Peripheral Devices

A peripheral device is any device that sends data to or receives data from the CPU. (Keyboard, mouse, monitor, printer, disk, etc.)

Some I/O devices are what we call *Human Interface Devices (HIDs)*. Such devices must be able to input and output data in a form suitable for humans. (Text, sound, motion, etc.)

ASCII (American Standard Code for Information Interchange) is the original 7-bit character encoding standard. It assigns a 7-bit binary code (not a number!) to each letter of the alphabet, digit, punctuation symbol, etc. It is used to input and output data in human-readable form.

```
0-31    Control chars.  Originally designed to control text-only
        printers.  Adopted by ASCII terminals (vt100, xterm)
        for analogous functions.

4       EOT     (ctrl+d)
7       BEL     (ctrl+g)    \007    Beep printer/terminal
8       BS      (ctrl+h)    \b      Move head/cursor left
                                   May shift rest of line on terminal
9       TAB     (ctrl+i)    \t      Move head/cursor to next tab stop
10      LF      (ctrl+j)    \n      Move head/cursor down (scroll)
12      FF      (ctrl+l)    \f      Scroll to start of next page
13      CR      (ctrl+m)    \r      Move head/cursor to col 1
```

Terminals: CR+LF goes to start of new line. Unix adds CR to each LF by default, so we write "Hello, world!\n" instead of "Hello, world!\n\r".

```
32-126  Printable characters.  Print/display char and move
        head/cursor right.  Wrap at eol of printer/terminal
        supports it.

32      Space
33      !
...
48      '0'
49      '1'
...
65      'A'
66      'B'
...
97      'a'
98      'b'

127     DEL     Delete char under cursor
```

```
Examples:

    char    ch = 'A';

    ch:     .byte   'A'

    What is in the variable ch? (binary)
```

ISO (International Standards Organization, now the International Organization for Standardization, extended the ASCII set to include non-English characters, graphic symbols, etc. The basic Latin-based ISO sets are 8 bits.

All ISO character sets are backward-compatible with ASCII, i.e. the character codes for the first 128 characters are the same as ASCII.

Some common ISO character sets: ISO-latin1, ISO-latin2

Unicode Transformation Format (UTF) is a set of multibyte character encodings that extend beyond 8 bits. UTF-8 is also backward-compatible with ASCII, and is the dominant character set used on the WEB.

Input-Output Interface

It would not be practical for every I/O device to be wired to the computer in a different way, so we must have a scheme where the hardware connections are fixed, and yet the communication with the device is flexible, so that the widely varying needs of devices can all be met.

An I/O device, from the viewpoint of the CPU, is a set of registers. The CPU communicates with and controls the I/O device by reading and writing these registers. For example, SPIM, the MIPS simulator, uses two registers to communicate with the keyboard.

- The keyboard data register contains the ASCII code of the last key pressed.
- The keyboard control register indicates when a new key has been pressed. If bit 0 is one, a key has been pressed since the last character was read. The keyboard controller sets this bit when a key is pressed. It clears this bit when the keyboard data register is read.

The CPU can find out whether a new character is available by reading the keyboard control register and testing bit 0. If bit 0 is 1, it then reads the keyboard data register to get the new key.

Accessing I/O devices at the hardware level is a lot like accessing memory. The registers in the I/O devices are connected to the CPU using buses. We need an address bus to specify which I/O device register is to be accessed. We need control lines to specify what kind of access is desired (read, write, reset, etc.) Finally, we need a data bus to transfer the data between the CPU and the device.

Each device has one or more control, status, and data registers at various I/O addresses. A hypothetical example:

```
Address Register
ff00    keyboard status
ff01    keyboard data
ff02    display status
ff03    display data
ff04    disk status
ff05    disk block address
ff06    disk block size
ff07    disk data address
...
```

I/O read and write operations can be more complex than memory read and write operations, but the basic idea is the same. I/O control generally involves more than just read and write control lines. In a sense, memory can be viewed as a very simple, fast I/O device.

