CHAPTER 11

11. PLANNING THE COMPUTER PROGRAM

In Chapter 10, computer programs have been described as the software of the computer system. We are now at the point in the system life cycle where programs are to be written. But before writing a computer program, one must be clear about the processing steps to be performed by the computer. Thus, to produce an effective computer program, one must first plan the logic (the various steps) of the program. If one attempts to plan the logic approach and write the program at the same time, he will likely become so involved with the required instruction formats that program logic will suffer. Hence, before we learn how to write a program (next chapter), we will first learn how to plan the logic of a computer program in this chapter.

PURPOSE OF PROGRAM PLANNING

Suppose you are asked by your teacher to solve an arithmetic problem and you are not familiar with the steps involved in solving that problem. In such a situation, you will not be able to solve the problem. The same principle applies to writing computer programs also. A programmer cannot write the instructions to be followed by a computer unless the programmer knows how to solve the problem manually.

Suppose you know the steps to be followed for solving the given problem but while solving the problem, you forget to apply some of the steps or you apply the calculation steps in the wrong sequence. Obviously, you will get a wrong answer. Similarly, while writing a computer program, if the programmer leaves out some of the instructions for the computer or writes the instructions in the wrong sequence, then the computer will calculate a wrong answer. Thus, to produce an effective computer program, it is necessary that the programmer writes each and every instruction in the proper sequence. However, the instruction sequence (logic) of a computer program can be very complex. Hence, in order to ensure that the program instructions are appropriate for the problem and are in the correct sequence, programs must be planned before they are written.

ALGORITHM

The term algorithm may be formally defined as a sequence of instructions designed in such a way that if the instructions are executed in the specified sequence, the desired results will be obtained. The instructions, however, should be precise and unambiguous and the result should be obtained after a finite number of executional steps. The latter condition actually states that an algorithm must terminate and should not repeat one or more instructions infinitely. In other words, the algorithm represents the logic of the processing to be performed. However, in order to qualify as an algorithm, a sequence of instructions must possess the following characteristics :

1. Each and every instruction should be precise and unambiguous.
2. Each instruction should be such that it can be performed in a finite time.
3. One or more instructions should not be repeated infinitely. This ensures that the algorithm will ultimately terminate.
4. After performing the instructions, that is after the algorithm terminates, the desired results must be obtained.

To gain insight into algorithms, let us consider a simple example.

Example 11.1. There are 50 students in a class who appeared in their final examination. Their marksheets have been given to you. Write an algorithm to calculate and print the total number of students who passed in first division.

Algorithm :

```
Step 1 :   Initialize TOTAL FIRST DIVISION and TOTAL MARK SHEETS CHECKED to zero.
Step 2 :   Take the marksheet of the next student.
Step 3 :   Check the division column of the marksheet to see if it is I : if no, go to step 5.
Step 4 :   Add 1 to TOTAL FIRST DIVISION.
Step 5 :   Add 1 to TOTAL MARKSHEETS CHECKED.
Step 6 :   Is TOTAL MARKSHEETS CHECKED = 50 : if no, go to step 2.
Step 7 :   Print TOTAL FIRST DIVISION.
Step 8 :   Stop.
```

It must be clear to the readers from this example that even for very simple problems, the development of algorithms is not so simple as it might initially appear and requires some thinking. It may also be noted from the given example that in order to solve a given problem, each and every instruction must be strictly carried out in a particular sequence. It is this fact which a beginner to problem solving by computers finds difficult to appreciate.

There are various ways in which an algorithm can be expressed. When an algorithm is expressed in a programming language, it becomes a program. Thus, any program is an algorithm although the reverse is not true. Besides represented as programs, algorithms are often expressed in the form of flowcharts which is discussed below.

FLOWCHARTS

A flowchart is a pictorial representation of an algorithm that uses boxes of different shapes to denote different types of instructions. The actual instructions are written within these boxes using clear and concise statements. These boxes are connected by solid lines having arrow marks to indicate the flow of operation, that is, the exact sequence in which the instructions are to be executed.

Normally, an algorithm is first represented in the form of a flowchart and the flowchart is then expressed in some programming language to prepare a computer program. The main advantage of this two step approach in program writing is that while drawing a flowchart one is not concerned with the details of the elements of programming language. Hence, he can fully concentrate on the logic of the procedure. Moreover, since a flowchart shows the flow of operations in pictorial form, any error in the logic of the procedure can be detected more easily than in the case of a program. Once the flowchart is ready, the programmer can forget about the logic and can concentrate only on coding the operations in each box of the flowchart in terms of the statements of the programming language. This will normally ensure an error-free program.

A flowchart, therefore, is a picture of the logic to be included in the computer program. It is simply a method of assisting the programmer to lay out, in a visual, two-dimensional format, ideas on how to organise a sequence of steps necessary to solve a problem by a computer. It is basically the plan to be followed when the program is written. It acts like a road map for a programmer and guides him how to go from the starting point to the final point while writing a computer program.

Experienced programmers sometimes write programs without drawing the flowchart. However, for a beginner it is recommended that a flowchart be drawn first in order to reduce the number of errors and omissions in the program. Moreover, it is a good practice to have a flowchart along with a computer program because a flowchart is very helpful during the testing of the program as well as while incorporating further modifications in the program.

FLOWCHART SYMBOLS

We have seen that a flowchart uses boxes of different shapes to denote different types of instructions. The communication of program logic through flowcharts is made easier through the use of symbols that have standardized meanings. For example, a diamond always means a decision. Only a few symbols are needed to indicate the necessary operations in a flowchart. These symbols have been standardised by the American National Standards Institute (ANSI). These symbols are shown in Figure 11.1 and their functions are discussed below.

Terminal. The terminal symbol, as the name implies, is used to indicate the beginning (START), ending (STOP), and pauses (HALT) in the program logic flow. It is the first symbol and the last symbol in the program logic. In addition, if the program logic calls for a pause in the program, that also is indicated with a terminal symbol. A pause is normally used in the program logic under some error conditions or if forms had to be changed in the computer's line printer during the processing of that program.

Input/Output. The input/output symbol is used to denote any function of an input/output device in the program. If there is a program instruction to input data from a disk, tape, card reader, terminal, or any other type of input device, that step will be indicated in the flowchart with an input/output symbol. Similarly, all output instructions, whether it is output on a printer, magnetic tape, magnetic disk, terminal screen, or any output device, are indicated in the flowchart with an input/output symbol.

Processing. A processing symbol is used in a flowchart to represent arithmetic and data movement instructions. Thus, all arithmetic processes of adding, subtracting, multiplying and dividing are shown by a processing symbol. The logical process of moving data from one location of the main memory to another is also denoted by this symbol. When more than one arithmetic and data movement instructions are to be executed consecutively, they are normally placed in the same processing box and they are assumed to be executed in the order of their appearance.

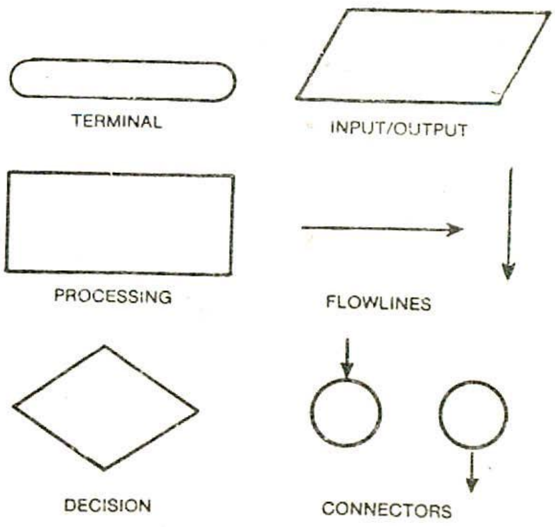


Figure 11.1. Various flowchart symbols.

Flowlines. Flowlines with arrowheads are used to indicate the flow of operation, that is, the exact sequence in which the instructions are to be executed. The normal flow of flowchart is from top to bottom and left to right. Arrowheads are required only when the normal top to bottom flow is not to be followed. However, as a good practice and in order to avoid ambiguity, flowlines are usually drawn with an arrowhead at the point of entry to a symbol. Good practice also dictates that flowlines should not cross each other and that such intersections should be avoided whenever possible.

Decision. The decision symbol is used in a flowchart to indicate a point at which a decision has to be made and a branch to one of two or more alternative points is possible. Figure 11.2 shows three different ways in which a decision symbol can be used. It may be noted from these examples that the criterion for making the decision should be indicated clearly within the decision box. Moreover, the condition upon which each of the possible exit paths will be executed should be identified and all the possible paths should be accounted for. During execution, the appropriate path is followed depending upon the result of the decision.

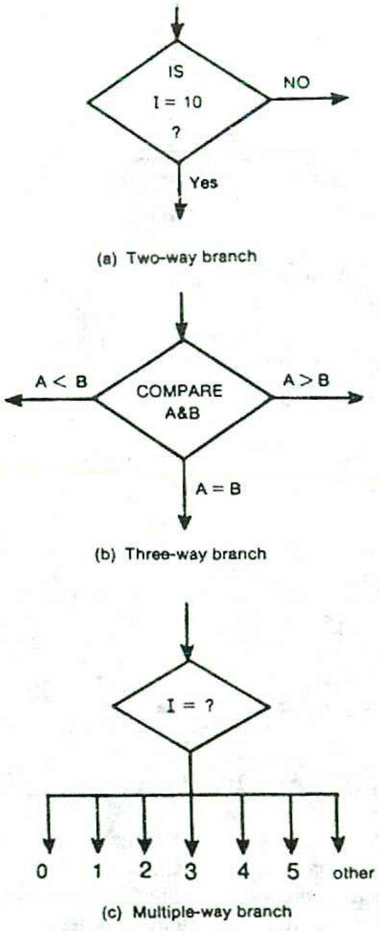


Figure 11.2. Examples of decision symbols.

Connector. If a flowchart becomes very long, the flowlines start criss-crossing at many places that causes confusion and reduces understandability of the flowchart. Moreover, there are instances when a flowchart becomes too long to fit in a single page and the use of flowlines becomes impossible. Thus, whenever a flowchart becomes complex enough that the number and direction of flowlines is confusing or it spreads over more than one page, it is useful to utilize the connector symbol as a substitute for flowlines. This symbol represents an entry from, or an exit to another part of the flowchart. A connector symbol is represented by a circle and a letter or digit is placed within the circle to indicate the link. A pair of identically labeled connector symbols is commonly used to indicate a continued flow when the use of a line is confusing. So two connectors with identical labels serve the same function as a long flowline. That is, they show an exit to some other chart section, or they indicate an entry from another part of the chart. How is it possible to determine if a connector is used as an entry or an exit point? It is very simple: if an arrow enters but does not leave a connector, it is an exit point and program control is transferred to the identically labeled connector that does have an outlet. It may be noted that connectors do not represent any operation and their use in a flowchart is only for the sake of convenience and clarity.

SAMPLE FLOWCHARTS

A flowchart should be drawn using the symbols mentioned above. To describe an algorithm in the form of a flowchart is not very difficult. What is required is some common sense and a little practice. The art of flowcharting is introduced below with the help of some simple examples.

Example 11.2. A student appears in an examination that consists of total 10 subjects, each subject having maximum marks of 100. The roll number of the student, his name, and the marks obtained by him in various subjects is supplied as input data. Such a collection of related data items that is treated as a unit is known as a *record*. Draw a flowchart for the algorithm to calculate the percentage marks obtained by the student in this examination and then to print it along with his roll number and name.

Solution :

The flowchart for the algorithm of this problem is shown in Figure 11.3. The first symbol is a terminal labeled START. It shows that this is the starting point or beginning of our flowchart logic. It does not mean that the computer is to be turned on or that anyone is to press a start button. The second symbol is an I/O symbol that is labeled specifically to show that this step is READ INPUT DATA. This step will input the roll number, name, and the marks obtained by the student from an input device into the main storage of the computer system. The third symbol is a processing symbol which is suitably labeled to indicate that at this step, the computer will add the marks obtained by the student in various subjects and then store the sum in a memory location which has been given the name TOTAL. The fourth symbol is again a processing symbol. The label inside it clearly indicates that the percentage marks obtained by the student is calculated at this stage by dividing TOTAL by 10 and the result is stored in a memory location which has been given the name PERCENTAGE. The fifth symbol is an I/O symbol and is labeled WRITE OUTPUT DATA. This logical

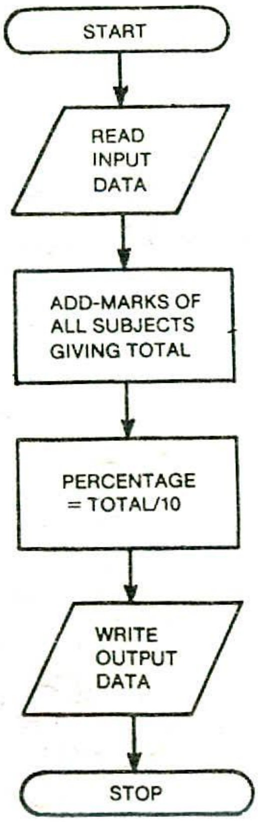


Figure 11.3. Flowchart for Example 11.2.

step in the flowchart indicates that the data desired as output will be outputted on the line printer. Note that details such as the roll number, name, and the marks or percentage being inputted or outputted or the specific positions being used are not a part of the logical steps of inputting or outputting. This information already appears in the system design documents and will be included in the computer program as input and output descriptions. The sixth symbol is a terminal symbol labeled STOP. This symbol indicates the conclusion of our logic - that is, the conclusion of the computer program. The various symbols used in the flowchart are connected by directed flowlines to indicate the sequence in which the instructions are to be executed.

The logic depicted in Figure 11.3 therefore, will read the students's record, calculate the percentage marks obtained by him, print one line, and then stop. One would certainly not like to use a computer to solve a trivial problem such as this. However, if we have to compute the percentage marks obtained by several students in the same examination then we may like to take the help of a computer. The next example illustrates how to do this.

Example 11.3. 50 students of a class appear in the examination of Example 11.2. Draw a flowchart for the algorithm to calculate and print the percentage marks obtained by each student along with his roll number and name.

Solution :

Since all the students have appeared in the same examination, so the process of calculation and printing the percentage marks obtained by each student will basically remain the same. The same process of reading the input data, adding the marks of all subjects, calculating the percentage, and then writing the output data has to be repeated for all the 50 students. Hence, an easy solution that comes to ones mind for this problem is to repeat the intermediate four symbols of Figure 11.3 fifty times. However if that is done, a total of 202 (50 x 4 + 2) flowchart symbols will have to be drawn. Obviously this will be a very time consuming and tedious job and hence is not desirable. We will now see how to solve this problem in a simpler way.

