

Three. Some Elements of Computing

We already know, this chapter is some kind of a help for all those readers coming from outside the realm of computing. To understand so many things that we get in the later chapters, we need it here. The ruling theme of the book is the political economy of power and control operating in the cyberspace as a particular moment of the hegemony of capital, and the politics of resistance towards this control. This word ‘cyberspace’ is more from the world of sci-fi and Internet-pop, than any word meant for serious communication. Wikipedia defines, ‘cyberspace’ means the “electronic medium of computer networks in which online communication takes place”. Weboepedia suggests, ‘cyberspace’ is “a metaphor for describing the non-physical terrain created by computer systems”. We are using this word ‘cyberspace’ in a pretty general way in this book. Wherever we are saying any computer-related thing – making of computers, or, computer industry with both its segments, software and hardware, or, the teaching of computer science, the understanding of it, on both the technical and common plane – that is, everything in/around/through the computers. We are deploying this term freely, as an all-encompassing one.

The power and control operating in this cyberspace is the backdrop, against which we want to explore the philosophical implications of GPL: how GNU GPL tweaked the concept of ‘property’ in the context of this cyberspace. By the time of the birth of GPL, this cyberspace was moving down the lane of primitive accumulation. Before going into *primitive accumulation*, let us have a note about this word ‘tweak’. This word has a surplus meaning specially for this book on political economy of computing. We are importing it from the world of computing into the field of political economy. The word ‘tweak’ means to displace and reconstruct, and indeed, something more than that. It carries the sense of ‘pluck’, ‘pinch’ or ‘twist sharply’. In the realm of software development, pieces of software continuously go through code-writing, editing, debugging, recycling and customization. Tweaking means how a software developer takes up an existing program, and goes on displacing and reconstructing it to achieve a program that suits the purpose more appropriately. For a full elaboration of all the nuances of the real process of ‘tweak’, its inherent dynamics and surplus meanings, see the seminal essay Raymond 2000.

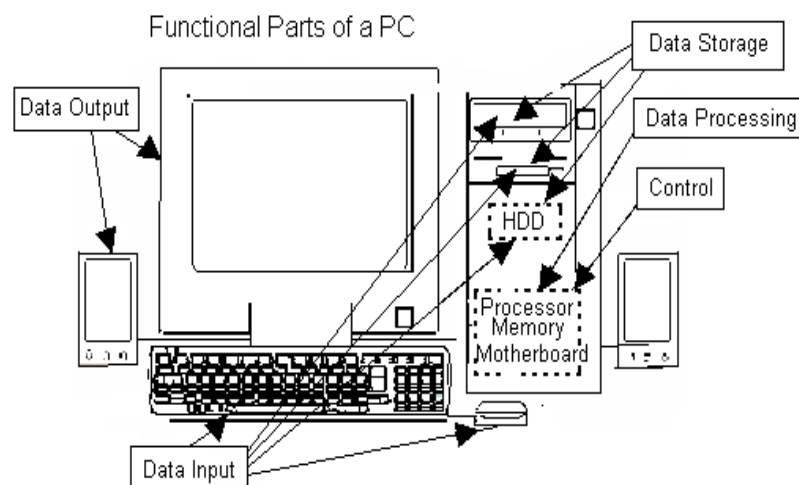
Once the primitive accumulation is complete, all petty producers, the owners of the means of production, land or tools or otherwise, get divorced from the ownership of these. This is done, in order to establish the capitalist market rules and capitalist property system. This is the way things have happened in history during the rise of capitalism. Something very much like this started happening in cyberspace too. And the stake here in cyberspace was of a huge import, both ethically and philosophically. The property that was getting divorced from its prior owner was a common property of all human beings. In fact, maybe it is the oldest form of property, the mother of all properties, that is, knowledge. Later we will know in details, how the same chain of events as in ‘primitive accumulation’ started repeating itself in cyberspace too. This time the expropriation involved software in particular, knowledge resource in general. We will know the story of its unfolding, through the supplements of resistance, a text called GPL, and a context called FLOSS that followed, in the negotiations and counter-negotiations around the proprietorship software. Some new horizons of political economy were encountered here, some new areas of postcolonial theory, the topology of which this book ventures to explore. Let us remember, software was one of the most capital-intensive and capital-generating form of reified

knowledge in history.

Now, here comes a problem. Talking about this newer kind of power and control involves materials that are not exactly from the realm of classical political economy. The kind of reason and logic that we are trying to develop in this book encloses and touches upon some elements definitely from hardcore computing. So, for the sake of making some new lands available to political economy, we now engage in an unavoidable digression. This digression, at times, verges on oversimplification, at times quite verbose. But we cannot avoid it in order to return to the crux of our discussion. In this chapter, we address a few questions like, what is FLOSS, or, what is Free and Open Source, or, before that, what is Source Code. And also we explicate some other elements that are necessary for understanding these questions and our chief strand of logic. Without these elements it is hardly possible to elaborate the process of power and control in the cyberspace, against which GPL emerged. Anyone with some elementary knowledge of computing can comfortably skip this chapter.

1. Functional Components of a Computer

Before understanding what is Source Code, and how it is controlled, we first need to understand a bit about computer basics. Let us begin from the beginning, with a PC sitting there on the table top. Obviously, a PC, or, a Personal Computer, is just one of the various forms of computers in today's world, but, here, for this book, it will be adequate to start from a schematic picture of a PC.

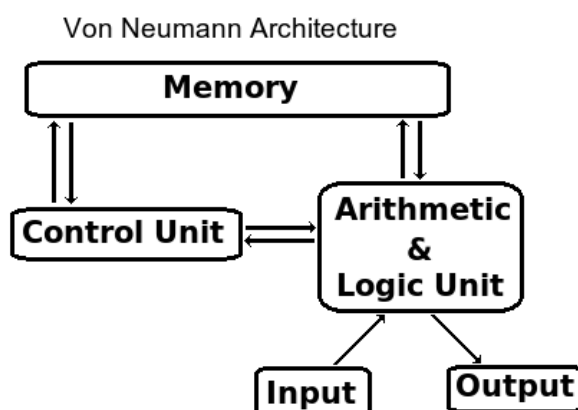


Here we attached five tags for the very common component types usually available in a PC. The keyboard and the mouse for Data Input, the console and the sound-box as Data Output. The HDD, Hard-Disk-Drive, the FDD, Floppy-Disk-Drive, and the CD-ROM Drive are listed both as Data Input and Data Storage components. And the Processor chip, the RAM and ROM chips held on the Mother-board are listed as Data Processing and Control components. These schematic categories are not watertight compartments. One component meant primarily for a particular use can very well overlap into other uses. Obviously, there are infinitely many kinds of peripherals, and quite a few of them come into usage that fall in more than one tags. Like, you can very much take some outputs on a

HDD and fix this HDD to some other computer, thus using it as an input device in the second machine. Without going into details of nonstandard use of components, or, debating about what standard use is, we are just trying to build a scheme of simple categories here. So, the categories are:

- I. Data Input
- II. Data Storage
- III. Data Processing
- IV. Data Output
- V. Control

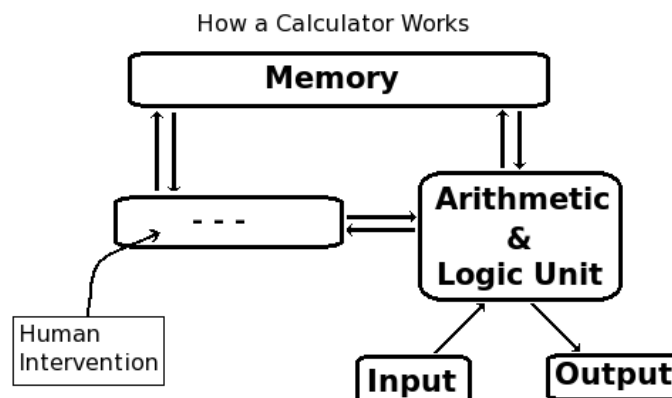
A scheme is already there, inscribed in the very way we wrote the list – we will come back to this scheme quite soon. Let us now start from the question: why in the history of all machines that man has made, does computer have a special position? The very special position of computer becomes very clear when we compare a computer with a calculator. That difference actually resides in the fifth element of our list: Control. Let us elaborate the difference – this difference has got a very interesting logical point implicit in it. There is a logical leap here that instantaneously makes computer and entirely different kind of machine, different from all other prior machines in the history of civilization. To properly understand this logical leap, we need a diagram of a scheme, created by Von Neumann, the celebrated physicist, mathematician, computer-scientist. It is called the ‘Von Neumann Architecture of Stored Program Computers’. Obviously there are many more interesting deployments of Von Neumann Architecture than the one here in this book. Here we are using a simplified version of the scheme, more fit for our purpose. Tanenbaum 2005 has good discussions about Von Neumann Architecture.