Whereas memory is just a large pool of slow, inexpensive registers for storing data, each I/O device register has a unique purpose in controlling a specific I/O device. This does not affect how the CPU accesses them at the hardware level, but it does affect how they are used by software.

Simple device control, such as stating whether an I/O register is to be read or written, can be done over the control lines. More complex devices are often controlled by sending special data blocks called *Peripheral Control Blocks (PCBs)* over the data lines. This is the primary method for communicating with disk drives, for example.

Since I/O devices are of a very different nature than CPU circuits, there must be interface hardware to connect each device to the CPU.

Methods for designing a CPU's I/O interface generally fall into one of the following categories:

- Completely separate memory and I/O buses
- Common buses, separate control lines
- Common buses and control lines

Regardless of the CPU's interface, the connections to the I/O device controllers are the same. In fact, the same devices are often used on multiple architectures with vastly different CPUs, e.g. in an x86 PC and a PowerPC Mac.

What is the advantage of using separate I/O and memory buses?

What is the advantage of using common buses for memory and I/O?

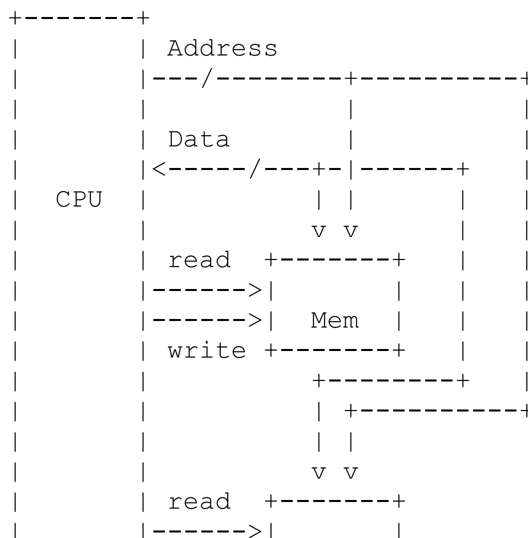
Peripherals are usually much slower than CPUs (keyboards, disks, printers), but are occasionally faster (high-speed networks, some video controllers). The high speed of some of today's I/O devices has driven the need for PCI-E serial network buses to replace older parallel buses like ISA, PCI, PCI-X, AGP.

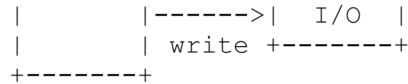
Isolated vs. Memory-mapped

If memory and I/O share address and data bus, there are two ways to distinguish them.

- In isolated I/O, common address and data bus lines are used to address both memory and I/O devices. Separate read/write control lines enable either memory or an I/O device, both never both, to accept the address. This scheme allows an I/O device and memory to use same address, since only one of them will be activated during any given clock cycle. In other words, memory and I/O time-share the buses, and have separate memory spaces. A given address can refer to a memory cell at one time and to an I/O device at another.

  Systems using isolated I/O and systems with separate memory and I/O buses have distinct I/O instructions that activate the I/O read or write control lines. Memory reference instructions activate the memory read or write control lines.





- In *memory-mapped I/O*, there is only one address space, and it is divided between memory cells and I/O device registers.

```
Address Purpose
0000    Memory
...     ...
fff0    Memory

fff1    Keyboard control register
fff2    Keyboard data register
fff3    Disk control register
...
```

Any given address is wired to either to a memory cell or an I/O device register, and cannot refer to anything else.

In memory-mapped I/O systems, there are no distinct I/O instructions. The CPU does not distinguish between memory access and I/O operations. The only difference between them is the address! Hence, we can used load and store instructions to access the registers in I/O devices on a load-store architecture. We can use almost any instruction (move, add, sub, etc.) to perform I/O on a memory-to-memory or register-memory computer.

Asynchronous Data Transfer

The internal operations of a CPU are synchronized by a common clock.

Generally, it is not possible, or at least not practical to synchronize I/O devices with the CPU's internal clock.

