATA x | ACK

Key: Host | Function

Success
Data Error
Halt
No Interrupt Pending
In Token Error

Failure
Halt
Error

Copyright © 2007 David Vernon (www.vernon.eu)
Universal Serial Bus - USB 2.0

- Isochronous Transfers
  - Occur continuously and periodically
  - Typically contain time sensitive information
    - Audio
    - Video
  - Characteristics of Isochronous Transfers
    - Guaranteed access to USB bandwidth
    - Bounded latency
    - Stream pipe – unidirectional
    - Error detection via CRC, but no retry or guarantee of delivery
    - Full and high-speed modes only
  - Maximum data payload is specified in the endpoint descriptor
    - 1023 bytes – full speed devices
    - 1024 bytes - high speed devices
    - Amount of data can be less than the pre-negotiated size and may vary from transaction to transaction
  - No handshaking
  - No error reports or STALL/HALT conditions

Copyright © 2007 David Vernon (www.vernon.eu)
Universal Serial Bus - USB 2.0
Universal Serial Bus - USB 2.0

◆ Bulk Transfers
  - Used for large bursty data transfer (e.g. printer, scanner)
  - Characteristics of Bulk Transfers
    - Error detection via CRC, with guarantee of delivery
    - No guarantee of bandwidth or minimum latency
    - Stream pipe – unidirectional
    - Full and high-speed modes only
  - Maximum data payload
    - 8, 16, 32, 64 bytes – full speed devices
    - 512 bytes - high speed devices
    - Amount of data can be less than the maximum packet size (zero padding not needed)
Universal Serial Bus - USB 2.0

Token Phase  Data Phase  Handshake Phase
Universal Serial Bus - USB 2.0

- **Bandwidth Management**
  - The host is responsible for managing bandwidth
  - Achieved at enumeration when configuring isochronous and interrupt endpoints (and throughout the operation of the bus)
  - **USB Specification Limits:**
    - Full speed bus: \( \leq 90\% \) allocated to periodic transfers
    - High speed bus: \( \leq 80\% \) allocated to periodic transfers
    - Control transfer get allocated from the remainder
    - Bulk transfers get what’s left!
Universal Serial Bus - USB 2.0

USB Descriptors

- All USB devices define a hierarchy of descriptors
  - What the device is
  - What version of USB it supports
  - How many ways it can be configured
  - Number of end-points and their types

Common descriptors

- Device descriptors
- Configuration descriptors
- Interface Descriptors
- Endpoint descriptors
- String descriptors
Universal Serial Bus - USB 2.0

- USB Descriptors
  - Device Descriptor
    - Only one of these
    - USB revision compliance
    - Product id and vendor id (needed to load the drivers)
    - Number of configurations supported
  - Configuration Descriptor
    - Amount of power used
    - Self- or bus-powered
    - Number of interfaces
    - Very few devices have more than one configuration
Universal Serial Bus - USB 2.0

◆ USB Descriptors
  - Interface Descriptor
    - Groups endpoints into functional groups
    - Each functional group corresponds to a feature of the device
    - For example, a fax/scanner/printer device might have three groups (and, hence, three interface descriptors)
  - Endpoint Descriptors
    - Describe endpoints other than endpoint zero
    - Endpoint zero is always assumed to be a control endpoint
  - String Descriptors
    - Provide human readable information
    - Optional
Universal Serial Bus - USB 2.0

- The Setup Packet
  - Every USB device must respond to setup packets on the default pipe
  - Used for detection and configuration of the device
    - Setting USB device’s address
    - Requesting device descriptor
    - Checking the status of an endpoint
  - All requests have to be processed within set time limits
  - Setup Packet is 8 bytes long
### Universal Serial Bus - USB 2.0