This diagram gives a customized and simplified version of Von Neumann Architecture, that is now a classic in the history of Computer Science, a regular text material of the elementary classes. We can readily recognize the five units within the scheme with the list given above. The first and fourth elements in our list, Input and Output, are shown in the lower part of the diagram. The arrows show the path of data going into the machine and coming out from it. Both *Input* and *Output* are linked to the *Arithmetic and Logic Unit* of the machine, frequently called by its acronym ALU. This ALU and *Memory* are both connected to *Control Unit*, the fifth element of our list. The second one of the list, *Data*

Storage, is obviously represented by *Memory* in the diagram. This includes both RAM, Random-Access-Memory, and ROM, Read-Only-Memory. Both RAM and ROM act as storage device or *Memory*. RAM and ROM are quite common terms in standard computer literature, and quite familiar to any average user. As the names suggest, they define two categories of memory. RAM is a kind of memory that can be accessed, that is read from or written to, at will or at random. And ROM is usually written once and read as many times as one needs. For the functioning of computer some amount of RAM is kept on silicon chips on the motherboard of a machine, called RAM chips. And the very primary instruction for activating a machine or booting it, is usually kept on a chip of the ROM kind and it is called BIOS, Basic-Input-Output-System, set on the motherboard of a computer. The other memory storage devices like HDD can be both used as RAM or ROM. Here, *Memory* in the diagram means all kinds of memory devices.

The remaining element in the list of functional components, *Data Processing*, the third one in the list, does actually reside in two parts, *Memory* and ALU in the diagram. The data that comes in through *Input* goes to *Memory* and resides there, waiting there to be recalled by ALU for Arithmetic and Logical processing. And at last, when the processing is finished, it goes out through *Output*. Or, it goes to a storage device, which is another form of output. So, in every sense of the term, the procedure of data processing is very definite and finite. And if it is really like that, some agency has to look after it, when and how every bit and piece of this procedure does start or end. And exactly this job is looked after by *Control*. A program, created by human beings, is put there for *Control* in the form of a 'stored program'. This program does the job of controlling the procedure. So, the activity of control, that was till now an entirely human prerogative, at last starts residing within a machine. Computer can represent human control – this is the mystery of the very special position of computer among all hitherto machines. This point will become clear very when we discuss about a calculator.



Now, let us discuss the difference between computer, and say, calculator. Both of them handle digital data, but, still, calculator is not computer. But, why? Let us iterate the steps during one simple calculation, say addition, of two and three, using calculator. So, mathematically, the question is $2 + 3 = ?$, the answer to which will be provided by calculator. The steps are pretty easy, we strike the keypad, the three switches one after another, '2', '+', and '3', and then, to get the answer we strike the key '=', and the result is displayed on the screen. So, let us structure the procedure. We go to *Input Device*, insert Raw Data, '2', then one Command Data, '+'. This tells calculator exactly what it has got

to do. Then again we insert another piece of Raw Data, ‘3’, then another Command Data, ‘=’. Now, with all data inserted till now into it, stored in its *Memory*, both Raw Data and Command Data, calculator processes all Raw Data with the actions meant by Command Data, and shows the result on *Output Device*. So, the Functional Parts used here are *Input Device*, *Memory*, *Processor*, and *Output Device*. Where is *Control*? It is not there, and exactly that is why calculator is not computer. All through the mathematical job done on calculator, *Control* remained with the human user. The human user feeds in data, both Raw data and Command data, in their proper sequence. And only then the result relevant to the human subject comes out from it. Calculator does everything except *control*. Exactly this lack is shown by the void in the box in the calculator diagram, waiting for human intervention to take place.

Later in this chapter we will cite a very short program, ‘add.c’, written in the high-level language C. This program does exactly the same thing – add 2 and 3 and give the sum. But, the difference in case of running this program on computer is that, the whole thing can now, through this small program, reside within a computer’s *Control*. And the program goes on doing the whole job without any human intervention. There are actually many layers of so-called ‘stored program’ residing within a computer, about which we will know later in details. The lowermost layer is that of the very primary program called Kernel. We can call this primary program as Operating System too, without going into many complex debates. This primary program called OS or kernel then allows and enables all later programs to run on the machine. So, all through the workings of computer, all the stored programs represent the element of *human control*. Understanding this method of control deployed through all these layers of ‘stored program’ is actually very crucial to us. Without it we cannot understand the political economy of computing that we are going to discuss in the later chapters of this book. We are coming back to this human control working in layers later in this chapter.

We are familiar with machines that do many kinds of jobs. But, in earlier times, many machines were built for some special purposes. A lot of these machines were specifically meant for doing mathematical, or better, numeric jobs. From time immemorial humankind is making machines for doing numeric jobs, once popularly called ‘number crunchers’. These machines are of all types, like Abacus, Logarithmic Tables, Slide-Rules, ‘Napier’s Bones’, Pascal’s ‘Pascaline’, ‘Analytical Engine’ by Babbage, and many such, till the arrival of calculator or computer in the recent times. Within numeric machines, computer has its pride of place, because, as we said, for the first time in history, humankind could vest the control thing into the heart of a machine. Till now this was absolutely a human action. That is the beauty of a ‘stored program’, and that is the beauty of the Von Neumann Architecture of a ‘stored program computer’. A stored program is the cleverest ruse in the history of human knowledge that relieves humankind of the responsibility of remaining in control, by supplying a flawless deputy.

2. ‘Bit’ and Representation of Data

We want to understand the concept of ‘stored program’ more intimately. *Source code* is actually the most primary form of ‘stored program’. To go into ‘source code’, we need to discuss some preliminaries a bit more. And, and in this case, this bit is a ‘bit’, literally. ‘Bit’

is the short form of ‘Binary-Digit’, the building block of data. Everything that happens in a computer happens around this thing called data, in both the forms – *raw data* and *command/instruction* data. Any work on computer starts with input of data, which is then processed by programs. A stored program is made of collections of instruction data, augmented by raw data for the execution of these instructions. After the stored program runs, the result is shown in output data. Now, the question is, what exactly is ‘data’ for computer? How does computer represent, structure and understand data?

The simplest answer of any question is either a ‘yes’ or a ‘no’, or, in numeric terms, a 1 or a 0. So, how much memory it would involve to hold or represent this simplest answer to a question? Obviously, the exact amount of memory space that can hold a 1 or 0. And, exactly this amount of memory space is called a ‘bit’ – the space that is enough to represent any one of the two binary digits, 0 or 1. This is the smallest unit of memory. So, mathematically speaking, the value of a bit can be either 1 or 0, depending on which number is stored at that exact moment in that bit of memory. In terms of computer architecture, this may mean presence or absence of a certain amount of current in a circuit, like when a machine is either ‘on’ or ‘off’ – it will get interpreted as 1 or 0, and get written or represented in that bit of memory space as 1 or 0. In fact, in the real case, this is done with a particular voltage getting interpreted as 1 and another as 0. This goes with common sense too. The computer cannot write a 0, or for that matter, cannot write anything at all, if it goes actually ‘off’. Handling information in terms of bits, in terms of 0/1, is actually a thing of convenience. The devices that operate around data – the input, output, and processing devices, their circuitry – all these can run most reliably when operating in this two-state binary mode. And the moment we start keeping information in terms of bits, this renders the size or volume of data as measurable – the amount of data can be exactly measured.