I/O devices operate at different speeds than the CPU (note that the same I/O device may be connected to computers with vastly different CPU speeds).

Data transfer links such as USB, Firewire, and Ethernet, operate on a clock that is separate from the CPU's internal clock, but common to devices at both ends of the link.

The operations of an I/O interface must somehow be synchronized with the CPU's activities. This is often done using *handshaking* signals. The I/O device and CPU set

wires or flip-flops to 0 or one to indicate their current state. (Recall the Mano FGI and FGO flags.)

The sender on an RS-232 serial interface sets a wire known as RTS (ready to send) to indicate that it has data to send. The receiver responds by by setting the CTS (clear to send) wire to indicate that it's OK to begin sending the next byte.

Newer interfaces such as USB, Ethernet, etc. use more complex setups, but basic 2-wire handshaking is still present in many modern interfaces.

Asynchronous Data Transfer

The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator. Clock pulses are applied to all registers within a unit and all data transfers among internal registers occur simultaneously during the occurrence of a clock pulse. Two units, such as a CPU and an I/O interface, are designed independently of each other.

If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers.

In that case, the two units are said to be asynchronous to each other. This approach is widely used in most computer systems. Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted. One way of achieving this is by means of a strobe pulse supplied by one of the units to indicate to the other unit when the transfer has to occur.

Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another control signal to acknowledge receipt of the data. This type of agreement between two independent units is referred to as handshaking.

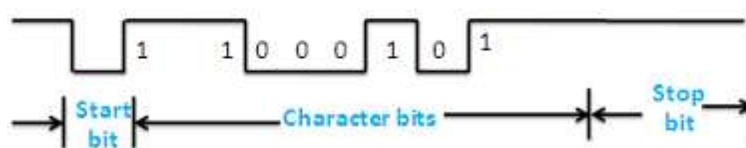
The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers. In fact, they are used extensively on numerous occasions requiring the transfer of data between two independent units. In the general case we consider the transmitting unit as the source and the receiving unit as the destination.

- For example, the CPU is the source unit during an output or a write transfer and it is the destination unit during an input or a read transfer. It is customary to specify the asynchronous transfer between two independent units by means of a timing diagram that shows the timing relationship that must exist between the control signals and the data in the buses. The

sequence of control during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination unit.

Asynchronous Serial Transfer

The transfer of data between tow units my be done in parallel or serial. in parallel data transmission, total message is transmitted at the same time. In serial data transmission, each bit in the message is sent in sequence one at a time. In asynchronous transmission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted.



Asynchronous serial transmission

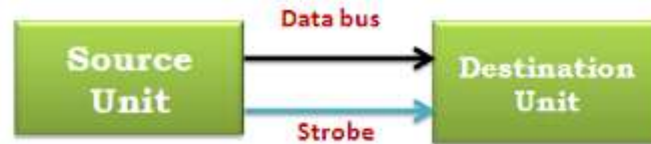
Asynchronous serial transmission is character oriented. Each character transmitter consists of a start bit, character bits, and stop bits. The first bit is called the start bit. It is always a 0 and is used to indicate the beginning of a character. The last bit called the stop bit is always a 1.

Strobe Control

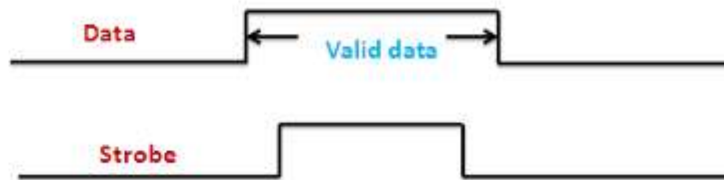
This method of asynchronous data transfer uses a single control line to time each transfer. The strobe may be activated by the source or the destination unit.