In a situation where the same logical steps can be repeated, the flowline symbols are used in a flowchart to indicate the repetitive nature of the logic in the form of a *process loop*. Figure 11.4 illustrates a flowchart with a process loop. Note the arrowhead on the flowline that forms the loop. It points upward indicating that as soon as the WRITE operation is over the control will flow back to the READ operation. Thus, the process loop of Figure 11.4

solves the problem of an exceedingly long flowchart by reusing the same logical steps over and over again. However, the flowchart of Figure 11.4 is incomplete because the process loop has introduced a new problem. The process loop shown does not has a logical ending. It will continue to attempt to repeat those four steps until someone manually cancels the job. This is an example of an *infinite loop* and hence the flowchart of Figure 11.4 does not represent an algorithm because an algorithm must terminate. So we have to find out a way of terminating the algorithm. This is done by the use of a decision symbol.

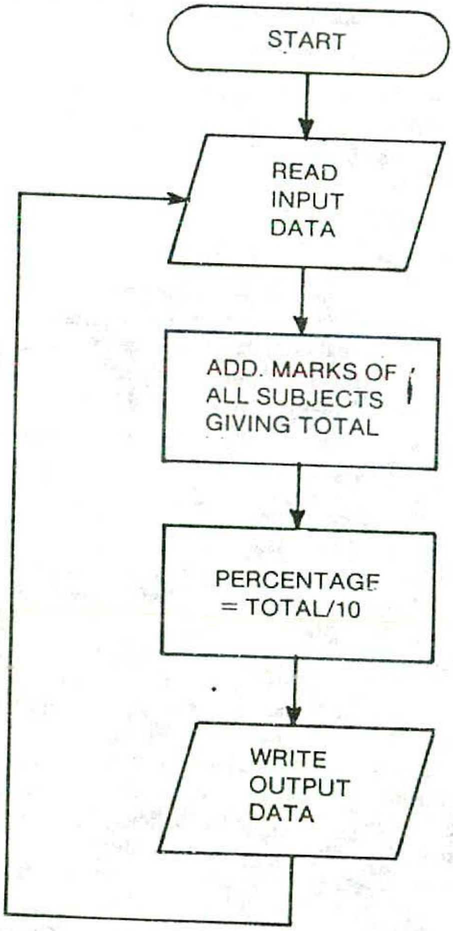


Figure 11.4. Flowchart for the solution of Example 11.3 with an infinite (endless) process loop.

Figure 11.5 shows a flowchart which uses a decision step to terminate the algorithm. In this flowchart, another variable COUNT has been introduced which is initialized to zero outside the process loop and is incremented

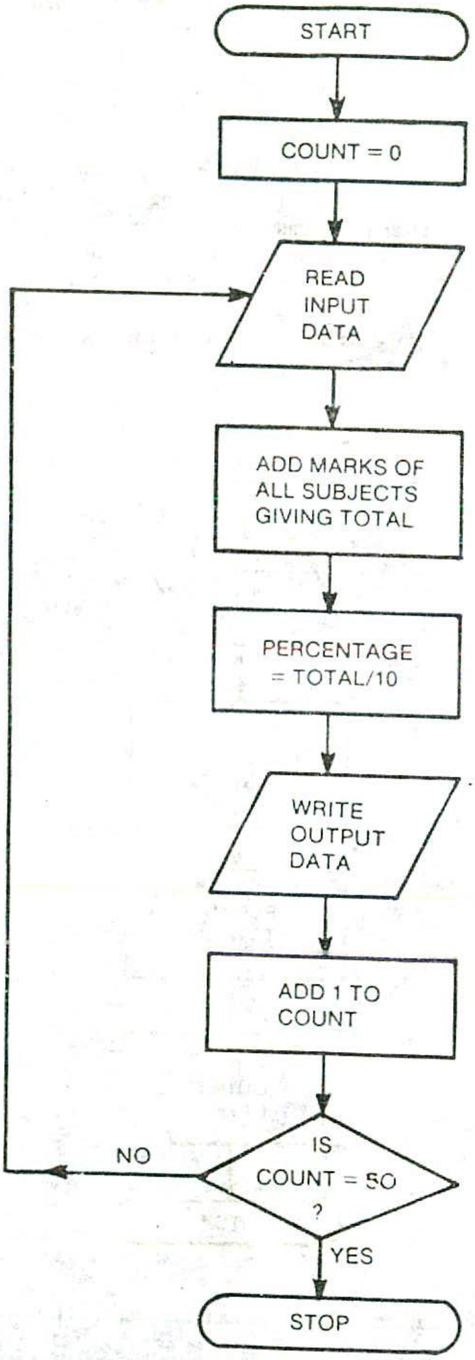


Figure 11.5. Flowchart for the solution of Example 11.3.

by 1 after processing the data for each student. Thus, the value of COUNT will always be equal to the number of students whose data has already been processed. At the decision step, the value of COUNT is compared with 50 which is the total number of students who have appeared for the examination. The steps within the process loop are repeated until the value of COUNT becomes equal to 50. As soon as the value of COUNT becomes equal to 50, the instruction at the decision step causes the control to flow out of the loop and the processing stops because a terminal symbol labeled STOP is encountered. Thus, the flowchart of Figure 11.5 is a complete and correct solution to the problem of Example 11.3.

Although the flowchart of Figure 11.5 is a correct solution to the given problem, it suffers from two major drawbacks. The first drawback is that in order to make the decision at the decision step, one must know the exact number of students who appeared in the examination. Suppose the examination of Example 11.2 is a university examination in which the total number of students who appeared for the examination is too big (say more than ten thousand). In such a situation, the counting of the total number of input records (data for each student) becomes a tedious job. Even then, if we do stick on counting the input records manually and supply the number of records to be compared against COUNT in the decision box and suppose while counting we make some error then the logic will not work. If the supplied number is less than the actual number of input records, then the computer will not process the data for last few students. And if the supplied number is more than the actual number of input records, the computer will try to read more records than what is supplied which will cause an error in the logic.

The second drawback is that the flowchart of Figure 11.5 is not a generalized solution to the given problem. Suppose the examination is conducted every year and so we will like to use the same program every year to process the students' data. However, the number of students appearing in the examination may not remain the same every year. This year it may be 50, but next year it can be 55 or 60 or anything. So if the computer program to solve this problem was based on the flowchart of Figure 11.5, the statement concerned with the decision step in that program will have to be changed again and again to supply the exact number of students. This is not a good programming practice. A good program should be general in nature. For example, in this case we should write a program that need not be modified every time even if the total number of students changes.

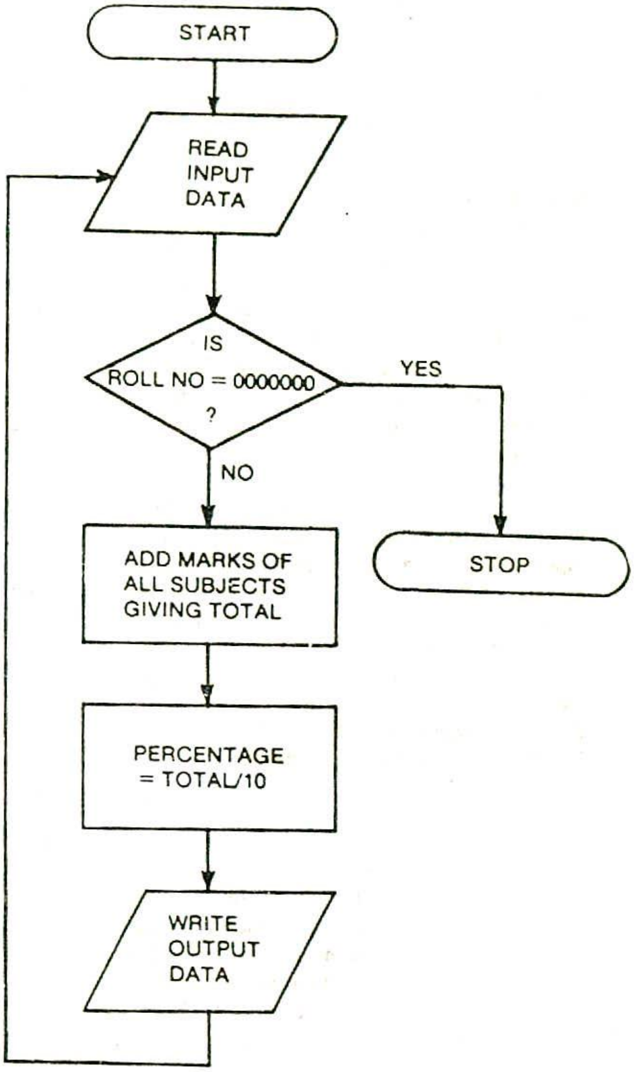


Figure 11.6. Generalised flowchart for the solution of Example 11.3 using the concept of trailer record. Here the process loop is terminated by detecting a special non-data record.

The flowchart of Figure 11.5 suffers from these drawbacks because in this flowchart the process loop is being controlled by counting. When the loop has been executed 50 times, the decision will cause execution to proceed to the STOP, thus terminating processing. (The reader should carefully step through the logic to ensure that the loop will be executed exactly 50 times and not 49 or

51.) To avoid these drawbacks, another method can be adopted to control the process loop. In this method, the end of input data is marked by a *trailer record*, that is, the last data record in the input is followed by a record whose sole purpose is to indicate that the end of the input data has been reached. Suppose the first 7 characters of the input record of a student represents his roll number (ROLLNO). Since 0000000 is never used as a roll number, a value of 0000000 as the first 7 characters can be used to represent the trailer record. As each input record is processed, the ROLLNO can be compared with 0000000 to determine if processing is complete. The logic of this process is illustrated in the flowchart of Figure 11.6. It is important to recognize that the programmer would have to include instructions in the program which specifically compare the ROLLNO to 0000000 in order to determine whether to continue or to terminate.

The concept of a trailer record centers around the notion of selecting a field (a particular item of data) in the input record which will be used to indicate the end of data and then selecting a *trailer value* also known as *sentinel value* which will never occur as normal data value for that field. The roll number of 0000000 is a good example. It may also be noted that when a trailer record is used to mark the end of input data, the decision box used for checking the trailer value should almost always be flowcharted immediately after the input symbol.

Example 11.4. For the examination of Example 11.2, we want to make a list of only those students who have passed (obtained 30% or more marks) in the examination. In the end, we also want to print out the total number of students who have passed. Assuming that the input data of all the students is terminated by a trailer record that has sentinel value of 9999999 for ROLLNO, draw a flowchart for the algorithm to do the above job.

Solution :

The flowchart in Figure 11.7 is a solution to this problem. There are two decision symbols in this flowchart. The first decision symbol checks for a trailer record by comparing ROLLNO against the value 9999999 to determine if processing is complete. The second decision symbol is used to check whether the student has passed or failed by comparing the percentage marks obtained by him against 30. If the student's PERCENTAGE is equal to or more than 30 then he has passed otherwise he has failed. Note from the flowchart that the operation WRITE OUTPUT DATA is performed only if the student has passed. If he has failed, we directly perform the operation READ INPUT DATA without performing the WRITE operation. This ensures that the output list provided by the computer will contain the details of only those students who have passed in the examination.

Another point to be noted in this flowchart is the use of variable COUNT. This variable has been initialized to zero in the beginning and is incremented by 1 every time the operation WRITE OUTPUT DATA is performed. But we have seen that the operation WRITE OUTPUT DATA is performed only for the students who have passed. Hence, the variable COUNT will be incremented by 1 only in case of students who have passed. Thus, the value of COUNT will always be equal to the number of students whose data has already been processed and who have been identified as passed. Finally, when the trailer record is detected, the operation WRITE COUNT will print out the final value of COUNT that will be equal to the total number of students who have passed the examination.

Example 11.5. Suppose the input data of each student for the examination of Example 11.2 also contains information regarding the sex of the candidate in the field named SEXCODE that can have values M (for male) or F (for female). We want to make a list of only those female students who have passed in second division (obtained 45% or more but less than 60% marks). In the end we also want to print out the total number of such students. Assuming that the input data of all the students is terminated by a trailer record that has a sentinel value of Z for SEXCODE draw a flowchart for the algorithm to do the above job.

Solution :

The flowchart in Figure 11.8 is a solution to this problem. There are four decision symbols in this flowchart. The first decision symbol checks for a trailer record by comparing SEXCODE against the value Z to determine if processing is complete. The second decision symbol is used to check whether the candidate is female or not by comparing the SEXCODE of that candidate against F. Note that if the SEXCODE is not F, that is, the candidate is not a female, we do not process the data of that student and return back to perform the operation of reading input data. This step ensures that the data of only female students will be taken for further processing.

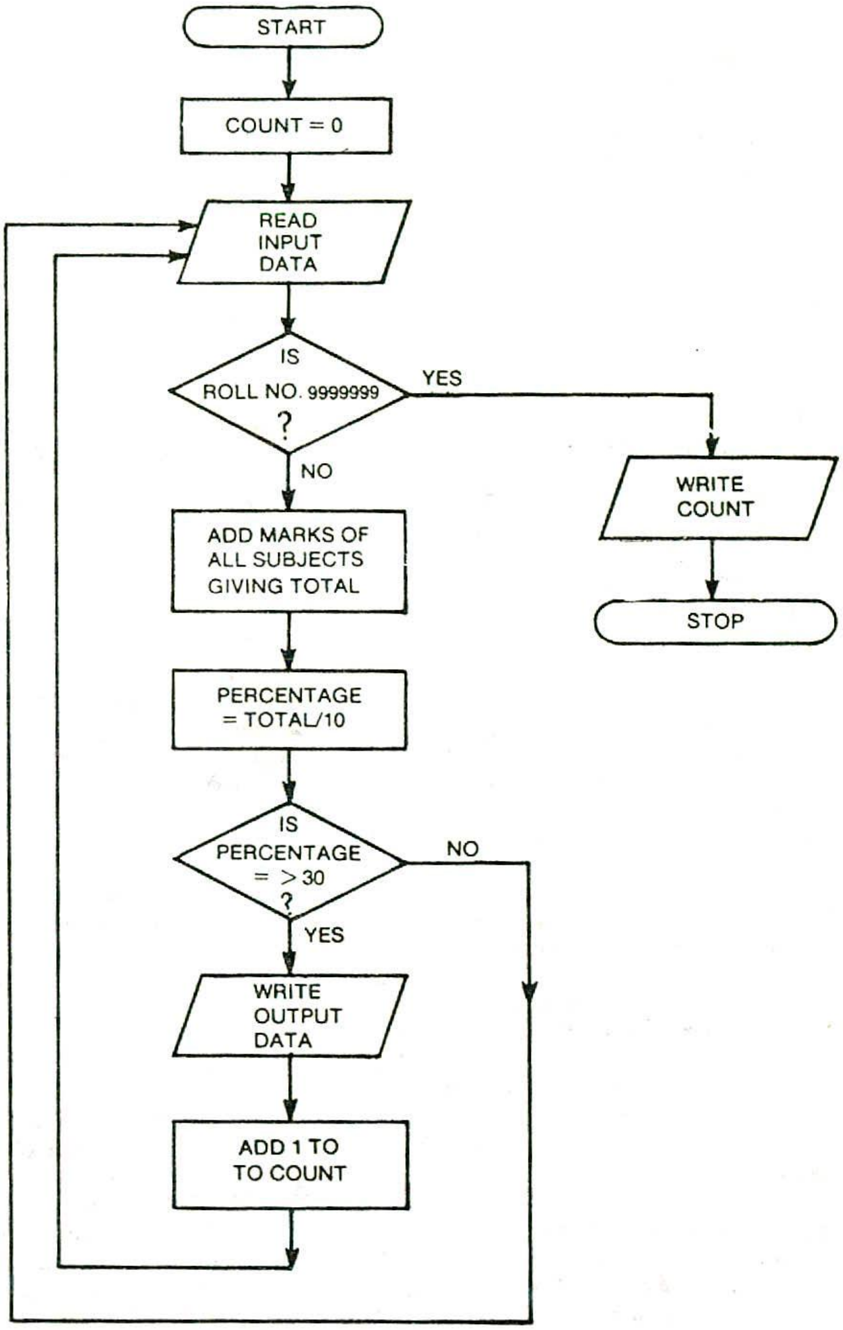


Figure 11.7. Flowchart for the solution of Example 11.4.

