The Outside World

What we have described so far is the whole computer. It has two parts, the RAM and the CPU. That's all there is. These simple operations are the most complicated things that a computer can do. The ability to execute instructions, modify bytes with the ALU, the ability to jump from one part of the program to another, and most importantly, the ability to jump or not jump based on the result of a calculation. This is what a computer is able to do. These are simple things, but since it operates so quickly, it can do huge numbers of these operations that can result in something that looks impressive.

These two parts make it a computer, but if all the computer could do is run a program and rearrange bytes in RAM, no one would ever know what it was doing. So there is one more thing that the computer needs in order to be useful, and that is a way to communicate with the outside world.

Dealing with anything outside of the computer is called 'Input/Output' or 'I/O' for short. Output means data going out of the computer; Input means data coming into the computer. Some things are input only, such as a keyboard, some things are output only, like a display screen, some things do both input and output, like a disk.

All we need for I/O is a few wires, and a new instruction.

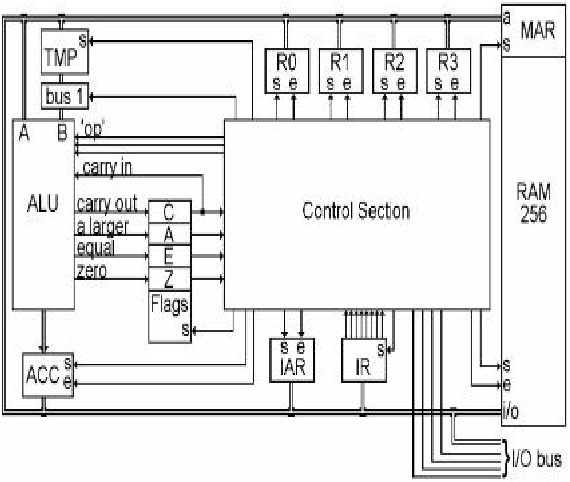
For the wires, all we are going to do is to extend the CPU bus outside of the computer and add four more wires to go with it. This combination of 12 wires will be called the I/O Bus. Everything that is connected to the computer is attached to this one I/O bus.

The devices that are connected to the I/O bus are called 'peripherals,' because they are not inside the computer, they are outside of the computer, on its periphery (the area around it.)

More than one thing can be attached to the I/O bus, but the computer controls the process, and only one of these things is active at a time.

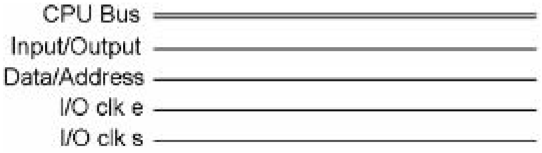
Each thing attached to the I/O bus has to have its own unique I/O address. This is not the same as the addresses of the bytes in RAM, it is just some 'number' that the peripheral will recognize when placed on the bus.

Here is what the I/O bus looks like in the CPU, there at the bottom right of the drawing.

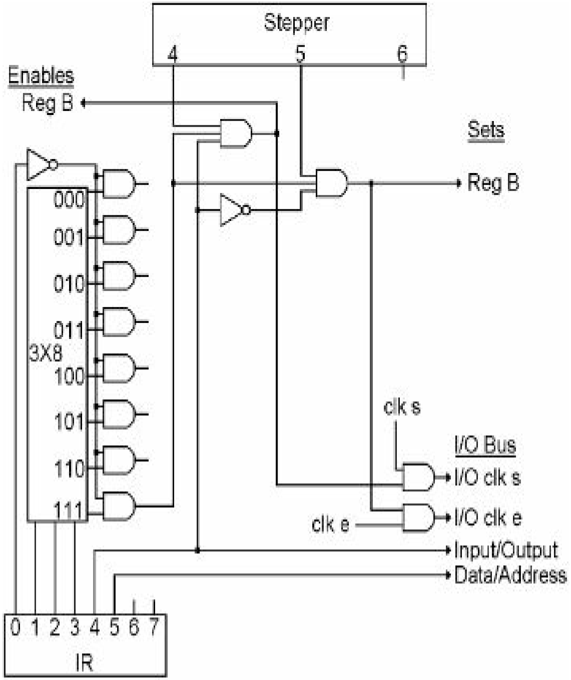


In the diagram below are the wires of the I/O Bus. The CPU Bus is the same eight-wire bundle that goes everywhere else. The 'Input/Output' wire determines which direction data will be moving on the CPU bus, either in or out. The 'Data/Address' wire tells us whether we will be transferring a byte of data, or an

I/O Address that selects one of the many devices that could be attached to the I/O bus. 'I/O Clk e' and 'I/O Clk s' are used to enable and set registers so that bytes can be moved back and forth.



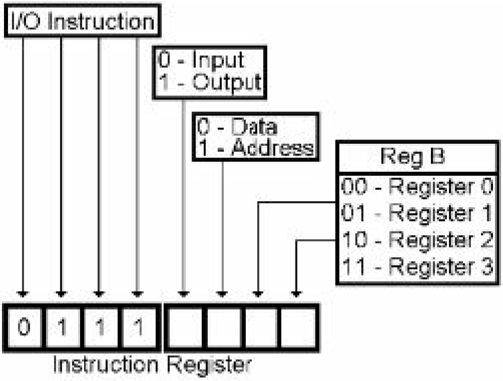
Here is the control section wiring for the new instruction that controls the I/O bus. This shows where the four new wires for the I/O bus come from. They are at the bottom right of the drawing. They were also shown on the full control section diagram a few chapters back. Sorry if that was confusing, but having that diagram in the book once was enough.



IR bits 4 and 5 are placed on the I/O bus at all times.

To make the I/O operation happen, only one step is needed. For Output, Reg B is enabled, and I/O Clk s is turned on and off during step 4. Steps 5 and 6 do nothing. For Input, I/O Clk e is enabled, and Reg B is set during step 5. Steps 4 and 6 do nothing.

Here is the Instruction Code for the I/O instruction:



This one instruction can be used in four different ways depending on IR bits 4 and 5, and therefore there are four new words for our language.

Language		Meaning
IN	Data,RB	Input I/O Data to RB
IN	Addr,RB	Input I/O Address to RB
OUT	Data,RB	Output RB to I/O as Data
OUT	Addr,RB	Output RB to I/O as Address

Each I/O device has its own unique characteristics, and therefore needs unique parts and wiring to connect it to the I/O bus. The collection of parts that connects the device to the bus is called a "device adapter." Each type of adapter has a specific name such as the 'keyboard adapter' or the 'disk adapter.'

The adapter does nothing unless its address appears on the bus. When it does, then the adapter will respond to the commands that the computer sends to it.

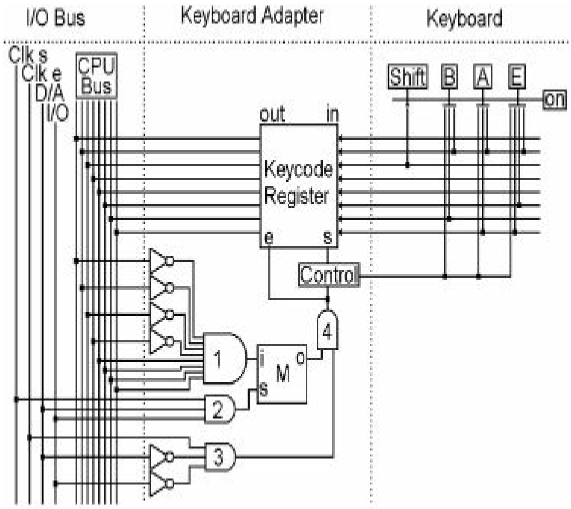
With an 'OUT Addr' instruction, the computer turns on the address wire, and puts the address of the device it wants to talk to, on the CPU bus. The peripheral recognizes its address and comes to life. Every other peripheral has some other address, so they won't respond.

We are not going to describe every gate in the I/O system. By this time, you should believe that bytes of information can be transferred over a bus with a few control wires. The message of this chapter is only the simplicity of the I/O system. The CPU and the RAM are the computer. Everything else, disks, printers, keyboards, the mouse, the display screen, the things that make sound, the things that connect to the internet, all these things are peripherals, and all they are capable of doing is accepting bytes of data from the computer or sending bytes of data to the computer. The adapters for different devices have different capabilities, different numbers of registers, and different requirements as far as what the program running in the CPU must do to operate the device properly. But they don't do anything fancier than that. The computer controls the process with a very few simple I/O commands that are executed by the CPU.

The Keyboard

A keyboard is one of the simplest peripherals connected to the I/O bus. It is an input only device, and just presents one byte at a time to the CPU.

The keyboard has eight wires inside, its own little bus as shown on the right. When you press a key, it simply connects electricity to the wires necessary to create the ASCII code corresponding to the key that was pressed. That little box that says ‘Control,’ is also notified when a key is pressed, and sets the ASCII code into the Keycode Register.



After pressing a key, there will be an ASCII code waiting in the Keycode Register. Here's how the CPU gets that code into one of its registers.

AND gate #1 has eight inputs. They are connected to the CPU bus, four of them through NOT gates. Thus this AND gate will turn on any time the bus contains 0000 1111. This is the I/O address of this keyboard adapter.

AND gate #2 comes on only during 'clk s' time of an OUT Addr instruction. It operates the 'set' input of a Memory bit. If the bus contains 0000 1111 at this time, the 'i' input will be on, and the Memory bit will turn on. When this Memory bit is on, it means that the keyboard adapter is active.

AND gate #3 comes on during ‘clk e’ time of an IN Data instruction. If the Memory bit is on, AND gate #4 will come on and the Keycode Register will be enabled onto the bus, which will be set into Reg B in the CPU.

Every adapter that is connected to the I/O bus needs to have the type of circuitry we see in gates #1 and #2 and the memory bit above. Each adapter will have a different combination that turns gate #1 on; this is what allows the CPU to select each adapter individually.

Here is a little program that moves the current keypress into Reg 3 in the CPU.

Instruction		Comments
Data	R2,0000 1111	* Put Addr of Keyboard in
OUT	Addr,R2	* Select Keyboard
IN	Data,R3	* Get ASCII of key presse
XOR	R2,R2	* Clear Address in Reg 2
OUT	Addr,R2	* Un-Select Keyboard

That little ‘Control’ box clears the Keycode Register after it has been sent to the CPU.

The program running in the CPU will check the keyboard adapter on a regular basis, and if the byte that it receives is all zeros, then no key has been pressed. If the byte has one or more bits on, then the program will do whatever the program has been designed to do with a keystroke at that time.

Again, we are not going to go through every gate in the Keyboard adapter. All device adapters have the same sorts of circuitry in order to be able to respond when they are addressed, and send or receive bytes of information as needed. But it is no more complicated than that. That is all that I/O devices and adapters do.

The Display Screen

Television and computer display screens work the same way, the main difference between them is only what they display. This is not actually computer technology, because you don’t need a display screen to have a computer, but most computers do have a screen, and the computer spends a lot of its time making the screen look like something, so we need to know a little bit about how it works.

Television appears to give you moving pictures with sound. The pictures and sound are done separately, and in this chapter, we are only concerned with how the picture works.

The first thing to know is that although the picture appears to be moving, it is actually a series of still pictures presented so quickly that the eye doesn’t notice it. You probably already knew that, but here’s the next thing. You have seen motion picture film. It is a series of pictures. To watch a movie, you put the film in a projector, which shines light through one picture, then moves the film to the next picture, shines light through it, etc. It usually runs at 24 pictures per second, which is fast enough to give the illusion of a constantly moving picture.

Television goes a bit faster, about 30 pictures per second, but there is another, much bigger difference between film and television. With the movie film, each still picture is shown all at once. Each picture is complete, when you shine the light through it, every part of the picture appears on the screen simultaneously. Television is not capable of doing this. It does not have a whole picture to put on the screen all at once.

All that a television can do at one instant in time, is to light up one single dot on the screen. It lights up one dot, then another dot, then another, very quickly until one whole picture’s worth of dots has been lit. This whole screen’s worth of dots makes up one still picture, thus it has to light up all of the dots within one thirtieth of a second, and then do it all over again with the next picture, etc. until it has placed 30 picture's worth of dots on the screen in one second. So the TV is very busy lighting up individual dots, 30 times the number of dots on the screen, every second.