(i) Source Initiated Data Transfer:

- The data bus carries the information from source to destination. The strobe is a single line. The signal on this line informs the destination unit when a data word is available in the bus.
- The strobe signal is given after a brief delay, after placing the data on the data bus. A brief period after the strobe pulse is disabled the source stops sending the data.



a) Block diagram



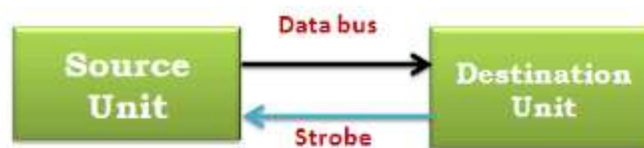
b) Timing diagram

Source - initiated strobe for data transfer

(ii) Destination Initiated Data Transfer:

- In this case the destination unit activates the strobe pulse informing the source to send data. The source places the data on the data bus. The transmission is stopped briefly after the strobe pulse is removed.

- The disadvantage of the strobe is that the source unit that initiates the transfer has no way of knowing whether the destination unit has received the data or not. Similarly if the destination initiates the transfer it has no way of knowing whether the source unit has placed data on the bus or not. This difficulty is solved by using hand shaking method of data transfer.



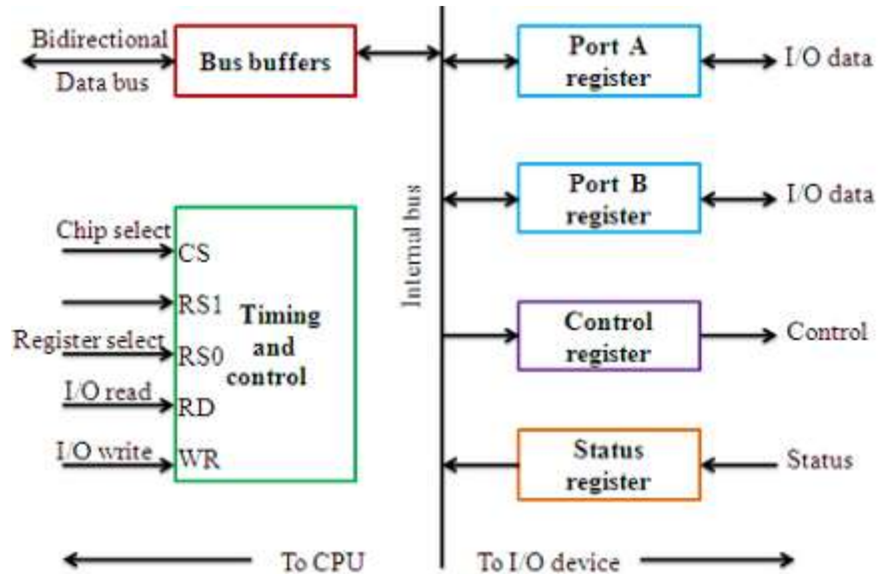
a) Block diagram

b) Timing diagram

Destination - initiated strobe for data transfer

Example of I/O Interface

An example of an I/O interface unit is shown in figure. It consists of two data registers called ports, a control register, a status register, bus buffers and timing and control circuits.



The four registers communicate directly with the I/O device attached to the interface. The I/O data to and from the device can be transferred into either port A or port B. Port A may be defined as an input port and port B may be defined as an output port. The output device such as magnetic disk transfers data in both directions. So bidirectional data bus is used. CPU gives control information to control register. The bits in the status register are used for status conditions. It is also used for recording errors that may occur during the data transfer. The bus buffers use the bidirectional data bus to communicate with the CPU. A timing and control circuit is used to detect the address assigned to the bus buffers.

CS	RS1	RS0	Register selected
0	X	X	None: data bus in high-impedance
1	0	0	Port A register

1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

Modes of transfer

Three Possible mode:

Data transfer between central computer and I/O devices may be handles in a variety of modes. The three possible modes are

1. Programmed I/O
2. Interrupt-initiated I/O
3. Direct Memory Access (DMA)

Programmed I/O

- Programmed I/O operations are the result of I/O instructions written in computer program. Each data item transfer is initiated by an instruction in the program. The I/O device does not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instructions by the CPU. The data transfer can be synchronous or asynchronous depending upon the type and the speed of the I/O devices.