This is not just true for the simplest form of data. Complex or composite forms of data can be transformed into simple data and handled that way. In terms of logical structure, this is more like a procedure that metes out a complex question or problem into simpler parts and replies them in the binary yes/no. This procedure of representing bigger blocks of information into the simplest terms of binary 1/0 can be clearer with a simple example. Let an archive library have some rare texts highly in demand. Say on the library message board there is a slot for the name of the text, and below it there is a light. If the text is available at the moment, the light will be on, otherwise off. For this simple piece of information the light-paradigm is quite adequate. But, this signal system breaks down the moment the library wants to convey some additional information, like, say, if the text is in paper form or microfilm. Obviously another light, a second one, solves the problem. For every state of the first light, ‘on’ or ‘off’, available or not, there can be two states of the second light, ‘on’ or ‘off’, paper or microfilm. So, in terms of composite signals there are four states:

```
00: 0 & 0: Unavailable & Microfilm: off & off
01: 0 & 1: Unavailable & Paper: off & on
10: 1 & 0: Available & Microfilm: on & off
11: 1 & 1: Available & Paper: on & on
```

These four number strings that the lights generated, 00, 01, 10, 11 are just the first four numbers in a binary system. These four binary numbers become 0, 1, 2, 3 in decimal

system, the system we generally use. This area of arithmetic, Binary or Decimal number system, is quite a large area on its own, with many dedicated text books. For the time being let us simply define it this way: while in Decimal system, we work with 10 digits, 0 to 9, in Binary system we work with 2 digits, 0 to 1. All other arithmetic rules apply to them in exactly the same way. There are other popular number systems too. Like Octal system that uses 8 digits, 0 to 7, or Hexadecimal that uses 16 digits, 0 to 9 and A to F. Binary, Octal and Hexadecimal systems are applied a lot in Computer Science. Mano 2003, Rajaraman and Radhakrishnan 2006 can provide good brief introduction to these.

We may want to handle a more complex situation than the earlier one with four possibilities. Say, the library wants to convey, if the texts are copyrighted or not, another bit can be used with the earlier two. With another bit place-holder, the strings it will generate are, 000, 001, 010, 011, 100, 101, 110, 111, in binary system. These binary numbers, in decimal terms, are 0, 1, 2, 3, 4, 5, 6, 7. Total number of composite signals in this last case is 8. In the second case it was 4. And, in the simplest one-question scenario, the total number of signals was 2. Maybe we can already see the regularity there. 2 is 2^1 , 4 is 2^2 , and 8 is 2^3 . This is pretty predictable, if we compare the results with decimal system. The number of digits in binary system is two, 1 and 2. And so, every added bit increases the number of possible signal by a factor of 2. For each and every combination of the earlier bits, this new added bit adds a possibility factor of 2, either a 1 or a 0.

But, as we can see, there is already a problem. Each of the three criteria that we used, 'availability or not', 'paper or not', 'copyrighted or not', is answerable in binary mode: yes/no, or, 0/1. If there are complex criteria, not answerable in binary, this method does not work any more. Though, it can be made to work, in a roundabout way, by writing down every possible situation generated by the complex criterion, and then converting this situation into a binary criterion. Like say, some texts are known for their spectral connection. Now, this question can be broken into composite simpler questions. Does a specter haunt us if we read it, answerable in binary. But, then comes a question, a complex one, which kind of specter it is: a spook, a phantom, a bogeyman, a hobgoblin? We can mete out this complex question into simpler ones answerable in binary, like, does the text cause a spook to haunt? Does it cause a phantom? Like this, it is always possible to create a signal system that can send composite signals in terms of simple bit signals. But, why should one go into all that trouble, an age-old time-tested signaling system always already remaining there? And that signaling system is our good old language, written or represented by our good old alphabet. But, this representation needs ASCII.

3. Text and ASCII

ASCII is the acronym for American-Standard-Code-for-Information-Interchange. ASCII is a coding scheme that assigns numeric values to letters, numbers, punctuation marks, and other characters. And thus, ASCII code enables computers and computer programs to exchange information between them. It has two sets of codes, Standard and Extended. The Standard system has 128 characters. And the Extended system has 256 characters, 128 characters added to the 128 of the Standard system. That means 128 or 2^7 characters for the Standard set, and 256 or 2^8 for the Extended set, with one added bit. Very soon we are going to understand its significance. In standard ASCII, values are assigned to

communication and control codes for print or display. Here is a very important point to understand. To display a piece of text we not only need alphabets to display, but also control characters too, like where to start, where to end, how much space to leave between them, and so on. There are quite a few of those non-printing characters without which the printing characters cannot be displayed in any meaningful combination. So, a code like ASCII must include them too. This phrase is not exactly thisphraseisnotexactly.

Once ASCII is there, the problem of meting out the complex data into bits of simple data, transfigures into a problem of representing language in terms of bits. First, let us get a measure of how many simple yes/no binary answers a particular amount of data can hold. We just saw that, 1 bit can represent two simple answers, 2 bits can represent four, and 3 bits can represent eight. And let us note another point that the position of the bits are not random. When more than one bits are involved, the order of the bits is a part of the data too. The position and value of a bit replies to a specific binary-answerable answer. For number of bits being 1, 2, 3, 4, 5, 6, 7, or 8, the number of representable composite events are respectively 2, 4, 8, 16, 32, 64, 128, or 256. So, now, let us take up the problem of representation of language. Representing language can be done by representing every character by a different number. This enables us to translate any other possible signaling system into language, and thus, to transform it into data. Once that is done, computer can process it. Let us remind ourselves once again, all these discussions here are more than an oversimplification. In terms of computing science all these things are quite commonplace and elementary, available in any standard textbook.

Let us take this language, English, as our example. This involves the Latin Alphabet, with 26 members, each of them in both upper and lower case, and hence 52 of them. Then there are 10 digits, 0 and 1 to 9. Then are the symbols of syntax like the period, the comma, the colon and so on. Let us add to them all the non-printing characters too, like the 'newline' or the 'tab' and so on. The effect of these characters are visually present, though absent in terms of symbols in the presented form of text – these are so necessary for any reasonable representation of text. The aggregate goes over seventy. So, at least how many bits must be there to represent these seventy-plus characters? As we have seen, six bits will allow at most 2^6 or only sixty-four of them. So, we shall need at least seven bits. These seven bits together, in the form of a ordered sequence of bits, can represent 2^7 or 128 different characters. The first of these ordered strings will be, in binary system, 0000000, that is 0 in decimal. And the last one will be 1111111, that is 127 in decimal. So, we get 0 to 127, or 128 of them in aggregate. Transforming to and fro between binary and decimal is quite easy, taking our familiar decimal arithmetic as a model. Like, say, 1111 in Binary. Let us get its decimal value. If it was in decimal, 1111 would be one thousand one hundred and eleven, or, $1 + 10 + 100 + 1000$, or $1 \times 10^0 + 1 \times 10^1 + 1 \times 10^2 + 1 \times 10^3$. They are all powers of 10 because it is in decimal or ten digits. In Binary, the number of digits is two, 0 and 1. So, binary 1111 in its decimal value will be $1 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 1 \times 2^3$, or, $1 + 2 + 4 + 8$, or, 15. It works the same way for binary 1111111, or decimal 127. So the total number of characters will be less than or equal to 128, numbered from 0 to 127.

The first consensus standard that came up for representation of English language characters in terms of digits, was ASCII, employing seven bits. Computer handles two kinds of basic data, number and character. And as we are seeing here, all characters are numbers too, and

hence can be handled by binary system in computer. The ASCII code works here as a two-way medium. Through it we translate characters into binary numbers. This enables representing language as data for computer. And whenever we want it back in language form, computer can counter-translate these preserved numbers back into characters. We said, early computers were predominantly number-crunching ones. These machines gradually started nibbling at characters as computers were getting progressively affordable. They first replaced calculators, then typewriters, and then maybe everything. Computers started devouring large chunks of human intellectual efforts in all walks of life. This required a ready method of transfiguring characters into numbers, and thus data, and then processing this data with rules constructed such a way that allowed us to process them according to the laws of language. ASCII played this important role.

Development of ASCII was going on since 1960, though its first commercial use came in the form of a 7-bit code for the teleprinters. Bell Laboratories published its first edition of this code in 1963, and a major revision in 1967. As we have already seen, the 7-bit code allows a total of 128 events to take place in this code system. From these 128, the first 33, that is, 0 to 32, are kept aside for the non-printing characters, like newline, tab, space, delete, and so on. Obviously, as the use of computers increased and proliferated into every possible walk of life, ASCII was progressively proving to be inadequate. This inadequacy becomes more prominent, when we want to handle more than one languages with it. Proliferation of computer use progressively called for code systems that can handle more than one languages, and thus, more than one character sets, simultaneously. And hence followed more intricate and advanced systems of data-character interchange systems. Unicode is the system that is now the order of the day. But we hardly need to discuss that here for the purpose of this book. This section, in a brief sketch, described, how computer reads data. But, obviously there are other different forms of data than numbers and text.