The last two decision symbols in the flowchart are used to check whether the student has passed in second division or not. The first of these decisions is used to ensure that the student has scored 45% or more marks. If she has scored less than 45% then it means that she is not a second divisioner and hence without making any further check we return back to the operation of reading input data. In case the student has scored 45% or more marks then we go one step further in the logic and by using the fourth decision in the flowchart we check whether her marks are less than 60% by comparing her PERCENTAGE against 60. If the condition at this step turns out to be false then it means that the student has scored 60% or more marks and hence she is a first divisioner and not a second divisioner. So once again we return back to read a new data without writing any output data. If the condition inside the fourth decision symbol turns out to be true then the female candidate can be classified to be a second divisioner. Hence in this case only we perform the operation WRITE OUTPUT DATA and subsequently increment the value of COUNT by 1.

It is suggested that the reader should go through the logic of this flowchart again and again until he/she is convinced that the output list provided by the computer will contain the details of only those female students who have passed in second division. The reader should also get convinced that finally when the trailer record is detected, the operation WRITE COUNT will print out the value of COUNT that will be equal to the total number of female students who have passed in second division. This flowchart is an example of a multiple-decision chart.

The flowchart of Figure 11.8 has been reproduced in Figure 11.9 to illustrate the use of connectors. There are four exit connectors having the label 1 all of which indicate a flow of control to the flowchart symbol having an entry connector labeled 1. This symbol is the input symbol in the flowchart. Similarly the exit connector having a label 2 indicates a flow of control to the entry connector labeled 2. The reader should compare the flowcharts of Figure 11.8 and Figure 11.9 and should get convinced that both the flowcharts represent the same logic.

A flowchart may seem simple to prepare, but you will find that much practice is needed in order to think through a problem in discrete, logical steps, to assume nothing and to forget nothing. Moreover, not everyone will tackle a problem in exactly the same way and, in consequence, several different flowcharts could be drafted for the same problem. It may also be noted that a completed flowchart is not a complete computer program. It is only an aid to programming. For a given problem, it defines the procedure and the logic involved. From the examples that have been discussed above, we are in a better position to understand what this 'logic' means.

LEVELS OF FLOWCHARTS

There are no set standards on the amount of detail that should be provided in a flowchart. A flowchart that outlines the main segments of a program or that shows less detail is a *macroflowchart*. On the other hand, a flowchart with more detail is a *microflowchart*, or detailed flowchart.

For example, let us consider the examination problem that we have already discussed. In all the flowcharts of the examination problem, there is a processing box having the instruction "ADD MARKS OF ALL SUBJECTS GIVING TOTAL". In order to display how the value of TOTAL is computed, a detailed flowchart can be drawn as shown in Figure 11.10. In a similar manner, the I/0 boxes for the READ and WRITE operations can also be converted to a detailed flowchart.

FLOWCHARTING RULES

While programmers have a good deal of freedom in creating flowcharts, there are a number of general rules and guidelines recommended by the American National Standards Institute (ANSI) to help standardize the flowcharting process. Various computer manufacturers and data processing departments usually have similar flowcharting standards. Some of these rules and guidelines are as follows :

1. First chart the main line of logic, then incorporate detail.

2. Maintain a consistent level of detail for a given flowchart.

3. Do not chart every detail or the flowchart will only be a graphic representation, step by step, of the program. A reader who is interested in greater details can refer to the program itself.

4. Words in the flowchart symbols should be common statements and easy to understand. It is recommended to use descriptive titles written in designer's own language rather than in machine oriented language.

5. Be consistent in using names and variables in the flowchart.

6. Go from left to right and top to bottom in constructing flowcharts.

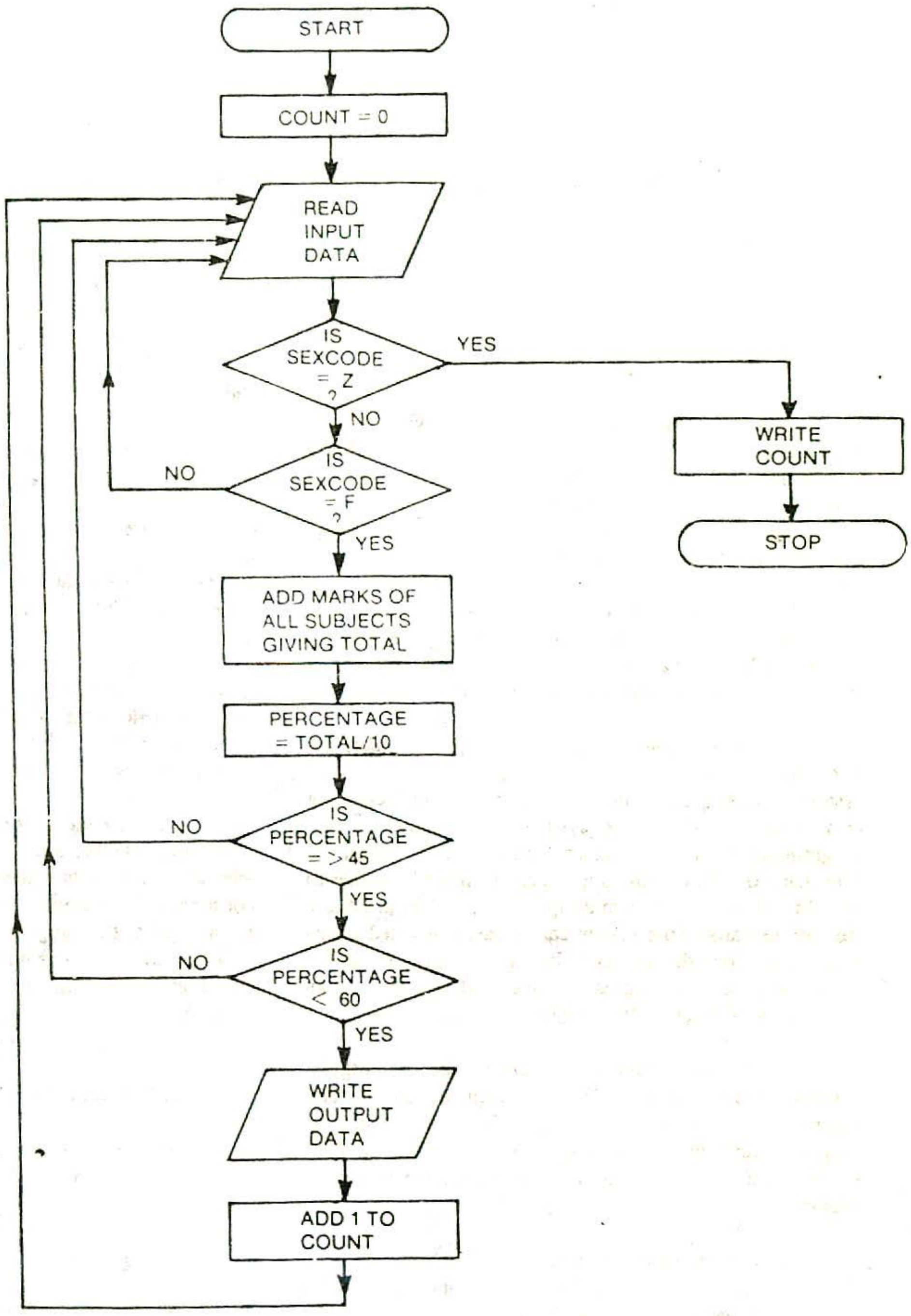


Figure 11.8. Flowchart for the solution of Example 11.5.

7. Keep the flowchart as simple as possible. The crossing of flowlines should be avoided as far as practicable.

8. If a new flowcharting page is needed, it is recommended that the flowchart be broken at an input or output point. Moreover properly labeled connectors should be used to link the portions of the flowchart on different pages.

ADVANTAGES OF FLOWCHARTS

The following benefits may be obtained when flowcharts are used for the purpose of program planning.

1. *Better Communication* : The old saying that "a picture is worth a thousand words" holds true for flowcharts also. Since, a flowchart is a pictorial representation of a program, it is easier for a programmer to explain the logic of a program to some other programmer or to his boss through a flowchart rather than the program itself.

2. *Effective Analysis* : A macro flowchart that charts the main line of logic of a software system becomes a system model that can be broken down into detailed parts for study and further analysis of the system.

3. *Effective synthesis* : A group of programmers are normally associated with the design of big software systems. Each programmer is responsible for designing only a part of the entire system. So initially, if each programmer draws a flowchart for his part of design, the flowcharts of all the programmers can be placed together to visualize the overall system design. Any problem in linking the various parts of the system can be easily detected at this stage and the design can be accordingly modified. Flowcharts are thus used as working models in the design of new programs and software systems.

4. *Proper Program Documentation* : Program documentation involves collecting, organizing, storing, and otherwise maintaining a complete historical record of programs and the other documents associated with a system. Good documentation is needed for the following reasons :

(a) Documented knowledge belongs to an organization and does not disappear with the departure (resignation/retirement) of a programmer.

(b) If projects are postponed, documented work will not have to be duplicated.

(c) If programs are modified in the future, the programmer will have a more understandable record of what was originally done.

From what we have seen of the nature of flowcharts, it is obvious that they can provide valuable documentation support.

5. *Efficient Coding* : Once a flowchart is ready, programmers find it very easy to write the concerned program because the flowchart acts as a roadmap for them. It guides them to go from the starting point of the program to the final point ensuring that no steps are omitted. The ultimate result is an error free program developed at a faster rate.

6. *Systematic Debugging* : Even after taking full care in program design, some errors may remain in the program because the designer might have never thought about a particular case. These errors are detected only when we start executing the program on a computer. Such type of program errors are called *bugs* and the process of removing these errors is known as *debugging*.

Once a bug is detected, it is easier to find out the reason for that bug by going through the logic of the program in flowchart form. A flowchart is very helpful in detecting, locating, and removing mistakes (bugs) in a program in a systematic manner.

7. *Systematic Testing* : *Testing* is the process of confirming whether a program will successfully do all the jobs for which it has been designed under the specified constraints. For testing a program, different set of data is fed as input to that program to test the different paths in the program logic. For example, to test the complete logic of the program for Example 11.5, the following set of data is necessary :

(a) Data for a male candidate.

(b) Data for a female candidate who has scored less then 45%.

(c) Data for a female candidate who has exactly scored 45%.

(d) Data for a female candidate who has scored more than 45% but less than 60%.

(e) Data for a female candidate who has exactly scored 60%.

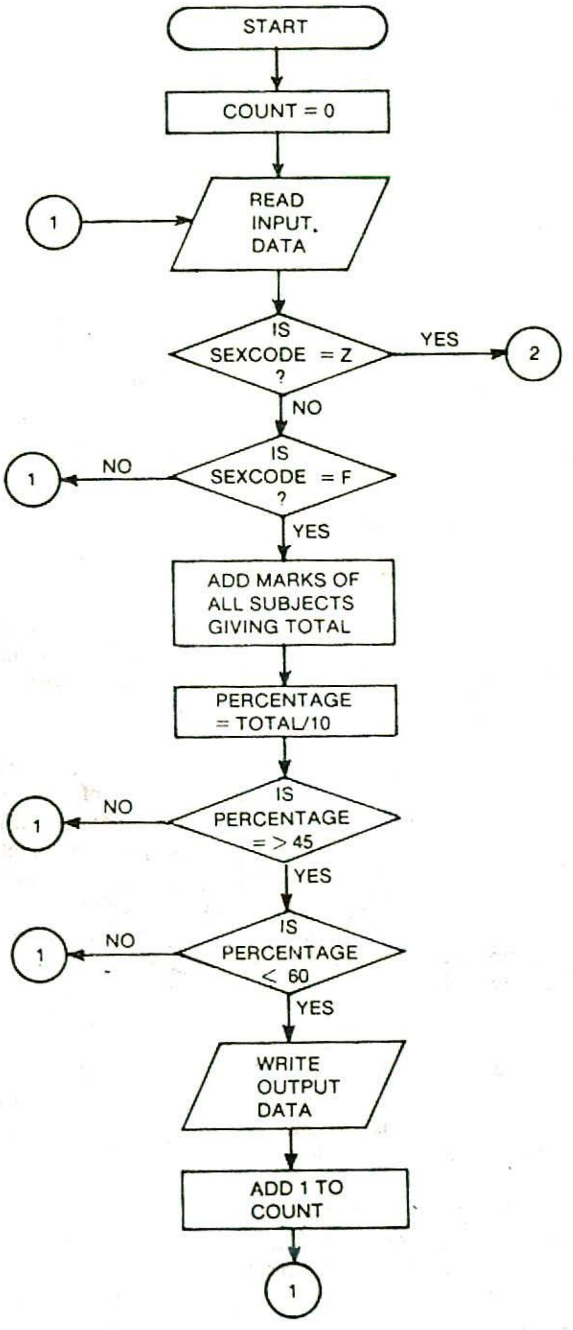


Figure 11.9. Flowchart of Figure 11.8 redrawn to illustrate the use of connectors.

(f) Data for a female candidate who has scored more than 60%.

(g) And obviously in the end the trailer data having sentinel value..

A flowchart proves to be very helpful in designing the test data for systematic testing of programs.

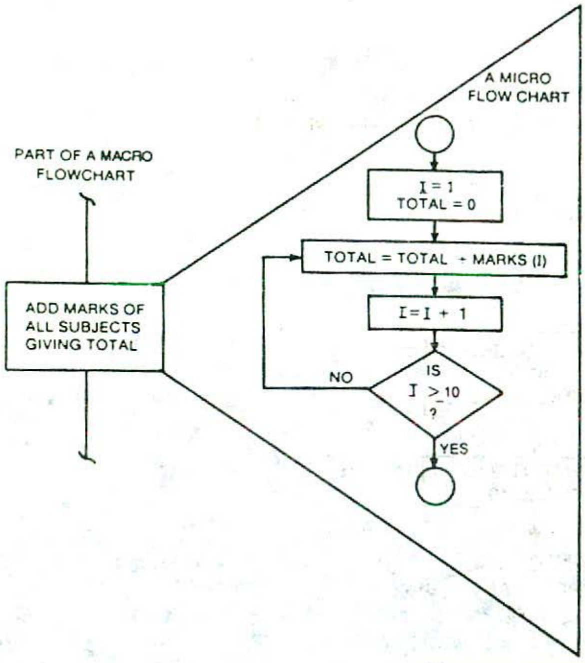

A MICRO FLOW CHART
PART OF A MACRO FLOWCHART
ADD MARKS OF ALL SUBJECTS GIVING TOTAL
I = 1
TOTAL = 0
TOTAL = TOTAL + MARKS (I)
I = I + 1
IS I ≥ 10 ?
NO
YES

Figure 11.10. Detailed flowchart of add marks of all subjects giving TOTAL

LIMITATIONS OF FLOWCHARTS

In spite of their many obvious advantages, flowcharts have several limitations which are as follows :

1. Flowcharts are very time consuming and laborious to draw with proper symbols and spacing, especially for large complex programs. In this chapter, you have seen examples of small program flowcharts developed for relatively small programs. You can very well imagine how difficult it would be to develop a detailed program flowchart for a program containing over 50,000 statements.

