Introduction to Software

I, Starting-up the computer

When you start-up your computer from a powered-down state (**cold boot**) or when you reset it because a program encounters an error from which it cannot recover (**warm boot**, e.g.: [Control]+[Alt]+[Delete]) your computer is going to carry out a series of initializations, tests and loading called boot. Boot is short for bootstrap, which in olden days was a strap attached to the top of your boot that you could pull to help get your boot on. Hence, the expression “pull oneself up by the bootstraps.” Similarly, bootstrap utilities help the computer get started.

The **BIOS** (Basic Input Output System) is responsible for booting the computer by providing a basic set of instructions and gives the computer a little built-in starter kit to run the rest of software from floppy disks (FDD) and hard disks (HDD). The BIOS performs all the tasks that need to be done at start-up time: **POST**. POST is a short for power-on self test, a series of diagnostic tests that run automatically when you turn your computer on. The actual tests can differ depending on how the BIOS is configured, but usually the POST tests the RAM, the keyboard, and the disk drives. If the tests are successful, the computer boots itself. If the tests are unsuccessful, the computer reports the error by emitting a series of beeps and possibly displaying an error message and code on the display screen. The number of beeps indicates the error, but differs from one BIOS to another. If the test is positive the BIOS proceeds to boot an operating system from FDD or HDD).

Furthermore, the BIOS provides an interface to the underlying hardware for the operating system in the form of a library of interrupt handlers, that all the code required to control the keyboard, display screen, disk drives, serial communications, and a number of miscellaneous functions. For instance, each time a key is pressed, the CPU (Central Processing Unit) perform an interrupt to read that key. This is similar for other input/output devices (Serial and parallel ports, video cards, sound cards, hard disk controllers, etc...). Some older PC's cannot co-operate with all the modern hardware because their BIOS does not support that hardware. The operating system cannot call a BIOS routine to use it; this problem can be solved by replacing your BIOS with a newer one, that does support your new hardware, or by installing a device driver for the hardware.

The BIOS is typically placed in a ROM (Read Only Memory) chip that comes with the computer (it is often called a ROM BIOS). This ensures that the BIOS will always be available and will not be damaged by disk failures for example. Because RAM is faster than ROM many computer manufacturers design systems so that the BIOS is copied from ROM to RAM each time the computer is booted. This is known as shadowing. Many modern PCs have a flash BIOS, which means that the BIOS has been recorded on a flash memory chip, which can be updated if necessary.

To perform its tasks, the BIOS need to know various parameters (hardware configuration). These are permanently saved in a little piece (64 bytes) of CMOS RAM (Complementary Metal Oxide Semiconductor Random Access Memory). The CMOS power is supplied by a little battery, so its contents will not be lost after the PC is turned off. Therefore, there is a battery and a small RAM memory on board, which never (should...) loses its information. The memory was in earlier times a part of the clock chip, now it's part of such a highly Integrated Circuit (IC). CMOS is the name of a technology that needs very low power so the computer's battery is not too much in use. Actually, there is not a battery on new boards, but an accumulator (Ni_Cad in most cases). It is recharged every time the computer is turned on. If your CMOS is powered by external batteries, be sure that they are in good operating condition. Also, be sure that they do not leak. That may damage the motherboard. Otherwise, your CMOS may suddenly "forget" its configuration and you may be looking for a problem elsewhere. Some new motherboards have a technology named the Dallas Nov-Ram. It eliminates having an on-board battery: There is a 10 year lithium cell epoxyed into the chip.

To change the parameters with which the BIOS configures your **chipset** (integrated circuits containing the core functionality of the motherboard and extension board) you will use a set of procedures named the SETUP. The original IBM PC was configured by means of DIP switches (Dual-In-line Package) buried on the motherboard.

FREE EBOOKS, NOTES , VIDEOS & PLACEMENT MATERIAL

For All Companies placement Material

@placementclasses

For CAT Exam Preparation Material

@cat_classes

For GATE Exam Preparation Material

@gate_classes

For Engineering Books & Material

@cs_ebooks

Codes of Following Programming Languages

C

@c_examples

C++

@cpp_examples

Java

@java_examples0

Python

@python_examples

Setting PC and XT DIP switches properly was something of an arcane art. DIP (Dual-In-line Package) switches/jumpers are still used for memory configuration and clock speed selection. When the PC-AT was introduced, it included the battery powered CMOS memory. CMOS was originally set by a program on the Diagnostic Disk, however later clones incorporated routines in the BIOS which allowed the CMOS to be (re)configured if certain magic keystrokes were used (e.g. [DEL]). Unfortunately as the chipsets controlling modern CPUs have become more complex, the variety of parameters specifiable in SETUP has grown. Moreover, there has been little standardization of terminology between the half dozen BIOS vendors, three dozen chipset makers and large number of motherboard vendors. Complaints about poor motherboard documentation of SETUP parameters are very common. To exacerbate matters, some parameters are defined by BIOS vendors, others by chipset designers, others by motherboard designers, and others by various combinations of the above. Parameters intended for use in Design and Development, are intermixed with parameters intended to be adjusted by technicians