4. Representation of Multimedia Data

Let us start with a picture – how this picture is represented in the computer, in the form of graphical data. Let us assume that we are printing the picture on a graph paper. A graph paper consists of a regular rectangular array of points, generated through intersections of vertical and horizontal lines. Each of these points has specific X and Y coordinates given by the order of horizontal and vertical line respectively. So, when the print is ready, we will see that some of these points from the array were overlapped by the points from the image, and some of the points have remained blank with no point of the image falling on them. Also let us assume that, for the time being, that the picture is a line drawing in black. So, now, the overlapped points have coincided with points printed in black, and the blank ones are the pure white ones. Obviously, as the number of horizontal and vertical lines increases, and thus, the number of intersections, the resultant array of the points on the graph will increasingly approximate the image better and better. Take this image of the first letter from the Latin Alphabet.



Obviously, the look of the image is quite kinky and rough. This will progressively become a smooth and comfortable one as we go on increasing the number of points, here shown by square black dots. In fact the number of lines in the graph grid being sufficiently large, they would not even seem as square or something. Even they would cease to look as disjoint points, they would seem like continuous regions filled with points as it happens in halftone prints of painted pictures.

Now, let us replace this graph paper grid of points with an electronic screen, for the grid the points being replaced by pixels. As it would happen on a rounded rectangular electronic screen the pixels that would build the alphabet are black and all other points are blank, that is, white. As we have seen in all our prior binary examples, this situation is pretty easy to represent in the form of numerical data, if we represent all 'black' ones by '1' and all 'white' ones by '0'. Now, usually, the electronic screens have some prescribed formats, like '640x480' or '800x600' or '1024x768', or for wide formats, '1440x900' or '1280x800'. So, if the size of screen is 640x480, the total number of points in the array or the total number of pixels will be 640x480, that is, 307200. In terms of data, it will be that many bits. We know 8 bits make 1 byte. And hence it will take 307200/8 or 38400 bytes of memory to represent all the visual data on a 640x480 screen. Some of these bits will carry 1, and others 0. That means, 38400/1024 or 37.5 Kilobytes of memory will adequately represent the line drawing in the form of graphical data meant for computer.

This example was a black and white line drawing, and hence we used 1 bit data for every point – black/white: 0/1. This was sufficient for an adequate representation of the drawing. If it was a picture in gray tones, then, say, we can use 1 byte or 8 bits per pixel. And as we know, 8 bits can represent 2^8 or 256 different signals, here, computer screen will represent every pixel in 256 different possible gray tones. And 256 different shades of gray seems quite adequate in representing a gray picture on the screen. In that case, that data of 37.5 kilobytes becomes multiplied by 8 and it is 300 kilobytes for one screen-full of gray picture. If it is a color picture, the situation is structurally the same, only with a different factor of multiplication. The perception of color comes from a mixture of different proportions of the three basic colors, red green and blue, RGB as they are called. If we take 8 bits for every color, that means three sets of 8 bits, one each for every color, red green and blue. So, we can depict 256 different shades for each of RGB or red green blue. And accordingly, the byte size of the picture becomes three times of its earlier size. We are taking 3x8 or 24 bits for RGB, in place of 8 bits for shades of gray, to represent each pixel. And so the size of the one screen-full of color picture becomes 3x300 or 900 kilobytes.

If this is what we do for a picture to represent it in terms computer data, video data comes the second in line. Video is essentially multiple frames of still pictures shown in quick succession such that it creates the illusion of moving images in our brain. And there is audio accompaniments with it. Though there are many layers of complexities involved here. The huge bulk of image data that goes into the making of video makes compression

of visual data a necessity. And this compression comes in different forms and standardized formats.

The other thing that remains here is the representation of audio data. Audio data has one very intrinsic difference with visual data. The difference is that all audio data has one dimension of real time involved in it. Audio always happens in real time. Essentially the method of representing audio waves in terms of digital data is to collect samples of the analog audio at regular intervals and collect the value of the sample signal in that interval. The algorithm for doing this is quite complex. In the first place it has to answer, at what time interval the samples should be taken, and then how many bits will be used to represent the value of a sample. There are many theories and methods involved here. But, essentially the logic is quite simple. The real analog data we cannot represent. But we can represent the value of the samples in terms of a fixed number of bits. The sampling rate and the number of bits we have to choose such a way that the digital data does represent the real analog audio quite adequately. And once it does so, the real analog audio becomes a string of bits that we can record in computer. When we want to get audio from this recorded data, we have to translate this data back into audio signals, in real time. We have to do it at the exact rate of sampling that we used while recording. And now we get a rebuilt audio from that digital data, represented in a form that computer can handle. What we have discussed in this section about representation of different kinds of data is quite elementary. Mano 2003, Rajaraman and Radhakrishnan 2006, Bartee 1991, or Tanenbaum 2005 give good introduction to these things.

5. A Stored C Program

Now that we are familiar with representation of different kinds of data, let us return to the question of a ‘stored program’. A computer works in terms of stored programs. And, we know, these stored programs consist of *raw data* and *instruction data*. Raw data is what we usually call data, and instruction data usually means the commands we issue. The control of the whole process through a stored program involves both these two kinds of data.

Now, let us take one small piece of program, consisting of only ten lines of code. Here we numbered the code-lines in a table to make the discussion simpler. Let us name this program as ‘add.c’, the ‘add’ part showing what it does and ‘.c’ being the usual surname of all C programs. This program adds ‘2’ and ‘3’ and gives the result. For those who are not familiar to programming, this is written in C. C is a major programming language. Some prefer to call C as the mother of all other programming languages. There are numerous books on C. Kernighan and Ritchie 2002 is an all time classic. Prata 1986 is a good help for beginners. This program ‘add.c’ involves exactly ten lines of code.

01	#include <stdio.h>
02	int main(void)
03	{
04	int x, y, sum;

05	<code>x=2;</code>
06	<code>y=3;</code>
07	<code>sum=x+y;</code>
08	<code>printf("\n 2 + 3 = %d \n", sum);</code>
09	<code>return 0;</code>
10	<code>}</code>

Let us discuss this C program ‘add.c’ line by line. Line #01 tells the program to include the library file ‘stdio.h’ within its body. It is outside the scope of this book to go into the details of a library file, but, let us know, simplistically, a library file is a collection of subprograms that is used to develop software. Libraries contain some common additional code and data that provide services to any individual program, whenever it needs. Including the library file within a program, is like, say, sending a salesman to an unknown country and inserting a dictionary of the language of that country within his bag. In this particular case, the name, ‘stdio.h’ suggests what this library file does. This file is concerned about standard input and output, ‘std’ for ‘standard’ and ‘io’ for ‘input-output’. Again, avoiding unnecessary complications, we can say, it makes some sense. Enabling ‘add.c’ to run involves taking inputs like ‘2’ or ‘3’, and producing an output like ‘5’. The part ‘.h’ in the file-name ‘stdio.h’ comes as a surname of ‘header’ files, as these library files are called.

Line #02 is a statutory declaration of all C programs. Simplistically speaking, any C program involves a function, a function that does something and has some value. And here ‘main ()’ is the function. This function ‘main ()’ is the entry point of any simple C program, from where the chain of execution starts. And the word ‘void’ indicates that the function ‘main ()’ in this case takes no parameter. The word ‘int’ in ‘add.c’ stands for ‘integer’. That means, the mathematical entities involved in the function ‘main ()’ will have integer values. Line #03 and #10 are the starting and ending parenthesis for the ‘main ()’ function in the program. These two define a block, within which the action of the program takes place.