Usually, the top left dot is lit first, then the one to its right, and so on across the top of the screen to the top right corner. Then it starts with the second line of dots, going across the screen again, the third line, etc. until it has scanned the entire screen. The brightness of each dot is high or low so that each part on the screen gets lit up to the proper brightness to make the screen look like the intended image.

At any one instant in time, the television is only dealing with one single solitary dot on the screen. So with television, there are two illusions – the illusion of motion coming from a series of still pictures, as well as the illusion of complete still pictures that are actually drawn one dot at a time. This second illusion is aided by what the screen in made of, each dot only gets lit up for a tiny fraction of a second, and it starts to fade away immediately. Fortunately, whatever the screen is made of that glows, continues to glow to some degree between one time when the dot is lit up and 1/30th of a second later when that same dot gets lit up again.

To the eye, you just see a moving picture, but there are a lot of things going on to make it appear that way.

In a computer, a single dot on the screen is called a 'picture element,' or 'pixel' for short.

Computer screens work just like televisions. They also have to scan the entire screen 30 times a second to light up each individual pixel and thereby make an image appear. Even if the content of the screen is not changing, something in the computer has to scan that unchanging image onto the screen 30 times every second. No scanning, no picture – that's just the way it works.

We're not going to go into the same amount of detail here that we did with the CPU and the RAM, those two are what make it a computer, but if we want to know how our computer is able to put something on the screen that we can read, we need to have the basic idea of how it works.

In this chapter we will look at the simplest kind of screen, the kind that is black and white, and whose pixels can only either be fully on or fully off. This type of screen can display characters and the type of pictures that are made of line drawings. Later in the book we will see the few simple changes that enable a screen to display things like color photographs.

The major parts are three. First there is the computer, we have seen how that works. It has an I/O Bus that can move bytes to and from things outside of the computer. Second is the screen. The screen is just a large grid of pixels, each of which can be selected, one at a time, and while selected, can either be turned on, or not. The third item is the 'display adapter.' The display adapter is connected to the I/O Bus on one side, and to the screen on the other side.

The heart of a display adapter is some RAM. The display adapter needs its own RAM so it can "remember" which pixels should be on, and which pixels should be off. In the type of screen we are going to describe here, there needs to be one bit in RAM for each pixel on the screen.

In order to make the screen scan every pixel 30 times every second, the Display Adapter needs its own clock that ticks at a speed that is 30 times the number of pixels on the screen. At each tick of the clock, one pixel is selected and it is turned on or not by the corresponding bit from the RAM.

As an example, lets use an old type of screen. It is a black and white screen that displays 320 pixels across the screen and 200 pixels down. That comes out to 64,000 individual pixels on the screen. Each pixel on the screen has a unique address consisting of two numbers, the first being the left-right or horizontal position, and the other being the up-down or vertical position. The address of the top left pixel is 0,0 and the bottom right pixel is 319,199

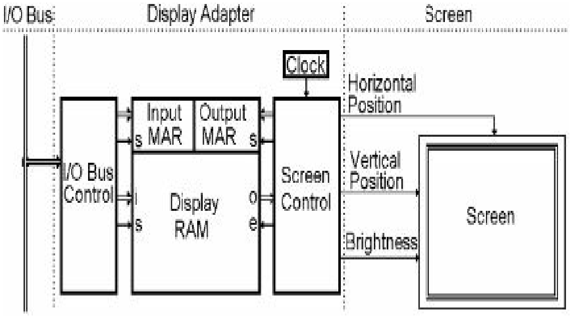
64,000 pixels times 30 pictures per second means that this Display Adapter's clock needs to tick 1,920,000 times per second. And since there are eight bits in a byte, we will need 8,000 bytes of display RAM to tell each of the 64,000 screen pixels whether to be on or off.

The display adapter has a register that sets the horizontal position of the current pixel. The display adapter adds 1 to this register at every tick of the clock. It starts at zero, and when the number in it gets to 319, the next step resets it back to zero. So it goes from zero to 319 over and over again. There is also a register that sets the vertical position of the current pixel. Every time the horizontal register gets reset to zero, the display adapter adds 1 to the vertical register. When the vertical register reaches 199, the next step will reset it to zero. So as the horizontal register goes from zero to 319 200 times, the vertical register goes from zero to 199 once.

The currently selected screen pixel is controlled by these registers, so as the horizontal register goes from 0 to 319, the current pixel goes across the screen once. Then the vertical register has one added to it, and the current pixel moves down to the first pixel on the next line.

Thus, the clock and the horizontal and vertical registers select each pixel on the screen, one at a time, going left to right in one row, then selecting each pixel in the next row down, then the next, etc. until every pixel on the screen has been selected one time. Then it starts all over again.

At the same time, there is another register that contains a display RAM address. This register also gets stepped through, although we only need one new byte for every eight pixels. The bits of each byte, one at a time, are sent to the screen at eight consecutive pixels to turn them on or off. After every eight pixels, the RAM address register has 1 added to it. By the time all of the pixels have been stepped through, the entire RAM has also been stepped through, and one entire picture has been drawn. When the horizontal and vertical registers have both reached their maximums, and are reset to zero, the RAM address is also reset to zero.



The display adapter spends most of its time painting the screen. The only other thing it has to do is to accept commands from the I/O Bus that will change the contents of the display adapter RAM. When the program running in the CPU needs to change what's on the screen, it will use the I/O OUT command to select the display adapter, and then send a display adapter RAM address and then a byte of data to store at that address. Then as the adapter continues to repaint the screen, the new data will appear on the screen at the appropriate spot.

The display adapter RAM is built differently than the RAM in our computer. It keeps the input and output functions separate. The inputs of all storage locations are connected to the input bus, and the outputs of all storage locations are connected to the output bus, but the input bus and the output bus are kept separate. Then there are two separate memory address registers, one for input and one for output. The input MAR has a grid that only selects which byte will be 'set,' and the output MAR has a separate grid that only selects which byte will be 'enabled.'

With this setup, the screen and the display RAM can both be continuously scanned using only the output MAR and the enable bit. When the I/O Bus is used to write into the display RAM, it uses only the input MAR and the set bit.

This is how the display adapter creates an image on the screen. Because of the way it works, there is an interesting relationship between which bits in the display RAM correspond to which pixels on the screen. As it scans the first eight pixels of the top line, it uses the individual bits of byte 0 of its RAM to turn the pixels on or off. As it scans the second eight pixels, it uses the individual bits of byte 1 of its RAM, etc. It takes 40 bytes of RAM to draw the first line, and so the last eight pixels, which are numbered 312 through 319, come from RAM byte 39. The second row uses byte 40 to draw its first 8 pixels, etc.

If you want to write letters and numbers on the screen, how do you do it? If you put the ASCII code for 'A' into a byte in the display RAM, you will just get eight pixels in a row where one is off, then one is on, then five are off and the last one is on. That's not what an 'A' should look like.

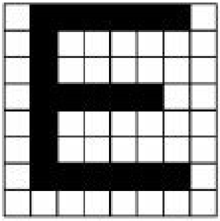
There is a solution for this, and it involves…

Another Code

When you want to print or display written language, you need to translate the ASCII code into something that is readable by a live person. We have a code, 0100 0101, that appears on the ASCII code table next to the letter 'E.' But how does the computer turn 0100 0101 into a readable 'E'?

We have a display screen, but the screen is a just a grid of pixels, there are no human readable 'E's in anything we have described so far. In order to get an 'E' on the screen, there has to be something that makes that shape that we recognize as a letter of the alphabet.

Therefore, we need another code. This code is really about little pictures made out of dots. For each character that we want to be able to draw on the screen, we need a little picture of that character. If you take a grid 8 pixels wide and 8 pixels high, you could decide which pixels had to be on to make a little picture that looks like the character that you want to draw on the screen, like this:



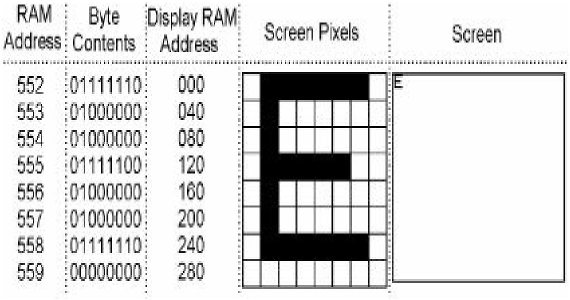
If you turn this picture into ons and offs, you could store it in eight bytes. If there are 100 different characters that you want to be able to display on the screen, then you'd need 100 different little pictures like this, and it would require 800 bytes of RAM to store it. Our little computer only has a 256 byte RAM, so this would be a good time to imagine that larger version that we described earlier.

These 800 bytes are a type of code known as a “font.”

If you want to make a character appear in a certain place on the screen, you need to choose the correct little picture from the font, and then use I/O instructions to copy the eight bytes of the picture to the proper bytes in the display adapter’s RAM.

If the pictures in our font are arranged in the same order as the ASCII code table, then we can use the numeric value of an ASCII code to find the corresponding picture within the font. The ASCII code for ‘E’ is 0100 0101. If you apply the binary number code to the same pattern of ones and zeros, you get the decimal number 69. ‘E’ then, is the 69th code in ASCII, and the picture of an ‘E’ will be the 69th picture within the font. Since there are eight bytes in each picture, you multiply the 69 by 8, and that tells you that the picture for ‘E’ will be the eight bytes starting at address 552.

Now we need to know where to copy these bytes to in the display RAM. Lets say that we want to display an ‘E’ at the very top left of the screen. Where are the bits that turn on the pixels that we are interested in? Well, the first line is easy, it is the first eight bits of the display RAM, Address 0. So we use a series of OUT instructions to copy RAM address 552 to display RAM address 0. Now, where is the second line in the display RAM? The display paints all 320 bits of the top row before it moves down to the second row. That means that it uses 40 bytes on each row, so the top row uses bytes 0-39. That means that the second byte of the picture of ‘E’ at RAM address 553 needs to be written at address 40 in the display RAM. Similarly, the third through eighth bytes get written at bytes 80, 120, 160, 200, 240 and 280. When you have done all of that, you would then see a complete ‘E’ on the screen. If you wanted to write an ‘X’ on the screen right next to the ‘E’, you would locate the eight bytes in the font for ‘X’ and copy them into display RAM bytes 1, 41, 81, 121, 161, 201, 241 and 281. If you need 27 ‘E’s on your screen, you just copy the one ‘E’ in your font to 27 different places in the display RAM.



RAM Address	Byte Contents	Display RAM Address
552	01111110	000
553	01000000	040
554	01000000	080
555	01111100	120
556	01000000	160
557	01000000	200
558	01111110	240
559	00000000	280

Of course, this seems like a lot of work just to make a single letter appear on the screen. The program that does this would need a loop of instructions that calculates the first ‘from’ and ‘to’ addresses, then issues the appropriate OUT instructions to copy the first byte to the display RAM. Then the loop would repeat, updating both addresses each time, until all eight bytes had been copied to the appropriate places. We’re not going to write this program, but it could easily be a 50 instruction program that has to loop around eight times before it’s finished. That means that it could take 400 instruction cycles just to put one character on the screen! If you drew 1000 characters on the screen, that might take 400,000 instruction cycles. On the other hand, that’s still only about one quarter of one percent of what this computer can do in one second.

This just goes to show you why computers need to be so fast. The individual things that they do are so small, that it takes a huge number of steps to get anything done at all.

The Final Word on Codes

We have seen several codes used in our computer. Each one was designed for a specific purpose. Individual coded messages are put in bytes, and moved around and used to get things done.

The bytes do not 'know' which code was used to choose the pattern that they contain. There is nothing in the byte itself that tells you which code it is supposed to be.

Certain parts of the computer are built with various codes in mind. In the ALU, the adder and comparator are built to treat bytes as though they contain values encoded with the binary number code. So are the Memory Address Register and the Instruction Address Register.

The Instruction Register is built to treat its contents as though it contains values encoded with the Instruction Code.