PC BIOS that can handle Plug-and-Play (PnP) devices are known as PnP BIOS, or PnP-aware BIOS. Plug-and-play refers to the ability of a computer system to automatically configure expansion boards and other devices. You should be able to plug in a device and play with it, without worrying about setting DIP switches, jumpers, and other configuration elements. The PnP BIOS are always implemented with flash memory rather than ROM. ESCD is a short for Extended System Configuration Data, a format for storing information about Plug-and-Play (PnP) devices in the BIOS. Windows and the BIOS access the ESCD area each time you re-boot your computer. SCAM is a short for SCSI Configuration Automatically, a subset of the PnP specification that provides plug-and-play support for SCSI devices.

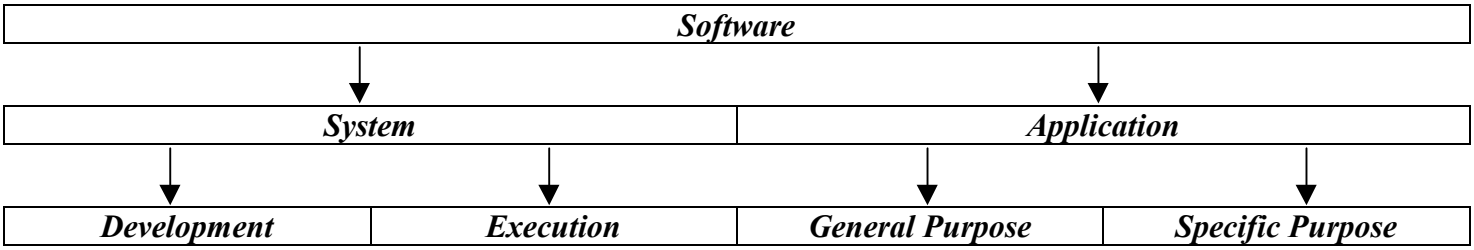
To conclude this part we can notice that the BIOS is always the first software to be executed by a computer.

II, Software types

Software is computer instructions or data. Anything that can be stored electronically is software. The distinction between software and hardware is sometimes confusing because they are so integrally linked. Clearly, when you purchase a program, you are buying software. But to buy the software, you need to buy the disk (hardware) on which the software is recorded. Without software, a computer is just a black box of electronic equipment that is incapable of any useful function. Software tells the computer what to do and when to do it.

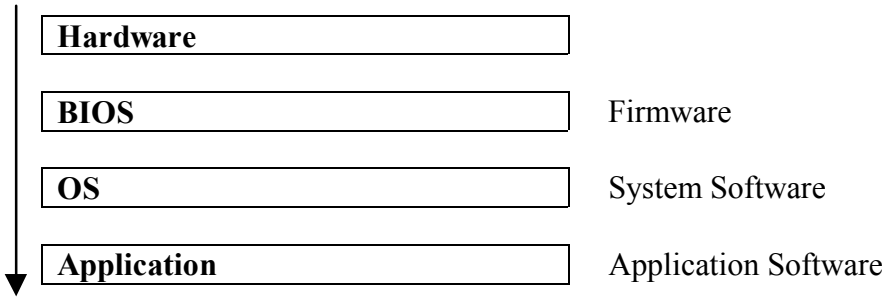
First there is a fundamental difference between programs and data:
- Distinct pieces of information, usually formatted in a special way and available for, or result of, processing.
- Programs are collections of instructions for processing data

Software is often divided into two categories:
- System software: Consists of low-level programs that interact with the computer at a very basic level. It is any software required to support the development or execution control of application programs but which is not specific to any particular application. This includes operating systems, compilers, loaders, linkers, debuggers and utilities for managing computer resources.
- Application software: Sits on top of systems software because it is unable to run without the operating system and system utilities. It includes programs that do real work for users. For example, word processors, spreadsheets, and database management systems fall under the category of general-purpose applications software. Whereas contract evaluation, stock control, payroll fall under the category of specific-purpose applications software



The term 'Software' has some derivatives:

Firmware: Software (programs or data) that has been written onto read-only memory (ROM). Firmware is a combination of software and hardware. ROMs, PROMs and EPROMs that have data or programs recorded on them are firmware. So we can sum up what have been said since the beginning of this document with the following figure:



Public-domain software: Refers to any program that is not copyrighted. Public-domain software is free and can be used without restrictions.

