Chapter 3: Digital Storage

Requirements

- ❑ Basic mathematics, Algebra helpful.
- ❑ Familiarity with the parts of a computer and how they work as explained in Chapter 1: Computers and Programming.
- ❑ Familiarity with binary and hexadecimal numbering systems, as explained in Chapter 2: Numbering Systems.

Introduction

You're probably sick and tired of seeing numbers, but I promise this is the last chapter you'll have to deal with them. I'm lying of course, because computers are based entirely in numbers using binary and you're going to be dealing with them a lot.

Digital Data

Computers and their corresponding digital storage devices use binary to store all of its data and thus all data to computers is *numerical* in nature. Binary is used because it relates directly to the construction of digital storage devices: on/off switches. Each bit represents a switch that is on (binary one) or off (binary zero). Combinations of these switches produce the ability to store larger numbers.

A sequence of bits that is used to store a single numeric value is known as a storage unit. The value contained in a storage unit is finite. Its range is limited to the amount of bits available.

For example, digital clock faces can display up to four decimal digits. The maximum number displayed is limited to the maximum of each digit put together. This is the same with storage units. The maximum value of a storage unit is limited to the amount of bits that can be turned on.

In a higher level sense, you may think of a storage unit as a water cup. It can store any amount of water that does not spill over the edge and it can also contain no water at all. Its *range* of water volume, therefore, is very finite and is not subject to change. However much water you put in that cup, or how you use it, is completely up to you. It is the same with this *digital data*.

There are several common storage units and each larger storage unit is typically measured in the amount of smaller storage units it makes up; so a *nibble* is four (4) bits, a *byte*[1] is two (2) nibbles, a *word* is two (2) bytes, a *double-word* is two (2) words, and a *quad-word* is two (2) double-words:

Storage Unit	**Bits**	**Bytes**
Bit	1	1/8
Nibble	4	1/4
Byte	8	1
Word	16	2
Double-Word	32	4
Quad-Word	64	8

The size of most storage units is commonly expressed in bits and bytes to easily identify differences and calculate total space. Nibbles are usually referred to as 4 bits or *half* bytes rather than as a solo storage unit. Bytes are the almighty building block of all digital data.

Bytes are the smallest amount of data that can be referenced individually by the computer. In order to check the individual bit or nibble values of digital data, the entire byte must be inspected. A byte is like an ice tray that can contain eight ice cubes. In order to get the ice cubes from the freezer to your cup, you have to take out the whole tray and individually bust out the ones you need. When you store your ice cubes, you put them in this tray rather than stacking them individually.

Digital storage devices in your personal computer will include the CPU which contains registers, your RAM which contains memory, and your mass storage devices, like your hard-drive and CD-ROM, which contain your permanent data files. All of these store data in terms of bytes. This is the common storage unit that allows digital devices to interact with each other.

The term "nibble", or nybble, comes from the similarity of "byte" to "bite". Nibbling is considered smaller than biting, which is why a nibble is half a byte. The two nibbles that make up a single byte are known as the high and low order nibbles respectively where the high order is a nibble whose bit weights are greatest. Thus, the left-most nibble in 8-bits, or a byte, is the high order nibble.

The terms high and low order apply not only to nibbles, but to the other storage units as well: a quard-word contains high and low order double-words, a double-word contains high and low order words, and a word contains high and low order bytes.

Vast Amounts

[1] A byte on some, mostly rare, systems is not 8-bits, but that is the assumption of this book. The term octet, on the other hand, refers explicitly to an 8-bit storage unit.

Where vast amounts of data are concerned, specifically bits or bytes, the amount is referred to with a smaller value whose name is prefixed. These prefixes are: *kilo*, *mega*, *giga*, and *tera*. Each of these represents a particular weight which, when multiplied by the value included, represents an approximate number of bits or bytes. The prefix values start with 'kilo' whose weight is one thousand. The prefix values increase from there. Each following prefix weight is equal to the last weight squared. So 'mega' is one thousand squared or one million or one thousand thousand.