The Display adapter RAM bits are just ons or offs for individual pixels. Pictures and fonts are strings of bytes that will result in something that can be recognized by a person when it is organized, and the brightnesses are set, by the wiring of a display adapter and screen.

The ASCII code table does not appear anywhere inside the computer because there is no way to represent a letter of the alphabet except by using a code.

The only places where ASCII gets converted between characters and the code for the character, are in the peripherals. When you press 'E' on the keyboard, you get the ASCII code for an 'E.' When you send the ASCII code for an 'E' to a printer, it prints the letter 'E.' The people who build these peripherals have an ASCII code table in front of them, and when they build a keyboard, the switch under the fourth button in the second row, which has the letter 'E' printed on it, is wired up to the proper bus wires to produce the code that appears next to the letter 'E' on the ASCII code table.

An 'E' is the fifth letter of an alphabet used by people to represent sounds and words in the process of writing down their spoken language. The only ‘E’s in the computer are the one on the keyboard and the ones that appear on the screen. All the ‘E’s that are in bytes are just the code that appears next to the ‘E’ on an ASCII code table. They are not ‘E’s, there is no way to put an ‘E’ in a computer. Even if you put a picture of an ‘E’ in a computer, it isn’t actually an ‘E’ until it is displayed on the screen. That’s when a person can look at it and say “That’s an E.”

Bytes are dumb. They just contain patterns of ons and offs. If a byte contains 0100 0101, and you send it to the printer, it will print the letter ‘E.’ If you send it to the Instruction Register, the computer will execute a Jump instruction. If you send it to the Memory Address Register, it will select byte number 69 of the RAM. If you send it to one side of the Adder, it will add 69 to whatever is on the other side of the Adder. If you send it to the display screen, it will set three pixels on and five pixels off.

Each of these pieces of the computer is designed with a code in mind, but once it is built, the mind is gone and even the code is gone. It just does what it was designed to do.

There is no limit to the codes that can be invented and used in a computer. Programmers invent new codes all the time. Like the cash register in the fast food restaurant mentioned earlier, somewhere in that machine is a bit that means ‘include French fries.’

The Disk

Most computers have a disk. This is simply another peripheral that is attached to the I/O bus. The disk's mission is very simple; it can do two things. You can send it bytes, which it will store, or you can tell it to send back some bytes, which were stored previously.

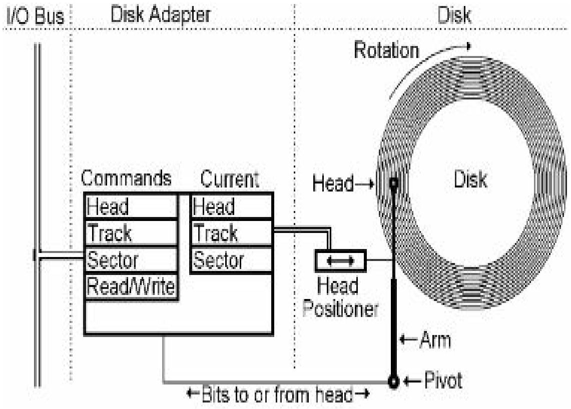
There are two reasons that most computers have a disk. First, they have the ability to store a huge number of bytes, many times greater than the Computer's RAM. The CPU can only execute programs that are in RAM, it can only manipulate bytes that are in RAM. But there is never enough RAM to store all of the things that you may want to do with your computer. And so a disk will hold everything, and when you want to do one thing, the bytes on the disk for that one thing will be copied into RAM and used. Then when you want to do something different, the bytes for the new activity will be copied from the disk into the same area of RAM that had been used for the first activity.

The second reason that computers have disks, is that the bytes stored on the disk do not disappear when you turn the power off. The RAM loses its settings when you turn the computer off, when you turn it back on, all bytes are 0000 0000, but the disk retains everything that has been written on it.

A computer bit has been defined so far as a place where there is or is not some electricity. But prior to that, we defined it as a place that can be in one of two different states. On a disk, the electric bits are transformed into places on the surface of the disk that have been magnetized one way or the other. Since magnets have north and south poles, the spot on the disk can be magnetized either north-south or south-north. One direction would represent a zero, and the other direction, a one. Once a spot is magnetized, it stays that way unless the same spot gets magnetized the other way. Turning the power off has no effect on the magnetized spots.

A disk, as its name implies, is a round thing, that spins around quickly. It is coated with a material that can be magnetized easily. Do you remember the telegraph? At the receiving end, there is a piece of metal with a wire wrapped around it. That piece of metal turns into a magnet when electricity moves through the wire. The disk has a tiny version of this called a ‘head’ mounted on an arm. The arm holds the head very close to the surface of the spinning disk, and the arm can swing back and forth, so that the head can reach any point on the surface of the disk. If you put electricity through the head, it can magnetize the surface of the disk. Also, it works the other way around; when the head passes over a magnetized area, it makes electricity appear in the wires wrapped around the head. Thus, the head can either write on the disk or read what has been previously written on the disk. The bits of the bytes are written one after another on the disk surface.

The surface of the disk is divided into a series of rings, called tracks, very close to each other. The head can move across the surface and stop on any one of the tracks. Each circular track is usually divided into short pieces called sectors. Since a disk has two sides, usually both sides are coated with the magnetic material and there is a head on each side.



In RAM, every byte has its own address. On a disk, there is also a way to locate bytes, but it is very different. You have to specify which head, which track and which sector at which a block of bytes is located. That is the type of "address" that the data on a disk has, like "Head 0, Track 57, Sector 15." And at that address, there is not just one byte, but a block of bytes, typically several thousand. For the examples in our book, since our RAM is so small, we will talk about a disk that stores blocks of 100 bytes.

When a disk is read or written, there is no way to access an individual byte in the block of bytes. The whole block has to be transferred to RAM, worked on in RAM, and then the whole block has to be written back to the disk.

The disk spins quickly, faster than that fan on your desk; many popular disks spin 7200 times a minute, which is 120 times per second. That’s pretty fast, but compared to the CPU, it is still pretty slow. In the time that the disk spins around one time, the Clock will tick over eight million times, and our CPU will execute well over a million instructions.

The disk, like every peripheral, is connected to its own adapter, which in turn is connected to the I/O bus. The disk adapter does a few things. It accepts commands to select a head, select a track and select a sector. It accepts commands to read from or write to, the block of bytes at the currently selected head, track and sector. There will also probably be a command where the CPU can check the current position of the arm and the disk.

The command to select a head can be completed immediately, but when it gets a command to select a track, it has to move the head to that track, which takes a long time in terms of instruction cycles. When it gets a command to select a sector, it has to wait for that sector to spin around to where the head is, which also takes a long time in terms of instruction cycles. When the CPU has determined that the head has arrived at the desired track and sector, then the I/O commands for reading or writing can be executed, and one byte at a time will be transferred over the I/O bus. A program that reads or writes a block of bytes has to continue the process until the whole block of bytes is complete. With our simple I/O system, the individual bytes move between the disk and a CPU register. The program that is running has to move these bytes to or from RAM, usually in consecutive locations.

This is all that a disk does. You have probably used a computer that had a disk, and didn’t need to know anything about heads, tracks and sectors. And that is a good thing, because it is pretty annoying to have to deal with a disk at that level of detail. We will look at how a disk is normally used later in the book.

Another language note: There are several words that mean virtually the same thing, but for some reason certain words go with certain technologies.

If you want to send someone a letter, first you write it on a piece of paper, then when the recipient gets the letter, he reads it.

In the days of tape recorders, you would start with a blank tape. Then you would record some music on the tape. When you wanted to hear the music again you would play the tape.

When it comes to computer disks, putting something on the disk is called writing. Getting something off the disk is called reading.

Putting something into RAM is called writing or storing. Getting something out of RAM is called reading or retrieving.

Putting something into a CPU register is usually called loading.

Putting music on a disk is sometimes called recording, sometimes burning. Listening to a disk is still usually called playing, but if you are copying it onto your computer, then it is called ripping.

Writing, recording, storing, loading and burning all mean pretty much the same thing. Reading, retrieving, playing and ripping are also very similar. They mean the same things, it’s just a difference of words.

Excuse Me Ma'am

There is one other thing that most computers have as part of their Input/Output system. A computer doesn't need one of these to be called a computer, so we will not go through every gate needed to build it. But it is a very common thing, so we will describe how it works.

You know if Mom is in the kitchen stirring a pot of soup, and little Joey comes running in and says "I want a glass of milk," Mom will put down the spoon, go over to the cabinet, get a glass, go to the refrigerator, pour the milk, hand it to Joey, and then she will go back to the stove, pick up the spoon and resume stirring the soup. The soup stirring was interrupted by getting a glass of milk, and then the soup stirring resumed.

This thing that most computers have, is called an "Interrupt," and it works very much like what happened with Mom and Joey.

An interrupt starts with one more wire added to the I/O Bus. This wire is used by certain device adapters to let the CPU know that it's a good time for the CPU to do an I/O operation, like right after someone presses a key on the keyboard. When a device adapter turns the Interrupt bit on, the next time the stepper gets back to step 1, the next instruction cycle will not do the usual fetch and execute, but rather it will do of the following:

Step 1	move binary 0 to MAR
Step 2	move IAR to RAM
Step 3	move binary 1 to MAR
Step 4	move Flags to RAM
Step 5	move binary 2 to MAR
Step 6	move RAM to IAR
Step 7	move binary 3 to MAR
Step 8	move RAM to Flags

The result of this sequence is that the current IAR and Flags are saved to RAM addresses 0 and 1, and they are replaced with the contents of RAM bytes addresses 2 and 3. Then the CPU returns to its normal fetch and execute operation. But the IAR has been replaced! So the next instruction will be fetched from whatever address was in RAM byte 2.

In other words, what the CPU had been doing is saved, and the CPU is sent off to do something else. If at the end of this new activity, the program puts RAM bytes 0 and 1 back into the IAR and Flags, the CPU will pick up from exactly where it left off, before it was interrupted.

This system is very useful for dealing with I/O operations. Without interrupts, the program running in the CPU would have to make sure to check all of the devices on the I/O Bus on a regular basis. With interrupts, the program can just do whatever it is designed to do, and the program that deals with things like keyboard input will be called automatically as needed by the interrupt system.

We have not included this in our CPU because it would just make our Control Section wiring diagram too big. It would need to add the following: two more steps to the stepper, wiring to do the above 8 steps in place of the normal instruction cycle, paths for the Flags register to get to and from the bus, a method of sending a binary 0, 1, 2 or 3 to MAR, and an instruction that restores RAM bytes 0 and 1 to the IAR and Flags register.

And that is an Interrupt system. As far as the language is concerned, the computer designers took an existing verb, 'interrupt,' and used it in three ways: It is a verb in "the keyboard interrupted the program," it is an adjective in "This is the Interrupt system," and it is a noun in "the CPU executed an interrupt."

That’s All Folks

Yes, this is the end of our description of a computer. This is all there is. Everything you see a computer do is a long concatenation of these very simple operations, the ADDing, NOTting, Shifting, ANDing, ORing, XORing of bytes, Storing, Loading, Jumping and I/O operations, via the execution of the instruction code from RAM. This is what makes a computer a computer. This is the sum total of the smarts in a computer. This is all the thinking that a computer is capable of. It is a machine that does exactly what it is designed to do, and nothing more. Like a hammer, it is a tool devised by man to do tasks defined by man. It does its task exactly as designed. Also like a hammer, if it is thrown indiscriminately it can do something unpredictable and destructive.

The variety of things the computer can be made do is limited only by the imagination and cleverness of the people who create the programs for them to run. The people who build the computers keep making them faster, smaller, cheaper and more reliable.

When we think of a computer, we probably think of that box that sits on a desk and has a keyboard, mouse, screen and printer attached to it. But computers are used in many places. There is a computer in your car that controls the engine. There is a computer in your cell phone. There is a computer in most cable or satellite television boxes. The things that they all have in common are that they all have a CPU and RAM. The differences are all in the peripherals. A cell phone has a small keyboard and screen, a microphone and a speaker, and a two-way radio for peripherals. Your car has various sensors and controls on the engine, and the dials of the dashboard for peripherals. The cash register in a fast food restaurant has a funny keyboard, a small display screen and a small printer for receipts. There are computers in some traffic lights that change the lights based on the time of day and the amount of traffic that crosses the sensors embedded in the roadway. But the CPU and RAM make it a computer, the peripherals can be very different.