Freeware: Copyrighted software given away for free by the author. Although it is available for free, the author retains the copyright, which means that you cannot do anything with it that is not expressly allowed by the author. Usually, the author allows people to use the software, but not sell it.

Shareware: Software distributed on the basis of an honor system. Most shareware is delivered free of charge, but the author usually requests that you pay a small fee if you like the program and use it regularly. By sending the small fee, you become registered with the producer so that you can receive service assistance and updates. The free use of the software is also usually limited to a period, in general '30 days trial period'. You can copy shareware and pass it along to friends and colleagues, but they too are expected to pay a fee if they use the product. Shareware is inexpensive because it is usually produced by a single programmer and is offered directly to customers. Thus, there are practically no packaging or advertising expenses. Note that shareware differs from public-domain software in that shareware is copyrighted. This means that you cannot sell a shareware product as your own.

You may also encounter the term **Demoware** that usually refers to a cut-down shareware version of a commercial product enabling the user to get the taste of the application without the access to all functionality.

The major par of the software is **commercial**, and this implies a five other types of license agreement:

- Single License: A single copy is bought and is supplied with the installation disks and the manuals. The software can only be installed on a single machine. Each extra machine is added by purchasing another complete package.
- Site License: A single copy of the software is bought with the permission to install the software on an agreed number of computers and only a few copies of the manuals are provided. This is a cheaper method than purchasing a single copy for each machine. An increase in the number of licensed users is achieved by paying for an extension to the existing licensed amount.
- License by Use: This allows the software to be installed on a large number of computers, but the license only allows a fixed number of users to be operating the software at any one time. Increasing the users on this system is identical to the site license arrangements
- License by Station: This allows a fixed number of machines to have the software installed. If it is a single-user license, the software must reside on a single machine.
- Network Multi-License: If an organization has a local area network, an individual software package for all the computers will reside as a single copy on a server. Many single-user packages will refuse to work over a network and special network versions have to be bought. Only a fixed number of users will be able to access the package on the server at any one time.

III, Software creation and programming languages

The 'raison d'être' of a piece of software is always a 'problem' that needs to be solved. The first step is then to develop an algorithm, that is to say a formula or set of steps for solving a particular problem. To be an algorithm, a set of rules must be unambiguous and have a clear stopping point. Algorithms can be expressed in any language, from natural languages like English or French to drawings, chart, and even programming languages. We use algorithms every day. For example, a recipe for baking a cake is an algorithm. Most programs, with the exception of some artificial intelligence applications, consist of algorithms. Inventing elegant algorithms -- algorithms that are simple and require the fewest steps possible -- is one of the principal challenges in programming.

Once you designed an algorithm to solve your problem, you have to code it in a programming language. This stage is sometime called 'implementation' and the result is program. The noun "program" describes a single, complete and more-or-less self-contained list of instructions, often stored in a single file, whereas "code" is uncountable noun describing some number of instructions which may constitute one or more programs or part thereof. To continue the parallel with a cooking recipe we can say that the program contains a list of ingredients (called variables) and a list of directions (called statements) that tell the computer what to do with the variables. The variables can represent numeric data, text, or graphical images.

When you buy software, you normally buy an executable version of a program. This means that the program is already in machine language -- it has already been compiled and assembled and is ready to execute. Therefore most programs rely heavily on various kinds of operating system software for their execution.

To code your algorithm you can choose low-level languages (closer to the language used by a computer) or high-level languages (closer to human languages), but eventually every program must be translated into a machine language that the computer can understand:

- The assembly code (low-level) must go through an assembler that translates it to machine language.
- The high-level language code must be compiled or interpreted

So there are two ways to run programs written in a high-level language. The most common is to compile the program; the other method is to pass the program through an interpreter.

To compile is to transform a program written in a high-level programming language from source code into object code. Programmers write programs in a form called source code. Source code must go through several steps before it becomes an executable program. The first step is to pass the source code through a compiler, which translates the high-level language instructions into object code which is often the same as or similar to a computer's machine language. The final step in producing an executable program is to pass the object code through a linker (or assemblers, binders, loaders). The linker combines modules and gives real values to all symbolic addresses, thereby producing machine code.