Now let's try these prefixes with data. A kilobyte is one thousand bytes. Ten kilobytes is ten thousand bytes. Thirteen megabits is thirteen million bits. Your hard drive is measured like this. If you have a twenty (20) gigabyte hard drive then it has twenty billion bytes available for storage. Just think; each one of those bytes can contain one of 256 different values. This means that, with your twenty gigabytes, you could have up to 1,310,720,000,000,000 different sets of data. That's a lot of possibilities!

Sometimes numbers will be trailed by an abbreviation rather than the whole word. The abbreviation is typically the first letter of the prefix and is sometimes followed by a 'b'. Instead of seeing '10 kilobytes' you'll probably see '10k' or '10kb'. The 'b', unfortunately, is extremely ambiguous and can mean *either* bytes or bits. The context of what it's used in is your only clue as to what it means.

When working with networks the transfer rates are always in terms of bits rather than bytes. Transfer rates are abbreviated by suffixing with 'bps' or *bits per second*. For example, '10kbps' is '10 kilobits per second'.

Unfortunately there are two ways of seeing the prefix values and I have only just shown you one, which is used in the world of hard drives mostly. The other way is the same except the weight of each prefix is a multiple of 1024, not 1000. This number is based on binary restrictions rather than simple metrics. So a 'kilo' is actually 1024, 'mega' is 1024 kilo, 'giga' is 1024 mega, and 'tera' is 1024 giga. Therefore ten kilobytes (10k) in this system is ten thousand two-hundred forty (10,240) bytes. This second system is used practically every where, save for specifying the size of a hard drive.

You won't have to deal much with the first because even operating systems will tell you hard drive information based on the 1024 variant. The only time you will need to know the first variant is when you actually want to buy a hard drive. A twenty gigabyte hard drive is twenty billion bytes not 21,474,836,480 ($20*1024^3$).

Multi-Byte Storage and Endians

Larger storage units, such as words and double-words, are made up of multiple bytes. That is, a single number that cannot be held in a single byte is spread across multiple bytes. The accumulative number of bits across these multiple bytes represents the numeric value in the storage unit. The arrangement of the bytes to represent a larger number, however, is device dependent. For example, imagine you got a package from your grandma that contained four unlabeled boxes, each containing eight chocolates, stacked sideways so you can see each box. The note inside the package says:

> *1st box = dark chocolate, 2nd box = milk chocolate, 3rd box = white chocolate, and 4th box = fake chocolate.*

How would you know which box was which? Automatically, we would assume that the first box is also the left-most. However, your grandma might have counted them the other way, *or* you may have the box turned around which means your left was her right. In the end, you really *don't* know which box represents which chocolates unless *you* or *someone thinking exactly like you* packed the boxes of chocolates. This is why *byte arrangement*, also known as *byte order*, can be confusing.

The part of a computer that assumes a specific byte order is the CPU. It reads instructions and performs them. Its instructions and the data that is used with them are all assumed by the CPU to be of a specific byte order. The two most common byte order types are known as *big endian*[2] and *little endian*.

Big endian, also known as *network byte order*, is the easiest to understand because it means that the left-most byte is the most significant and vice versa for the right-most. Thus, just like with bit significance, the weight of the left-most byte is the greatest and this weight gradually decreases as you move to the right. The following shows the weight of each bit of each byte for the common storage units in big endian format:

Big endian diagram

Little endian is the opposite of big endian and more difficult to humanly conceive. A little endian number indicates that the left-most byte is least significant and vice versa for the right-most. When you look at the bits, and each of their weights, of a little endian number it's easy to get confused. The following shows the weight of each bit of each byte for the common storage units in little endian format.

Little endian diagram

When one knows the endian-ness of a particular set of bytes, they can convert from one format to the other. This is done all the time with networks and thus the internet. Any computer running with a little endian CPU such as Intel or AMD must convert multi-byte numbers from any network, because they use big endian. It is *known* that networks use big endian so these little endian CPU's *know* to convert the numbers before using them.