For the rest of the book we will look at miscellaneous subjects related to understanding how computers are used, a few interesting words that are related to computers, some of their frailties and a few other loose ends.

Hardware and Software

You’ve heard of hardware. That word has been around for a long time. There have been hardware stores for a century or more. I think that a hardware store originally sold things that were hard, like pots and pans, screwdrivers, shovels, hammers, nails, plows, etc. Perhaps ‘hardware’ meant things that were made out of metal. Today, some hardware stores no longer sell pots and pans, but they sell huge variety of hard things, like bolts and lawnmowers, also lumber and a lot of soft things too, like carpet, wallpaper, paint, etc. But these soft things are not called software.

The word ‘software’ was invented somewhere in the early days of the computer industry to differentiate the computer itself from the state of the bits within it. Software means the way the bits are set on or off in a computer as opposed to the computer itself. Remember that bits can be either on or off. The bit has a location in space, it is made of something, it exists in space, it can be seen. The bit is hardware. Whether the bit is on or off is important, but it’s not a separate part that you bolt into the computer, it is the thing in the computer that is changeable, the thing that can be molded, it is ‘soft’ in that it can change, but you can’t pick it up in your hand all by itself. This thing is called software.

Think of a blank videotape. Then record a movie on it. What is the difference between the blank videotape and the same videotape with a movie on it? It looks the same, it weighs the same, you can’t see any difference on the surface of the tape. That surface is coated with very fine particles that can be magnetized. In the blank tape, the entire surface of the tape is magnetized in random directions. After recording the movie on the tape, some little places on the tape are magnetized in one direction and other little places are magnetized in the other direction. Nothing is added to or taken away from the tape, it’s just the way the magnetic particles are magnetized. When you put the tape into a VCR it plays a movie. The tape is hardware, the pattern of the directions of magnetization on the tape is software.

In a computer, there are a great many bits. As we have seen, a lot of bits have to be set in certain ways in order to make the computer do something useful. The bits in the computer are always there. If you want the computer to do a certain thing, you set those bits on or off according to the pattern that will make the computer do what you want it to do. This pattern is called software. It is not a physical thing, it is just the pattern in which the bits are set.

So the difference between hardware and software isn't like metal versus rubber. Both metal and rubber are hardware as far as the computer definition is concerned. Hardware is something you can pick up, see, handle. Software is the way the hardware is set. When you buy software, it is recorded on something, usually some kind of disk. The disk is hardware, the specific pattern recorded on that disk is software. Another disk may look just like it, but have completely different software written on it.

Another way to see the difference between hardware and software is how easy it is to send it across a distance. If you have a vase that you want to send to your aunt Millie for her birthday, you have to pack the vase in a box and have a truck take it from your house to her house. But if you want to give her the present of music, you might go to the store, buy her a disk and mail it, but you might also buy her a gift certificate on the Internet, send her an e-mail, and have her download the music. In that case, the music will get to her house without a truck having to go there. The music will be transported solely by the pattern of electricity that comes over the Internet connection to her house.

Another way to see the difference between hardware and software is how easy it is to make a copy of the item. If you have a lawnmower, and want a second lawnmower, there is no machine that will copy the lawnmower. You could photograph the lawnmower, but you'd only have a flat photograph of a lawnmower. You couldn't mow any lawns with the photo. To get a real second lawnmower, you'd have to go back to the lawnmower factory and build another one out of iron and plastic and rope and whatever else lawnmowers are made out of. This is hardware.

Software can be copied easily by machine. All you need is something that can read the disk or whatever it is recorded on, and something else to write it onto a new disk. The new one will be just like the original, it will do all the same things. If the original is your favorite movie, the copy will also be your favorite movie. If the original is a program that will prepare your tax papers, so will the copy.

Software is not a physical thing, it is just how the physical things are set.

By far the most commonly used definition of 'software' is to refer to a package of computer instruction code. I think that the way it got this name is that once you have built a device as versatile as a computer, there are many different things that it can be made to do. But when there are no instructions in it, it can't do anything. So the software is an absolutely necessary part of a computer that is doing some task. It is a vital part of the total machine, yet it isn't like any other part in the machine. You can't weigh it or measure it or pick it up with a pair of pliers. So it is part of the 'ware,' but it isn't hardware. The only thing left to call it is 'software.'

Programs

As mentioned earlier, a series of instructions in RAM are called a program.

Programs come in many sizes. Generally, a program is a piece of software that has everything needed to do a specific task. A system would be something larger, made up of several programs. A program might be made up of several smaller parts known as ‘routines.’ Routines in turn may be made up of sub-routines.

There are no hard and fast definitions that differentiate between system, program, routine and sub-routine. Program is the general term for all of them, the only difference is their size and the way they are used.

There is another distinction between two types of programs that is not related to their size. Most home and business computers have a number of programs installed on them. Most of these programs are used to do something that the owner wants to do. These are called application programs because they are written to apply the computer to a problem that needs to be solved. There is one program on most computers that is not an application. Its job is to deal with the computer itself and to assist the application programs. This one program that is not an application is called the Operating System.

The Operating System

An "Operating System," or "OS" for short, is a large program that has many parts and several objectives.

Its first job is to get the computer up and running when you first turn the computer on.

Another one of its jobs is to start and end application programs and give each one time to run. It is the 'boss' of every other program on that computer. When more than one program is in RAM, it is the operating system that switches between them. It lets one program run for a small fraction of a second, then another program, then another program. If there are ten programs in RAM, and each one gets to run for one hundredth of a second at a time, each program would be able to execute millions of instructions in that time, several times per second. It would appear that all ten programs were running simultaneously because each one gets to do something, faster than the eye can see.

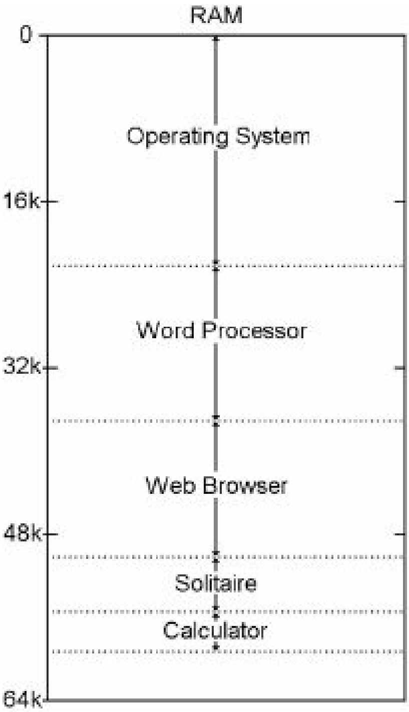
An Operating system also provides services to application programs. When an application program needs to read from, or write to the disk, or draw letters on the screen, it does not have to do all of the complicated I/O instructions necessary to accomplish the task. The OS has a number of small routines that it keeps in RAM at all times for such purposes.

All an application needs to do to use one of these routines is to load up some information in the registers, and then jump to the address of the proper OS routine. Here's an example of how it might be done. Lets say you want to draw a character on the screen. First, put the ASCII code of the desired character into R0. Then put row and column numbers of where you want it to appear on the screen into R1 and R2. And here's the tricky part: You put the address of the next instruction of your application program, into R3. Now just jump to the OS routine. The routine will take care of all of the details of drawing the character on the screen, and then its last instruction will be JMPR R3. Thus, these routines can be 'called' from any application, and when done, the routine will jump back to the next instruction in the application that called it.

There are several reasons for having the OS do all of the I/O functions. One is that it makes it easier to write application programs, the programmer does not even need to know how the peripherals actually work. Another reason is that it would waste a lot of RAM if every application had its own copy of all of the I/O routines. One of the most important reasons is that the OS can check to see whether the program should be allowed to do what it is asking to do. This is part of the OS’s other job of being the boss.

The heart of the OS is basically a loop of instructions that asks the following questions: Do I need to input anything? Do I need to output anything? Do I need to let any program run? Then it starts over again. If the answers to all of these questions is no, the CPU just executes the instructions in this loop over and over, millions of times per second. When there is something to do, it jumps to the beginning of the program that takes care of it, and when that is done, it jumps back to this loop where the OS ‘waits’ for something else to do.

Here is a diagram of our larger RAM version, showing what parts of RAM might be occupied by an Operating System and several other programs.


RAM
0
16k
32k
48k
64k
Operating System
Word Processor
Web Browser
Solitaire
Calculator

Within each program's RAM, there is all of the instruction code that makes the program work. Each program may be divided up into its own main loop, and many routines that are used for the various tasks that it needs to do. As mentioned, the OS also has routines that can be called by other programs.

Each program also uses part of its 'address space' for the data that it is working on. The calculator, for example, needs to have a few bytes where it stores the numbers that the user enters into it. Solitaire needs some bytes that specify which cards are in which positions. The word processor needs some RAM for all of the ASCII codes that make up the document you are working on. The OS also needs bytes where it can store fonts, keep track of where application programs have been loaded, receive the data that it reads from the disk, and for many other purposes.

And so this is what goes on inside your average computer. There are many different programs and data areas in RAM. The OS jumps to a program, the program jumps to a routine, the routine jumps to a sub-routine. Each program works on its data or calculates something or does an I/O operation. As each one finishes, it jumps back to where it came from. The CPU executes one instruction from one program at a time, and if they are written intelligently, each program will get its job done piece by piece, without interfering with the rest.

If our computer had included an 'interrupt system' like we described a few chapters back, every time someone pressed a key on the keyboard or moved the mouse, there would be an interrupt that would call a part of the OS that determines which I/O device caused the interrupt, and then calls the proper routine to take care of whatever it was. When that was done, the CPU would continue on with the next instruction of whatever program had been running when the interrupt happened.

This can all seem very complex, with so many millions and billions of instructions being executed in the blink of an eye. There are ways of organizing programs and good programming practices that can make it much more understandable. A study of these would simplify software in the same manner that I hope this book has simplified the hardware. But that would be the subject for another entire book.

Languages

Writing programs is very hard to do when you're just writing ones and zeros, but that is the only code that the CPU 'understands.'

What is a language? A spoken language, such as English, is a way to represent objects, actions and ideas with sounds. A written language is a way to represent the sounds of a spoken language with symbols on paper. Sounds like another code, and a code representing a code. We just can't get away from these things!

Do you remember that shorthand we used when we were looking at the CPU instruction code and the wiring in the Control Section? Well, that is actually something more than just a handy tool that was invented for this book. It is a computer language. Here are a few lines of it:

Instruction Code		Language		Meaning
1000	rarb	ADD	RA,RB	Add
1001	rarb	SHR	RA,RB	Shift Right
1010	rarb	SHL	RA,RB	Shift Left
1011	rarb	NOT	RA,RB	Not

A computer language is a way to represent the instruction code. Its purpose is to make it easier to write computer programs.

In order to use this language, you write the program you want with ASCII characters, and save it into a file. Then you load a special program called a 'compiler' into RAM and jump to its first instruction. The compiler will read the ASCII file, translate each line into the Instruction Code that it represents, and write all of the Instruction Code bytes into a second file. The second file may then be loaded into RAM, and when the CPU jumps to its first instruction, the program you wrote in ASCII will hopefully do what you intended it to do.

Of course, when computers were first invented, all programs had to be written directly in ones and zeros. Then somebody got tired of the tedium of programming that way, and decided to write the first compiler. Then ever after, programs were written in this easier language, and then translated into Instruction Code by the compiler. With the original compiler, you could even write a better compiler.

So in order for a computer language to exist, you need two things, a set of words that make up the language (another code,) and a compiler that compiles the written language into computer instruction code.

The language that we have seen in this book has only about 20 words in it. Each word correlates directly to one of the instructions of which this computer is capable. Each line you write results in one computer instruction. When you write an 87 line program in this language, the instruction code file that the compiler generates will have 87 instructions in it.