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bmRequestType</td>
<td>1</td>
<td>Bit-Map</td>
<td><strong>D7 Data Phase Transfer Direction</strong>&lt;br&gt;0 = Host to Device&lt;br&gt;1 = Device to Host</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>D6..5 Type</strong>&lt;br&gt;0 = Standard&lt;br&gt;1 = Class&lt;br&gt;2 = Vendor&lt;br&gt;3 = Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>D4..0 Recipient</strong>&lt;br&gt;0 = Device&lt;br&gt;1 = Interface&lt;br&gt;2 = Endpoint&lt;br&gt;3 = Other&lt;br&gt;4..31 = Reserved</td>
</tr>
<tr>
<td>1</td>
<td>bRequest</td>
<td>1</td>
<td>Value</td>
<td>Request</td>
</tr>
<tr>
<td>2</td>
<td>wValue</td>
<td>2</td>
<td>Value</td>
<td>Value</td>
</tr>
<tr>
<td>4</td>
<td>wIndex</td>
<td>2</td>
<td>Index or Offset</td>
<td>Index</td>
</tr>
<tr>
<td>6</td>
<td>wLength</td>
<td>2</td>
<td>Count</td>
<td>Number of bytes to transfer if there is a data phase</td>
</tr>
</tbody>
</table>
Universal Serial Bus - USB 2.0

◆ The Setup Packet

- bRequest field determines the request being made
  - Standard device requests (section 9.4 of the USB specification)
    - Must be implemented for every USB device
      - GET_STATUS
      - SET_ADDRESS
      - 6 more
    - Class requests (e.g. communications class, mass storage class)
    - Vendor defined requests
Universal Serial Bus - USB 2.0

◆ Enumeration

- The process of determining
  - what device has just been connected to the bus
  - The power requirements
  - Number and type of endpoints
  - Class of product
- Host then assigns an address to the device
- Host also enables a configuration allowing the device to transfer data on the bus
- See section 9.1.2 of the USB specification
We have concentrated on the USB interface from the point of view of the device or function.

However, something has to manage all the transactions on the bus ... this is the job of the host.

We mentioned three USB host controller specifications:

- **UHCI (Universal Host Controller Interface)**
  - Developed by Intel
  - More work is done in software
  - Hence cheaper hardware

- **OHCE (Open Host Controller Interface)**
  - Developed by Compaq, Microsoft, and National Semiconductor
  - More work done in hardware
  - Hence simpler software!

- **USB 2.0**
  - **EHCI (Enhanced Host Controller Interface)**
Universal Serial Bus - USB 2.0

- The job of each of these interfaces is to schedule transactions on the bus, based on the information provided by each function/device at enumeration
  - Meet the needs of all isochronous, interrupt, control, and bulk transfers
- Transactions are generated by cycling through a list of frames (one frame every 1ms), each frame comprising a list of transfer descriptors
  - Transfer descriptor
    - USB device address
    - Type of transaction to be performed
    - Transfer size
    - Speed of transaction
    - Location of memory data buffer (where data will be read from or written to)
The IEEE-488 Bus
IEEE-488 Bus

- Digital Interface for Programmable Instrumentation
- Also known as the General Purpose Instrument Bus (GPIB)
  - Provides a way of interconnecting a microcomputer controller with a large range of test and measuring instruments
    - Signal generators, oscilloscopes, network analysers ...
  - The objective of the IEEE-488 bus is to enable different instruments using different data transfer rates, different message lengths, and different capabilities to communicate with each other on a single bus
  - Allows a PC to supervise operation of instruments, and collect & process data they provide

(Based in part on information abstracted from: http://www.interfacebus.com/Design_Connector_GPIB.html and http://www.hit.bme.hu/people/papay/edu/GPIB/tutor.htm)
IEEE-488 Bus

Supports both star and daisy-chain topologies
IEEE-488 Bus

◆ In 1965, Hewlett-Packard designed the Hewlett-Packard Interface Bus (HP-IB) to connect their line of programmable instruments to their computers.

◆ Because of its high transfer rates (nominally 1 Mbytes/s), this interface bus quickly gained popularity.

◆ It was later accepted as IEEE Standard 488-1975, and has evolved to ANSI/IEEE Standard 488.1-1987.

◆ Today, the name General Purpose Interface Bus (GPIB) is more widely used than HP-IB.

◆ ANSI/IEEE 488.2-1987 strengthened the original standard by defining precisely how controllers and instruments communicate.
IEEE-488 Bus

- The IEEE-488 interface bus is an 8 bit wide byte serial, bit parallel interface system which incorporates:
  - 5 control lines (interface management)
  - 3 handshake lines
  - 8 bi-directional data lines.

- The entire bus consists of 24 lines, with the remaining lines occupied by ground wires.