High and low order units in multi-byte numbers are determined by the bit weights, not the physical byte order in data. So the high and low order units cannot be properly exacted without first re-arranging the bytes so they can be read naturally. Thus, multi-byte storage units in big endian format need no change to determine the high and low order units, while units little endian format do:

[2] Also known as *network byte order*.

Insert picture comparing high and low orders of big and little endian numbers.

Each byte in a multi-byte number has an order number. The first byte in a big endian number is the right-most, whereas in a little endian it is left-most. A double-word storage unit has four bytes, which could be labeled in order 1, 2, 3, and 4 according to their bit weights. In a little endian number, these would be in that order, but that means you'd have to turn them around to get the correct value. A big end number would have these in the opposite order (4, 3, 2, 1) which is a natural representation of their values.

Integers

A set of digital bits and bytes can be interpreted directly as a binary number as mentioned. These binary numbers are known as *integers*, which imply *only* whole numbers and so far, only positive. Integers are the most common numbers used in computers for calculations and such because of their simplicity and whatnot. But they are not the *only* types of numbers.

Although digital data is represented in bytes which are made of bits, it does not necessarily mean that it will be used as an integer. If you take a step back, you'll realize that the bits are merely switches that can be on or off. Combinations of these switches commonly form integer numbers using the binary numbering system, but they can also just be switches or be used in different ways.

For example, if you take a piece of graph paper and color in some of the squares, does it necessarily mean you're representing a binary number? No, you could be drawing a picture with those filled and unfilled squares. Digital data can represent pure values as such or even different kinds of numbers such as *real* numbers.

While we can look at something and determine what it actually is by look and intuition, a computer must be told. That is, you can flip a bunch of bits in digital data and then tell the processor it's whatever you want it to be. It's important to remember that everything in digital data boils down to a bunch of zero's and one's.

Negative Integers

Any number that is specified as being either negative or positive is known as a *signed number*. Numbers whose sign is irrelevant or unspecified are known as *unsigned numbers*. Up until this point I have only explain unsigned integers, which are whole numbers without a specified sign.

When we write out a number, we can make it negative by preceding it with the negative sign or specify that it is positive by preceding it with the positive (plus) sign. Digital integers, however, are not the same as they are on paper because they must be represented ultimately as a binary number, i.e. zeroes and ones, so there is no way to "just write" a dash.

The most common way signed integers are stored in binary is *two's compliment representation*. It represents numbers as negative by inversing all their bits and adding one. Another method for negative integers is *sign-magnitude representation* in which the left-most (most significant) bit is simply a sign-bit where zero (0) is positive and one (1) is negative. I won't bother explaining this latter method because of its simplicity and rarity[3].

A two's compliment integer uses the most significant (high order) bit to represent the "signed-ness" of a number. However, it doesn't do this by simply being on or off as is the case with signed magnitude integers. Rather, a negative number is the *inverse + 1* of its positive counter part. That is to say, the negative version of a number is all of its bits switched (off to on, on to off) with one (1) added to the result. Thus, the left-most bit of a negative number in this system will always be one (1). The rest of the bits in the number, however, are completely different.

The strangeness of this system is due to its direct correlation to how processors devour binary data and perform subtractions and such. Whether you realize it or not, every calculation in a computer boils down to binary *additions* and this system makes the subtractions translate into those additions more naturally.

Because of the way the two's compliment system works the maximum negative number value is always greater than the maximum positive value. The reason is that zero is considered unsigned or loosely "positive". For example, a signed byte can contain 256 different values, which means 128 positive and 128 negative; since zero is counted as positive, that range is -128 to +127. The following table shows the signed and unsigned ranges of numbers in the common storage units:

Insert range table here.

Like byte order, the CPU must already know if a storage unit is signed or unsigned, otherwise it will deal with the number incorrectly. If an integer is intended to be negative and signed, it will be treated much larger if used as unsigned. The reason is that left-most bit, when not used to represent sign, has a very large weight. Add to that and the fact that all the other bits are inversed which usually means an otherwise small negative number becomes *very* large. For example, a negative one (-1) signed integer is represented as a storage unit with all bits on; that means it also equates to the largest possible *unsigned* integer if used as unsigned.