Then someone invented a "higher level" language where one line of the language could result in multiple computer instructions. For example, our computer does not have an instruction that does subtraction. But the compiler could be designed so that it would recognize a new word in the language like 'SUB RA,RB' and then generate however many machine instructions were necessary to make the subtraction happen. If you can figure out how to do something fancy with 47 instructions, you can have a word in your language that means that fancy thing.

Then someone invented an even higher level language where the words that make up the language don't even resemble the CPU's actual instructions. The compiler has a lot more work to do, but still generates instruction code that does the things that the words in that language mean. A few lines from a higher level language might look like this:

```
     Balance = 2,000
     Interest Rate = .034
     Print "Hello Joe, your interest this year is: $"
     Print Balance X Interest Rate
```

The compiler for this language would read this four-line program, and generate a file that could easily contain hundreds of bytes of instruction code. When that instruction code was loaded into RAM and run, it would print:

```
     Hello Joe, your interest this year is: $68
```

Writing software in higher level languages can result in getting a lot more done in a shorter amount of time, and the programmer no longer needs to know exactly how the computer actually works.

There are many computer languages. Some languages are designed to do scientific work, some are designed for business purposes, others are more general purpose. Lower level languages are still the best for certain purposes.

The File System

As we saw earlier, the way a disk actually works is pretty foreign to most people who use a computer.

To make things easier, someone invented an idea called a "file." A file is supposed to be similar to the kind of paper files that people use in offices. A paper file is a sheet of cardboard folded in half and placed in a file cabinet. This folder has a tab on it where you can write some sort of name for the folder, and then you can put one or many pieces of paper in the folder.

A computer file is a string of bytes that can be any length, from one byte up to all of the bytes available on the disk. A file also has a name. A disk may have many files on it, each with its own name.

Of course, these files are just an idea. To make a file system work, the operating system provides a bunch of software that makes the disk appear to be like a filing cabinet instead of having heads, tracks, sectors and blocks of bytes.

This file system gives application programs an easy way of using the disk. Applications can ask the OS to create, read, write or erase something called a file. All the application needs to know is the name of the file. You open it, request bytes, send it bytes, make it bigger or smaller, close the file.

The OS uses part of the disk to maintain a list of file names, along with the length of each file and the disk address (head, track, sector) of the first sector of the data. If the file is smaller than a disk sector, that's all you need, but if the file is larger than one sector, then there is also a list which contains as many disk-type addresses as needed to hold the file.

The application program says create a file with the name "letter to Jane." Then the user types the letter to Jane and saves it. The program tells the OS where the letter is in RAM and how long it is, and the OS writes it to disk in the proper sector or sectors and updates the file length and any necessary lists of disk-type addresses.

To use the file system, there will be some sort of rules that the application program needs to follow. If you want to write some bytes to the disk, you would need to tell the OS the name of the file, the RAM address of the bytes that you want to write, and how many bytes to write. Typically, you would put all of this information in a series of bytes somewhere in RAM, and then put the RAM address of the first byte of this information in one of the registers, and then execute a Jump instruction that jumps to a routine within the Operating System that writes files to the disk. All of the details are taken care of by this routine, which is part of the OS.

If you ask the OS to look at your disk, it will show you a list of all the file names, and usually their sizes and the date and time when they were last written to.

You can store all sorts of things in files. Files usually have names that are made up of two parts separated by a period like “xxxx.yyy.” The part before the period is some sort of a name like “letter to Jane,” and the part after the dot is some sort of a type like “doc” which is short for “document.” The part before the period tells you something about what is in the file. The part after the dot tells you what type of data is contained in this file, in other words, what code it uses.

The type of the file tells both you and the OS what code the data in the file uses. In one popular operating system “.txt” means text, which means that the file contains ASCII. A “.bmp” means BitMaP, which is a picture. A “.exe” means executable, which means it is a program and therefore contains Instruction Code.

If you ask the OS what programs are available to execute, it will show you a list of the files that end with “.exe”. If you ask for a list of pictures that you can look at, it will show you a list of files that end with “.bmp”.

There are many possible file types, any program can invent its own type, and use any code or combination of codes.

Errors

The computer is a fairly complex machine that does a series of simple things one after another very quickly. What sorts of things could go wrong here?

In the early days of computing, when each gate in the computer was relatively expensive to build, sometimes there were components that actually had moving parts to make electrical connections. Two pieces of metal had to touch to make the electricity go to where the builders wanted it to go. Sometimes when the machine stopped working correctly, the fixit guy would look inside to find out what was wrong, and he would find that a spider had crawled inside the machine and had gotten itself wedged in between two of these pieces of metal that were supposed to touch each other. Then when one piece of metal moved to touch the other, the spider was in the way and they wouldn't touch. So the electricity wouldn't get to where it needed to go, and the machine would not operate correctly anymore. The fixit guy would remove the bug, clean up the contacts, and report "There was a bug in the computer." And he literally meant a bug.

Over time, whenever a computer appeared to be operating incorrectly, people would say that the computer had a bug. There are two main classes of computer bugs: hardware and software.

A hardware bug actually means that the computer is broken. This could be as serious as you turn the computer on, and it catches fire, to there is one byte in the RAM where one bit is always off.

Now one bit in RAM that refuses to change may be a problem or it may not. If the byte where that bit is located somehow never gets used, then the computer will work just fine. If that byte is part of a place where a name is stored, then the name may get changed from "Joe" to "Jod." If that byte has some program instructions in it, you may get an XOR instruction changed to a JMP instruction. Then when the program gets to that instruction, it will not do the XOR like it is supposed to, but rather it will jump somewhere else and start executing whatever is at the new location as though it was a series of instructions. The contents of those bytes will determine what happens next, but it will almost certainly be as wrong as a train falling off its track.

If a gate is broken in the stepper, for instance, so that step 4 never comes on, then the computer will not really be able to operate at all. It would still be able to fetch instructions in steps 1, 2 and 3, but every instruction would execute incorrectly. Certainly the program would make a mess of things after 'executing' just a few instructions.

Software bugs can take many forms, but they are all ultimately programmer mistakes. There are probably many more ways to write a program incorrectly than correctly. Some errors just create some kind of incorrect results, and other errors cause the computer to "crash."

One of my favorite stupid programmer stories is this: Someone bought a car on credit. He got a coupon book with the loan, one coupon to be sent in with each payment. But when he made his first payment, he accidentally used the last coupon in the book instead of the first one. A few weeks later, he received a computer-generated letter from the loan company saying, "Thank you for paying off your loan in full, next time you need a loan please use us again." Obviously, the program just checked the coupon number and if it was equal to the highest number coupon in the book, jump to the routine for a paid-in-full loan. It should have at least checked the balance remaining on the loan before deciding that it was paid off. This is a subtle error, it might not be caught by the loan company until they audited their books months later. The computer did exactly what it was told to do, and most of the time it was adequate, but the program was not written to anticipate all of the situations that sometimes occur in the real world.

One of the worst software bugs is getting stuck in a loop. The program executes a series of instructions, and then jumps back to the beginning of the series and executes it over and over again. Of course, loops are used all the time in programming, but they are used to do something that has a finite number of similar steps.

It may repeat until 50 bytes have been moved somewhere, or keep checking for the user to press a key on the keyboard. But the computer will exit the loop at some point and continue on to its next task. But if there is some sort of programming error where there is a loop that has no way out, the computer will appear to be completely stuck. This is sometimes called being 'hung,' the whole computer may need to be turned off and restarted to get out of the loop and back into useful operation.

There are all sorts of errors that end up with the CPU trying to execute something other than instruction code. Lets say you have your program residing at address 10 through 150, and you have some ASCII data such as names and phone numbers at addresses 151 through 210. If the program is written incorrectly so that under certain conditions it will jump to address 180, it will just continue fetching and executing the bytes starting at address 180. If 180-189 was filled with the ASCII for "Jane Smith," the "program" will now be executing complete garbage, a series of bytes that were not designed to be Instruction Code. It may put itself into a loop, or jump back somewhere into the program, or issue the command to erase the disk drive. And it will be doing garbage at its usual high speed. If you looked at the patterns in the bytes, you could see what it would do, but it could be just about anything. If the name at address 180 was "Bill Jones", it would do something completely different. Since it is not designed to be useful, most likely it will just keep making a bigger mess out of what is in memory until the computer will have to be powered off to get it to stop.

Another type of error could occur if a program accidentally wrote "John Smith" into the place where a font was stored. In that case, every letter "E" that got drawn on the screen thereafter would look like this: '
.'

The computer executes hundreds of millions of instructions every second, and it only takes one wrong instruction to bring the whole thing to a screeching halt. Therefore, the subject of programming computers in a manner that will be completely 'bug free' is something that gets a lot of attention. Almost all programming is done with languages, and the compilers for these languages are designed to generate Instruction Code that avoids the most serious types of errors, and to warn the programmer if certain good programming practices are violated. Still, compilers can have errors, and they will never be able to spot an error like the one above with the car loan.

As you can see, the computer and its software are pretty fragile things. Every gate has to work every time, and every instruction that gets executed has to be correct. When you consider all of the things that could go wrong, the high percentage of things that normally go right is actually quite impressive.

Computer Diseases?

Another place where human characteristics get assigned to computers is something called a computer virus. This implies that computers can come down with a disease and get sick. Are they going to start coughing and sneezing? Will they catch a cold or the chicken pox? What exactly is a computer virus?

A computer virus is a program written by someone who wants to do something bad to you and your computer. It is a program that will do some sort of mischief to your computer when it runs. The motivation of people who write virus programs ranges from the simple technical challenge of seeing whether one is capable of doing it, to a desire to bring down the economy of the whole world. In any case, the people who do such things do not have your best interests in mind.

How does a computer 'catch' a virus? A virus program has to be placed in your RAM, and your computer has to jump to the virus program and run it. When it runs, it locates a file that is already on your hard disk, that contains a program that gets run on a regular basis by your computer, like some part of the operating system. After the virus program locates this file, it copies the virus program to the end of this file, and inserts a jump instruction at the beginning of the file that causes a jump to where the virus program is. Now your computer has a virus.

When a computer with a virus is running, it does all of the things it is supposed to do, but whenever it runs the program that contains the virus, the inserted jump instruction causes the virus program to be run instead. Now the virus usually will do something simple, like check for a predetermined date, and if it is not a match, then the virus program will jump back to the beginning of the file where the operating system program still exists.

Thus, your computer will appear totally normal, there are just a few extra instructions being executed during its regular operations. The virus is considered dormant at this point. But when that date arrives, and the virus ‘decides’ to do whatever is in the rest of its program, it can be anything. When the virus program is running, it can do whatever mischief the person who wrote it could think of. It can erase files on your disk, or send them somewhere else via the internet. One humorous virus would, every once in a while, make the letters on the screen appear to come loose and fall into a pile at the bottom of the screen.

Here’s an example of how to catch a virus. Let’s say that you have a friend who finds a funny movie on the Internet. It makes him laugh, and he thinks that you will enjoy it too, so he emails the movie file to you. You receive the movie file and play it, and you do enjoy it.

There are two different things that could have occurred here. If your friend sent you a file named “funny.mov,” and your OS includes a program that plays ‘.mov’ files, then the OS will load that program into RAM, and that program will read the pictures in the “funny.mov” file and display them on your screen. This is fine, the program that ran was something that was already on your computer. The “funny.mov” file just provided a series of pictures that were displayed on your screen.

But if your friend sent you a file named “funny.exe,” then when you ask the OS to play the movie, it will load “funny.exe” into RAM and jump to its first instruction. Now you have a program running in your computer that came from somewhere else. If it is a virus program, it will probably play the movie for you so that you don’t suspect anything, but it can do anything else that it wants, to the files on your disk while you are watching the movie. It will probably install itself and go into a dormant state for days or weeks, and you won’t even know that your computer is ‘infected.’ But sooner or later it will come alive and do whatever damage it was designed to do.