2. Owing to the symbol-string nature of flowcharting, any changes or modifications in the program logic will usually require a completely new flowchart. Redrawing a flowchart is again so tedious that many companies either do not redo them or produce the flowchart by using a computer program to draw it. There are several computer programs available that will read the program's instructions and draw a flowchart of its logic, but these programs are fairly expensive to acquire and use a lot of computer time.

3. There are no standards determining the amount of detail that should be included in a flowchart.

Because of such limitations, many organizations are now reducing the amount of flowcharting used. In its place, they are using alternative tools for program analysis, two of which are briefly discussed below.

DECISION TABLES

Decision tables are used to define clearly and concisely the word statement of a problem in a tabular form. They can prove to be a powerful tool for defining complex program logic. As the name implies, decision tables reveal what decisions or actions the computer is to take as a result of the input data. When the computer has to make a large number of decisions or if there are a large number of different branches within a program, decision tables are particularly useful. In these cases, decision tables are preferred to flowcharts.

The steps to be followed for constructing a decision table are as follows :

1. Properly define the problem that has to be solved by computer.

2. List out all the conditions to be tested in the problem.

3. List out the corresponding actions that should be taken with each combination of conditions.

4. Form a decision table using the two lists.

Most decision tables have six parts. The basic format of a decision table is shown in Figure 11.11. The first part of the decision table contains the name and/or number of the table. For some larger applications, two or more decision tables may be used in the analysis part of program development. The second part of the table, known as condition stub, descirbes the conditions that could exist in the program logic. Thus, the contents of condition stub correspond to the conditions contained in the decision symbols of a flowchart. Action stub, the third part of the decision table, contains the action statements. These statements correspond to the statements located in nondecision symbols of a flowchart. While the condition statements reveal the possible states of the input data, the action statements describe the possible actions of the computer system. The right hand side of the decision table contains the rule numbers (part 4), the actual conditions (part 5), and the actions taken by the computer (part 6). The condition entries correspond to the paths leading out from decision symbols.

TABLE HEADING	DECISION RULES
CONDITION STUB	CONDITION ENTRIES
ACTION STUB	ACTION ENTRIES

Figure 11.11. Format of a decision table.

A decision table for the problem of Example 11.5 that was charted in Figure 11.8 is shown in Figure 11.12 In this table, each rule number is a given condition followed by a specific action to be taken by the computer. The six rule numbers, the six actual conditions, and the associated actions taken by the computer system are discussed below.

Rule 1 : The student is a female, and the percentage marks obtained is 45% or more, and the percentage marks obtained is less than 60%. The computer should write the output data, add 1 to COUNT, and then read the next student's record. It is a case of a female student who has passed in second division.

Rule 2 : The student is female, and the percentage marks obtained is 45% or more, and the percentage marks obtained is not less than 60%. The computer should directly read the next student's record without performing any other operation. It is the case of a female student who has passed in first division (60% or more marks).

Rule 3 : The student is a female and the percentage marks obtained is not 45% or more. A dash (-) in this column against the last condition means that this condition is irrelevant in arriving at the action. In fact if PERCENTAGE is not equal to or greater then 45 then it has to be less than 60. PERCENTAGE greater than or equal to 60 is logically impossible. Such conditions in the decision table which are irrelevant in arriving at an action are known as don't care conditions and are denoted by a dash. For this rule, the computer should directly read the next student's record without performing any other operation. It is the case of a female student who has scored less than 45% and hence she is not a second divisioner.

Rule 4 : The student is a female, and the other conditions are don't care conditions. The computer should calculate PERCENTAGE after adding marks of all subjects and then proceed to test further conditions. It is the case of a female student whose PERCENTAGE has yet to be calculated.

Rule 5 : The student is a male, and the other conditions are don't care conditions. The computer should directly read the next student's record without performing any other operation. It is the case of a male student.

Rule 6 : In all the previous five rules, it was ascertained that the current record is not a trailer record because all these rules have a value N (No) for the condition SEXCODE = Z. In this rule, the SEXCODE is equal to Z which indicates a trailer record and hence the computer should write the value of COUNT and then STOP. The other conditions in this rule are don't care conditions. It is the case of a trailer record.

In the decision table, "Y" means yes, "N" means no, "-" means don't care, and "X" means the computer should take this action.

Examination problem of Example 11-5	Decision rule number.					
	1	2	3	4	5	6
Condition Statements						
SEXCODE = Z	N	N	N	N	N	Y
SEXCODE = F	Y	Y	Y	Y	N	—
PERCENTAGE = > 45	Y	Y	N	—	—	—
PERCENTAGE < 60	Y	N	—	—	—	—
Actions Taken						
Calculate PERCENTAGE				X		
Write output data	X					
Add 1 to COUNT	X					
Read next student record	X	X	X		X	
Write COUNT						X
STOP						X

Figure 11.12. Decision table for the examination problem of Example 11.5.

ADVANTAGES AND LIMITATIONS OF DECISION TABLES

Decision tables are normally used in place of flowcharts because of the following reasons:

1. They are easier to draw and change than charts.

2. They provide more compact documentation. A small table can replace several pages of charts.

3. It is also easier to follow a particular path down one column than through several flowchart pages.

However, decision tables are not very popular and are not so widely used as charts because :

1. Flowcharts are better able to express the total sequence of events needed to solve a problem.

2. Flowcharts are more familiar to, and are preferred by, many programmers and beginners

PSEUDOCODE

Pseudocode is another programming analysis tool that is used for planning program logic. "Pseudo" means imitation or false and "Code" refers to the instructions written in a programming language. Pseudocode, therefore, is an imitation of actual computer instructions. These pseudoinstructions are phrases written in ordinary natural language (e.g., English, French, German, etc.). Instead of using symbols to describe the logic steps of a program, as in flowcharting, pseudocode uses a structure that resembles computer instructions. Because it emphasises the design of the program, pseudocode is also called Program Design Language (PDL).

Pseudocode is made up of the following basic logic structures that have been proved to be sufficient for writing any computer program :

1. Sequence

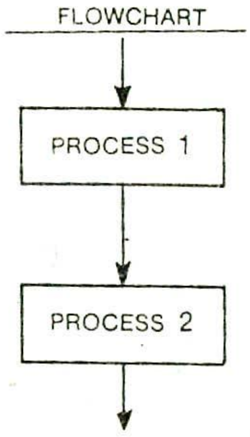
2. Selection (IF. . .THEN. . .ELSE
   or IF. . .THEN)

3. Iteration (DO...WHILE or REPEAT...UNTIL)

Sequence logic is used for performing instructions one after another in sequence. Thus, for sequence logic, pseudocode instructions are written in the order, or sequence, in which they are to be performed. The logic flow of pseudocode is from the top to the bottom. Figure 11.13 shows an example of sequence logic structure.

Selection logic, also known as decision logic, is used for making decisions. It is used for selecting the proper path out of the two or more alternative paths in the program logic. Selection logic is depicted as either an IF...THEN or an IF...THEN...ELSE structure. The flowcharts of Figure 11.14 and 11.15 illustrate the logic of these structures.

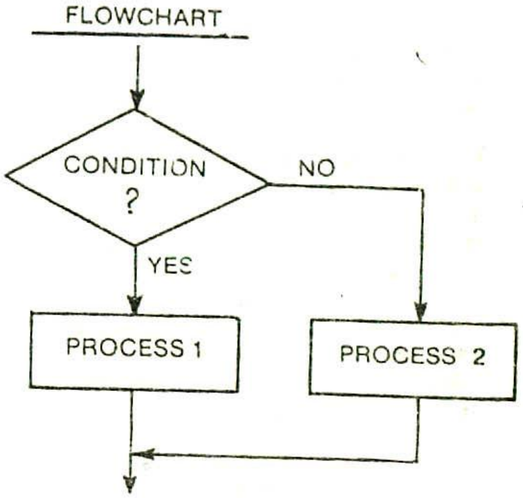
Their corresponding pseudocode is also given in these figures. The IF...THEN...ELSE construct says that if the condition is true, then do process 1, else (if the condition is not true) do process 2. Thus, in this case either process 1 or process 2 will be executed depending on whether the specified condition is true or false. However, if we do not want to choose between two processes and if we simply want to decide if a process is to be performed or not, then the IF...THEN structure is used. The IF...THEN structure says that if the condition is true, then do process 1 and if it is not true then skip over process 1. In both the structures, process 1 and process 2 can actually be one or more processes. They are not limited to a single process. ENDIF is used to indicate the end of the decision structures.



PSEUDOCODE

```
•
•
•
Process 1
Process 2
•
•
•
```

Figure 11.13. Flowchart and pseudocode for sequence structure.



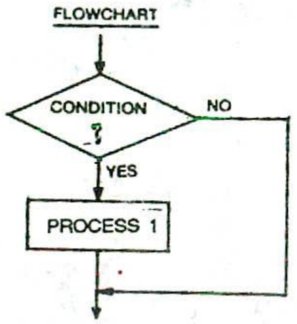
PSEUDOCODE

```
•
•
•
IF Condition
   THEN  process 1
ELSE
              process 2
END IF
•
•
•
```

Figure 11.14. Flowchart and pseudocode for IF...THEN...ELSE selection structure.

Iteration logic is used to produce loops when one or more instructions may be executed several times depending on some condition. It uses two structures called the DO...WHILE and the REPEAT...UNTIL. They are illustrated by flowcharts in Figure 11.16 and Figure 11.17 respectively. Their corresponding pseudocodes are also given in these figures. Both DO...WHILE and REPEAT...UNTIL are used for looping. The differences are that in the DO...WHILE, the looping will continue as long as the condition is true. The looping stops when the condition is not true. On the other hand, in case of

REPEAT...UNTIL, the looping continues until the condition becomes true. That is, the execution of the statements within the loop is repeated as long as the condition is not true. In both the DO...WHILE and REPEAT...UNTIL, the loop must contain a statement that will change the condition that controls the loop. If it doesn't, the looping will continue without end which is the case of an infinite loop. Remember that no program should contain an infinite loop. Also note that the condition is tested at the top of the loop in the DO...WHILE and at the bottom of the loop in the REPEAT...UNTIL. ENDDO marks the end of a DO...WHILE structure and UNTIL followed by some condition marks the end of the REPEAT...UNTIL structure.

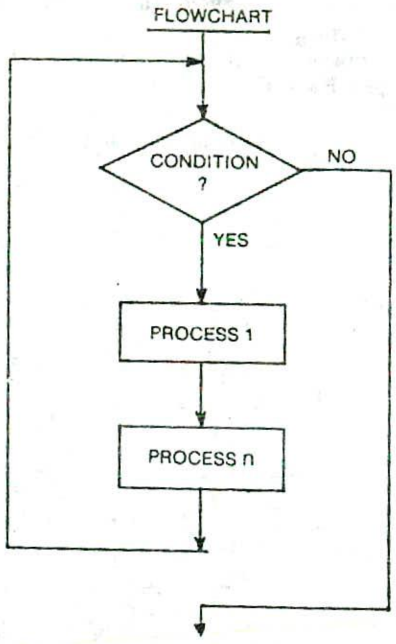


PSEUDOCODE

```
•
•
•
IF CONDITION
    THEN PROCESS 1
END IF
•
•
•
```

Figure 11.15. Flowchart and pseudocode for IF...THEN selection structure.

The pseudocode version of the logic of the problem of Example 11.5 that was charted in Figure 11.8 is shown in Figure 11.18. In the pseudocode example, the first line initializes the value of COUNT to zero and the second line reads the input data of the first student. The third line is the beginning of a loop using the DO...WHILE structure. It indicates that the loop will continue so long as the value of SEXCODE is not equal to Z - that is, as long as the trailer record is not found. In this example, a series of decisions followed by an instruction to read next student's record are included within the loop. The first statement within the DO...WHILE loop asks, "Is the SEXCODE equal to F?" If the answer is yes, PERCENTAGE is calculated and again the third statement within the loop asks, "Is PERCENTAGE equal to or greater than 45?" If it is, then "Is PERCENTAGE less than 60?" This is a series of three IF...THEN decision structures. Each one ends with an ENDIF vertically aligned below the appropriate IF.

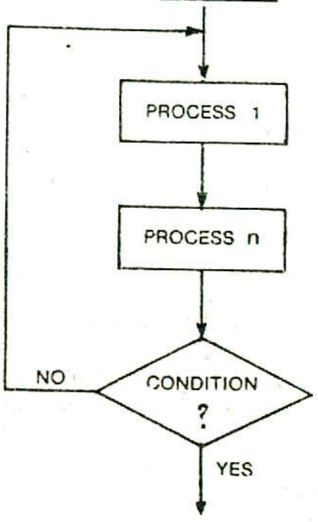


PSEUDOCODE

```
•
•
•
DO WHILE CONDITION
    PROCESS 1
    PROCESS n
ENDDO
•
•
•
```

Figure 11.16. Flowchart and pseudocode for DO...WHILE iteration structure.

FLOWCHART



PSEUDOCODE

```
•
•
•
REPEAT
     PROCESS 1
     PROCESS 2
UNTIL CONDITION
•
•
•
```

Figure 11.17. Flowchart and pseudocode for REPEAT...UNTIL iteration structure.

The two instructions - write output data, and add 1 to COUNT are performed only if all three conditions (that of SEXCODE being F, PERCENTAGE being equal to or more than 45, and PERCENTAGE being less than 60) are found to be true (answered yes). If any of the three conditions is not true, the logic path goes to the statement that reads next student's record. After the last student's record is processed, the trailer record for which the value of SEXCODE is Z is encountered. This will cause the DO...WHILE loop to stop, because the condition (SEXCODE not equal to Z) is no longer true. When the DO...WHILE condition is no longer true, the next logical step will be the instruction following the ENDDO. At this stage, the value of COUNT will be printed (write COUNT) and finally the program execution will stop (STOP).

```
Set COUNT to zero
Read first student record
DO WHILE SEXCODE is not equal to Z
  IF SEXCODE = F THEN
    calculate PERCENTAGE
    IF PERCENTAGE => 45 THEN
      IF PERCENTAGE < 60 THEN
        Write output data
        Add 1 to COUNT
      ENDIF
    ENDIF
  ENDIF
  Read next student record
ENDDO
Write COUNT
STOP
```

Figure 11.18. Pseudocode for the examination problem of Example 11.5.