- If the speeds match then synchronous data transfer is used. When there is mismatch then asynchronous data transfer is used. The transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. This method requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated the CPU is required to monitor

- The interface to see when a transfer can again be made. In this method the CPU stays in a loop till the I/O unit indicates that it is ready for data transfer. This is time consuming process which can be solved by using interrupt.

Interrupt initiated I/O

In the programmed I/O method, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly.

It can be avoided by using an interrupt facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device.

In the meantime the CPU can proceed to execute another program. The interface meanwhile keeps monitoring the device. When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer.

Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing.

Example of Interrupt initiated I/O:

1. Vectored interrupt
2. Non vectored interrupt

Vectored interrupt :

In vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the interrupt vector.

Non vectored interrupt

In a non vectored interrupt, the branch address is assigned to a fixed location in memory.

Priority Interrupt

A priority interrupt is a system that determines, which condition is to be serviced first when two or more requests arrive simultaneously. Highest priority interrupts are serviced first. Devices with high speed transfers are given high priority and slow devices such as keyboards receive low priority. When two devices interrupt the computer at the same time the computer services the device, with the higher priority first. Establishing the priority of simultaneous interrupts can be done by software or hardware.

Daisy-Chaining Priority

The hardware priority function can be established by either a serial or a parallel connection of interrupt lines. The serial connection is called daisy chaining method.

In daisy chaining method all the devices are connected in serial. The device with the highest priority is placed in the first position, followed by lower priority devices.

Direct Memory Access (DMA)

Imagine a computer can execute 3 billion instructions per second. Suppose it is connected to the network with a 10-gigabit network interface. 10 gigabits of data represents a little more than 1 gigabyte when considering 8-bit characters, plus protocol overhead such as error checking/correction, packet headers, etc.

If this interface generated an interrupt for each character sent or received, the CPU would be interrupted about 1 billion times per second, or every third instruction cycle. Since each interrupt must be serviced by an interrupt service routine, what is the implication here?

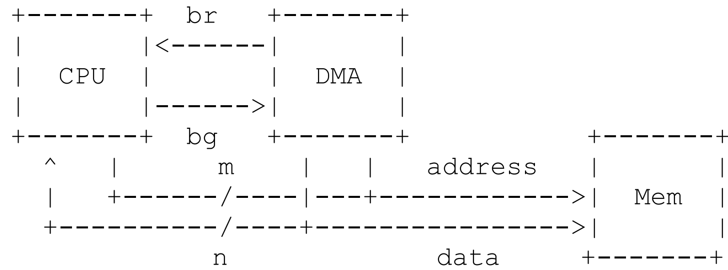
Obviously, such high-speed devices cannot be serviced one byte or word at a time. We must have a way to transfer a large block of data between the device and memory with intervention of the CPU for every byte.

DMA is a hardware solution that allows blocks of data to be transferred between memory and an I/O device with no involvement of the CPU. It is often used in conjunction with interrupts, where one interrupt is generated for each block transferred rather than each byte transferred.

Not necessarily for increasing I/O performance, but for reducing CPU load.

The size of the block transferred may vary, but in any case DMA reduces the number of interrupts generated by at least an order of magnitude, and reduces the amount of work the ISR must perform, since the actual data transfer has already been performed by hardware.

When a DMA transfer occurs, the CPU relinquishes access to memory, and the *DMA controller* associated with the I/O device becomes the *bus master*. To initiate a transfer, the DMA controller issues a *bus request*, usually by asserting a wire to to the CPU (like FGI in the Mano machine). When the CPU is ready to relinquish the memory buses, it responds with a *bus grant*, usually by asserting a wire telling the DMA controller it's OK to proceed.



The CPU and DMA controller will typically keep themselves disconnected from the buses using 3-state buffers when they are not the bus master.

Transfers may occur as a *burst*, where a large block of data is transferred all at once (and the CPU is locked out of memory access for many memory cycles), or for individual memory cycles dispersed over time. The latter scenario is known as *cycle stealing*.