Unless specified, all numbers I explain in binary will be unsigned. Even though it's usually assumed, an unsigned number doesn't necessarily mean that it is a positive value.

[3] Most computers do *not* use sign-magnitude natively as their method of expressing negative integers. In fact I can't think of any that do. In the realm of PC computing, you can be assured that the common method is two's compliment.

It implies a raw amount, not a positive or negative value. For instance, when you play a board game involving dice rolling you always use the rolled number at its face value. The number is unsigned because it does not imply a positive amount. If the game tells you to roll the dice it may also tell you to use that dice roll against yourself rather than for. I.e. "roll the dice, this is how much you lose". A negative or positive effect can be taken from that number. And if you think about it, the negative sign is actually just a symbol for subtraction from zero.

Floating Point Numbers

Real numbers can be represented in binary with a bit more difficulty than integers. While an integer can only have whole parts, real numbers can fractional parts. Think of it this way. How many values exist between zero (0) and one (1)? The answer is *infinite*. Fractional values can get ever smaller as you go into halves of halves, quarters of quarters, etc. The problem then, is that the number of bits in any given storage unit is *finite*, meaning it has a limit. You can't store an infinite number of values in a byte; you can only store two hundred fifty-six (256) different ones.

One solution to representing real numbers is floating-point which implies a calculation to represent a fraction using four parts: a sign, a mantissa, a radix, and an exponent. The sign is either a one (1) or negative one (-1). The mantissa is a positive number that holds the *significant digits* of the floating-point number. The radix is essentially the base numbering system involved (decimal is base 10, binary is base 2). The exponent indicates the positive or negative *power* of the radix that the mantissa and the sign should be multiplied by. Basically, significant digits represent the number while the power is the size. You can see this in how these four parts are combined (* = multiplication):

$$\text{sign} * \text{mantissa} * \text{radix}^{\text{exponent}}$$

This combination is a scientific notation for writing very large and very small numbers. Numbers in this format are also known as *floats*. Five can be represented in this way using one of the following:

$1 * 5 * 10^0$
$1 * 0.5 * 10^1$
$1 * 0.05 * 10^2$

The IEC[4] 559 (also known as IEEE[5] 754) standard, which is incidentally used by C++, defines a binary format in two storage units for expressing the floating-point format[6]. The first *type* is known as *single-precision floating point*, or *single*, which resides in a 32-bit (double-word) storage unit. In this type, the most significant bit represents the sign where zero (0) is positive and one (1) is negative. Following that are eight (8) bits which represent the exponent as an unsigned integer. Lastly in this type are twenty-three (23)

[4] International Electrotechnical Commision
[5] Institute of Electrical and Electronics Engineers.
[6] There are plenty of detailed resources on floating-point; what I give here is merely a brief overview.

bits for the mantissa, also interpreted as an unsigned integer. The radix, which seems to have been left out, is assumed to be two (2). This type also defines 127 as a value to be added to the exponent result; this is known as a *bias*.

The second type defined by IEC 559 is known as a double-precision floating point, or double, and resides in a 64-bit (quad-word) storage unit. It is much like a single except that eleven (11) bits are used for the exponent, fifty-two (52) bits for the mantissa, and has a bias of 1023.

Both types also contain a "hidden bit" which is the whole number part of the mantissa, being either one (1) or zero (0). The value of this leading bit is determined by the exponent. If the exponent is all one's (all bits are on), then the hidden bit is zero (0) and the mantissa is denormalized. In all other exponent cases, the mantissa is normalized and the leading (hidden) bit is one (1).

Floating point numbers require additional calculations making them slower than dealing with integers. Most modern computers have a floating-point unit, sometimes inside a "math co-processor", that speeds up these calculations making them almost as fast as integers. Floating point numbers can also be slightly inaccurate when stored in the standard IEC 559 format, because it is base 2 rather than base 10. For example, the closest you can get to representing '0.2' in an IEC 559 single is '0.199999'. There are also many other considerations not detailed here such as infinites, rounding, truncation, and "not-a-number"'s. Standard floats and these issues are covered in more detail in Chapter 4: Arithmetic.