- Uses negative logic with standard TTL levels
  - When DAV is true, for example, it is a TTL low level ($\leq 0.8$ V)
  - When DAV is false, it is a TTL high level ($\geq 2.0$ V)

- The maximum data transfer rate is determined by a number of factors, but is assumed to be 1Mb/s.
IEEE-488 Bus

- Devices exist on the bus in any one of 3 general forms:
  1. Controller
  2. Talker
  3. Listener

- A single device may incorporate all three options, although only one option may be active at a time.
IEEE-488 Bus

◆ CONFIGURATION REQUIREMENTS

- To achieve the high data transfer rate for which the GPIB was designed, the physical distance between devices and the number of devices on the bus are limited.
- A maximum separation of 4 m between any two devices and an average separation of 2 m over the entire bus.
- A maximum total cable length of 20 m.
- No more than 15 device loads connected to each bus, with no less than two-thirds powered on.
IEEE-488 Bus

- The Controller makes the determination as to which device becomes active on the bus.
- The GPIB can handle only one ‘active’ controller on the bus, although it may pass operation to another controller.
- Any number of active listeners can exist on the bus with an active talker as long as no more than 15 devices are connected to the bus.
- The controller determines which devices become active by sending interface messages over the bus to a particular instrument.
IEEE-488 Bus

◆ Each individual device is associated with a 5 bit BCD code which is unique to that device.
  □ By using this code, the controller can coordinate the activities on the bus and the individual devices can be made to talk, listen (un-talk, un-listen) as determined by the controller.
  □ A controller can only select a particular function of a device, if that function is incorporated within the device; for example a ‘listen’ only device can not be made to talk to the controller.

◆ The Talker sends data to other devices
◆ The Listener receives the information from the Talker
IEEE-488 Bus

- Device dependent messages are moved over the GPIB in conjunction with the data byte transfer control lines.
- These three lines (DAV, NRFD, and NDAC) are used to form a three wire ‘interlocking’ handshake which controls the passage of data:
  - The active talker would control the ‘DAV’ line (Data Valid).
  - The listener(s) would control the ‘NRFD’ (Not Ready For Data), and the ‘NDAC’ (Not Data Accepted) line.
IEEE-488 Bus

- In the steady state mode the talker will hold ‘DAV’ high (no data available) while the listener would hold ‘NRFD’ high (ready for data) and ‘NDAC’ low (no data accepted).
- After the talker placed data on the bus it would then take ‘DAV’ low (data valid).
- The listener(s) would then send ‘NRFD’ low and send ‘NDAC’ high (data accepted).
- Before the talker lifts the data off the bus, ‘DAV’ will be taken high signifying that data is no longer valid.
IEEE-488 Bus
IEEE-488 Bus

- If the ‘ATN’ line (attention) is high while this process occurs the information is considered data (a device dependent message), but with the "ATN’ line low the information is regarded as an interface message; such as listen, talk, un-listen or un-talk.

- The other five lines on the bus (‘ATN’ included) are the bus management lines.

- These lines enable the controller and other devices on the bus to enable, interrupt, flag, and halt the operation of the bus.
IEEE-488 Bus

- Most devices operate either via front panel control or GPIB control (REMOTE)
- While using the front Panel the device is in the Local state, when receiving commands via the GPIB, the device is in the Remote state
- The device is placed in the Remote state when ever the System Controller is reset or powered on; also, when the system controller sends out an Abort message
- In addition, if the device is addressed, it then enters the Remote state.
## IEEE-488 Bus