Line #04 tells that the function in this program will involve three variables. Their names are ‘x’, ‘y’, and ‘sum’. This line also tells that all of them will involve only integer values, shown by the word ‘int’. This means, this program commands the computer to demarcate three slots in its memory, and name them as ‘x’, ‘y’, and ‘sum’. These three slots now become the addresses for these three variables, addresses where their values will be kept. And it additionally tells the computer that these three slots will hold only integer values in them. In line #05 and #06 the values of ‘x’ and ‘y’ are assigned. Line #05 tells the computer to put the numeric integer value ‘2’ in the slot named ‘x’. And line #06, similarly, tells to put the value ‘3’ in the slot named ‘y’. The command for addition is given in line #07. It tells the computer to add ‘x’ and ‘y’, that is, take the values kept in ‘x’ and ‘y’ and put them together in the slot ‘sum’. The variable slot called ‘sum’, from now on, will represent the togetherness of ‘x’ and ‘y’, that is, the sum of the values kept in ‘x’ slot and ‘y’ slot.

Line #08 tells the computer to print the result of the summation of 'x' and 'y', or the value of the variable 'sum', that is, the integer value kept in the memory slot 'sum'. The other details of this line specifies the format of this output line. The '\n' part tells to start a new line before giving the output, the second '\n' tells to leave one line of open space after the output. And the '%d' part tells to give the output in the integer-value format. So, now, if we run the program we will get the output: "2 + 3 = 5", with one line open space on both the top and the bottom of the line.

So, now, we have completed a C program that can add '2' and '3' and display the result. But, the problem is, this program, exactly in this form, will never run. Why? We are coming back to that in a bit, when we talk about compilers. But, before that, let us explore the novelty of this stored program, as we showed this in the Von Neumann Architecture of stored program computers. Let us note the difference between summing '2' and '3' on a calculator and doing the same thing with this C program, 'add.c', on a computer. As we said, in case of doing it with a calculator, the control remains with the human subject. This human subject controls the process of addition: when to start, where to start, how long to continue, through what steps, where to call it a day, and deciding the end product as the result. Whereas, in case of 'add.c', all these control elements are vested to the stored program. As many times as we may run the program, every time this control job will be done by this flawless deputy of a program according to the commands once written there in 'add.c'. This is the novelty and the pride of place of computer among all the machines produced till date by human civilization. For the first time we can vest the control of the process on the machine by means of a stored program. In fact every day we are doing that, in millions and millions of programs.

6. Low Level and High Level Language

Just now, in the earlier section we said that, this program 'add.c' will not run, though, obviously, 'add.c' has no particular flaw of its own in terms of C syntax, grammar or logic. Actually, this program was never meant to run. Like all '*.c' files, it was meant just to be read and written and edited and corrected and debugged. This program, or any program like this, does never run. These are only text files, carrying some C sentences and statements, just the way a piece of English text carries some English sentences and statements. The programs that run are altogether of a different kind, they are not text files, they are executable files. But, first let us understand the difference between High Level Language and Low Level Language before we go into compilers.

The term 'High Level' does not signify any superiority of this language over the 'Low Level' ones. It shows a higher level of abstraction over the layer of machine languages. There are many intricate details and differences among the categories of machine code, assembly language and things like that. But, for the time being, let us use the two concepts: 'human-understandable' and 'machine-executable' in place of those two terms 'High Level' and 'Low Level'. Obviously, it is a bit oversimplified. But that does not hamper us in this discussion. The program 'add.c' in the last section represents high level 'human-understandable' code. Those ten lines of code in 'add.c' were written by a human being, meant to be read and understood and corrected and debugged by human beings. This is, obviously, human-understandable. In fact, that is the whole *raison d'être* of the high level

programming languages: why they were made the way they were made. They are meant to be written and understood by human beings. But, then, there is obviously another kind of language that we cannot understand. To understand its nature let us try to read something that cannot be read.

```
??ELFAA^A^@^@^@^@^@^@^@^@B^@C^@A^@^@F^D^H4^@^@^@^@^@^@^@4^@
^@^H^@(^@J^@^AD^HH<81>^DH^@^@^@^@^@^@D^@^@P<E5>td<D0>^D^@^E^D^HC^D^H^A^
^@^@^A^@^@^D^@^@^@^@^D^@^@Q<E5>td^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
F^@^@^@D^@^@^@
^@/lib/ld-linux.so.2^@^@D^@^@^P^@^@^A^@^@GNU^@^@^@^@^@B^@^@^F^@^@^@
^@^@^@B^@^@^@D^@^@^@A^@^@^@E^@^@^@^@^@^@^@D^@^@^@<AD>K<E3><C0>^@^@^@
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
A^@^@R^@^@^@J^@^@^@^@^@^@^@^@9^@^@^@R^@^@^@Z^@^@^@<B8><84>^DH^D^@^@Q^@N^@A^
@_gmon_start_^@libc.so6^@_IO_stdin_used^@printf^@_libc_start_main^GLIBC_2.0^@^@^@^@B^@B^@A^A^
^@^@^@A^@A^@P^@^@^@P^@^@^@^@^@^@^@^@Pii^M^@^B^@B^@^@^@^@^@^@B^@^@^@^@^@
2 + 3 = %d ^@^@AESCA^C;X^X^@^B^@^@^@<FF><FF><FF>4^
^@^@<0><FF><FF><FF>P^@^@^T^@^@^@^@^@^@AzR^@A^AESCL^L^D^D<88>^A^@^X^@^@^@^@^@
^@^@<E4><FE><FF><FF><FF>^E^@^@^@^@^@
```

This is nothing but the executable binary file called ‘add’ that we generate from the code ‘add.c’ by a compiler. This ‘add’ is an executable file. That means, it runs, what ‘add.c’ cannot. We are going into those details shortly. But, I opened this binary file with a text reading package called ‘less’ and copied a portion, only to represent in a way the drama involved here. This binary file does run, but cannot anymore be read. Obviously, it is a tall order for us, human beings, to read, write or understand this. But, computer reads this file and runs it very easily. So, a machine understands it adequately well, adequate to run and execute it. This is an executable file and it is not in any way any text meant to be read or written by human beings. It may catch our eyes that in the ending section of this gibberish, we get something that is already very known to us. The problem that we are trying to solve, as quoted in this inhuman piece of language: “2+3=%d”. This gibberish actually is a portion from an executable file that we generated with a Compiler from the C code of ‘add.c’. An executable file that, when executed, will add ‘2’ and ‘3’, and print out this result in the prescribed format of ‘%d’. In this paragraph we have used two new terms, ‘decompiler’ and ‘compiler’. Now let us know their functions.

7. Compiling Source Code

By definition, a Compiler is a computer program that translates a text written in one computer language, say the Source Language, to another computer language, say the Target Language. Usually, the original sequence in the Source Language is called Source Code, and the translated sequence in the Target Language is called the Object Code. This is, obviously, a very general definition of what Compiler is. Commonly, a compiler is used to translate Source Code (anything like 'add.c'), written in High-Level languages, to an Executable file, written in Machine Language, that is, something understood by the machine. The machine then runs the commands and instructions given there in the executable file. In computer science an executable file means a file whose contents are interpreted as a executable program by a computer. This is not a piece of text that is meant to be read or written by human beings. In some cases, even text files may be executable too, as a list of commands and interpretations that get chronologically executed by the machine. They are the so-called shell scripts. Another complexity also remains concerning the Object Code. In some cases it may call for some kind of Linker operation. But, we are

not going into those details here. We just want to understand compiling in the most plain and simple way possible, only in order to enable us to understand what Source Code is, and how it becomes in reality Free or Captive.

Let us go directly to the case of the C code, 'add.c'. This is not an executable program, it does not run. It is a text file. So, now, how to get a program, or, an executable file from this piece of text held in a text file called 'add.c'? On different Operating Systems, with different compilers there are a lot of different ways to generate the program or the executable file. The thing that was quoted in the earlier section from the program called 'add', was from a binary or executable file generated on a Fedora 7 GNU-Linux Operating System, with a compiler called GCC. The command that created the executable file was:

```
gcc -o add add.c
```

The 'gcc' part of the command calls the GCC program into its action of compiling. The '-o' part specifies the name of the executable file. And obviously, 'add.c' is the name of the code file that we want to compile. This command makes GCC generate an executable file called 'add'. This is an executable file, and hence it can run. It can be run with a command, issued from within the directory where it is:

```
./add
```

When run, it gives this output:

```
2 + 3 = 5
```

So, the compiler works in one direction: it transforms human-understandable code to object code or machine-executable programs. And the decompiler works exactly in the reverse direction, transforming binary or executable programs into human-understandable high-level code. Later, when we discuss about the rights of an user of a piece of software, we will need both the concepts of *compiling* and *decompiling*.