BCD Numbers

There are times when it is beneficial to store a number in digital data without losing it's "decimal form". That is, each digit in decimal corresponds to a specific amount of bits in binary. This format is known as BCD or *binary coded decimal*.

The most common form of BCD is *packed*, sometimes known as *packed numbers*, where each decimal digit value is stored in a nibble. Another, less efficient, method of BCD is where each one is stored in a whole byte. The amount of space wasted for the latter is **very significant** which is why it is rarer (BCD in modern computing is rare in the first place). The last form of BCD that I am aware of is where each decimal digit is represented by six (6) bits. That, however, is an ancient form from a time of 48 and 60 bit storage units and I won't touch on it. Since packed BCD is the only one worth mentioning, I'll continue explaining it.

Unlike floating point, BCD is *extremely* accurate because it is a direct representation of whole or fractional numbers in decimal. The value of each decimal digit is simply plopped into a nibble in the same order as they were written.

Insert diagram of decimal number inserted into a storage unit.

Both the decimal point and sign can also be added into the equation using a special code for each. Since each nibble can store sixteen (16) different values (0 – 15), there are six (6) values (10 – 15) that are *never* needed for a decimal digit. Thus, a decimal point can be represented by the value 10 and a sign by the value 11. The value fifteen (1111b) usually means an *empty* digit. That is, one that does not contain a numeral, sign, or decimal point.

BCD numbers have the advantage of easy translation to and from decimal, but are lacking in speed and also waste space.

Fixed Point Numbers

When precise real numbers are needed in a fast-paced environment, fixed point numbers may be used[7]. These are simply integers where the decimal point is implied at a certain point in the number. Implying the decimal point is like a form which requires you to pencil in numbers but *not* decimal points or signs. Thus, if the decimal point is implied at two places from the right, the number 1000 would actually be 10.00. The advantage is that the number is always accurate, however at the same time this format cannot be used to represent exponentially large or small numbers without using a *lot* of data.

Another hazard with fixed-point numbers is arithmetic. Multiplying two numbers will yield a result that has too many digits. When we multiply two fractional decimal numbers, the decimal point must be moved over in the result for the correct answer. This "moving" of the decimal point for fixed-point numbers requires additional calculations.

Some fixed-point implementations do things differently. Rather than implying the decimal point at a specific point in the *decimal* representation of the number, they imply it at the hex or binary level. For instance, a double-word storage unit might use two (2) bytes for the whole part and two (2) for the fractional part:

***Picture of storage unit split by implied decimal*.**

Because computers work in binary, it's a simple task to figure out where the decimal should be when performing arithmetic on two like fixed-point numbers. The problem now, and before as well, is *overflow*.

Overflow occurs when a calculation causes numbers that are bigger than the storage unit used in storing them. What happens in most cases, as in integer arithmetic, is *truncation*, where the additional bits are simply "chopped off"; more on this and other fixed-point details in Chapter 4: Arithmetic.

[7] Before floating-point units were up to par, most games for personal computers used integers and fixed-point exclusively. Modern games rely on floating-point more than fixed which is all but dead in that realm.

Hexadecimal Helps

A storage unit's minimum value is always zero, but its maximum value is determined by the number of bits that makes it up. Let us take a look at a nibble first. It is made up of four bits, so at its range is '0000b' to '1111b'. If we use our brains, and previously learned process of converting binary to decimal, we will find that the range is '0' to '15'. It's a little funky, but '15' in hexadecimal is simply '0xF'. Therefore a nibble represents a single hexadecimal digit just like a bit is a single binary digit.

The hexadecimal numbering system is extremely helpful when dealing with storage units. Two hex digits equate to a byte, four to a word, eight to a double word, and sixteen to a quad-word. This is a lot easier than dealing with the massive amounts of bits associated with each. An added plus is the ease at which you can convert binary to hex and vice versa. Most of the time when dealing with raw storage you will see and use hexadecimal rather than decimal or even binary. It lends itself much to that purpose.