One important feature of pseudocode as a programming tool is the use of indentation. Each statement within the DO...WHILE loop is indented, to show that it is part of the loop. Similarly, the statements within each IF...THEN structure is indented properly to clearly specify the statements which are part of each structure. The use of indentation in pseudocode is the same technique used with the various programming languages. Its sole purpose is to clarify the logical structure of the program. With this technique, we can tell at a glance which statements make up each of the logic structure of the total program logic. To fully appreciate this factor, the reader should compare the equivalent non-indented pseudocode of Figure 11.19 to that of Figure 11.18. The difference in clarity would be far greater if this were a longer pseudocode covering, for instance, one or more pages.

ADVANTAGES AND LIMITATIONS OF PSEUDOCODE

Pseudocode has three main advantages:

1. Converting a pseudocode to a programming language is much more easier as compared to converting a flowchart or a decision table.

2. As compared to a flowchart, it is easier to modify the pseudocode of a program logic when program modifications are necessary.

```
SET COUNT to Zero
Read first student record
DO WHILE SEXCODE is not equal to Z.
IF SEXCODE = F
THEN Calculate PERCENTAGE
IF PERCENTAGE = > 45
THEN IF PERCENTAGE < 60
THEN Write output data
Add 1 to COUNT
ENDIF
ENDIF
ENDIF
Read next student record
ENDDO
Write COUNT
STOP
```

Figure 11.19. Non-indented version of pseudocode of Figure 11.18.

3. Writing of pseudocode involves much less time and effort than drawing an equivalent flowchart. Pseudocode is easier to write than an actual programming language because it has only a few rules to follow, allowing the programmer to concentrate on the logic of the program.

However, pseudocode suffers from the following limitations :

1. In case of pseudocode, a graphic representation of program logic is not available.

2. There are no standard rules to follow in using pseudocode. Different programmers use their own style of writing pseudocode and hence communication problem occurs due to lack of standardization.

3. For a beginner, it is more difficult to follow the logic of or write pseudocode, as compared to flowcharting.

QUESTIONS

1. Why is it advisible to plan the logic of a program before writing it ?

2. What is an algorithm? What are the characteristics necessary for a sequence of instructions to qualify as an algorithm ?

3. What is a flowchart ?

4. How does a flowchart help a programmer in program development ?

5. Can a flowchart be drawn for a task if the person drawing the flowchart cannot perform the task manually ? Discuss.

6. What are the various symbols used in flowcharting ? Give their pictorial representation.

7. Describe the function of the various flowcharting symbols.

8. Why are there standards for the symbols used in drawing flowcharts ?

9. What is a record ? A trailer record ?

10. What is a sentinel value ? Discuss its use.

11. What is a process loop ? An infinite loop ?

12. Why is it necessary to avoid infinite loops in program design ?

13. "A loop consists of a body, a test for exit condition, and a return provision." Discuss this statement.

14. What is a generalized algorithm ? Why should programs be general in nature ?

15. Discuss the difference between loop control by counting and loop control by the use of sentinel value. Which is preferable and why ?

16. How can a counter be used to keep track of the number of times a loop has been executed ?

17. Is it possible to have more than one flowchart for a given problem ? Give reasons for your answer.

18. What is the difference between a macroflowchart and a microflowchart? Illustrate with an example.

19. What are the various guidelines to be followed while drawing a flowchart ?

20. Discuss the various advantages and limitations of flowcharting.

21. W y is proper documentation of a program necessary ?

22. What are program bugs ? What is debugging ?

23. What is meant by testing a program ? How is it done ?

24. What are decision tables? When are they preferred to flowcharts ?

25. What are the various steps to be followed for constructing a decision table ?

26. Draw the basic format of a decision table and discuss the role of each part.

27. What are don't care conditions in decision tables ?

28. List out the various advantages and limitations of decision tables.

29. What is a pseudocode ? Why is it so called ? Give another name for pseudocode.

30. What are the three basic logic structures used in writing pseudocode ? Discuss the use of each.

31. Draw flowcharts for the two different structures used for selection logic.

32. What is the difference between the IF...THEN and the IF...THEN...ELSE structures ?

33. Draw flowcharts for the two different structures used for iteration logic.

34. Both DO...WHILE and REPEAT...UNTIL are used for looping. Discuss the difference between the two structures.

35. What is the purpose of the ENDIF and ENDDO?

36. What is indentation ? Why is it used in writing pseudocodes ?

37. Discuss the advantages and limitations of pseudocode.

38. Three numbers, denoted by the variables A, B, and C are supplied as input data. Draw a flowchart for the logic to pick and print the largest of the three numbers.

39. Draw a flowchart of the logical steps needed to produce a printed listing of all students over the age of 20 in a class. The input records contain the name and age of the students. Assume a sentinel value of 99 for the age field of the trailer record.

40. Draw a flowchart of the logical steps needed to print the name and age of the oldest and the youngest student in a class. The input records contain the name and age of the students. Assume a sentinel value of 99 for the age field of the trailer record.

41. The first 20 records in a data set are to be read and printed. Draw a flowchart for the algorithm to do this job. Make sure that the processing stops after the twentieth record.

42. Input data regarding the information of employees of a company has been supplied. The first field of each input record contains the employee number (EMPNO). Assume that the input data of all the employees is terminated by a trailer record having a trailer value of 99999 for EMPNO. Draw a flowchart for the logic to count and print the total number of input records, that is, the total number of employees.

43. For the employees problem of Question 42, we want to count and print the number of only male employees in the age range of 25 to 30. Assume that the input records contain SEXCODE and AGE fields to provide this information. Draw a flowchart for the algorithm to perform this job.

44. Suppose that a population survey has been carried out in a given city, and that the information received from the survey has been transcribed onto punched cards. Since the cards have 80 colums each, one card contains the name, address, sex, age, profession, etc., of one employee. That is, each card contains one record pertaining to one employee. Our problem is to print the details of all the adults (aged 18 years or more) in the city under survey. Finally, we also want to print the total number of adults. Assume a suitable sentinel value for any field in the trailer record and draw a flowchart for the algorithm to do this task.

45. A set of examination papers which have been graded with scores from 0 to 100 is to be searched to find how many of them are above 90. The total has to be printed. Prepare a flowchart to illustrate this job. Assume a suitable sentinel value for the trailer record.

46. Each paper in a set of examination papers includes a grade of A, B, C, D, or E. A count is to be made of how many papers have grade of A and how many have grade of E. The total count of both types have to be printed at the end. Prepare a flowchart to perform this function. Assume a suitable sentinel value for the trailer record.

47. A shopkeeper wants to have a general program for his personal computer that will prepare bills for each customer as and when he sells goods to them. His idea is that as soon as the customer purchases some goods from his shop, he will supply the description, unit price, and the quantity purchased for each item, as input to the computer. He wants that with this information, the computer should print each item along with its unit price, quantity purchased and the total price. Finally, the computer should also print the total cost of all the items purchased by the customer. Assuming that a sentinel value of zero is used for the quantity purchased field in the trailer record, draw a flowchart for the logic to do this job.

48. An employer plans to pay a bonus to each employee. Those earning Rs.2000 or above are to be paid 10 percent of their salary; those earning less than Rs.2000 are to be paid Rs.200. The input records contain the employee number, name, and salary of the employees. The output to be printed should contain the employee number, name, and the amount of bonus to be paid for each employee. Draw a flowchart for the algorithm to do this job. Assume a suitable sentinel value for any of the fields of the trailer record.

49. Each employee pay record includes the hours worked and the pay rate. The gross pay is to be determined as hours worked times pay rate and is to be printed for each employee. For all hours worked in excess of 40, the overtime rate, which is 1.5 times the regular rate, is to be paid. Draw a flowchart to illustrate the problem logic. Assume a suitable sentinel value for any of the input fields of the trailer record.

50. A data file contains a set of examination scores and is followed by a trailer record with a value of -1. Draw a flowchart for the logic to calculate and print the average of the scores.

51. The data file of Question 50 is expanded to include several sets of data, each requiring calculation of its average. Each data set is followed by a trailer record with a value of -1; however, the last data set is followed by a trailer record with a value of -2. Draw a flowchart for the logic to perform this task.

52. Suppose five numbers denoted by the variables A, B, C, D, and E are supplied as input. Draw a flowchart for the logic to print these numbers in descending order of magnitude.

53. Draw a flowchart for the logic to find out whether a given triangle ABC is isosceles.

Assume that the angles of the triangle are supplied as input. Print the answer as yes or no.

54. Draw a flowchart for the logic to find out whether a given triangle ABC is a right angled triangle. Assume that the sides of the triangle are supplied as input data. Print the answer as yes or no.

55. Draw a flowchart for the logic to find out whether a given quadrilateral ABCD is a rectangle. Assume that all the four angles and four sides of the quadrilateral are supplied as input data. Print the answer as yes or no.

56. Draw a flowchart to illustrate the logic to convert a number from base 10 to a new base using the division-remainder technique.

57. Draw a flowchart to illustrate the logic to convert a number to decimal from another base.

58. Obtain a decision table to solve the problem described in Question 38.

59. Obtain a decision table to solve the problem described in Question 43.

60. Obtain a decision table to solve the problem described in Question 46.

61. Obtain a decision table to solve the problem described in Question 53.

62. Obtain a decision table to solve the problem described in Question 54.

63. Write the pseudocode to solve the problem described in Question 39.

64. Write the pseudocode to solve the problem described in Question 40.

65. Write the pseudocode to solve the problem described in Question 43.

66. Write the pseudocode to solve the problem described in Question 46.

67. Write the pseudocode to solve the problem described in Question 47.

68. Write the pseudocode to solve the problem described in Question 48.

69. Write the pseudocode to solve the problem described in Question 49.

70. Write the pseudocode to solve the problem described in Question 53.

CHAPTER 12

12. COMPUTER LANGUAGES

This chapter continues the development of computer programs that was begun in Chapter 11. After the programming analysis phase, discussed in the previous chapter, has been completed, the second step in the development of computer programs is to write the specific instructions, needed to process an application, into a language and form acceptable to a computer system. The process of writing such program instructions for an analysed problem is called *coding*. In this chapter, we will see how the logical steps of our program plan will be coded as program instructions. The goal of this chapter is to introduce some of the common computer languages used in writing computer programs.

ANALOGY WITH NATURAL LANGUAGES

A language is a system of communication. With our natural language such as English, we communicate to one another our ideas and emotions. Similarly, a computer language is a means of communication used to communicate between people and the computer. With the help of a computer language, a programmer tells a computer what he wants it to do. All natural languages (English, French, German, etc.) use a standard set of symbols for the purpose of communication. These symbols are understood by everyone using that language. We normally call this set of symbols the vocabulary of that particular language. For example, the words we use in English are the symbols of English language that make up its vocabulary. Each word has definite meaning which can be looked up in a dictionary. In a similar manner, all computer languages have a vocabulary of their own. Each symbol of the vocabulary has definite unambiguous meaning which can be looked up in the manual meant for that language. Hence, each symbol of a computer language is used to tell the computer to do a particular job. The main difference between a natural language and a computer language is that natural languages have a large vocabulary but most computer languages use a very limited or restricted vocabulary. This is mainly because a programming language by its very nature and purpose does not need to say too much. Each and every problem to be solved by a computer has to be broken down into discrete (simple and separate), logical steps which basically comprise of four fundamental operations - input and output operations, arithmetic operations, movement of information within the CPU, and logical or comparison operations.

Each natural language has a systematic method of using symbols of that language. In English, this method is given by the rules of grammar. These rules tell us which words to use and how to use them. Similarly, the symbols of a particular computer language must also be used as per set rules which are known as the syntax rules of the language. In case of a natural language, people can use poor or incorrect vocabulary and grammar and still make themselves understood. However, computers, being a machine, are receptive only to exact vocabulary used correctly as per syntax rules of the language being used. Thus, in case of a computer language, we must stick by the exact rules of the language if we want to be understood by the computer. As yet, no computer is capable of correcting and deducing meaning from incorrect instructions. Computer languages are smaller and simpler than natural languages but they have to be used with great precision.

Unless a programmer adheres exactly to the syntax rules of a programming language, even down to the correct punctuation marks, his commands will not be understood by the computer.

Programming languages have improved throughout the years, just as computer hardware has improved. They have progressed from machine-oriented languages that use strings of binary 1s and 0s to problem-oriented languages that use common mathematical and/or English terms. However, all computer languages can be classified in the following three broad categories :

(a) Machine Language

(b) Assembly Language

(c) High-level Language.

We shall now examine the evolution and nature of each type of language.

MACHINE LANGUAGE

Although computers can be programmed to understand many different computer languages, there is only one language understood by the computer without using a translation program. This language is called the *machine language* or the *machine code* of the computer. Machine code is the fundamental language of a computer and is normally written as strings of binary 1s and 0s. The circuitry of a computer is wired in such a way that it immediately recognizes the machine language and converts it into the electrical signals needed to run the computer.

An instruction prepared in any machine language has a two-part format, as shown in Figure 12.1. The first part is the command or operation, and it tells the computer what function to perform. Every computer has an *operation code* or *opcode* for each of its functions. The second part of the instruction is the operand, and it tells the computer where to find or store the data or other instructions that are to be manipulated. Thus, each instruction tells the control unit of the CPU what to do and the length and location of the data fields that are involved in the operation. Typical operations involve reading, adding, subtracting, writing, and so on.

OPCODE (operation code)	OPERAND (Address/Location)

Figure 12.1. Instruction format.

We already know that all computers use binary digits (0s and 1s) for performing internal operations. Hence, most computers' machine language consists of strings of binary numbers and is the only one the CPU directly understands. When stored inside the computer, the symbols which make up the machine language program are made up of 1s and 0s. For example, a typical program instruction to print out a number on the printer might be

```
101100111111010011101100
```

The program to add two numbers in memory and print the result might look something like the following :

```
001000000000001100111001
001100000000010000100001
011000000000011100101110
101000111111011100101110
000000000000000000000000
```

This is obviously not a very easy language to learn, partly because it is difficult to read and understand and partly because it is written in a number system with which we are not familiar. But it will be surprising to note that some of the first programmers, who worked with the first few computers, actually wrote their programs in binary form as above.

Since human programmers are more familiar with the decimal number system, most of them preferred to write the computer instructions in decimal, and leave the input device to convert these to binary. In fact, without too much effort, a computer can be wired so that instead of using long strings of 1s and 0s we can use the more familiar decimal numbers. With this change, the preceding program appears as follows :

```
10001471
14002041
30003456
50773456
00000000
```

The set of instruction codes, whether in binary or decimal, which can be directly understood by the CPU of a computer without the help of a translating program, is called a machine code or machine language. Thus, a machine language program need not necessarily be coded as strings of binary digits (1s and 0s). It can also be written using decimal digits if the circuitry of the computer being used permits this.

Advantages and Limitations of Machine Language

Programs written in machine language can be executed very fast by the computer. This is mainly because machine instructions are directly understood by the CPU and no translation of the program is required. However, writing a program in machine language has several disadvantages which are discussed below.