This sort of malicious program is called a virus because the way it works is similar to the way that real viruses infect living things. A real virus is a thing that is smaller than a one celled animal. It doesn’t quite qualify as being alive because the virus by itself cannot reproduce. They do reproduce, however, by invading a cell of something that is alive. Once in the cell, the virus uses the mechanisms of that cell to make copies of itself, which can then go on and infect other cells.

The computer virus also cannot reproduce or do anything else by itself. It needs to get into a computer, and somehow get itself executed one time by that CPU. When it runs that first time, it inserts itself somewhere into the operating system so that it will thereafter get executed on a regular basis. Those instructions will do whatever damage they are designed to do to the computer on which they are running, and they will also usually do something that is designed to spread the virus to other computers.

Firmware

Of course, RAM is an essential part of any computer. The ability to write bytes into RAM, and read them back out again is an integral part of how the machine works.

But in some computers, there are sections of the RAM that only get written to when the computer starts up, and thereafter these sections remain unchanged as the computer operates. This could be true in any computer that always runs the same program. Perhaps half of the RAM is used to contain the program, and the other half of the RAM is used to contain the data that the program is working on. The half with the program has to be loaded at some point, but after that, the CPU only has to read the bytes of the program in order to fetch and execute them.

When you have this sort of situation, you can build half of your computer's RAM the normal way, and with the other half, you skip the NAND gates, and just wire each bit directly to an on or an off in the pattern of your program.

Of course, you can't write into the pre-wired RAM, but you can read from it just fine. This type of RAM was given the name Read Only Memory, or ROM for short. You use it the same way you use RAM, but you only read from it.

There are two advantages to ROM. In the early days of computers, when RAM was very expensive, ROM was a lot less expensive than RAM.

The other advantage is that you no longer have to load the program into RAM when you first turn the computer on. It is already there in ROM, ready to be executed by the CPU.

The point here is a new word. Since software was named 'soft' because it is changeable, when it comes to ROM, you still have a pattern in the bits, but they're not so soft anymore. You can't write into a ROM, you can't change the bits. And so this type of memory came to be known as 'firmware.' It is software that is permanently written into hardware.

But that isn't the end of the story. The ROM described above had to be built that way at the factory. Over the years, this idea was improved and made easier to use.

The next advance was when someone had the bright idea of making ROM where every bit was set on at the factory, but there was a way of writing to it with a lot of power that could burn out individual connections, changing individual bits to an off. Thus this ROM could be programmed after leaving the factory. This was called 'Programmable ROM' or 'PROM' for short.

Then someone figured out how to make a PROM that would repair all of those broken connections if it were exposed to ultraviolet light for a half an hour. This was called an 'Erasable PROM', or 'EPROM' for short.

Then someone figured out how to build an EPROM that could be erased by using extra power on a special wire built into the EPROM. This was called 'Electrically Erasable PROM', or 'EEPROM' for short. One particular type of EEPROM has the name 'Flash memory.'

So there is RAM, ROM, PROM, EPROM, EEPROM and Flash. These are all types of computer memory. The thing they have in common is that they all allow random access. They all work the same way when it comes to addressing bytes and reading out the data that is in them. The big difference is that RAM loses its settings when the power goes off. When the power comes back on, RAM is full of all zeros. The rest of them all still have their data after power off and back on.

You may ask then, "Why don't computers use EEPROM for their RAM? Then the program would stay in RAM when the computer was off." The answer is that it takes much longer to write into EEPROM than RAM. It would slow the computer down tremendously. If someone figures out how to make an EEPROM that is as fast and as cheap and uses the same or less power as RAM, I'm sure it will be done.

By the way, the word ROM has also come to be used to mean any type of storage that is permanently set, such as a pre recorded disk, as in 'CD ROM,' but its original definition only applied to something that worked just like RAM.

Boots

What do boots have to do with computers? Well, there is an old phrase that goes “pull yourself up by your own bootstraps.” It is kind of a joke, it literally refers to the straps that are sewn into many boots that are used to help pull the boots onto your feet. The joke is that if you are wearing such a pair of boots, and want to get up off the ground, instead of getting a ladder or climbing a rope, you can get yourself off the ground by simply pulling hard enough on those bootstraps. Of course this would only work in a cartoon, but the phrase has come to mean doing something when there is no apparent way to do it, or doing something without the tools that would normally be used, or accomplishing something by yourself without help from anyone else.

In a computer, there is a problem that is similar to needing to get off the ground and having no tools available to accomplish it. When a computer is operating, the memory is full of programs that are doing something, and when the operator of the computer enters a command to start another program, the operating system locates the program on disk, loads it into memory, and jumps to the first instruction of the program. Now that program is running.

But when you first turn on a computer, how do you get the operating system into memory? It takes a program running in memory to tell the disk drive to send over some instruction code, and the program needs to write that code into memory at an appropriate place, and then jump to its first instruction to get the new program running. But when you turn the computer on, every byte in memory is all zeros. There are no instructions in memory at all. This is the impossible situation, you need a program in memory to get a program in memory, but there is nothing there. So in order for the computer to get going in the first place, the computer has to do something impossible. It has to pull itself up by its bootstraps!

A long time ago, in the early days of computers, the machine had switches and push buttons on the front panel that allowed the operator to enter bytes of data directly into the registers, and from there, into RAM. You could manually enter a short program this way, and start it running. This program, called a “bootstrap loader,” would be the smallest possible program you could write that would instruct the computer to read bytes from a peripheral, store them in RAM, and then jump to the first instruction. When the bootstrap loader executes, it loads a much larger program into memory, such as the beginnings of an operating system, and then the computer will become usable.

Nowadays, there are much easier ways of loading the first program into the computer, in fact it happens automatically immediately after the computer gets turned on. But this process still happens, and the first step is called “booting” or “booting up” and it only means getting the first program into memory and beginning to execute it.

The most common solution to this problem has three parts. First, the IAR is designed so that when the power is first turned on, instead of all of its bits being zero, its last bit will be zero, but the rest of its bits will be ones. Thus for our little computer, the first instruction to be fetched will be at address 1111 1110. Second, something like the last 32 bytes of the RAM (235-256) will be ROM instead, hardwired with a simple program that accesses the disk drive, selects head 0, track 0, sector 0, reads this sector into RAM, and then jumps to the first byte of it. The third part then, had better be that there is a program written on that first sector of the disk. This sector, by the way, is called the ‘boot record.’

This word ‘boot’ has become a verb in computer talk. It means to load a program into RAM where there are no programs. Sometimes people use it to mean loading any program into RAM, but its original meaning only applied to loading the first program into an otherwise blank RAM.

Digital vs. Analog

You’ve no doubt heard these terms bandied about. It seems that anything associated with computers is digital, and everything else is not. But that’s not quite close enough to the truth.

What they mean is quite simple, but where they came from and how they ended up in their current usage is not so straightforward.

The word ‘digital’ comes from digit, which means fingers and toes in some ancient language, and since fingers and toes have been used for counting, digital means having to do with numbers. Today, the individual symbols that we use to write numbers (0, 1, 2, 3, etc.) are called digits. In the computer, we represent numbers with bits and bytes. One of the qualities of bits and bytes is their unambiguous nature. A bit is either on or off; there is no gray area in between. A byte is always in one of its 256 states; there is no state between two numbers like 123 and 124. The fact that these states change in steps is what we are referring to when we say digital.

The word ‘analog’ comes from the same place as ‘analogy’ and ‘analogous,’ thus it has to do with the similarity between two things. In the real world, most things change gradually and continuously, not in steps. A voice can be a shout or a whisper or absolutely anywhere in between. When a telephone converts a voice into an electrical equivalent so that it can travel through a wire to another telephone, that electricity can also vary everywhere between being fully on and fully off. Sound and electricity are two very different things, but the essence of the voice has been duplicated with electricity. Since they are similar in that respect, we can say that the electrical pattern is an ‘analog’ of the voice. Although the meaning of ‘analog’ comes from this ‘similarity’ factor, when you make an analog, you are usually making an analog of something that is continuously variable. This idea of something being continuously variable has come to be the definition of analog when you are comparing digital and analog. Something that is analog can be anywhere within the entirety of some range, there are no steps.

Digital means change by steps and analog means change in a smooth continuous manner. Another way to say it is that digital means that the elements that make up a whole come from a finite number of choices, whereas analog means that a thing is made of parts that can be selected from an unlimited number of choices. A few non-computer examples may help to clarify this.

If you have a platform that is three feet above the floor, you can either build stairs for people to climb up to it, or a ramp. On the ramp, you can climb to any level between the floor and the platform; on the stairs, you only have as many choices as there are steps. The ramp is analog, the stairs are digital.

Let’s say that you want to build a walkway in your garden. You have a choice of making the walkway out of concrete or out of bricks. If the bricks are three inches wide, then you can make a brick walk that is 30 inches wide, or 33 inches wide, but not 31 or 32. If you make the walk out of concrete, you can pour it to any width you want. The bricks are digital, the concrete is analog.

If you have an old book and an old oil painting, and you want to make a copy of each, you will have a much easier time making a copy of the book. Even if the pages of the book are yellowed, and the corners are dog-eared, and there are dirt smudges and worm holes inside, as long as you can read every letter in the book, you could re-type the entire text, exactly as the author intended it. With the oil painting, the original colors may have faded and are obscured by dirt. The exact placement of each bristle in each brush stroke, the thickness of the paint at every spot, the way adjacent colors mix, could all be copied in great detail, but there would inevitably be some slight differences. Each letter in the book comes from a list of a specific number of possibilities; the variations of paint colors and their positions on the canvas are limitless. The book is digital, the painting is analog.

So there you have the difference between analog and digital. The world around us is mostly analog. Most old technologies were analog, like the telephone, phonograph, radio, television, tape recorders and videocassettes. Oddly enough though, one of the oldest devices, the telegraph, was digital. Now that digital technology has become highly developed and inexpensive, the analog devices are being replaced one by one with digital versions that accomplish the same things.

Sound is an analog thing. An old fashioned telephone is an analog machine that converts analog sound into an electrical pattern that is an analog of the sound, which then travels through a wire to another phone. A new digital telephone takes the analog sound, and converts it into a digital code. Then the digital code travels to another digital phone where the digital code is converted back into analog sound.

Why would anyone go to the trouble of inventing a digital phone when the analog phone worked just fine? The answer, of course, is that although the analog phone worked, it was not perfect. When an analog electrical pattern travels over long distances, many things can happen to it along the way. It gets smaller and smaller as it travels, so it has to be amplified, which introduces noise, and when it gets close to other electrical equipment, some of the pattern from the other equipment can get mixed in to the conversation. The farther the sound goes, the more noise and distortion are introduced. Every change to the analog of your voice becomes a part of the sound that comes out at the other end.

Enter digital technology to the rescue. When you send a digital code over long distances, the individual bits are subjected to the same types of distortion and noise, and they do change slightly. However, it doesn't matter if a bit is only 97% on instead of 100%. A gate's input only needs to 'know' whether the bit is on or off, it has to 'decide' between those two choices only. As long as a bit is still more than half way on, the gate that it goes into will act in exactly the same way as if the bit had been fully on. Therefore, the digital pattern at the end is just as good as it was at the beginning, and when it is converted back to analog, there is no noise or distortion at all, it sounds like the person is right next-door.

There are advantages and disadvantages to each method, but in general, the benefits of digital technology far outweigh its shortcomings.

Probably the biggest advantage of digital has to do with the making of copies. When you make a copy of something like a vinyl record, you could record it to a tape recorder, or I guess you could even get all of the equipment to cut a new vinyl record. But there will be some degree of difference between the original and the copy. In the first place, all machinery has accuracy limitations. A copy of any physical object can be very close to the original, but never quite exact. Second, if there are any scratches or particles of dust on the original, the copy will then have duplicates of these defects. Third, friction between the record and the needle actually wears away a tiny amount of vinyl every time you play it. If you use a tape recorder, there is always a low level of 'hiss' added to the sound. If you make a copy of a copy, and a copy of that, etc. the changes will get larger and larger at each stage.

When it comes to something that is digital, as long as every bit that was on in the original is also on in the copy, we get an exact copy every time. You can make a copy of the copy, and a copy of that, etc., and every one of them will be exactly the same as the original. Digital is definitely the way to go if you want to be able to make an unlimited number of copies and preserve something for all time.