Text Data

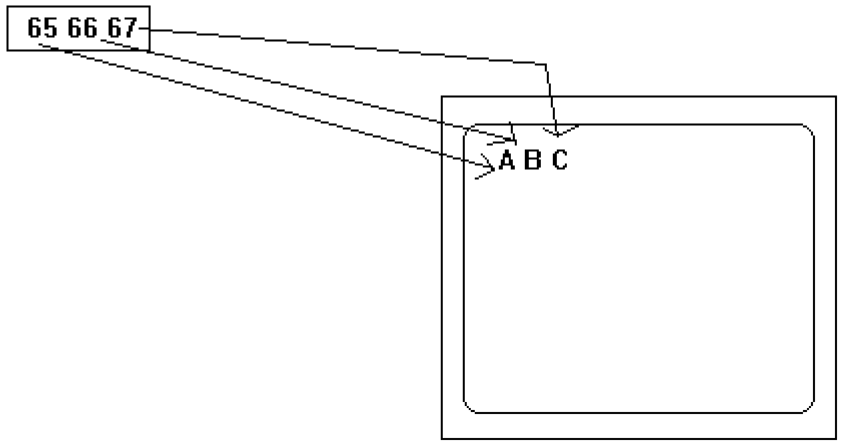
Although computes deal only in numbers, we use them for much more than simple calculation. Text is one of the most common entities within a computer. We write it, we read it, and we use it for many things. Even though we see letters and other characters, the data is still stored using *numbers*.

Anytime a set of items has a specific order, they can be represented by numbers. The English alphabet is an example of this. The first letter is 'a' and the last is 'z'. Each of these letters could be represented by the numbers one (1) through twenty-six (26). This is a lot like how computers deal with textual data.

Understanding text is similar to understand numbers. A number, as I've shown, is an expression of a value *amount*. Computers store numbers in on/off switches that can be expressed in the binary number system which can be easily converted to hexadecimal. Text, unlike numbers, is a series of characters that are read singly and in order. Each character has a specific shape defined as a *glyph* used in display. That's all text is, is a series of drawn symbols. Computers draw symbols using an output device such as your monitor or a printer.

Each individual item or symbol is known as a character and includes things beyond just letters. Punctuation and spaces, for example, are characters. Lower-case English letters are characters with different codes than their upper-case counterparts. If you were to give someone a list of numbers corresponding to English letters, how would they know where to put the spaces between the words or which letters were capitalized? Most of us would intuitively know, but computers do not because they are based on *logic* rather than *intuition*.

You might think of text in computers as graph paper. Each square would contain a number that corresponds to a letter. When a program in the computer needs to display some text, it will run through the list of numbers. For each number it finds, it will display a glyph that corresponds to the character that the number represents. This is like you taking a separate piece of paper and writing out each character (letter, number, etc.) represented by the numbers on the graph paper:



Note that the analogy I used implied a computer *program* doing something with the text. This is always how it is with *any* data in a computer. Because digital data is always just a bunch of numbers, it is up to a computer program (i.e. a set of instructions) to decipher the meaning of that data.

A list of numbers representing text is known as a *character string*, or simply *string*. Each number in the string is known as a *character code* and corresponds to a textual character. The storage unit required for each code and what character each code actually represents depends on the *character set* of the number series.

Character sets, also known as *code sets*, are like the legends on a map. They define exactly what each number represents just like a map legend defines each symbol. The computer stores these strings as simply numbers, so unless you know what character set it is encoded it in you won't be able to understand it. Unless someone tells you that the dots and dashes you see are Morse code, for example, you might not recognize it. Luckily, most character strings are encoded in *US-ASCII*, known usually as just "ASCII":

Insert ASCII table here.