The compiler derives its name from the way it works, looking at the entire piece of source code and collecting and reorganizing the instructions. Thus, a compiler differs from an interpreter, which analyzes and executes each line of source code in succession, without looking at the entire program. An interpreter is a program that directly interprets and executes instructions written in a high-level language.

The advantage of a compiler is that once the program is compiled it does not need any other program (except the OS) to run. Moreover, programs produced by compilers run much faster than the same programs executed by an interpreter. Every high-level programming language (except strictly interpretive languages) comes with a compiler. Because compilers translate source code into object code, which is unique for each type of computer, many compilers are available for the same language. For example, there is a C++ compiler for PCs and another for Apple Macintosh computers. In addition, the compiler industry is quite competitive, so there are actually many compilers for each language on each type of computer. More than a dozen companies develop and sell C compilers for the PC.

The advantage of interpreters is that they can execute a program immediately. Compilers require some time before an executable program emerges, and the process can be time-consuming if the program is long. For this reason, interpreters are sometimes used during the development of a program, when a programmer wants to add small sections at a time and test them quickly. In addition, interpreters are often used in education because they allow students to program interactively. Both interpreters and compilers are available for most high-level languages. However, BASIC and LISP are especially designed to be executed by an interpreter. In addition, page description languages, such as PostScript, use an interpreter. Every PostScript printer, for example, has a built-in interpreter that executes PostScript instructions.

Assembler, compiler, interpreter and linker are system software.

In the history of computing we generally distinguish several generations of languages:

First generation language (1GL): The machine codes

The machine code is the (binary) representation of a computer program that is actually read and interpreted by the computer. When computers were first "programmed" from an input device, rather than by being rewired, they were fed input in the form of numbers, which they then interpreted as commands. A program in machine code consists of a sequence of machine instructions (possibly interspersed with data). Instructions are binary strings which may be either all the same size (e.g. one 32-bit word for many modern RISC microprocessors) or of different sizes, in which case the size of the instruction is determined from the first word (e.g. Motorola 68000) or byte (e.g. Inmos transputer). A program fragment might look like "010101 010110". The collection of all possible instructions for a particular computer is known as its "instruction set". Execution of machine code may either be hard-wired into the central processing unit or it may be controlled by microcode. The basic execution cycle consists of fetching the next instruction from main memory, decoding it (determining which operation it specifies and the location of any arguments), executing it by opening various gates (e.g. to allow data to flow from main memory into a CPU register) and enabling functional units (e.g. signalling to the ALU - arithmetic logic unit of the CPU- to perform an addition). Almost no one programs in machine language anymore.

Second generation language (2GL): The Assembly languages

Assembly languages have the same structure and set of commands as machine languages, but they enable a programmer to use names instead of numbers. Each type of CPU has its own machine language and assembly language, so an assembly language program written for one type of CPU won't run on another. In the early days of programming, all programs were written in assembly language. Now, most programs are written in a high-level. Programming in assembly language is slow and error-prone but is the only way to squeeze every last bit of performance out of the hardware and therefore programmers still use assembly language when speed is essential or when they need to perform an operation that isn't possible in a high-level language. An assembler is a program that translates programs from assembly language to machine language.

Third generation language (3GL): The High level languages

A high-level language is designed to be easier for a human to understand, including things like named variables. A fragment might be:

```
let c = c + 2 * d
```

Fortran, ALGOL and COBOL are early examples of this sort of language. Most "modern" languages (BASIC, C, C++, Java…) are third generation. Most 3GLs support structured programming, and they are the languages used to develop the major part of the actual packages.

Fourth generation language (4GL): The Application-specific languages

The term was invented by Jim Martin to refer to non-procedural high level languages built around database systems. The first three generations were developed fairly quickly, but it was still frustrating, slow, and error prone to program computers, leading to the first "programming crisis", in which the amount of work that might be assigned to programmers greatly exceeded the amount of programmer time available to do it. Meanwhile, a lot of experience was gathered in certain areas, and it became clear that certain applications could be generalized by adding limited programming languages to them. Thus were born report-generator languages, which were fed a description of the data format and the report to generate and turned that into a COBOL (or other language) program which actually contained the commands to read and process the data and place the results on the page.

Some other successful 4th-generation languages are: database query languages (e.g. SQL), PostScript, Mathematica, HTML,…

Fifth generation language:

A myth the Japanese spent a lot of money on. In about 1982, MITI decided it would spend ten years and a lot of money applying artificial intelligence to programming, thus solving the software crisis. The project spent its money and its ten years and in 1992 closed down with a whimper.