8. Source Code, Object Code, Portability

The technical difference between these two forms of code, source code and object code, is now clear. But, this is going to be very important for us later, when we enter into the principles of Software Licensing. So, let us make things a bit clearer here. As we will see later, in the context of a license, software can be distributed in both the forms, Source Code and Object Code.

Gradually, with the advent of international trade, jurisdiction of the copyright concepts and laws began to transcend borders, and thus, it became necessary to standardize these things on an international plane. The idea of 'copyright' originated from the first real copyright act *Statute of Anne* passed in 1710 in Britain. Then copyright act came up in different countries, but they all applied locally, that is, within the country. On the international level, one pioneer venture in this area was the 'The Berne Convention for the Protection of Literary and Artistic Works', an international agreement about copyright laws. It is usually called as 'Berne Convention'. It took place in Berne, Switzerland in 1886. Berne Convention represented a kind of consensus position about market of 'thought products' in

the West. From a Cultural Studies point of view it may be interesting to note that it was none other than Victor Hugo who took a leading role in the organization of the Berne Convention and in a way it was a fusion of two concepts. One was the ‘right-of-the-author’ or ‘*droit d’auteur*’. This concept is obviously bigger in its cultural scope than the strict economic connotations of the second concept of ‘copyright’ from the Anglo-Saxon world. The time of the Berne Convention becomes more important in its culture studies surplus meanings if we cross-check it with the European history. The British Glorious Revolution, that changed the face of the world for good, was just complete in every sense at the time of Berne Convention. And, at this moment, obviously, the balance of European power was in an absolute tilt against the “Norman” in favor of the “Saxon”, to use the timeless Walter Scott paradigm of Robin Hood and Ivanhoe. Obviously it was more than a century before the *Rule Britannica* would leave the scene in a total exit and it will be a solitary show of the “New World”, the US of A.

According to the Berne Convention, copyrights for creative works are not subject to any registration or application or something like that. It is entirely *automatic*, even if there is an absolute lack of assertion or declaration by the author. The moment the work is complete, written or recorded on a medium, the author is by default entitled to copyrights in the work, or to any derivative work, in any country under the Berne Convention. Only an express and explicit declaration of renunciation or disclaiming by the author can negate these rights, at any point of time before the copyright expires. During the 1990s, WTO, World-Trade-Organization, and TRIPS (Agreement on Trade Related aspects Intellectual Property Rights) tried to bring the realm of computer software under a strong surveillance of copyright laws. Any country that wanted to become a member of WTO was required to sign TRIPS. And it was obligatory for any TRIPS signatory to conform to the laws. According to TRIPS, the copyright laws apply to both the forms of Source Code and Object Code. In the market reality of computer software, the proprietary software companies release their product in the form of object code. They keep the source code hidden – the source code that was actually compiled to produce this object code is kept as a trade secret. There are many more layers of complexity here: later we will go into the details of the copyright laws. For the time being let us make the dichotomy between object code and source code a bit clearer, and what real difference it generates in the market.

We have already witnessed a compiler, GCC, in action, when we transformed our source code into object code. The end product was an executable file or binary file, or, in that sense, an object file. An object file is a file used for storing the compiled object code and data related to it, after the compiler compiles the source code. Without going into many related additional complexities, we can say that these object files contain the machine code, that is, instructions meant for the machine, instructions that enable the machine to run and execute the things that we intend it to do in our source code. Object files can come in various file formats. Gradually some particular formats like COFF, Common-Object-File-Format, and ELF, Executable-and-Linkable-Format, have become standardized, by formulating and defining these formats, and using them on various kinds of operating systems and machine architectures. As their names suggest, COFF and ELF contain the executable machine code, and all the linked libraries and all that they need to run.

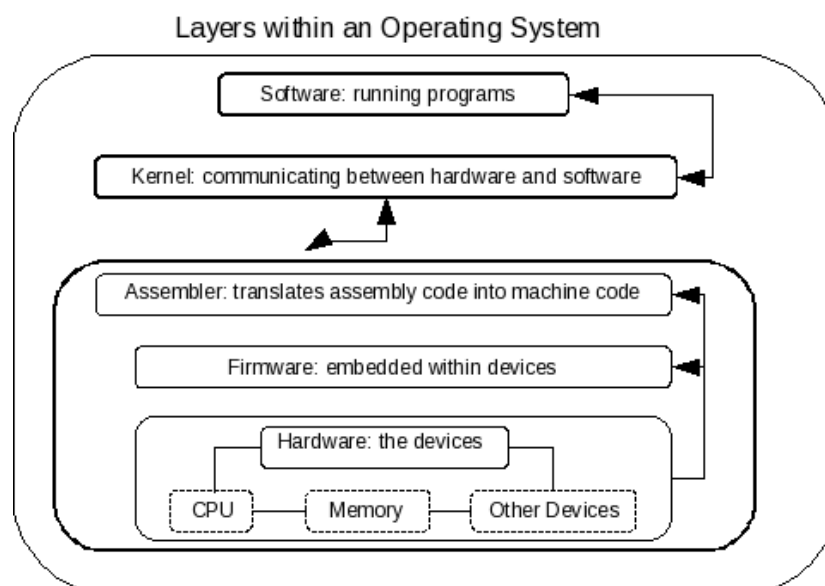
Actually, before the very advent of ‘portable’ operating systems like Unix, every computer

had its own brand of Object File. There are various kinds of machines and architectures. And each one is different from the others. Earlier in this chapter too, we wanted to give hints about it. But, now, as we proceed, we are going to discover some very practical implications of these things in the coming chapters. In an oversimplified way, computer architecture is the fundamental structure of computer operation. It is built in layers. The lowest layer is obviously the hardware, that is the pieces of metals and plastic, the wires, and most importantly, the silicon chips. A myriad of different kinds of devices that, when put together, under the supervision and direction of operating system, become a working computer. But, without all the codes and instructions working inside them, these devices are dead.

Immediately over the hardware layer comes the layer of firmwares or instructions embedded within the pieces of hardware, if any. Just above this layer of firmwares is the work-space of the device drivers. When any piece of hardware is attached to a machine, the operating system, in order to use this device, needs a command set that is called the device driver. Firmware and device driver together enable the operating system running on a machine to work with the devices attached to computer.

Just above this layer of firmware and device drivers comes the layer of assembler. This assembler, in a nontechnical way, is a kind of an utility that translates the codes in the low level assembly language into machine codes. These machine codes are the very instructions that are finally and actually run by the operating system through the hardware devices when computer operates.

Then comes the kernel layer. Kernel is the central-most component of the operating system that actually is responsible for the allocation and use of the resources. By ‘resources’ we mean the totality of all the devices. Kernel allocates them to all the running processes within operating system. Every running software generates one or more processes within operating system. And so, by allocating the resources, kernel actually enables these processes to work. At any point in time, several different programs maybe running simultaneously together. Say, a browser is displaying some pages from the Net, a word-



processor is creating pages, a graphics program is drawing a diagram, and a multimedia program is playing some music. All these programs are generating a plethora of processes, which are all simultaneously demanding the resources of the CPU, the RAM, the hard-disk and everything. And all these must work without any of them disturbing any other. Kernel actually communicates between hardware and software components of the working machine and enables the whole computer system to run, together with all its component processes. So, now the point is, whenever we are referring to the term ‘computer architecture’, that means the whole conceptual structure shown in the diagram. And every layer in the diagram is related to its upper one in a causal chain. If we start from a particular make of CPU and other related devices, obviously the firmware would be specific to it. And so will be the drivers, and thus the whole structure would become very specific. Like the popular architecture categories of i386, PowerPC, Sparc, and so on.

The ‘i386’ category is one of many ways like ‘Intel386’, ‘i386’, or simply ‘386’, to refer to the series of microprocessor chips made by Intel, used as CPU since 1986. The generic name of the ‘instruction set’ for this genre of CPU is known as ‘x86’. Starting from the lowest layer of this instruction set, through all the layers in the diagram, this instruction set, together with all the specific instruction sets related to the specific components, generates a particular kind of architecture. Another important example of architecture is ‘PowerPC’. This is another CPU architecture of the kind called RISC, Reduced-Instruction-Set-Computer. This is a kind of CPU designing strategy where the instructions are do-less, do-simple and thus do-fast type. There are a lot of architectures that fall within the RISC category, but this is obviously outside the scope of our discussion. PowerPC was created by AIM, or Apple-IBM-Motorola alliance, in 1991. Though it was primarily intended for the PC, PowerPC became quite popular in embedded systems. Sparc is another RISC architecture originally designed by Sun Microsystems in 1985. These are just a few examples of architectures, there are many more, and many other that follow one of the generic architectures, though deviate in minor ways.