1. *Machine dependent*. Because the internal design of every type of computer is different from every other type of computer and needs different electrical signals to operate, the machine language also is different from computer to computer. It is determined by the actual design or construction of the ALU, the control unit, and the size as well as the word length of the memory unit. Hence, suppose after becoming proficient in the machine code of a particular computer, a company decides to change to another computer, the programmer may be required to learn a new machine language and would have to rewrite all the existing programs.

2. *Difficult to program*. Although easily used by the computer, machine language is difficult to program. It is necessary for the programmer either to memorize the dozens of code numbers for the commands in the machine's instruction set or to constantly refer to a reference card. A programmer is also forced to keep track of the storage location of data and instructions. Moreover, a machine language programmer must be an expert who knows about the hardware structure of the computer.

3. *Error prone*. For writing programs in machine language, since a programmer has to remember the opcodes and he must also keep track of the storage location of data and instructions, it becomes very difficult for him to concentrate fully on the logic of the problem. This frequently results in program errors. Hence, it is easy to make errors while using machine code.

4. *Difficult to modify*. It is difficult to correct or modify machine language programs. Checking machine instructions to locate errors is about as tedious as writing them initially. Similarly, modifying a machine language program at a later date is so difficult that many programmers would prefer to code the new logic afresh instead of incorporating the necessary modifications in the old program.

In short, writing a program in machine language is so difficult and time consuming that it is rarely used today.

ASSEMBLY LANGUAGE

One of the first steps in improving the program preparation process was to substitute letter symbols-mnemonics for the numeric operation codes of machine language. A *mnemonic* (or memory aid) is any kind of mental trick we use to help us remember. Mnemonics come in various shapes and sizes, all of them useful in their own way. For example, a computer may be designed to interpret the machine code of 1111 (binary) or 15 (decimal) as the operation `subtract', but it is easier for a human being to remember it as SUB.

Use of Symbols Instead of Numeric OpCodes

All computers have the power of handling letters as well as numbers. Hence, a computer can be taught to recognize certain combination of letters or numbers. It can be taught (by means of a program) to substitute the number

14 every time it sees the symbol ADD, substitute the number 15 every time it sees the symbol SUB, and so forth. In this way, the computer can be trained to translate a program written with symbols instead of numbers into the computer's own machine language. Then we can write program for the computer using symbols instead of numbers, and have the computer do its own translating. This makes it easier for the programmer, because he can use letters, symbols, and mnemonics instead of numbers for writing his programs. For example, the preceding program that was written in machine language for adding two numbers and printing out the result could be written in the following way :

```
CLA  A
ADD  B
STA  C
TYP  C
HLT
```

Which would mean "take A, add B, store the result in C, type C, and halt." The computer, by means of a translating program, would translate each line of this program into the corresponding machine language program.

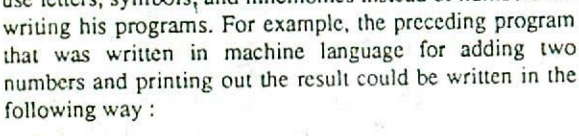


Figure 12.2. Illustrating the translation process of an assembler.

At this point we must learn a few more terms. The language which substitutes letters and symbols for the numbers in the machine language program is called an *assembly language* or *symbolic language*. A program written in symbolic language that uses symbols instead of numbers is called an *assembly code* or a *symbolic program*. The translator program that translates an assembly code into the computer's machine code is called an *assembler*. The assembler is a system program which is supplied by the computer manufacturer. It is written by system programmers with great care. It is so called because in addition to translating the assembly code into machine code, it also, 'assembles' the machine code in the main memory of the computer and makes it ready for execution. A symbolic program written by a programmer in assembly language is called a *source program*. After the source program has been converted into machine language by an assembler, it is referred to as an *object program*. As shown in Figure 12.2, the input to an assembler is a source program written in assembly language and its output is an object program which is in machine language. Since the assembler translates each assembly language instruction into an equivalent machine language instruction, there is a one-to-one correspondence between the assembly instructions of source program and the machine instructions of object program.

By now, it must have been clear to the readers that when we write a program in symbolic language, we first run the assembler (program) to assemble the symbolic program into machine language, and then we run the machine language program to get our answer. You will notice that this means more time spent by the computer - it not only has to run the main program to get the answer, but it also must first translate the original symbolic program into machine language. But symbolic programming saves so much time and effort of the programmer that the extra time spent by the computer is worth it.

To see how symbolic programming works, let us first write a short machine language program and then see how we would write the same program in assembly language. For this, let us assume that the computer uses the following mnemonics for the operation codes mentioned against

Mnemonic	Op Code	Meaning
HLT	00	Halt used at end of program to stop
CLA	10	Clear & Add into A register
ADD	14	Add to the contents of A register
SUB	15	Substract from contents of A register
STA	30	Store A register

each. For simplicity, here we have considered only five operation codes that will be used in writing our program. Like this there can be more than hundred operation codes available with a particular computer.

The program we will write is quite simple : adding two numbers and storing the sum. The regular machine language program for this will be as follows :

Memory Location	Contents		comments
	op Code	Address	
0000	10	1000	Clear & add first number to A register
0001	14	1001	Add second number to the contents of A register
0002	30	1002	Store the answer from A register
0003	00		Halt
.			
.			
.			
1000			Reserved for first number
1001			Reserved for second number
1002			Reserved for the answer.

It has been assumed here that the computer is capable of handling decimal numbers instead of only binary numbers. The two numbers to be added are stored in memory locations 1000 and 1001, and the answer obtained after adding the two numbers is to be stored in location 1002. The first instruction at location 0000 clears (makes zero) the A register (accumulator) and puts the contents of location 1000 (first number) in it. The second machine instruction at location 0001 adds the contents of location 1001 (second number) to the contents of A register (first number) and stores the sum in A register. The third instruction at location 0002 stores the answer from A register into memory location 1002. Finally, the fourth machine instruction at location 0003 stops the execution of the program.

Now we will see how to write the same program in assembly language. We can easily replace the op-code in each of the preceding instructions by the corresponding mnemonic and write the program as given below instead; and let the computer handle the rest. The assembler program would then translate CLA to 10, ADD to 14, STA to 30 and HLT to 00, thus producing the machine language program.

Memory Location	Contents	
	Mnemonic	Address
0000	CLA	1000
0001	ADD	1001
0002	STA	1002
0003	HLT	

Use of Symbols Instead of Addresses

But there is no need to stop here; after all, it is not much more work to memorize a few numbers for op-codes, and using symbols for op-codes just makes more work for the computer. We might as well let the computer do a large share of the work.

One of the greatest problems in machine language programming comes from keeping track of addresses. Every time we write a large program, we need a pad of paper off to the side on which we keep a running list of what numbers are stored where. Each time we want a number, we must look up the address on this list. This takes time and can lead to mistakes if we are not careful. Why not let the computer do this part of work for us ? We can include, as part of the assembly program, a section that does nothing but keeps a list of addresses for numbers.

In the preceding sample program to add two numbers, we might start the symbolic program by telling the computer something like the following :

"From now on address 1000 will be called FRST, address 1001 will be called SCND, and address 1002 will be called ANSR."

In response to this, the assembler sets up a table somewhere in the computer memory which looks something like the following:

```
FRST = 1000
SCND = 1001
ANSR = 1002
```

From now on, we just call numbers by their names. The assembler will look up the name in the table and provide the right address, thus saving us even more work. This means we can write our sample program as shown below:

Memory Location	Contents	
	Mnemonic	Address
0000	CLA	FRST
0001	ADD	SCND
0002	STA	ANSR
0003	HLT	

The assembler then looks up the address for each of the numbers, translates the mnemonic op-code into the numerical op-code and comes out with a machine language program. For example, in the first instruction CLA FRST, the assembler translates the CLA into an op-code of 10 and looks up the address of FRST in the table. It finds the address as 1000, and so it puts the first machine language instruction as 10 1000. In the same way it translates the ADD SCND instruction into 14 1001, and so on, making the machine language program step by step.

But there is one more part of our work that we can hand over to the computer. We need not even tell the computer where to place each number and where to place each instruction, as we have been doing till now. Instead of saying something like "Place FRST in 1000, SCND in 1001, and ASNR in 1002," we need only tell the computer to start putting the numbers into memory starting with address 1000. In the same way we do not have to specify that the instruction CLA FRST goes into 0000, ADD SCND into 0001, and so on - we merely tell the computer to start the program at location 0000. The symbolic program would therefore go something like this :

START PROGRAM AT 0000 AND START DATA AT 1000

SET ASIDE AN ADDRESS FOR FRST

SET ASIDE AN ADDRESS FOR SCND

SET ASIDE AN ADDRESS FOR ANSR

CLA FRST

ADD SCND

STA ANSR

HLT

Then we start the assembler working on this symbolic program. We see that the first four steps are not really parts of the main program to add the two numbers; they are instructions to the assembler on what to do, and are called *pseudo-instructions*. The word pseudo, from the greek word meaning false, fake or pretended, describe them quite well : they are instructions that do not do anything in the main program, but only provide information to the assembler to tell it how we want the program assembled.

To see how the assembler changes this symbolic program into machine language, let us follow it step by step.

The first step tells the assembler that the program should start at address 0000 and each following instruction should be in the following address, and that the data (in this example this includes the FRST and SCND numbers, as well as the ANSR) should start at memory location 1000.

The next step tells the assembler to set aside an address for FRST. The assembler therefore picks the first free address in the group set aside for data, which is 1000, and will call it FRST from now on. Note that because we will refer to the address with the symbol FRST from now on, we really do not care what the address exactly is : whether it is 1000 or 1001 or 4253 makes no difference because the assembler will keep track of it.

The next step tells the assembler to set aside an address for SCND. The next free address after 1000 is 1001 (because 1000 is already taken by FRST), and so the assembler will supply the address 1001 every time we use the symbol SCND.

In the same way, the next step tells the assembler to set aside an address, this time 1002, for ANSR. This is the last pseudo-instruction, and the next instruction will start the program itself.

The next instruction is CLA FRST, which the computer translates into 10 1000 by translating CLA into 10 and looking up the address of FRST in its table which is 1000. Similarly, the assembler will translate the instruction ADD SCND into 14 1001 and the instruction STA ANSR into 30 1002. Finally it translates HLT into 00, thus providing the complete machine language program for the given assembly language program.

Advantages of Assembly Language Over Machine Language

Assembly languages have the following advantages over machine languages :

1. *Easier to understand and use.* Assembly languages are easier to understand and use because mnemonics are used instead of numeric op-codes and suitable names are used for data. The use of mnemonics means that comments are usually not needed; the program itself is understandable. Symbolic programming also saves a lot of time and effort of the programmer because it is easier to write as compared to machine language programs.

2. *Easy to locate and correct errors.* While writing programs in assembly language, fewer errors are made, and those that are made are easier to find and correct because of the use of mnemonics and symbolic field names. Furthermore, assemblers are so designed that they automatically catch errors. If we use an invalid mnemonic or a name that has never been defined, the assembler will print out an error indication. For example, suppose one instruction in the symbolic program reads ADD AREA, and we forget to define what AREA is, the assembler will look through its table to find AREA and not finding it will indicate the error.

3. *Easier to modify.* Assembly language programs are easier for people to modify than machine-language programs. This is mainly because they are easier to understand an hence it is easier to locate, correct, and modify instructions as and when desired. Moreover, insertion or removal of certain instructions from the program does not require change in the address part of the instructions following that part of the program. This is required in case of machine language.

4 *No worry about addresses.* One of the greatest advantage of assembly language is that it eliminates worry about address for instructions and data. This is more important than it seems at first glance. Suppose we have written a long machine language program involving many steps and many references to itself within the program, such as looping, and address modifications, and so on. At the very end we may suddenly discover that we have left out an instruction in the middle. If we insert that instruction, we will have to remember all the following instructions, and go through the entire program to check any references to other steps. This is a tedious job. But if we write the same program in symbolic language, we merely add the extra instruction, and the assembler will take care of the step numbering automatically.

5. *Easily relocatable.* Suppose that an assembly language program starts at address 1000 and we suddenly find that we have another program to be used with this program and this program also starts at location 1000. Obviously, one of the two programs will have to be rewritten to be moved somewhere else. In machine language, this can be a complicated job. But in case of assembly language, we merely have to change the first statement; for example instead of :

   START PROGRAM AT 1000 AND START DATA AT 2000

   we merely change this first statement to

   START PROGRAM AT 3000 AND START DATA AT 4000

   and run the symbolic program once more through the assembler. The equivalent machine language program will this time start at memory location 3000 instead of 1000, and there will be no conflict with the other program. In other words, using symbolic language we can easily move programs from one section of the memory to another; we say that assembly language programs are easily relocatable because their location is easily changed merely by changing the first instruction. This is not easily done with machine language programming.

6. *Efficiency of machine language.* In addition to the above mentioned advantages, an assembly language program also enjoys the efficiency of its corresponding machine code because there is one to one correspondance between the instructions of an assembly language program and its corresponding machine language program. Except for pseudo-instructions, which are simply instructions to the assembler, every other instruction of an assembly language program is translated into one machine language instruction. For every machine language instruction, there is a corresponding symbolic

instruction and for every symbolic instruction (except the pseudo-instructions) there is a corresponding machine instruction. In other words, the symbolic program will be just as long as the resulting machine language program. So leaving out the translation time required by the assembler, the actual execution time of an assembly language program and its equivalent machine language program (written independently) will be the same. The reason we are stressing this important fact is that there are languages (called macro-languages) in which a single instruction may get translated into an entire series of machine language instructions. Assembly language, in its basic form, is not one of these - there is one-to-one relationship between symbolic and machine languages.

Limitations of Assembly Language

The following disadvantages of machine language are not solved by using assembly language :

1. *Machine dependent.* Because each instruction in the symbolic language is translated into exactly one machine language instruction, assembly languages are machine dependent. That is, they are designed for the specific make and model of the processor being used. A decision to change to another computer still usually requires learning a new language and the conversion of all existing programs - a very expensive undertaking.

2. *Knowledge of hardware required.* Since assembly languages are machine dependent, so the programmer must be aware of a particular machine's characteristics and requirements as the program is written. An assembly language programmer must know how his machine works and should have a good knowledge of the logical structure of his computer in order to write a good assembly language program.

3. *Machine level coding.* In case of an assembly language, instructions are still written at the machine-code level - that is, one assembler instruction is substituted for one machine-code instruction.

Machine and assembly languages being machine dependent, are referred to as *low-level languages.*

Assembly Languages With Macro Instructions

In general, assembly languages are termed one-for-one in nature, that is, each assembly language instruction will result in one machine language instruction. However, quite often a certain set of machine language or assembly language instructions have to be used over and over. For example, three instructions, one after the other, might be needed to print out a number on a particular computer. These three instructions, always in the same order, might be used over and over in the same program. Instead of forcing the programmer to write out the set of three instructions every time he wants to print a number, we might as well design the assembler (program) in such a way so as to take care of these instructions. Every time the programmer gave the PRINT instruction, for example, the assembler would translate it into three machine language instructions instead of one, thús supplying the complete set of instructions required for printing.