US-ASCII, which stands for "United States-American Standard Code for Information Interchange", is a 7-bit character set that was born in the sixties when computers were young and has managed to persist even today. The English alphabet, Arabic numbers, punctuation, and more are all defined in ASCII and even modern character sets, such as

Unicode, inherit the same codes for these things[8]. Because each character code in ASCII is 7-bits, each code is stored in a single byte.

A byte can store 8-bits while a standard ASCII character only requires 7. Codes using the 8th bit, and thus exceeding 127, are known as *extended ASCII*. There are multitudes of extensions, so you can really only be guaranteed the first 127 characters or codes 0 to 127. From this point on I will speak of textual data as it pertains to ASCII and the first 128 codes because of its ubiquity.

Some of the codes in a character set do not represent actual characters; instead they represent an action to perform when reached. These are known as *control codes* which represent *control characters*. These control characters allow blocks of text to be formed with primitive formatting such as tabs, new-lines, and form feeds (also known as page breaks). Control characters can be used to store several individual lines of text in a single character string. At the end of each line a new-line character is placed to signify the end of the current line and the beginning of the next.

Data Organization

If you lived in a home that was completely empty, how would you store all of your stuff? Computers rely on data being organized in some way so that they can find what they need, rather than just keeping a pile of bytes somewhere.

Mass storage devices (permanent storage) and memory (temporary storage) take a typically agnostic approach to organization. They are like boxes that you stick stuff in. Although you can organize the stuff you plop in a box, you can also just dump stuff in without any organization at all. A CPU with accessing a storage device or memory can simply specify what bytes it wants to use. Thus, it is a computer program's job to bring organization to storage and is usually done by the *operating system*.

While data in memory is typically unorganized, data in mass storage devices is typically organized using a *file system*[9] where data is stored in files which are contained in directories[10]. A directory can contain zero or more files as well as more directories. Each file and directory has a name and sometimes either can have additional attributes attached. The data of a file is simply zero or more bytes that are inexplicably tied to the location, known as the *path*, of the file using its name and the name of all parent directories it's contained in. The end result is a *file tree*:

Insert picture of a file tree (perhaps from windows explorer?)

[8] EBCIDIC is the only character set, to my knowledge, with *no* relation to ASCII; it which stands for "Extended Binary Coded Decimal Information Code" and is an 8-bit, proprietary character set created by IBM long ago that persists today mainly on mainframes.
[9] Chapter 1: Computers and Programming gives a high-level explanation of files, directories, and paths.
[10] The term 'folder' is a modern term, but equates to the same thing as 'directory'.

The format of a file system is how it organizes these logical files among the pile of bytes. It's like how you organize stuff versus your family or friends. Your technique and resources is individual to you. Most computers use one of several common techniques enforced by the operating system.

Author's Opinion: In the olden days, but persisting amongst computer science courses, people sought to categorize how files themselves were organized. They would describe files as data spread amongst one or more *fields* within zero or more *records*. Having dealt with several different file formats (such as *PDF*, *XML*, and *MPEG-2 TS*) I recommend you **not** think of files in this way. Each file can have its own organization depending on its advertised format or the format the creating program put it in; and it is difficult to break these into the record-field schema.

A storage device meaning to be organized by a file system, but not yet formatted is known as an unformatted device or *partition*. A partition is a part of a storage device that has been sectioned off so as to be seen as an independent logical (rather than physical) storage device. Partitioning is a way to organize multiple file systems onto a single mass storage device:

Insert picture of hard drive that has been partitioned into a couple file systems.

Data in memory is typically organized into *pages* where each page is a specific amount of bytes. Think of this organization as a single notebook while file systems is a file cabinet filled with multiple notebooks.

Processing Data

Although the CPU ultimately controls a computer by performing instructions, it must gather its data in some way before using it. Loading data for use is like us remembering something; it is pulling the data from storage into an area where it can be immediately used. A CPU loads data into *registers* which it can then use for its instruction(s). A register is a single storage unit that can store a single number and is immediately accessible by the CPU.