That a program runs on one particular architecture does not necessarily mean that it will run on other architectures. Here, by the word ‘program’ we mean the executable aspect of the program, or object code. Take our ‘add.c’ source code for example. From source code ‘add.c’, with GCC, we compiled object code named ‘add’. The binary object code was built in such a way that it can run on that Fedora 7 GNU-Linux environment on a PC. Even on a PC, in some other environment like MS-Windows Vista, XP or 95, or MacOS, it will not run. And obviously it will not run on some other architecture like PowerPC or Sparc or something. But, the portability of source code is here that, the same source code ‘add.c’ can be compiled on different systems in different ways to make the object code run on any architecture or any operating system. So, the point is that, source code remains the same, while the machine-understandable instructions imbibed within object code become different depending on the architecture or OS environment, for which it was compiled. This is the issue of portability that is going to assume an important aspect in this book.

Portability is one of the key concepts of programming in a high-level language like C. By this feature of portability, the same code-stock is able of being reused, in place of creating a new set of code when moving software from one environment or architecture to another. This is an important point in terms of our discussion in the coming chapters. Object code is

never portable. It is specific to one environment or architecture. Though, some combined kind of object code is possible. Combined code means more than one set of object code put together, enhancing its versatility. But it is way out of scope of our discussion. And, the point is, object code hides the source code behind it. So, some object code that runs on one architecture or environment, is useless in some other setting. And more important is that, because it hides the source-code, nothing can be done with the object code if there is a change in perspective in terms of architecture or operating-system. But, source-code, like our 'add.c', is very plainly understandable to anyone who knows the grammar of C, and hence can be easily modified, customized, and recompiled to suit any changed perspective. And it opens up ways for new explorations too, when someone wanting to add '4' and '5', or subtracting them, and so on, can easily tweak this code and reuse. And this modified source-code again can be compiled according to any environment or architecture.

Let us take an example. When one buys a copy of the Microsoft word-processor 'MS-Word', one buys object code. Not only this is meant to run only on one architecture or environment, the source code behind this object code is unknowable too. So, no one can modify, improve or customize it by changing source code. While another word-processor 'OpenOffice.org-Writer' is available in both its versions, source code and object code. This is a FLOSS piece of software. As we will learn it in the later chapters, this is obligatory for software under GPL to make the source-code available. So, any user of this software, if he has the ability, is free to change it, or, modify it according to any need of architecture or environment or personal taste or anything. And more than that, as we will see later, that modified version too can be distributed at will, both object code and source code. In the coming chapters we are going take up this issue again and again, while discussing GPL and FLOSS.

9. Layers of Hardware and Software

After the discussion of a model of layers within computer architecture, let us now elaborate the layers within the space of man-machine interface within computer. Within an operating computer, when some work is getting done on the machine, actually a very large number of activities are happening simultaneously together in different layers. They include multitudes of user-level software through the hardware layers of silicon chips, metal connections and plastic and wires.

First, let us understand a little, what is the role of a program or a group of programs in an operating system. Let us remember, this operating system is again a program, or, a group of programs. A simple definition of *operating system* or OS is something like this: the first piece of program (or, a collection of intertwined programs) that operates on a machine, and makes the machine usable to all other later programs to follow. The word 'usable' means a lot of things. It means that operating system, OS, is the piece of software that manages the sharing of the resources of a computer, and provides users with an interface to access the resources. Every machine is one particular assembly of different available internal and external components. Every component has its own way of working. OS looks after every possible mode of working of every component in a machine through all the drivers that literally drive the components. When some user wants to work on a particular component, the user can forget all the intricate details of the way of working of that particular piece of

component. OS looks after these details.

OS generates the very man-machine interface that allows human subjects to go on working with the machine. OS makes the two ends meet. It processes system data and user input in a parallel and simultaneous way. OS has to do it that way, because, OS is the sole authority that allocates and manages tasks and internal system resources among different processes. These processes are always running within a working machine as a service to users using the system and programs running on the system. Users of applications are oblivious towards this intense back-end operation almost all the time. In fact, the very job of OS is to generate this comfort of oblivion for users of application programs. But, whether users know it or not, at the foundation of all running software, OS is endlessly and ceaselessly going on performing the very basic tasks of controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating computer networking and managing files.

For any modern OS, computer works in a multi-user, multi-task mode. That is, even if there is only one human user, so many programs do run together, for the maintenance, well-being and efficiency of the system: both the system programs and the user programs. And in many cases, there are more than one users using the system, with many different programs. All these programs are sending their demands to the same CPU, or the same pool of memory devices. Like say, the text someone is writing, and the music someone else is playing, and some video another is downloading from the Net, are all working through the same CPU. And all the programs are writing and reading data to the same hard-disk. And more important is the fact that, CPU can do only one thing at a time. So, it calls for a kind of breaking of every period of time into infinitely large number of small segments. Once this segmenting is done, it is easy to allocate one segment each to one process at a time. Then this allocation just goes through a continuous rotation for all the processes one by one. We do not keep it in mind. That is possible only because OS does it for us. It takes away all the trouble of remembering all the intricate hardware details, generating an easy-understandable virtual structure of the machine. This allows us to use it comfortably. Tanenbaum 2002 deals this whole thing quite exhaustively.

Layers from Hardware to Software within an Operating Computer			
Accounting ...	Railways ...	Internet ...	Application Layer
Compiler	Editor	Command Interpreter	Central Layer of System Programs
Operating System			
Machine Language			Hardware Layer of Physical Components
Silicon, Metal, Plastics			

Now, with this scheme, let us understand the deep layers within the working machine. The lowest layer is the Hardware Layer of Physical Components. That includes the chips, the transistors, the connectors, the boards, the wires, and all. This is the layer where Machine Language or Machine Code works. Machine Language is a system of instructions that is directly executed by the CPU of the machine. In a sense, Machine Language is a

programming language too, that works with the lowest level of abstraction for representing the commands. We know that, any program, however big or small, smart or dumb, is a list of commands to be executed by the machine, and some data relevant for doing this. And in this layer, this whole work is done at the lowest level of abstraction. This Hardware Layer is actually divided into sub-layers of machine details and the most elementary form of communication. This communication works through a pre-language that sends the commands and instructions directly into hardware. This is the world of micro architecture and machine language residing in silicon chips and wires and metal junctions and all those. This is the layer that we can totally forget while running a machine – and OS makes that possible.

The next layer is the Central Layer of System Programs. This is the layer of operation of OS. OS enables the system programs to run, while OS itself is the most primal and central system program. This layer of system programs comes above the lowest layer of Physical components. This layer is divided into two sub-layers. OS is the lower sub-layer here. OS actually creates a kind of blanket over the hardware details, and generates a virtual space, as we said. This is a space that is intelligible and understandable by human beings. This space enables users to create and use programs. Three other things are mentioned in this layer: Editor, Compiler, and Command Interpreter.

Editor means a simple text editor. That means, not the usual word-processors like OpenOffice.org Writer or MS-Word. For this kind of editor, text is simply an array of characters, without any special formatting attached to them, like ‘bold’ or ‘italic’, this font or that font, and so on. Earlier in this chapter we wrote a C program, ‘add.c’, and compiled it into an executable or binary file ‘add’, and then ran it. What things we used for it? To write the code, we needed editor, something that edits simple and plain text. Then to compile this code we needed *compiler*. In this particular case compiler was GCC. To run these programs, to invoke editor or compiler, we issued several commands. When the C code ‘add.c’ was compiled into an executable file called ‘add’, we gave a command ‘gcc -o add add.c’. To run ‘add’, the command was ‘./add’. When we are issuing these commands, who is taking these commands and performing as commanded?