<table>
<thead>
<tr>
<th>Pin</th>
<th>Signal Name</th>
<th>Function</th>
<th>Pin</th>
<th>Signal Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DIO1</td>
<td>Data input/output bit 1</td>
<td>13</td>
<td>DIO5</td>
<td>Data input/output bit 5</td>
</tr>
<tr>
<td>2</td>
<td>DIO2</td>
<td>Data input/output bit 2</td>
<td>14</td>
<td>DIO6</td>
<td>Data input/output bit 6</td>
</tr>
<tr>
<td>3</td>
<td>DIO3</td>
<td>Data input/output bit 3</td>
<td>15</td>
<td>DIO7</td>
<td>Data input/output bit 7</td>
</tr>
<tr>
<td>4</td>
<td>DIO4</td>
<td>Data input/output bit 4</td>
<td>16</td>
<td>DIO8</td>
<td>Data input/output bit 8</td>
</tr>
<tr>
<td>5</td>
<td>EOI</td>
<td>End-or-identify</td>
<td>17</td>
<td>REN</td>
<td>Remote enable</td>
</tr>
<tr>
<td>6</td>
<td>DAV</td>
<td>Data valid</td>
<td>18</td>
<td>SHIELD</td>
<td>Ground (DAV)</td>
</tr>
<tr>
<td>7</td>
<td>NRFD</td>
<td>Not ready for data</td>
<td>19</td>
<td>SHIELD</td>
<td>Ground (NRFD)</td>
</tr>
<tr>
<td>8</td>
<td>NDAC</td>
<td>Not data accepted</td>
<td>20</td>
<td>SHIELD</td>
<td>Ground (NDAC)</td>
</tr>
<tr>
<td>9</td>
<td>IFC</td>
<td>Interface clear</td>
<td>21</td>
<td>SHIELD</td>
<td>Ground (IFC)</td>
</tr>
<tr>
<td>10</td>
<td>SRQ</td>
<td>Service request</td>
<td>22</td>
<td>SHIELD</td>
<td>Ground (SRQ)</td>
</tr>
<tr>
<td>11</td>
<td>ATN</td>
<td>Attention</td>
<td>23</td>
<td>SHIELD</td>
<td>Ground (ATN)</td>
</tr>
<tr>
<td>12</td>
<td>SHIELD</td>
<td>Chassis ground</td>
<td>24</td>
<td>SIGNAL GND</td>
<td>Signal ground</td>
</tr>
</tbody>
</table>
IEEE-488 Bus

◆ NRFD: Not Ready For Data
  □ Part of a three wire handshake
  □ Used to indicate a device is ready for data, active low

◆ DAV: Data Valid
  □ Part of a three wire handshake
  □ Used to indicate valid data on the bus, active low

◆ NDAC: Not Data Accepted
  □ Part of a three wire handshake
  □ Used to indicate a device has not yet accepted data, active low
IEEE-488 Bus

- ATN: Attention
  - When low (true) the system places all devices in Command Mode
  - When high (false) the system places all devices in the Data Mode
  - In Command Mode the Controller passes data to devices,
  - In Data Mode the Talker passes data to the Listener.
  - All devices must monitor the ATN line and respond within 200nS.
IEEE-488 Bus

- EIO: End or Identify
  - Indicates the last data transfer of a multi-byte sequence or used by the system controller to indicate a Parallel Poll to a device (in conjunction with the ATN line)
IEEE-488 Bus

- IFC: Interface Clear;
  - Used only by the system controller to halt current operations
  - Placing all devices in an idle state
  - All talkers are set to Un-talk and all Listeners are set to Un-listen
  - Serial Poll is disabled
  - All devices must monitor IFC, and respond within 100μS
IEEE-488 Bus

◆ **REN: Remote Enable**
  - Used by the system controller to place devices in programming mode
  - All Remote capable Listeners are set to remote operation (if they were addressed to Listen), when REN is true (low)
  - When false, devices are set to Local control
  - All device must monitor the REN line, and respond within 100μS.
IEEE-488 Bus

- **SRQ: Service Request**
  - Used by any device to indicate that a device needs service
  - Any device may use this line
  - Normally used as an interrupt line
  - The controller may mask this line, in favor of Polling
  - The SRQ line is cleared by a Serial Poll
IEEE-488 Bus

☀ DIO1 to DIO8: Data Input-Output bus

☐ Bi-directional, Byte Serial-Byte Parallel.
☐ Per data line, bits are serial, ASCII data is sent as parallel data.
   Standard data is 7 bit ASCII
☐ But no coding format is defined with IEEE-488.
IEEE-488 Bus

- **IEEE 488.2 AND SCPI**
  - The SCPI and IEEE 488.2 standards addressed the limitations and ambiguities of the original IEEE 488 standard
- **IEEE 488.2** makes it possible to design more compatible and productive test systems
IEEE-488 Bus

- The ANSI/IEEE Standard 488-1975, now called **IEEE 488.1**, greatly simplified the interconnection of programmable instrumentation by clearly defining mechanical, electrical, and hardware protocol specifications.
- For the first time, instruments from different manufacturers were interconnected by a standard cable.
- Although this standard went a long way towards improving the productivity of test engineers, the standard did have a number of shortcomings.
- Specifically, IEEE 488.1 did not address data formats, status reporting, message exchange protocol, common configuration commands, or device-specific commands. As a result, each manufacturer implemented these items differently, leaving the test system developer with a formidable task.
IEEE-488 Bus

