For the computer to be of use to us it must be able to obtain data from the outside world, process it (this is the part we have been discussing), and then communicate the information back to the outside world.

We will start by looking at the following question:

How does the CPU implement input and output transactions?

A partial answer is it doesn’t do all of the work itself. Instead the fine details of I/O are handled by special purpose interface chips. These chips handle the transfer of data to the external device which then does further processing before the information is presented to the end user.

**Input / Output Fundamentals**

We divide I/O into three areas:
1. Strategy
2. Interface circuitry
3. I/O device – the peripheral.

The output device is the CRT. It accepts a form of serial communication and needs to reconstruct a parallel signal used, for instance, to select a character from a table of symbols.

We can visualize the discussion so far as:

**Relationship between computer and peripheral**

Example: Consider the example of a computer connected to a CRT and a Keyboard.

Input and output is done via a strategy called programmed data transfer – when we want data an instruction reads data from the input port the keyboard is connected to. When the computer wishes to display information data is written to the output port that communicates with the CRT.

A port is thought of as a gateway between the computer and an external device.

How the data is actually moved between the CPU and the external device is handled by the interface circuitry between the two.

In our example data is being sent from the CPU to the CRT by way of the computers output port. This port is itself a sophisticated package of circuitry. The connection between port and the CRT may consist of nothing more than a twisted-pair (two parallel wires twisted at regular intervals). Typically data is written from CPU to the output port in parallel form at 8/16/32 bits at a time. The output port must serialize the information and transmit it bit by bit over the twisted pair to the CRT. In addition, the CRT and output port must synchronize their communications.

The CPU views the interface circuitry (chip) as a memory location which data can be written to or read from. The chip contains specialized logic which allows for communication with the CRT.

A second example: consider a block of data needing to be written to a disk drive. Again we start by looking at the strategy. Programmed data transfer is probably not the scheme used. There is no real need to load the data from memory into the CPU registers and then write to a port. Instead what is often used is a Direct Memory Access (DMA) strategy.
The CPU instructs special purpose DMA hardware to move the block of memory, thereby freeing the CPU to continue with its calculations.

The interface circuit is a chip called a DMA Controller (DMAC). The DMAC is responsible for:

- providing the addresses for source or destination of data,
- signaling the peripheral data is needed or data is ready,
- coordinating with the CPU for control of data and address buses during the data transfer.

The peripheral, the disk drive is a complex mixture of electronic and high-precision mechanical devices. Data is stored by affecting magnetic properties of the surface of a high-speed rotating disk.

**Handshaking and buffering**

The process of two devices communicating is itself a complex task. It must involve making sure the devices are indeed communicating with each other and must take into account the vast differences in the speed between devices.

All data transfers can be grouped into one of two classes:

- Open loop
  - Data is sent and reception is assumed
- Closed loop
  - Data is sent and an acknowledgement of receipt is returned.

**Open loop data transfer:**

- **Point A** – Data from the computer becomes valid
- **Point B** – The computer asserts a control signal called DAV (data valid)
  
  The peripheral can start its read of data at this point. It must complete its read by the time point D is reached, when the data vanishes.
- **Point C** – DAV is negated. This indicates to peripherals that the data is no longer valid.
- **Point D** – Data vanishes.

Notice in this open-loop transfer the peripheral does not communicate with the CPU. There is no indication that data was received.

**Timing diagram for Open Loop:**

![Timing diagram for Open Loop](image)

**Closed-loop data transfer:**

Next we discuss **Closed-Loop** data transfer.

Many devices will communicate information back to the CPU indicating they have received data or are ready to receive data. The device that initiates the data transfer is often referred to as the master and the device addressed by the master is often called the slave.

The master and slave communicate via a sequence of signals indicating data ready, data accepted, data no longer valid. The sequence of communications is called **handshaking**. Handshaking can be used to deal with the difference in speed of the master and slave.