Here's an example. You want to dial the phone number of your grandma who you haven't spoken to in a while and you don't remember her phone number. This is like a CPU that needs to perform an instruction, but the data is stored somewhere other than its registers. You must get your grandma's phone number from your address book and memorize temporarily to dial the number. A CPU must load data from memory into its registers so it can perform the desired instruction. When done dialing the number, you'll probably forget the number again and have to look it up later. A CPU has a limited amount of registers, so it must "forget" the data that were in them to make room for new data to use.

A CPU is actually just like a calculator with more resources at its disposal. A standard calculator can only deal with two numbers at a time and stores a third in memory at the user's request. A CPU will usually only deal with two numbers for a single instruction, but it can store many more in its immediate registers and many, *many* more in temporary memory, e.g. RAM.

In order for a CPU to use data from memory or mass storage, it must be *read* in. When a CPU puts data into memory or mass storage, it is known as *writing* it. Your plain paper notebook works the same way for your notes. When studying to do something, you read what's on the paper and in order to put down more data you write to it. Storage works the same way.

There are several ways in which stored data is accessed, but usually it comes down to *random access* and *sequential access*. Random access implies the ability to read or write arbitrarily to any part in a logical group of data. Sequential means you can only read *or* write data in a forward direction within its logical group.

Data can be randomly accessed when you can arbitrarily ask for data from any position in its logical group; albeit file or the whole chunk. Memory that is divided into pages can be accessed this way by individual bytes across all pages or bytes within single pages. The term RAM, *random access memory*, comes from the fact that practically all memory is randomly accessible like this.

Sequential accessed data is read or written to in specific order. This works best for mass storage devices which divide data into logical files. When you want data from a file, you read it out sequentially and vice versa for writing. Mass storage hardware is limited in its ability to "jump around" to any point like you can with RAM and for that reason it is best to access data within it sequentially. Usually data from a mass storage device is read sequentially and copied into RAM where it can then be accessed randomly.

Sequential access would be like a stack of papers in a box. You can only get to the next page, by taking out the one on the top. Random access, on the other hand, would be like a book. You can flip to any page at any time.

There are pro's and con's to both RAM and mass storage. RAM is fast, but temporary and typically smaller in size than mass storage. Mass storage is permanent and has much more space available, but is slower to read from. The less interaction with mass storage, the faster programs become.

When the CPU is processing multiple programs at once, it sometimes requires more memory to work with than is available which is why operating systems provide *virtual memory*. This is memory whose data is kept on a mass storage device and *simulates* RAM.

Plain Text Files

There are many types of files contained within file systems, but one very important one which you will be working with extensively are plain text files. Files can be distinguished as either being plain text files or binary files. A plain text file is a group of data where each and every byte pertains to a character code whereas a binary file is a group of data whose every byte can represent almost anything.

Plain text files, herein known simply as text files, are very *portable* because they are predictable. The term portable is subjective, but in this case I mean that the files can be transferred from one mass storage device to any other, even those using different operating systems, and they can still be understood. Their contents are immediately recognizable.

Most text files, and the assumption I make about the ones you'll use, are based in the ASCII character set and each byte represents one character code. A text file, then, is just a list of numeric character codes. Each line in a text file is specified by ending it with a new-line *sequence*.

Although ASCII defines standard control codes, there are three different interpretations of how to specify a new-line in text files. I call this a new-line sequence, and you may have seen the term elsewhere. Simply put, if you use a specific new-line sequence, an operating system that uses a different one may not recognize it. Most modern operating systems and programs on them are smart enough to recognize all three types, *but* many are not.

Byte Sequence[11] (in hex)	**Code Name**	**Used On**
0x0D	CR	Macintosh
0x0D0A	CRLF	DOS, Windows, OS/2
0x0A	LF	Unix, Linux, BSD

As stated, many programs are smart enough to check for any of these three sequences to represent a new-line, but there are some that are not. Depending on the operating system you are using, you will be relying on one of those three sequences when creating text files.

Summary

Blarg.

[11] The byte sequence is the hex codes for each byte needed to represent a new-line sequence. Two of the three only use a single byte to represent a new-line, but the other needs two bytes. The two bytes *must be in order*!

Exercises

Blarg.