- **IEEE 488.2** enhanced and strengthened IEEE 488.1 by standardizing data formats, status reporting, error handling, Controller functionality, and common commands to which all instruments must respond in a defined manner.
- By standardizing these issues, IEEE 488.2 systems are much more compatible and reliable.
- The IEEE 488.2 standard focuses mainly on the software protocol issues and thus maintains compatibility with the hardware-oriented IEEE 488.1 standard.
IEEE-488 Bus

IEEE-488 Bus


Copyright © 2007 David Vernon (www.vernon.eu)
IEEE-488 Bus
Digitial Input/Output
Digital I/O

- Digital input is the simplest form of inputting signals created by external devices to a computer
- Digital output is the simplest form of outputting from a computer to external control devices
- Digital input/output (I/O) provides for the inputting of digital logic signals to the computer from ‘real-world’ and outputting of compatible logic signals to the ‘real-world’
- Typically uses an I/O Data Acquisition & Control (DA&C) board mapped on the PC bus to connect to switches, actuators, sensors, relays, lights, ...
Digital I/O

- Digital I/O can be achieved using the Programmable Peripheral Interface (PPI) device

  - Intel 8255 IC
  - 40 pins
  - Normally mapped into the isolated I/O memory map of a PC
  - TTL voltage compatibility (5.5V maximum input)
  - Maximum current source / sink: 1mA
Digital I/O

Connection to the system microprocessor is via the data bus (D0-D7), with handshaking lines:

- **RD** Read
- **WR** Write
- **A1** Address line to select I/O port or control register
- **A0**
- **RESET** low input on RESET initializes the device
- **CS** chip select (low to activate the device)
Digital I/O

[Diagram showing digital I/O connections]
Digital I/O

- 24 I/O lines, grouped into three groups of 8 bits
  - Port A (address 00 [A1A0] – BASE_ADDRESS)
  - Port B (address 01 – BASE_ADDRESS+1)
  - Port C (address 10 – BASE_ADDRESS+2)
    - Port C is divided into two parts: upper and lower
Functionality programmed using the 8-bit Control Register (address 11 – BASE_ADDRESS+3)

- D7  0    inactive
    1    active

- D4  0    Port A is an output port
    1    Port A is an input port

- D1  0    Port B is an output port
    1    Port B is an input port

- D0  0    Port C lower is an output port
    1    Port C lower is an input port

- D3  0    Port C upper is an output port
    1    Port C upper is an input port
Digital I/O
Digital I/O

Mode set active
0 – Inactive
1 – Active

Port A Mode
00 – Mode 0
01 – Mode 1
10 – Mode 2

Port A
0 – Output
1 – Input

Port C (upper)
0 – Output
1 – Input

Port B Mode
0 – Mode 0
1 – Mode 1

Port B
0 – Output
1 – Input

Port C (lower)
0 – Output
1 – Input
Port A can be operated in one of three modes (0, 1, 2) set by D6 & D5

- Mode 0 (D6 D5 = 00)
  - Simplest mode: no handshaking
  - Bits on Port C can be programmed as inputs or outputs
Digital I/O

- Port A can be operated in one of three modes (0, 1, 2) set by D6 & D5

  - Mode 1 (D6 D5 = 01)
    - Handshaking for synchronization (required when speed of devices differs significantly)
    - Port C is used for the handshaking signals
    - Input:
      - PC4: STB drive low to write data into Port A
      - PC5: IBF Input Buffer Full; automatically goes high when data has been written into Port A & remains high until data has been read

    - Output:
      - PC7: OBF Output Buffer Full Port A, automatically goes low when data is written to the port
      - PC6: ACK drive low to acknowledge data has been read (and OBF then goes high again)
Port A can be operated in one of three modes (0, 1, 2) set by D6 & D5

Example: Port A input, mode 1 handshaking
- External device placed data on Port A data lines
- External device asserts STB (i.e. 0, driven low)
- Data is latched on port A register
- 8255 asserts IBF (i.e. 1, driven high)

Example: Port A output, mode 1 handshaking
- CPU writes data to Port A
- OBF is asserted (i.e. 0, driven low)
- External device reads data on port A data lines
- External device asserts ACK (i.e. 0, driven low)
- OBF is cleared (i.e. 1, driven high)
Digital I/O