Any instruction, such as PRINT, which gets translated into several machine language instructions, is called a *macro instruction.* There might be many such macro instructions permitted by a particular assembler. Thus, to speed up the coding process, assemblers were developed that could produce a variable amount of machine language instructions for each macro instruction of the assembly language program.

The use of macro instructions adds much work to the computer because the translation process becomes more than just changing each word into a number. The assembler must be able to supply the missing steps as well, but it means a tremendous saving of work for the programmer. The programmer gets relieved of the task of writing an instruction for every machine operation performed. It reduces the length of the programs he writes, cuts down on his errors, and simplifies his programming.

The macro instruction capability was developed very early in the evolution of computer languages. In fact, it is this concept of multiple machine instructions from one macro instruction around which today's machine-independent higher level languages are designed.

HIGH-LEVEL LANGUAGE

We have already seen that writing of programs in machine language or assembly language requires a deep knowledge of the internal structure of the computer. While writing programs in any of these languages, a programmer has to remember all the operation codes (numeric or mnemonic) of the computer and know in detail what each code does and how it affects the various registers of the computer. However, we have also seen that in order to write a good computer program. the programmer should mainly concentrate on the logic of the problem rather than be concerned with the details of the internal structure of the computer. In order to facilitate the programmers to use computers without the need to know in detail the internal structure of the computer, high-level languages were developed.

High-level languages, instead of being machine based, are oriented more towards the problem to be solved. These languages enable the programmer to write instructions using English words and familiar mathematical symbols. So it becomes easier for him to concentrate on the logic of his problem rather than getting involved in programming details. Obviously, the two-part format shown in Figure 12.1, that was required for writing instructions in machine language or assembly language, is not necessary for writing instructions in a high-level language. For example, let us consider the same problem of adding two numbers (FRST & SCND) and storing the sum in ANSR. We have already seen that three low-level (machine/assembly) instructions are required for performing this job. However, if we use a high-level language, say FORTRAN for instance, to instruct the computer to do this job, only one instruction need be written :

ANSR = FRST + SCND

This instruction is obviously very easy to understand and write because it resembles the familiar algebraic notation for adding two numbers : $a = b + c$.

High-level languages are basically symbolic languages that use English words and/or mathematical symbols rather than mnemonic codes. In other words, a high-level language is a symbolic language with nothing but macro-instructions. Every instruction which the programmer writes in a high-level language is translated into many machine language instructions. This is one-to-many translation and not one-to-one as in the case of assembly language. It is due to this reason that high-level languages are so called.

High-level languages are also known as *problem-oriented languages* because the macro instructions are especially picked to be useful for solving particular types of problems. Each such language is then best to solve a particular class of problems and may be completely useless for solving other types of problems. For example, if a high-level language is capable of handling business-type applications that consist of high input volume, relatively little processing, and a high output volume, then the language is a *business-oriented language*. On the other hand, languages excellent at performing sophisticated computations but not adept at handling large data files are *mathematically-oriented languages*. Thus, a problem-oriented language is designed in such a way that its instructions may be written more like the language of the problem. For example, a scientist using a science-oriented language can use scientific formulas, and a business man with a business-oriented language can use business terms. Hence, high-level languages are generally easier to learn and write.

COMPILERS

Since a computer hardware is capable of understanding only machine level instructions, so it is necessary to convert the instructions of a program written in high-level language to machine instructions before the program can be executed by the computer. We have seen that assembly languages use an assembler to perform this conversion process. In case of a high-level language, this job is carried out by a *compiler*. Thus, a compiler is a translating program (much more sophisticated than an assembler) that translates the instructions of a high-level language into machine language. A compiler is so called because it compiles a set of machine language instructions for every program instruction of a high-level language. A program written by a programmer in a high-level language is called a *source program*. After this source program has been converted into machine language by a compiler, it is referred to as an *object program*. As shown in Figure 12.3 the input to a compiler (program) is a source program written in a high-level language and its output is an object program which consists of machine language instructions. Note that the source program and the object program are the same program, but at different stages of development.

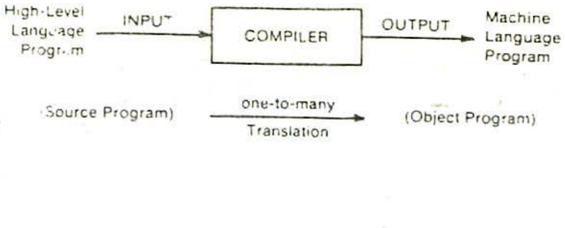


Figure 12.3. Illustrating the translation process of a compiler.

A compiler can translate only those source programs which have been written in the language for which the computer is meant. For example, a FORTRAN compiler is only capable of translating source programs which have been written in FORTRAN and, therefore, each machine requires a separate compiler for each high-level language that it supports. This is illustrated in Figure 12.4. Moreover, since an object program for one machine will not be the same as the object program for another machine, it is necessary that each machine must have its own `personal' compiler for a particular language, say L1. Figure 12.5 illustrates how machine-independence is achieved by using different compilers to translate the same high-level language program to machine languages of different computers.

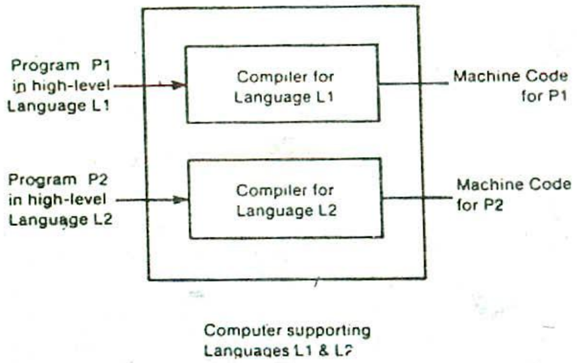


Figure 12.4. Illustrating the requirement of separate compilers for each high-level language supported by a computer.

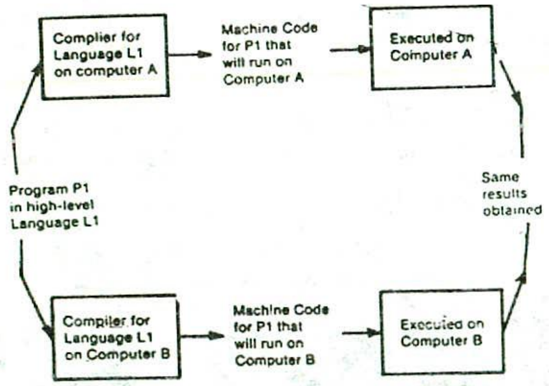


Figure 12.5. Illustrating the machine independence characteristic of a high-level language. Separate compilers are required for the same language on different computers.

Compilers are large programs which reside permanently on secondary storage. When the translation of a program is to be done, they are copied into the main memory of the computer. The compiler, being a program, is executed in the CPU. While translating a given program, the compiler analyses each statement in the source program and generates a sequence of machine instructions which, when executed, will precisely carry out the computation specified in the statement. As the compiler analyses each statement, it uncovers certain types of errors. These are referred to as *diagnostic errors*. The compiler can diagnose the following kinds of errors in a source program :

a. Illegal characters

b. Illegal combination of characters

c. Improper sequencing of instructions in a program.

A source program containing an error diagnosed by the compiler will not be compiled into an object program. The compiler will print out a suitable message indicating this, along with a list of coded error messages which indicate the type of errors committed. The error diagnostics is an invaluable aid to the programmer.

A compiler, however, cannot diagnose *logical errors*. It can only diagnose grammatical (syntax) errors in the program. It cannot know ones intentions. For example, if one has wrongly punched -25 as the age of a person, when he actually intended +25, the compiler cannot diagnose this. Programs containing such errors will be successfully compiled and the object code will be obtained without any error message. However, such programs, when executed, will not produce the right answers. So logical errors are detected only after the program is executed and the result produced does not tally with the desired result. Hence, it is essential to be precise in writing a program and pay careful attention to the smallest detail.

INTERPRETERS

An *interpreter* is another type of translator used for translating high-level languages into machine code. It takes one statement of a high-level language and translates it into a machine instruction which is immediately executed. Translation and execution alternate for each statement encountered in the high-level language program. In other words, an interpreter translates one instruction, and the control unit executes the resulting machine code, the next instruction is translated, and the control unit executes the machine code instruction, and so on. This differs from a compiler which merely translates the entire source program into an object program and is not involved in its execution. In case of a compiler, the whole source program is translated into an equivalent machine language program. The object code, thus obtained, is permanently saved for future use and is used every time the program is to be executed. So repeated compilation is not necessary for repeated execution of a program. However, in case of an interpreter, no object code is saved for future use because the translation and the execution processes alternate. The next time an instruction is used, it must once again be interpreted and translated into machine language. For example, during the repetitive processing of the instructions within a loop, each instruction in the loop will have to be reinterpreted every time the loop is executed.

Interpreters are often employed with microcomputers (small computers). The advantage of an interpreter over a compiler is fast response to changes in the source program. The interpreter eliminates the need for a separate compiling run after each program change to add features or correct errors. Moreover, a compiler is a complex program compared to an interpreter. Interpreters are easy to write and they do not require large memory space in the computer. The interpreter, however, is a time consuming translation method because each statement must be translated every time it is executed from the source program. Thus, a compiled machine language program runs much faster than an interpreted program.

Assemblers, compilers, and interpreters are systems software that translate a source program written by the user to an object program which is meaningful to the hardware of the computer. These translators are also referred to as *language processors*. since they are used for processing a particular language.

Based on the above discussions, we conclude that, in general, a programming language should possess the following characteristics to be considered high-level :

1. The language should be relatively independent of a given computer system. That is, instead of being machine based, it should be oriented more towards the problem to be solved.
2. Each statement of the language should be a macro instruction that gets translated into many machine language instructions.
3. The language should enable the programmers to write instructions using familiar words and mathematical symbols. It should be natural and should use abbreviations and words used in everyday communication.
4. The language should be independent of machine language instructions and other pieces of system software except for the compiler or the interpreter.
5. The language should not be experimental in nature and should exist on more than one computer system.

Advantages of High-Level Languages

High level languages enjoy the following advantages over assembly and machine languages :

1. *Machine independence*. High-level languages are machine independent. This is a very valuable advantage because it means that a company changing computers - even to one from a different manufacturer - will not be required to rewrite all the programs that it is currently using. Even for programs written in the high-level languages, some modifications are almost always required, but these modifications are relatively minor and can be easily done without much effort. In other words, a program written in a high-level language can be run on many different types of computers with very little or practically no effort.

2. *Easy to learn and use*. These languages are very similar to the languages normally used by us in our day-to-day life. Hence they are easy to learn and use. The programmer need not learn anything about the computer he is going to use. He need not worry about how to store his numbers in the computer, where to store them, what to do with them, etc. That is the programmer need not know the machine instructions, the data format, and so on.

However, such a knowledge is desirable since it allows the programmer to utilize the system more efficiently.

3. *Fewer errors.* In case of high-level languages, since the programmer need not write all the small steps carried out by the computer, he is much less likely to make an error. The computer takes care of all the little details, and will not introduce any error of its own unless something breaks down. Furthermore, compilers are so designed that they automatically catch and point out the errors made by the programmer. Hence, diagnostic errors, if any, can be easily located and corrected by the programmer.

4. *Lower program preparation cost.* Writing programs in high-level languages requires less time and effort which ultimately leads to lower program preparation cost. Generally, the cost of all phases of program preparation (coding, debugging, testing, etc.) is lower with a high-level language than with an assembly language or with a machine language.

5. *Better documentation.* A high-level language is designed in such a way that its instructions may be written more like the language of the problem. Thus the statements of a program written in a high-level language can be easily understood by a person familiar with the problem. For the documentation of such programs, very few or practically no separate comment statements are required.

6. *Easier to maintain.* Programs written in high-level languages are easier to maintain than assembly language or machine language programs. This is mainly because they are easier to understand and hence it is easier to locate, correct, and modify instructions as and when desired. Insertion or removal of certain instructions from a program is also possible without any complication. Thus, major changes can be incorporated with very little effort.

Limitations of High-level Languages

Two disadvantages of high-level languages are :

1. *Lower efficiency.* Generally, a program written in assembly language or machine language is more efficient than one written in high-level language. That is, the programs written in high-level languages take more time to run and require more main storage.

2. *Lack of flexibility.* Because the automatic features of high-level languages always occur and are not under the control of the programmer, they are less flexible than assembly languages. An assembly language provides programmers access to all the special features of the machine they are using. Certain types of operations which are easily programmed using the machine's assembly language, are impractical to attempt using a high-level language. This lack of flexibility means that some tasks cannot be done in a high-level language, or can be done only with great difficulty.

In most cases, the advantages of high-level languages far outweigh the disadvantages. Most computer installations use a high-level language for most programs and use an assembly language for doing special tasks that cannot be easily done otherwise.

SOME-HIGH-LEVEL LANGUAGES

The credit for the development of the first high-level language is usually given to Dr. Grace Hopper who described the idea of a compiler and its language as early as 1952. Two languages were developed under Dr. Hopper's directions : FLOWMATIC was a commercial and business language which could easily be put together from the contents of a flowchart, whereas MATHEMATIC was a mathematical language. These two languages were an early example of the development of high-level languages in different directions - business and commercial, and science and mathematics. Since then, many other high-level languages have been produced. Today there are over 200 high-level languages. However, most of these are for very special purposes or are designed to solve problems in a very specific applications area. Some of the most common high-level languages have been briefly described below. The primary objective is to provide some insight into these languages rather than to provide detailed knowledge required to write programs.

FORTRAN

One of the oldest and the most popular high-level language is FORTRAN which stands for *FOR*mula *TRAN*slation. Originally developed by IBM (International Business Machine Corporation) for its 704 computer in 1957, FORTRAN has undergone several revisions so that the language has been evolving into a wider and more useful language with time. The original FORTRAN was soon followed by FORTRAN II. The next popular and advanced version was FORTRAN IV. In order to allow a program that was written for one computer system to be run on another computer system, this version was standardized by the American National Standards Institute (ANSI) in the year 1966. FORTRAN thus has the distinction of being the first standardized language. In 1977 an updated version of FORTRAN IV, known as FORTRAN 77, was announced and standardized by ANSI. It contains several additional features which are not a part of FORTRAN IV, e.g. character and file handling, constructs related to a more structural approach to programming.