This is *command interpreter*. When we gave the command, ‘./add’, and pushed the ‘Enter’ key, this meant, take the executable file ‘add’ from the current directory and run it. Someone searched the current directory, found ‘add’ and then executed it. This someone was command interpreter. Command Interpreter, usually called a shell, is the middleman between us and the OS. Though, the problem is, in a lot of cases, the average user is hardly aware of this omnipresent shell or command-interpreter. This is particularly so in the case of the MS-Windows systems. The GUI, Graphical-User-Interface, the way of doing things with pictures and mouse, in place of the ways with typing commands, the CLI, Command-Line-Interface, tries a lot to hide the workings of shell or command interpreter. This is particularly true in case of the proprietary systems like MS-Windows. We will come to this point of GUI and CLI in the next section.

Above this central layer of System Programs comes the third and final layer of Application Software. Usually this is the layer where the realm of a common computer user starts and ends. Though it is very hard to define a ‘common computer use’. Here we mean all those things that concern an average user, like the word-processor, the database application, the

browser, the music player, the movie player, and so on. Later we will see, defining this user, and then interrogating that definition will be very much within the scope of the discussions of political economic and cultural questions in the coming chapters. For now, let us settle it here that, this third and final Application Layer is the layer in use pretty everywhere: in railways ticket counters, in banks, or while surfing the Net.

10. GUI and CLI

A UI or User-Interface allows people to interact with computers and their peripherals. GUI and CLI – these two are the two basic UI things before us. In a GUI we work with pictures and a mouse, and hence it is called a graphical UI or GUI. It generates a graphical environment made of icons, widgets, windows, where we can point with a pointing device like a mouse and thus work with it. These graphical elements like icons, widgets, windows are used in conjunction with text, wherever necessary. A ‘widget’, literally meaning a small mechanical device or control, like a gadget, is a technical term in the realm of GUI literature. It is a basic element of a GUI. A widget is a visual building block that constructs all the components of the GUI environment. When working with a mouse in a GUI environment one of the most frequent things that we do is clicking buttons. These buttons are widgets.

There are different kinds of buttons, like a ‘toggle button’ that can take any one of the two states at any moment: ‘on’ or ‘off’, or a ‘check box’ where a tick appears when we select the option represented by the check box. And it is a toggle button too in the sense that it is either selected or not. There is a ‘radio button’ which represents one of several predefined options and it is a toggle button too, changing the shape or color when we click it to show which state it is in. All these are examples of widgets. Other frequently used widgets include a ‘slider’ indicating a variable value, a ‘scroll bar’ that is a kind of slider that shows the coordinates of the position within a window, a ‘list box’ that displays a list, or a ‘drop-down list’ that drops a list when we click on it. The windows are the primary things that we interact with when in a GUI environment. These windows are made of many widgets.

These widgets serve functions of different kinds. Like, say, the ‘windows’ or the things that open on the desktop, the ‘menus’ or the lists full of different elements that can be clicked with a pointing device like the mouse, the ‘radio buttons’ or the ‘check boxes’ or the ‘icons’ or the small pictorial representation of different elements of the machine or the devices connected with it, like the CD-Drive or the printer. These widgets create the visual environment. These widgets are kind of virtual switches that we click with the pointing devices like a mouse or a trackball, in contrast to real physical switches that we click with our fingers or otherwise. This ‘virtual dimension’ issue will come back in our later discussions.

In contrast to the GUI, the CLI, Command-Line-Interface, allows a user to do the same things by typing in different commands. We have already experienced some ways of working in a CLI in working with ‘add.c’. We issued different commands invoked the compiler GCC to compile object code that we named as ‘add’ from source code ‘add.c’, and ran this binary called ‘add’. We did all this in a CLI. All the command that we used were typed in on the command prompt and were interpreted and executed by shell or

command interpreter. The CLI is the method of interacting with the objects, the files, the programs, and the OS as a whole, through typed-in commands.

A CLI usually works in a terminal, obviously, where we can work with keyboard and text. In a CLI, every time we issue a command, we end it with hitting the ‘Enter’ key that enters the command into computer. Here, as we can see, directly the text signs are used, without using any visual sign. From those extremely early days in the life-story of computers, say around 1960, much before the GUI was invented at all, this CLI was the most primary means of user interaction with the computers. The CLI allows the users to issue the commands in a terse and precise way, along with the parameters and options of the commands, obeying the syntax rules of shell or command interpreter working behind the command prompt. In fact, if seen this way, the presence of shell is an implicit one in a CLI too, not an explicit one. With some special commands, in some special situations, the presence of shell becomes conspicuous. But this implicitness of shell or command interpreter in a CLI becomes a full-fledged and elaborate hiding in the case of a GUI.

There is another possibility here: that of GUI-CLI. This happens when within a GUI environment, we open a terminal window, and within this terminal we type in commands. Though it is a GUI, we interact with it in a CLI manner. The terminal window is made with widgets, exactly the way any other window is made. The essay, Das 2005, “Tongue to Fingers: Colonizing IT in a Postcolonial World” looks into the politics of mentality. The GUI-only environment on the computer while a student is getting trained makes her/him unduly dependent on ‘mediation’. The GUI things allows one to work only with those options which are always already built into these GUI environments, the environments that were mediated through the software engineers that created it. While the CLI environment intrinsically motivates the student towards a higher level of innovation and customization. And so a good choice is GUI-CLI, with the comfort of a GUI and the flexibility of a CLI.

Section 11. Software: FLOSS or Other

In our times, using computer means using software. Software, as we said, has different forms and purposes, meant for different layers in the hierarchy that starts from micro-architecture and hardware components and ends in the user domain. Now, this book is focused on FLOSS, and what is FLOSS software? It is better to start the definition the other way round, that is, from the other of FLOSS software – the proprietary kind of software. Let us make it very simple. For users of the Microsoft products, using these pieces of software on a machine involves at least two layers. One is the OS layer. For the users of the Microsoft products this layer consists of, say, Microsoft Windows XP or Vista, and related packages that enable this OS to run. Another layer is made up of user level applications, like Microsoft Office. Other possible and frequent user applications are browsers like Internet Explorer from Microsoft, or Graphics things like Adobe Photoshop from Adobe, or multimedia packages like Windows Media Player from Microsoft, and so on for different kinds of activities.

And on a FLOSS system, like the one on which this book is getting written, the OS is Fedora 12, using OpenOffice Writer for writing, and graphics packages like OpenOffice Draw, Xfig and Gimp for creating the diagrams. For multimedia there are things like

Mplayer, Exaile, or Rhythmbox and so on. But, where lies the difference, except the difference in brand-names?

The difference, really, is much bigger for the pieces of software in the second group. The difference starts from the deepest software level, that is the level of kernel, just above the hardware layers. In case of these FLOSS pieces of software operating through particular package-distribution systems like Fedora, the name of this kernel is Linux. This kernel is licensed under GNU GPL. GPL provides any user of this kernel four different kinds freedom. The first freedom is to run and use it. The second freedom is to study how the program works and modify it to suit the user's needs. The third freedom is the freedom to redistribute copies of this kernel to anyone who wants it. The fourth freedom is the freedom to improve the kernel and redistribute the improved copy.

And this is very different with the Microsoft EULA, End-User-License-Agreement, which the Microsoft products are licensed under. Only the first freedom, or freedom to run and use it is granted there, that too, in more cases than not, under many restrictions. Here the Microsoft example was used, but it is true for all proprietary software, though with many variations on the theme of restriction. The freedom to modify or improve is not at all possible there, in the first hand, because they are all closed-source. When someone is purchasing a copy of Microsoft Windows or Microsoft Office, actually a copy of object code is getting purchased. For them, source code is closed. And obviously, rules of the proprietary software market rules out any kind of redistribution.

For FLOSS software, the aspects of freedom are not just true for the kernel, they are true for every single package. Though there are variations of license, and many differences in interpreting freedom. We will come back to these things in fullest possible details later in this book. There are quite a lot of licenses that are recognized under as FLOSS, Free-Libre-Open-source-Software. Later we will get familiar with some of these details, and how they are all related to GPL in terms of genealogy, and what it all means in terms of ethics or right, and also in terms of market or capital. But, for the time being this rudimentary definition will work for us. In the next chapter we will start exploring the history of Source Code, how it was all open in the age of primitive accumulation of computing knowledge, or primitive FLOSS, as we called it. This was before people started trying to close the free and open tradition of cooperative knowledge.