The computer and peripherals we have built are entirely digital so far. And if all we ever wanted to do with them were digital things such as arithmetic and written language, we could leave it that way. However, if we want our computer to play music and work with color photographs, there is one more thing we need to look at.

I Lied – Sort of

There is one piece of hardware in a computer that is not made completely out of NAND gates. This thing is not really necessary to make a computer a computer, but most computers have a few of them. They are used to change from something that is analog to something that is digital, or digital to analog.

Human eyes and ears respond to analog things. Things that we hear can be loud or soft, things that we see can be bright or dark and be any of a multitude of colors.

The computer display screen that we described above had 320 x 200 or 64,000 pixels. But each pixel only had one bit to tell it what to do, to be on or off. This is fine for displaying written language on the screen, or it could be used to make line drawings, anything that only has two levels of brightness. But we have all seen photographs on computer screens.

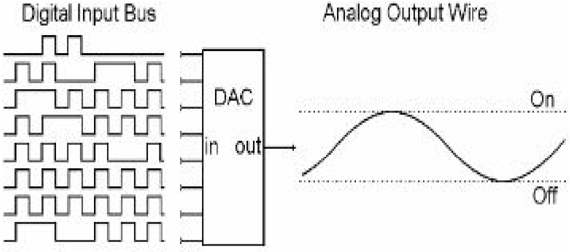
First of all, there needs to be a way to put different colors on the screen. If you get out a magnifying glass and look at a color computer or television screen, you will see that the screen is actually made up of little dots of three different colors, blue, red, and green. Each pixel has three parts to it, one for each color. When the display adapter scans the screen, it selects all three colors of each pixel at the same time.

For a computer to have a color screen, it needs to have three bits for each pixel, so it would have to have three times the RAM in order to be able to control the three colors in each pixel individually. With three bits, each color could be fully on or off, and each pixel would therefore have eight possible states: black, green, red, blue, green and red (yellow,) green and blue (cyan,) blue and red (magenta) and green, blue and red (white.)

But this is still not enough to display a photograph. To do that, we need to be able to control the brightness of each color throughout the range between fully on and fully off. To do this, we need a new type of part that we will describe shortly, and we need more bits in the display RAM. Instead of one bit for each color in each pixel, we could have a whole byte for each color in each pixel. That's three bytes per pixel, for a total of 192,000 bytes of RAM just for this small display screen.

With these bytes, using the binary number code, you could specify 256 levels of brightness for each color in each pixel. This would amount to 16,777,216 different states (or colors) for each pixel. This is enough variety to display a reasonably good-looking photograph.

In order to make this work – a number specifying 256 different levels of brightness – you need a thing called a "digital to analog converter" or "DAC" for short. A DAC has eight digital inputs, and one analog output. The way it works is that it is wired up to treat the input as a binary number, and the output has 256 levels between off and on. The output has 256 gradations between off and on, and it goes to the level that the input number specifies. If the input is a 128, the output will be halfway on. For a 64 the output will be one quarter on. For 0, the output will be fully off.



In order to make this color screen work, the display adapter needs to access three bytes at a time, connect them to three DACs, and connect the outputs of the DACs to the three colors in the current pixel being painted. That's how a color screen works.

When we defined 'analog' in the last chapter, we said that it was something that was continuously variable from fully off to fully on. But our DAC really only has 256 different levels at its 'analog' output. It's a lot closer to being analog than a bit, but it still has steps. What the computer is doing is approximating an analog thing in steps small enough to fool the intended audience. When it comes to the eye, 256 different levels of brightness is sufficient.

If something requires smaller steps to fool the intended audience, you can make a DAC that has 16 bits on the digital side. Thus you can present the digital input with a number anywhere from 0 to 65535. The analog side can still only vary from fully off to fully on, but the size of the steps will be much smaller since there are now 65536 of them.

When it comes to the ear, it can hear very small differences, and so a 16 bit DAC is required for high quality sound.

All sounds, from music to speech to thunder crashes are vibrations of the air. They vary in how fast the air vibrates, and in exactly how it vibrates. The human ear can hear vibrations from about 20 Hz at the low end to 20,000 Hz (20 kHz) at the high end, so this is the range of vibrations that computers are designed to deal with. For any electronic machine to make sounds, there is a device called a speaker. All that a speaker does is move back and forth in the air, making the air vibrate. If it makes the air vibrate in precisely the same way as the original thing that was recorded, it will sound just like the original.

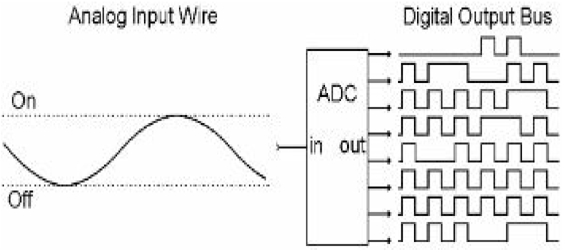
In order to store a sound in a computer, the position of the speaker is divided into 65536 possible positions. Then a second is divided into 44,100 parts. At each one of those parts of a second, the desired position of the speaker is stored as a two-byte number. This is enough information to reproduce sound with very high quality.

To play top quality stereo music, a computer would need a 'sound peripheral.' This would have two 16 bit DACs with their analog outputs connected to speakers. It would also have its own clock that ticks at 44,100 Hz. At each tick, it would get the next two two-byte numbers, and connect them to the digital side of the

DACs.

As far as speed goes, this would be 176,400 bytes per second. Certainly that is fast, but remember that our computer clock ticks a billion times per second. That means that the computer can send four bytes to the sound peripheral, and go off and execute about 4000 instructions on some other task before it needs to send the next four.

For going the other way, there is an “Analog to Digital Converter,” or “ADC” for short. This is used to convert the sound from a microphone into a series of bytes, or for a camera to convert a picture into a series of bytes. The input has one wire that can be anywhere from all the way off to all the way on. The ADC makes its outputs into a number from 0-255 for an 8-bit ADC or from 0-65,535 for a 16-bit ADC. This number represents how much the input is on or off. Half on is 128 or 32,768, one quarter on is 64 or 16,384, etc. This process is just the reverse of what a DAC does.



DACs and ADCs are not made out of NAND gates, they have electronic parts like radios have. How they do what they do is not a proper subject for this book. So maybe I lied when I said that everything in a computer is made out of NAND gates? Well, not really, because DACs and ADCs are only used in certain types of peripherals, not in the computer itself.

Full Disclosure

We have built a very small computer here. It is about the smallest computer that could be invented that does everything necessary to be worthy of the name computer. I don't think that anyone has built such a small computer since about 1952, and no one has ever built this exact computer in the real world.

If a real computer designer ever read this book, I'm sure he'd be pulling his hair out over all of the opportunities that have been missed here to make a better machine. But again, the goal has been to illustrate computer principles as simply as possible.

This is an eight-bit computer. That means that the registers in the processor are eight bits, the bus is eight bits, and in this machine, even the Memory Address Register is eight bits.

With most of the computers that actually get built, while the individual bytes in RAM remain 8 bits, everything else is expanded to 16 bits, 32 bits or 64 bits or a combination of these in different parts of the machine.

Our RAM only has 256 bytes, which is ridiculously small, but that's all you can have with an eight-bit Memory Address Register. If you use 16 bits, you can have 65,536 bytes of RAM (that's 64kb), if you use 24 bits you can have 16mb, if you use 32 bits you can have 4 gigabytes of RAM.

Real computers have things that this one does not, but they are not capable of doing things that this computer cannot do.

In our computer, if you want to shift a byte three bits to the left, you would put three shift left instructions in your program. In most real computers, they have shifters that will shift any number of bits in one instruction. But the result is the same, your byte ends up looking the same in either case, the real computer just gets the job done faster.

In our computer, the adder can add two eight-bit numbers. If you want to add 16 bit numbers, you have to employ some software to do it. In most computers, the adder can add 16 or 32 bit numbers in one instruction. Again, the results are the same, one is just faster than the other.

The stepper in our computer is a simplification of something that most computers have, called a 'state machine.' State machines provide steps, but start the next instruction as soon as possible, do what is necessary for an interrupt system, can create more complex instructions, etc. Since all we needed was six consecutive steps, we built a simpler thing and just made up the term 'stepper.'

So yes, our computer is a simple, small, relatively slow computer, but it can do everything that more complicated machines can do. The things that make a bigger machine bigger, are designed to get the job done faster, do it in fewer clock cycles, do the same task with fewer instructions, operate on several bytes at the same time. But the nature of what the machines do is exactly the same. Every task they can do comes down to shifting, ANDing, ORing, XORing, ADDing and NOTing bytes. There are no other fancy types of operations that have been left out of this book.

In a bigger machine, you can do addition, subtraction, multiplication and division in a single instruction. That is because they have huge numbers of gates arranged into things like a 'hardware multiplier.' There is no reason to show you the details of how you construct one of these, it is a very complicated job for the few people who need to build one. It is understandable, and it all ultimately comes down to NAND gates just like everything else. But we have seen how to do all the math operations there are with just an adder, shifter, NOT gates and some software. The hardware multiplier gets there faster, but the results are exactly the same.

Bigger machines have more registers, the registers are each multiple bytes, they have adders that can add three numbers at the same time, but still the instructions come down to the same simple operations. Your understanding of computers is not small because we have looked at a small computer.

Philosophy

Why do we have a chapter called "Philosophy" in a book about computers? The only thing in this book that even comes close to being a philosophical question is its title, "But How do it Know?" We will attempt to answer this question a little later on.

This book has been about the computers that we have today. But what about the future? As computers and software continue to advance, how soon if ever, will the day come when there are walking talking computerized robots that look and act just like people? Will the day come when we have to decide whether or not to give these robots the same legal rights as people? Will computers eventually take over the world and replace people altogether?

To answer these sorts of questions, people often refer to a major question that has been outstanding in the field of philosophy for many years.

The question is, whether man is composed solely of the structural body that we can see and dissect, or whether there is an integral spiritual component to every human being which accounts for the qualities of consciousness, love, honor, happiness, pain, etc.

That question is far beyond the scope of this book, and it remains unconvincingly answered despite many books arguing each viewpoint. There are people in the sciences who say that we are on track to building conscious computers, and it will happen. There are people in the humanities who say that it is impossible because you can't manufacture a spirit. Each side has been unable to sway the other.

If we define the brain as that funny looking chunk of gray meat enclosed by the skull, and define the mind as whatever it is that is responsible for consciousness, memory, creativity, thinking, and everything else that we notice going on in our heads, then we can restate the big philosophic question as: "Are the brain and the mind one and the same thing?"

Then when it comes to the question about building a convincing human robot, there would be two possibilities.

If the brain and the mind are the same thing, you might not be able to build a synthetic person today, but as time went on, eventually you could understand every structure and function in the brain, and build something of equal complexity that would generate true consciousness, and that really should act just like any other person.

If the brain and the mind are not the same thing, then building a robot buddy will always be about simulating humanity, not building something of equal quality and value.

Restating the question doesn’t make it any easier to answer, but this idea of separating what we know about minds from what we know about brains may be useful. Early on, we said that we were going to show how computers work so that we could see what they were capable of doing, and also what they were not capable of doing. We are going to take what we know about brains and what we know about minds and compare each individually to our new knowledge about computers. In doing so we can look for differences and similarities, and we may be able to answer a few less controversial questions.

Computers do certain things with great ease, such as adding up columns of numbers. A computer can do millions of additions in a single second. The mind can barely remember two numbers at the same time, never mind adding them up without a pencil and paper.

The mind seems to have the ability to look at and consider relatively large amounts of data at the same time. When I think of my favorite cat, I can re-experience seeing what he looks like, hearing the sounds of his purring and mewing, feeling the softness of his fur and his weight when picked up. These are some of the ways that I know my pet.

What would it mean for our computer to think about a cat? It could have pictures of the cat and sounds of the cat encoded in files on a spinning disk or in RAM. Is that thinking? If you ran the bytes of these files one by one through the ALU, would that be thinking? If you put the picture on the screen, would that be thinking? If you played the sounds to the speakers, would that be thinking?