- Port A can be operated in one of three modes (0, 1, 2) set by D6 & D5
  - Mode 2 (D6 D5 = 10)
    - Bi-directional I/O
    - Port C is used for the handshaking signals
    - Input:
      - PC4 STB drive low to write data into Port A
      - PC5 IBF Input Buffer Full; automatically goes high when data has been written into Port A & remains high until data has been read
    - Output:
      - PC7 OBF Output Buffer Full Port A, automatically goes low when data is written to the port
      - PC6 ACK drive low to acknowledge data has been read (and OBF then goes high again)
Port B can be operated in one of two modes (0, 1) set by D2

- Mode 0 (D2 = 0)
  - Simplest mode: no handshaking
  - Bits on Port C can be programmed as inputs or outputs

- Mode 1 (D2 = 1)
  - Handshaking for synchronization (required when speed of devices differs significantly)
  - Port C is used for the handshaking signals
  - Input:
    - STB
    - IBF
  - Output:
    - OBF
    - ACK

- Input:
  - PC2
  - PC1

- Output:
  - PC1
  - PC2

Input:
- STB: drive low to write data into Port B
- IBF: Input Buffer Full; automatically goes high when data has been written into Port B & remains high until data has been read

Output:
- OBF: Output Buffer Full Port B, automatically goes low when data is written to the port
- ACK: drive low to acknowledge data has been read (and OBF then goes high again)
Digital I/O
Digital I/O

(Figures based on W. H. Rigby and T. Dalby, *Computer Interfacing*)

Copyright © 2007 David Vernon (www.vernon.eu)
Digital I/O

// ppi_2.c – read Port A and write same data to Port B

#define  BASE_ADDRESS 0x3B0  /* change this as required */
#define  PORTA BASE_ADDRESS
#define  PORTB (BASE_ADDRESS+1)
#define  PORTC (BASE_ADDRESS+2)
#define  CNTRL_REG (BASE_ADDRESS+3)
#include <stdio.h>
#include <time.h> /* included for term() */
#include <dos.h> /* included for inputb and outputb */

void my_delay(int secs);

int main(void)
{
    unsigned char i=0;
    outportb(CNTRL_REG, 0x90);  /* what does this do? */
    do
    {
        i = inportb(PORTA);
        outportb(PORTB, i);
        my_delay(1);  /* wait 1 second */
        printf(Input value is %d\n", i);
    } while (i != 0xff);
    return(0);
}
void my_delay(int secs)
{
    time_t oldtime, newtime;
    time(&oldtime);
    do {
        time(&newtime);
    } while ((newtime-oldtime)<secs);
}
Digital I/O

( Figures based on W. H. Rigby and T. Dalby, Computer Interfacing)

Copyright © 2007 David Vernon (www.vernon.eu)
Digital I/O

- Optical Isolation
  - Discrete Input Modules
    - Discrete I/O modules are installed as interfaces when discrete field devices are used to sense and control process variables. A discrete field device has only two functional states. These states may be on/off, open/closed, or true/false.
    - Typical discrete input devices include push buttons, limit switches, temperature switches, and pressure switches. Input field devices have different operating voltage requirements, such as 120 VAC, 24 VDC, and 5 VDC. Discrete input modules with different voltage ratings are used to handle the different voltage requirements of these input devices.
    - Regardless of the specific voltages involved, all input modules perform the same overall functions. They convert the input signals from the field devices to low-level DC signals the CPU can use. And, they electrically isolate the processor from the higher input voltages.
Digital I/O

- **Optical isolation (coupling)**
  - isolates the processor circuitry from the input circuitry
  - the input module provides the CPU with a low-level DC signal which indicates the status of the input device.
  - Any high voltage spikes which might occur at the input side will thus be prevented from reaching the processor and damaging it.
Digital I/O

- The signal coming from the field input device is converted to a suitable level and filtered, then it passes through an optical isolation unit which consists of a light-emitting diode and a phototransistor.
- If the signal level is sufficient to represent an on input status (i.e. when the field device is providing a signal to the input interface), the LED will turn on.
- If the input signal voltage is too low, the LED will turn off.
- When the LED turns on, the phototransistor turns on. This on signal is provided to the processor, which records the status of the input bit as 1.
- When the LED turns off, the phototransistor will turn off. The status of the input bit will then be changed to 0.
Digital I/O