**Timing diagram for Handshaking:**

![Timing diagram for Handshaking](image)
A view of the handshaking arrangement:

The handshaking process can be extended to have the devices acknowledge the de-assertion of signals (we refer to this strict sequence of steps as delay insensitive since a new step can not take place without the proper acknowledgements of activations and deactivations of signals).

We will call such a data transfer a fully interlocked data transfer.

Typically the handshaking sequence is carried out by special-purpose hardware. The CPU only gets involved when an error occurs. When do errors occur and how are the detected?

- When the transmitter wants to send data it starts a timer concurrently with the DAV signal. If no DAC signal is transmitted by the receiver in a given interval of time the operation is aborted. The period of time between the timer start and the declaration of failure is called a timeout.

Buffered I/O

A typical problem with input and output is devices tend not to operate at optimal speeds. It is often the case that data items arrive faster than they can be processed. Rather than ignore or lose the data it can be loaded into a special device called FIFO memory (First In First Out). As data arrives at the input port it is stored in order of arrival. The designer of the port is responsible for supplying sufficient memory to handle worse-case input bursts.

The process of saving data in a store until it is needed is called buffering.

FIFO – the data structure used to implement a FIFO is called a queue. Typically a queue will grow from lower addresses to higher addresses. New data are added at higher addresses while data is retrieved at the lower addresses. Thus as described so far, a queue would tend to move through memory toward higher and higher addresses. For a variety of reasons this is not practical. The way to limit a queue to a fixed region of memory is to use a circular buffer. Through the use of pointers we keep track of where the beginning and end are. When we hit the end of the region allocated for the queue we move the end pointer back to the beginning address. One would think this would be a problem because isn’t this where the start pointer is?

Probably not since we have been processing data on the queue the start pointer was moving up in memory at the same time the end pointer was moving.
Programmed I/O

The term *programmed I/O* is used for input or output that is initiated by an instruction in the program such as:

\[
\text{MOVE.B keyboard,D0}
\]

Which reads a byte of data from the Keyboard into D0.

Some processors have special instructions used only for I/O:

\[
\text{OUT 123}
\]

might be an instruction which moves the contents of a data register to the data bus while placing 123 (01111011) on the eight least significant bits of the address bus and generating a pulse on the systems I/O write line. If an I/O interface sees its own address together with a read-port or write-port signal it executes an I/O data transfer.

We can visualize a simplified version of this set up:

![Simplified I/O Interface Diagram](image)

Many common processors lack I/O instructions. Instead the I/O devices and the memory share the same address space. Interface ports are treated as an extension to main memory. Thus part of the CPU’s normal memory space is dedicated to I/O operations. We call this arrangement the *Memory-Mapped I/O* model.

A typical access of I/O ports would look like:

\[
\text{MOVE IO_PORT,D0}
\]

Or

\[
\text{MOVE D0,IO_PORT}
\]

A Hypothetical Example Program:

\[
\begin{align*}
&* \text{ For I=1 to 256} \\
&* \text{ Move data from Table_I to output_port} \\
&* \text{End For} \\
&* \\
&\text{PORT EQU $008000} \quad \text{Location memory-mapped port} \\
&\text{CNT EQU 256} \quad \text{size of block to move} \\
&\text{ ORG $000400} \quad \text{origin of program} \\
&* \\
&\text{MOVE #$CNT,D1} \quad [D1] \leftarrow 256 \\
&\text{LEA TAB,A0} \quad [A0] \leftarrow \text{TAB} \\
&\text{LEA PORT,A1} \quad [A1] \leftarrow \text{PORT} \\
&* \\
&\text{LOOP MOVE.B (A0)+,D0} \quad [D0] \leftarrow [M([A0])] \\
&* \quad [A0] \leftarrow [A0]+1 \\
&\text{SUB #1,D1} \quad [D1] \leftarrow [D1]-1 \\
&\text{BNE LOOP} \quad \text{Repeat till done} \\
&* \\
&\text{ORG $002000} \quad \text{Origin of Data} \\
&\text{TAB DS.B 256} \quad \text{reserves 256 bytes}
\end{align*}
\]

As far as the computer is concerned, sending data to a monitor is a matter writing data to memory address $008000$.