The sounds and pictures encoded in the computer are just byte patterns sitting where they are. They don't look like anything or sound like anything unless they are sent to the peripherals for which they were designed. And if they are sent to the screen and speakers, the computer doesn't see them or hear them. Of course, your computer could have a camera pointing at the screen, and a microphone listening to the sounds, but the computer still wouldn't see a picture or hear a sound, it would just collect more strings of bytes very similar to the ones sent to the screen and speakers in the first place.

There could be programs that perform mathematical operations on the picture files in order to discover patterns, and store the results of these calculations in other files. There could be files that relate one picture file to other similar picture files, and pictures to sounds, etc., creating more files.

But no matter how much programming is applied to the picture files, there is something that the mind can do that the computer simply doesn't have any facility for.

The mind can consider the whole of some thing all at the same time. You can think of the whole of the cat all at once. Its sort of like the difference between the movie film and the TV screen. The movie film has whole pictures, the TV screen only has one pixel at a time. You could say that your mind works so quickly that you don't notice the details, it gets integrated into a whole just like the pixels get integrated into an entire picture. But what does the integrating? And when it's integrated, what is it and where is it? And what looks at the integrated whole?

We've just seen everything that's in a computer. The computer moves one byte at a time over the bus. The fanciest thing it does is to add two bytes into one. Everything else it 'does' amounts to nothing more than the simple warehousing of bytes. A stored byte doesn't do anything beyond maintaining its own current setting.

A computer just doesn't have any facilities that integrate the elements of a picture into anything else, nowhere to store that something else, and nothing with which to look at it.

I'm not saying that something couldn't be built that would perform these functions, I'm just saying that computers as we know them today don't currently include any such device.

Here is another question. If a brain works like a computer, then it needs to have a program for the CPU to run. Where would this program come from?

Although the brain has trillions of cells, the entire human body starts with one fertilized egg cell. So any program that the brain has, would have to be present in this single cell, presumably in the DNA.

Scientists have now decoded the entire DNA sequence of humans. DNA is interesting in that it is a long string of only four types of things. It's digital! A lot of the pieces of this string are used for making chemical reactions take place to make proteins, etc. but the majority of it is called 'junk DNA' because no one knows what its purpose is. But even if you consider that the entirety of the DNA is devoted to computer software, then there could be about a billion instructions in this program. Now that's a lot of software, but the average home computer probably has that much software loaded onto it's hard drive, and that wouldn't be anywhere near enough to run a human being.

Some have said that the human computer programs itself. As a programmer myself, I just can't imagine how this would work. While it's true that a program can accumulate data and modify the way it works based on the collected data, this is not the same thing as writing a new program. If someone ever writes this program that can write any new needed program, there will be a huge number of computer programmers put out of work forever.

Then there are the kinds of errors that computers make versus the kind that people make. If a computer gets stuck in a loop, it appears to have stopped completely.

Have you ever seen a person walking down the street suddenly stop working? All functions just cease. The person would just fall down until somehow his computer re-booted. People do collapse from time to time, but it is usually because some other part broke, like having a heart attack, and you can see the person recognize the pain as it takes them down. But if the human computer got stuck in a loop, there would be an instant loss of consciousness and the body would just fall completely limp with no struggle. I have never seen that, but if the brain operated just like a computer, you would expect to see it on a fairly regular basis.

Then there is the matter of speed. As we have seen, a simple computer can do a billion things in a second. When it comes to the brain, it has nerves that have some similarity to the wires in computers. Nerves can also carry electricity from place to place. In a computer, wires come out of gates and go into other gates. In the brain, nerves are connected together by "synapses." These synapses are spaces between nerves where the electricity in one nerve creates a chemical reaction, which then causes the next nerve to create its own electricity. These chemical reactions are painfully slow.

No one has shown that these nerves are connected up anything like the wires in a computer, but their lack of speed makes it very unlikely that it would do much good even if the connections were similar. After the electricity travels quickly through the nerve cell, it reaches the synapse, where the chemical reaction takes about one five hundredth of a second to complete. That means that our simple computer built out of NAND gates could do two million things in the same time that only one thing could be done by a computer built out of nerves and synapses.

Another area where the difference between the mind and computers is quite obvious, is in the area of recognizing faces. The mind is very good at it. If you walk into a party with fifty people present, you will know in a matter of seconds whether you are among a group of friends or of strangers. A lot of research has been done into how people accomplish this feat, and a lot of very interesting information has been uncovered. There is also a lot of speculation, and there are many fascinating theories about the underlying principles and mechanisms. But the complete and exact structures and functions have not been uncovered.

If you give a computer a picture file of a person, and then give it the same file again, it can compare the two files byte by byte and see that each byte in one file is exactly equal to the corresponding byte in the other file. But if you give the computer two pictures of the same person that were taken at different times, or from different angles, or with different lighting, or at different ages, then the bytes of the two files will not match up byte by byte. For the computer to determine that these two files represent the same person is a huge task. It has to run very complex programs that perform advanced mathematical functions on the files to find patterns in them, then figure out what those patterns might look like from different angles, then compare those things to every other face it has ever stored on its disk, pick the closest match, then determine if it's close enough to be the person or just someone that looks similar.

The point is that computers have a method of dealing with pictures based on the principles on which computers work. Using these principles alone has not yet yielded computers or software that can recognize a face with anywhere near the speed and accuracy of any ordinary person.

Voice recognition by computers is another technology that has come a long way, but has much further to go to rival what the mind does easily.

So in comparing a computer to a brain, it just doesn't look very likely that they operate on the same principles. The brain is very slow, there isn't any place to get the software to run it, and we don't see the types of problems we would expect with computer software errors.

In comparing a computer to the mind, the computer is vastly better at math, but the mind is better at dealing with faces and voices, and can contemplate the entirety of some entity that it has previously experienced.

Science fiction books and movies are full of machines that read minds or implant ideas into them, space ships with built-in talking computers and lifelike robots and androids. These machines have varying capabilities and some of the plots deal with the robot wrestling with consciousness, self-realization, emotions, etc. These machines seem to feel less than complete because they are just machines, and want desperately to become fully human. It’s sort of a grown-up version of the children’s classic “Pinocchio,” the story about a marionette who wants to become a real boy.

But would it be possible to build such machines with a vastly expanded version of the technology that we used to build our simple computer?

Optimism is a great thing, and it should not be squashed, but a problem will not be susceptible to solution if you are using a methodology or technology that doesn’t measure up to that problem. In the field of medicine, some diseases have been wiped out by antibiotics, others can be prevented by inoculations, but others still plague humanity despite the best of care and decades of research. And let’s not even look into subjects like politics. Maybe more time is all that’s needed, but you also have to look at the possibility that these problems either are unsolvable, or that the research has been looking in the wrong places for the answer.

As an example, many visions of the future have included people traveling around in flying cars. Actually, several types of flying cars have been built. But they are expensive, inefficient, noisy and very dangerous. They work on the same basic principles as helicopters. If two flying cars have any sort of a minor accident, everyone will die when both cars crash to the Earth. So today’s aviation technology just won’t result in a satisfactory flying car. Unless and until someone invents a cheap and reliable anti-gravity device, there will not be a mass market for flying cars and traffic on the roads will not be relieved.

If you want to build a machine that works just like a person, certainly the best way to do it would be to find out how the person works and then build a machine that works on the same principles, has parts that do the same things, and is wired up in the same way as a person.

When Thomas Edison invented the phonograph, he was dealing with the subject of sound. Sound is a vibration of the air. So he invented an apparatus that captured the vibrations in the air and transformed them into a vibrating groove on the surface of a wax cylinder. The sound could then be recreated by transferring the vibrations in the groove back into the air. The point is, that in order to recreate sound, he found out how sound worked, and then made a machine that worked on the same principle. Sound is a vibration, the groove in a phonograph is a vibration.

A lot of research has been done on the subject of what makes people tick. A lot of research has been done on the subject of how to make computers do the things that people do. A lot of things have been discovered and a lot of things have been invented. I do not want to minimize any of the work done, or results achieved in these areas.

But there are many things that have not yet been discovered or invented.

Many dead brains have been dissected and their parts have been studied and classified. The brain does contain nerve cells which move electricity from one place to another. This is a similarity between brains and computers. But research into the actual operation of living human brains is necessarily limited. Most observations have been made during surgeries that were necessitated by accident or disease. Many observations have been made of changes to behavior after an injury or disease has disabled certain parts of the brain. From this research, it has been possible to associate certain functions with certain areas of the brain.

But no one has discovered a bus, a clock, any registers, an ALU or RAM. The exact mechanism of memory in the brain remains a mystery. It has been shown that nerves grow new connections over time, and it is assumed that this is the mechanism of learning, but no one has been able to say that this particular nerve does this exact function, as we can do with the individual wires in a computer.

Everything that goes into a computer gets turned into one code or another. The keyboard generates one byte of ASCII per keystroke, a microphone generates 44,100 binary numbers per second, a color camera generates three binary numbers per pixel, 30 times a second, and so on. No one has isolated the use of any codes like ASCII, binary numbers, fonts or an instruction code in the brain. They may be there, but they have not been isolated. No one has traced a thought or located a memory in the same way that we could follow the operation of a program in a computer.

It is widely assumed that the brain works in some much more spread out way than a single computer, that there are thousands or billions of computer elements that cooperate and share the work. But such elements have not yet been located. In the world of computing, this idea is called ‘parallel processing’ and computers with dozens or hundreds of CPUs have been built. But these computers still haven’t resulted in a human substitute.

Think of it all as a puzzle. How people work is one side of the puzzle. Making computers do things that people do is the other side of the puzzle. Pieces of the puzzle are being assembled on both sides. The problem is that as progress is being made on both sides, it looks more and more like these are two different puzzles, they are not coming together in the middle. They are not converging into a single picture.

The researchers are very aware of these developments. But when it comes to pop culture, people hear about new inventions all the time, and see the future portrayed in science fiction films, and the logical conclusion seems to be that research will continue to solve the problems one by one until in 10 or 20 or 30 years we will have our electro-mechanical friends. In the past century we conquered electricity, flight, space travel, chemistry, nuclear energy, etc. So why not the brain and/or mind? The research, however, is still at the stage where every time one new answer is found, it creates more than one more new question.

So it appears that whichever way we look at it, neither the brain nor the mind work on the same principles as computers as we know them. I say 'as we know them' because some other type of computer may be invented in the future. But all of the computers we have today come under the definition of 'Stored Program Digital Computers,' and all of the principles on which they operate have been presented in this book.

Still, none of this 'proves' that a synthetic human could never be built, it only means that the computer principles as presented in this book are not sufficient for the job. Some completely different type of device that operates on some completely different set of principles might be able to do it. But we can't comment on such a device until someone invents one.

Going back to a simpler question, do you remember Joe and the Thermos bottle? He thought that the Thermos had some kind of a temperature sensor, and a heater and cooler inside. But even if it had had all of that machinery in it, it still wouldn't "know" what to do, it would just be a mechanical device that turned on the heater or cooler depending on the temperature of the beverage placed in it.

A pair of scissors is a device that performs a function when made to do so. You put a finger and thumb in the holes and squeeze. The blades at the other end of the scissors move together and cut some paper or cloth or whatever it is that you have placed in their way. Do the scissors "know" how to cut shapes out of paper or how to make a dress out of cloth? Of course not, they just do what they're told.

Similarly, NAND gates don't "know" what they are doing, they just react to the electricity or lack of it placed on their inputs. If one gate doesn't know anything, then it doesn't matter how many of them you connect together, if one of them knows absolutely zero, a million of them will also know zero.

We use a lot of words that give human characteristics to our computers. We say that it "knows" things. We say it "remembers" things. We say that it "sees," and "understands." Even something as simple as a device adapter "listens" for its address to appear on the I/O bus, or a jump instruction “decides” what to do. There is nothing wrong with this as long as we know the truth of the matter.

Now that we know what is in a computer, and how it works, I think it is fairly obvious that the answer to the question “But How do it Know?” is simply “It doesn’t know anything!”