◆ Discrete Output Modules

- One function of an output module is to convert the low-level DC output signals from the processor into appropriate voltage signals that switch the output devices on or off.
- Typical discrete output devices include solenoids, motor starters, and alarm indicators.
- Since different output devices have different voltage requirements, output modules are available with different AC and DC voltage ratings.
- There are a variety of output modules. Two types that are commonly used are DC output modules and AC output modules. Only the DC output module will be discussed here as an example.
Digital I/O

**Discrete Output Modules**

- A DC output module converts the output from the processor to the desired DC voltage signal that turns the output device on or off.
Digital I/O

- This circuit uses optical coupling to isolate the processor circuitry from the output device circuitry
- The power transistor acts as a switch, turning the output circuit on and off in response to signals from the processor
- For example, when the LED in the processor circuit receives an ON signal, it turns on
  - This causes the phototransistor to complete the circuit to the power transistor
  - This activates the power transistor and switches the output circuit on, signalling the output device to activate.
- When the signal in the processor circuit goes off, the LED goes off
  - This turns off the photo-transistor
  - As a result, the power transistor turns off, and the output circuit is de-energised.
Analogue I/O
Analogue Input

- Most sensors produce an analogue electrical signal
  - DC voltage 0-10v, for example
  - DC current 4-20mA, for example
  - AC voltage, variable frequency
- Example: Thermistor temperature sensor
  - Resistance inversely proportional to temperature
  - 10k ohms at 25 degrees centigrade

( Figures based on W. H. Rigby and T. Dalby, Computer Interfacing)
Analogue Output

8-Bit A/D Converter
Input Voltage Range -5 to +5 Volts

<table>
<thead>
<tr>
<th>Voltage</th>
<th>Binary Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
<td>0000 0000</td>
</tr>
<tr>
<td>0</td>
<td>1000 0000</td>
</tr>
<tr>
<td>+5</td>
<td>1111 1111</td>
</tr>
</tbody>
</table>

Copyright © 2007 David Vernon (www.vernon.eu)
Analogue Input

Analogue-to-Digital Conversion

- Converts analogue signals (mainly voltage) to a digital signal compatible with the TTL voltage required internally by the PC.

- Resolution of ADCs depend on the number of bits used and the range of the analogue input. There are $2^n-1$ levels for an n-bit ADC:

$$\text{resolution} = \frac{V_{\text{max}} - V_{\text{min}}}{2^n-1}$$

$$= \frac{5 - (-5)}{2^8-1}$$

$$= 0.0392 \text{ volts}$$
Analogue Input

- Analogue-to-Digital Conversion

- It is important to choose an ADC with the proper resolution.

- If the analogue voltage changes by an amount which is less than the resolution then it will not be detected, and the computer will not know about the change.

- Cost is a major factor when selecting the resolution.
Analogue Input

- Three types of Analogue-To-Digital Converters are commonly used in DA&C boards
  - **Dual-Slope converters**: simplest and least expensive (slow ... 10-100ms)
  - **Successive approximation converters**: most popular converter, faster than DS
  - **Flash converters**: very fast and expensive
Analogue Input

- Sample and Hold Circuits
  - Used to stabilise the changing analogue signal while the conversion process is taking place
  - A typical technique is to include a capacitor which is charged to the value of the input analogue voltage
Analogue Input

- Multiplexed-inputs
  - This allows more than one channel (more than one analogue input) to be connected to the ADC
  - Typically 8 to 16 channels are multiplexed on DA&C boards.
Analogue Input

- In order to program any DA&C card, you need to know the register structure of the card and the base address.

- Then you simply follow the manufacturer’s programming procedure which involves writing to and reading from specific registers.

- And you know how to do this in C (outportb and inportb).
Analogue Output

Digital-to-Analogue Conversion

- Convert a value represented by a discrete (integer) variable to a corresponding analogue quantity.

Again, resolution is important: there are $2^{n-1}$ levels for an n-bit DAC; e.g. 8-bit DAC, 5V voltage range:

Resolution: $\frac{V_{\text{max}} - V_{\text{min}}}{2^{n-1}}$

= $\frac{5 - 0}{(2^8-1)}$

= 0.0196 volts per step
Analogue Output

- **Digital-to-Analogue Conversion**
  - Programming a DAC is done in a similar way to ADC
  - A data acquisition and control card typically contains both ADC and DAC
  - Programming the DAC is achieved through writing to and reading from appropriate registers
LabJack USB AD&C: www.labjack.com