A typical Output port would occupy two or more memory locations. One holds the actual data while another might hold a status byte (bit) which could indicate port ready or port busy.

Suppose we have an 8 MHz machine (The Macintosh 128K released January 1984 had an 8 MHz 68K processor).

Discussion of clock cycles
- The clock defines regular time intervals called cycles. In order to execute a machine instruction the processor breaks it into a sequence of steps such that each step can be performed in one clock cycle.

\[\text{Clock rate} = 1 / \text{length of cycle}\]
Cycles per second are called hertz (Hz)

1 million cycles per second is 1 Megahertz (MHz)
1 billion cycles per second is 1 Gigahertz (GHz)

Now from the definition of clock rate (Clock rate = 1 / length of cycle)
we see the time for one cycle would be = 1 / Clock rate.
Thus for our 8MHz machine, each cycle takes:
1/8,000,000 seconds or 0.000000125 or 0.125 \times 10^{-6} seconds

ah that’s just 1/8 microsecond.

Now looking at our code, one pass through the loop requires
8 + 8 + 8 + 10 = 34 cycles
an we are making 256 passes through the loop for a total of
34\times256 = 8704 cycles

Now if each cycle takes 0.125 \times 10^{-6} seconds the elapsed time to move
the table will be
0.125 \times 10^{-6} \times 8704 \approx 0.001088 seconds or 1.088 milliseconds

Data is being transferred at the
34\times0.125 \times 10^{-6} \text{ seconds per character (4.25 microseconds)}

The problem with our example assembly language program is that
most devices connected to the output port are SLOW. If we write
data at this rate in a continuous stream, much of the data would be
lost (once we overflowed the buffer).

To accommodate the speed differential between computer and peripheral
a form of software handshaking is set up (recall we indicated earlier that an
I/O port would occupy more than one memory location – it's time to use the
status byte!)

Modified hypothetical program:
* For I=1 to 256
  * Repeat
    * read Port_Status_byte
    * Until Port_Not_Busy
    * Move data from Table_I to output_port
  * End For

PORT EQU $008000 Location memory-mapped port
PORTS EQU $008002
CNT EQU 256 size of block to move
ORG $000400 origin of program

MOVE #CNT,D1 \[D1\]
LEA TAB,A0 \[A0\]
LEA PORT,A1 \[A1\]
LEA PORTS,A2 \[A2\]

LOOP MOVE.B (A0)+,D0 \[D0\]
     AND.B #1,D2
     BEQ WAIT \[D2\] ← [M\[A2\]]
     MOVE.B D0,(A1) \[M\[A1]\] ← D0
     SUB #1,D1 \[D1\] ← [D1] - 1
     BNE LOOP \[D1\] ← [D1] - 1

* ORG $000200 Origin of Data
TAB DS.B 256 reserves 256 bytes

The lines in
WAIT MOVE.B (A2),D2 \[D2\] ← M\[A2\]
AND.B #1,D2 \[D1\] ← [D1] - 1
BEQ WAIT \[D1\] ← [D1] - 1

Constitute what we call a polling loop – the output device is continuously
polled until it signals it is free and then the program continues.

Consider the following analysis: (2001)
Okidata ML-520 Dot Matrix printer $289.00, 430 CPS

430 * 10^6 characters/second = 4.3 \times 10^4 characters/microsecond

or about 2326 microseconds/character. We can calculate on our 8MHz
machine that the loop would consume 3.25 microseconds/Loop.

2326 microseconds/character \times 3.25 microseconds/Loop = 715.7 Loops/Character

Clearly this is extremely inefficient!

Fortunately, in a multiprogramming environment other tasks can be done while
we wait for the printer. A strategy often employed is to ignore the printer until it
sends a request for attention. Such a strategy is called interrupt-driven I/O.