FORTRAN was designed to solve scientific and engineering problems and is currently the most popular language among scientists and engineers. The language is oriented towards solving problems of a mathematical nature and has been designed as an algebra-based programming language. Any formula or those mathematical relationships that can be expressed algebraically can easily be expressed as a FORTRAN instruction, e.g. A = B + C - D. To illustrate the nature of FORTRAN programs, a simple FORTRAN program to compute and print the sum of 10 numbers is given below.

```
C     FORTRAN PROGRAM TO COMPUTE
C     THE SUM OF 10 NUMBERS
      SUM = 0
      DO 50 I = 1, 10
      READ (5, 10) N
10    FORMAT (F6 . 2)
      SUM = SUM + N
50    CONTINUE
      WRITE (6, 20) SUM
20    FORMAT (1X, 'THE SUM OF GIVEN
    - NUMBERS=', F10 . 2)
      STOP
      END
```

From the example you can see that a FORTRAN program consists of a series of statements. These statements supply input/output, calculation, logic/comparison, and other basic instructions to the computer. The words READ, WRITE, DO, and STOP in the statements mean exactly what you would expect. A FORTRAN program requires that certain parts of every statement be placed in certain columns. Statement numbers, which are optional in FORTRAN, are placed in columns 1-5 (10, 50 and 20 in this example). A comment statement starts with a C in the first column (the first statement of this example). Comment statements are used in programs for the purpose of documentation or explanation designed to assist anyone reading the source program listing. Comments do not form a part of the program logic and are ignored by the computer. The actual FORTRAN statement is placed in columns 7-72. A character in column 6 means that the statement in the previous line is being continued in this line. Columns 73-80 are ignored by the computers. The programmer may use these columns for any purpose, such as numbering each statement or writing a program code name.

In the example above, the value of SUM is first initialized to 0. The next statement tells the computer to do a loop that starts at the DO statement and ends in line having label 50, which is a CONTINUE statement. Inside the loop, values of N are read and added to SUM one by one. After the computer loops 10 times, reading and accumulating the sum of 10 numbers, the computer goes out of the loop and drops down to the next statement. This is the WRITE statement which prints the message : THE SUM OF GIVEN NUMBERS = followed by the computed value of SUM. The next statement, which is a STOP statement tells the computer to stop the execution of the program. Finally, the END statement tells the computer that there are no more instructions or statements in the program. The data for this program is contained in a separate file and is not shown in the program.

COBOL

COBOL is an *acronym* for *CO*mmon *B*usiness *O*riented *L*anguage. As its name implies, this language was designed specifically for business data processing and today it is the most widely used business-oriented programming language. Unlike FORTRAN, which gradually developed into a full-fledged language, the vocabulary and grammar of COBOL were worked out in 1959-1960 by a committee of the *CO*nference on *DA*ta *SY*stems *L*anguages (CODASYL) as a joint effort of computer users, manufacturers, and the United States government. After the vocabulary and grammar were defined by this committee, the various manufacturers wrote the compilers for their computers. Since 1960, the language has been revised, but revision by manufacturers have been rare, because other CODASYL committees have continued to maintain, revise, and extend the initial specifications. An ANSI COBOL standard was first published in 1968, and a later version was approved in 1974. As long as these standards are followed, a COBOL program can be run on any computer system with an ANSI COBOL compiler.

COBOL was designed to have the appearance and structure of a business report written in English. Thus, a COBOL program is constructed from sentences, paragraphs, sections, and divisions. All COBOL programs must have four divisions namely, the identification division, the environment division, the data division, and the procedure division. The nature of COBOL program is illustrated below with the help of a simple COBOL program to compute and print the sum of given numbers.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SUMUP.
AUTHOR. P K SINHA.
* THIS PROGRAM COMPUTES AND PRINTS
* THE SUM OF GIVEN NUMBERS.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. BURROUGHS-6700.
OBJECT-COMPUTER. BURROUGHS-6700.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT DATA-FILE ASSIGN TO
        CARD-READER.
        SELECT OUTPUT-FILE ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
FD DATA-FILE
        RECORD CONTAINS 80 CHARACTERS
        LABEL RECORD IS OMITTED
        DATA RECORD IS INPUT-DATA-RECORD.
01      INPUT-DATA-RECORD.
        05 N      PICTURE 9(6)V99.
        05 FILLER PICTURE X(72).
FD      OUTPUT-FILE
        RECORD CONTAINS 132 CHARACTERS
        LABEL RECORD IS OMITTED
        DATA RECORD IS OUTPUT-RECORD.
01      OUTPUT-RECORD.
        05 FILLER PICTURE X.
        05 TITLE PICTURE X(25).
        05 SUM PICTURE 9(10)V99.
        05 FILLER PICTURE X(94).
WORKING-STORAGE SECTION.
77      MESSAGE PICTURE X(25) VALUE IS
                "THE SUM OF GIVEN NUMBERS=".

PROCEDURE DIVISION.
OPEN-FILES.
        OPEN INPUT DATA-FILE.
        OPEN OUTPUT OUTPUT-FILE.
INITIALIZATION.
        MOVE SPACES TO OUTPUT-RECORD.
        MOVE ZERO TO SUM.

PROCESS-LOOP.
        READ DATA-FILE AT END CO TO
        PRINT-PARA.
        ADD N TO SUM.
        GO TO PROCESS-LOOP.
PRINT-PARA.
        MOVE MESSAGE TO TITLE.
        WRITE OUTPUT-RECORD.
END-OF-JOB.
        CLOSE DATA-FILE.
        CLOSE OUTPUT-FILE.
        STOP RUN.
```

It can be easily seen from this example that COBOL is a self-documenting language. Self-documenting languages are those that do not require much explanation in order to be understood by some one reading the program instructions.

The self-documenting aspect of COBOL is made possible by its English like sentence and paragraph structure and the very generous maximum symbolic field-name length of 30 characters. With a field-name length of up to 30 characters, the name can clearly identify the field and its purpose. COBOL programmers should ensure that the field name that are used are meaningful, so that the self-documenting feature of the language is not lost.

Like FORTRAN, most COBOL compilers require that certain parts of every statement be placed in certain columns. There are 80 columns in a line out of which columns 1-3 are used for page number and columns 4-6 identify line numbers. The use of sequence numbers is optional and can be omitted. An asterisk (*) in column 7 indicates a comment line and the entry is not compiled to produce object code. Comment lines are actually some notes revealing the intentions of the programmer and are used for program documentation. The actual COBOL statements are placed in columns 8-72. Notice that there are two margins : A and B. The A margin that starts at column 8 is used to start a new division, section, or paragraph. The B margin that starts at column 12 is used to start any sentence. Finally, columns 73-80, which the computer ignores, can be used to write some identification.

BASIC

BASIC (*B*eginners *A*ll-purpose *S*ymbolic *I*nstruction *C*ode) was developed by Professors John Kemeny and Thomas Kurtz in the year 1964 at Darmouth College in the United States. Their purpose was to develop a language which would be very easy to learn and thus can be used by the undergraduate students in all fields of study. The language has few grammatical rules and can be learnt in a few hours of concentrated study. In order to understand and write programs in BASIC, it is not necessary to learn complex programming techniques. A person with little or no knowledge of computers or programming can learn to write BASIC programs in a short period of time. Because of its simplicity and bias towards the user, BASIC is even being used by school students. It is a language well suited for use in education and has become extremely popular with microcomputer users.

Unlike FORTRAN or COBOL, BASIC is an interpreter based language. Instead of compilers, interpreters are generally used in microcomputer systems to translate BASIC instructions into machine-language code. Thus, as a BASIC program is being entered, its statements are checked for syntax errors which can be immediately corrected. This feature of BASIC makes it one of the most popular computer languages used in microcomputer systems. It is available in almost all microcomputers and even in some pocket calculators. Though simple and easy to learn, BASIC is quite flexible and reasonably powerful. It can be used for both business and scientific applications. Probably the greatest drawback of this language is that it has not yet been standardized. The language varies significantly from one computer system to another. Thus, a BASIC program written on one computer may not work on another unless modified.

A Basic program to compute and print the sum of 10 numbers is given below :

```
5    REM PROGRAM TO COMPUTE
6    REM THE SUM OF 10 NUMBERS
10   LET S = 0
20   FOR I = 1 TO 10
30   READ N
40   LET S = S + N
50   NEXT I
60   PRINT "THE SUM OF GIVEN NUMBERS="; S
70   DATA 4, 20, 15, 32. 48
80   DATA 12, 3, 9, 14, 44
90   END
```

It can be observed from this example that a BASIC program is made up of a series of statements. Each statement starts with a statement number and a key word, followed in most cases by some type of action. For example, in the statement, "40 LET S = S + N", 40 is the statement number, LET is the key word, and S = S + N is the action. The first statement of the program which is a REM statement is a remark being made for the purpose of explaining or documenting a program step. It has no effect on the program logic. The instruction must have a line number, the key word REM, and any remark that the programmer wishes to make. In our example, the remark statement was used to name the program. The two DATA statements in the program are used to furnish the input data. The FOR and NEXT statements control the loop that is meant for adding the 10 numbers one by one. The END statement stops the program execution.

PASCAL

Named after the famous French mathematician Blaise Pascal, this language was first introducted in the year 1971 by Professor Niklaus Wirth of the Federal Institute of Technology in Zurich Switzerland. His aim was to develop a language after the concepts associated with structured programming. Thus, PASCAL was the first language to fully embody in an organised way the concepts of structured programming. The language is relatively easy to learn, and it allows the programmer to structure the programming problem. This means that the program must be written in logical modules which are in turn called by a main controlling module. In other words, PASCAL is designed to force us to look at a problem in a logical way and to lay out a solution before we begin writing the program. It is based on the theory that the use of too many GO TO statements in a program makes the program clumsy and unstructured. The logic of such programs becomes complex and these programs are difficult to understand and maintain because they lack any recognizable structure or flow of control. Hence, as a good programming practice, a programmer should avoid the use of GO TO statements in his program as far as practicable. The features of PASCAL help us in overcoming this problem. The language is designed in such a way that complete PASCAL programs can be written without the use of GO TO statements. The process of looping or repeating a sequence is handled automatically by special loop control statements. PASCAL is thus widely recognised as a language that instills good programming habits in a programmer. Owing to this reason, this language is also extensively used to teach programming to beginners. The features of PASCAL allow it to be used for both scientific and business applications. Hence, it is a very powerful language and has been implemented on several different computers including minicomputers and microcomputers. A PASCAL version of a summing program similar to those presented earlier in other languages is shown below.

```
PROGRAM SUMNUMS (INPUT, OUTPUT);
(* PROGRAM TO COMPUTE THE SUM OF 10
NUMBERS *)
(* DECLARATION OF VARIABLES *)
VAR SUM, N   : REAL;
VAR I : INTEGER;

BEGIN
 SUM := 0;
 FOR  I := 1 TO 10 DO
 BEGIN
  READ (N);
  SUM : = SUM + N;
 END;
 WRITELN ('THE SUM OF GIVEN NUMBERS=',
 SUM);
END.
```

The first line of the program contains the name of the program which is SUMNUMS. This is followed by two comment lines which are used for documentation purpose. Any comment can be placed within the symbols (* and *) to document a PASCAL program. Then, all the variables are declared. The variables SUM and N are declared as real and hence they can be assigned any real number. Similarly, the variable I, that has been declared to be an integer variable, can be assigned any integer value. The heart of the program starts with the word BEGIN and ends with the word END. First, the variable SUM is initialized to zero. The next statement starts a DO loop that reads and computes the sum of the 10 numbers. Finally, the statement having WRITELN prints the result of the program. It may be observed that PASCAL programs are composed of blocks starting with BEGIN and terminating with END. All variables are declared at the beginning of the program and program statements proceed in a logical flow from start to finish.

PL/I

PL/I stands for programming language one. It was designed in the mid-1960s by IBM as a general purpose language having features similar to COBOL for business applications and features similar to FORTRAN for scientific applications in addition to other features such as string manipulation and list processing. The intention was to create a universal language which would be adequate for programming any kind of application. A PL/I standard was produced by an ANSI committee in the year 1976. A subset of this full standard, known as PL/I-G is also available for use with small computers.

Although PL/I is one of the most versatile and the most powerful of the programming languages, it is not the most commonly used. The main reason behind this is that since PL/I has features found in both FORTRAN and COBOL, it is a sophisticated language. It is enormous by any standards and is not easy to learn in its totality. Furthermore, the complexity of the language makes a compiler and support packages for the full language quite large. Because of its size and heritage, the language so far has been available primarily on IBM equipment and is not very widely used or accepted.

A PL/I program for the summing problem is given below to illustrate the nature of the language.

```
SUMNUMS : PROCEDURE OPTIONS (MAIN);
/* PROGRAM TO COMPUTE THE SUM OF 10
NUMBERS */
DECLARE (SUM, N) FIXED;
DECLARE I FIXED;
SUM = 0;
DO I = 1 TO 10;
  GET (N);
  SUM = SUM + N;
END;
PUT EDIT ('THE SUM OF GIVEN NUMBERS =', SUM)
(A,F5);
END;
```

It can be seen from this example, that the basic element of a PL/I program is a statement. Each statement is terminated by a semicolon (;) and several statements may be combined to form a procedure. A procedure may represent an entire small program (as in this example) or a block or module of a more complex program. Because of its modular structure, a beginner need only learn a small part of the language in order to write programs for a particular type of application. Moreover, modular procedure blocks and other features of the language allow the use of the various concepts of structured programming.

OTHER HIGH-LEVEL LANGUAGES

The high-level programming languages that were discussed above are not necessarily the most important or most popular languages. These languages were presented in some detail to give you a better understanding of computer programming and nature of high-level languages in general. There are several other programming languages which are equally important and popular. Some of these languages are briefly discussed in this section.

RPG. Report program Generator (RPG) is a business-oriented, general purpose programming language. As the name implies, the language is designed to generate the output reports resulting from the processing of common business applications. The language was developed by IBM as a result of their customer requests for an easy and economic mechanism for producing reports and was launched in the year 1961 for use on the IBM 1401 computer. The latter version of RPG, called RPG II, greatly improved the language and gave it additional capabilities.

RPG is considerably different from other programming languages. Instead of writing instructions or statements, the programmer uses very detailed coding sheets to write his specifications about input, calculations, and output. These sheets specify exactly what is to be done, and then the computer uses them to generate the necessary instructions to perform the desired application. Thus, RPG is easier to learn and use as compared to COBOL. Moreover, RPG can duplicate any COBOL program. Owing to these reasons, RPG is commonly used on many small computers and in small businesses. It is well suited for applications where large files are read, few calculations are performed, and output reports are created. However, RPG has restricted mathematical capability and cannot be used for scientific applications.

ALGOL. Like FORTRAN, ALGOL (*ALGO*rithmic *L*anguage) is also one of the earliest and the most influential high-level languages that was designed for scientific applications. It was designed by an international group of mathematicians, and developed by groups in Europe and the United States. The language was first introduced in 1958, resulting in ALGOL58. Later on, it was revised in 1960 and this revised version, known as ALGOL60, was the most widely used version of the language. The most recent, and the most powerful, version is ALGOL68. Like PASCAL and PL/I, ALGOL is a block-structured or modular language that is well suited for use in a structured programming setting. One area in which ALGOL, as originally defined, is quite deficient is that of input/output. The language was primarily designed as a way of expressing algorithms, and I/O statements were not made a part of the official language. ALGOL has not proven to be as popular in the United States as in Europe for practical work, although it is widely used in universities, especially in computer science departments because of its elegance and power. The lack of I/O facilities, plus the fact that the largest computer manufacturer, IBM, did not favour ALGOL early in its existence, has contributed to its lack of widespread usage.

APL. APL (*A P*rogramming *L*anguage) was developed by Kenneth in 1962. It is a very powerful programming language that is well suited for specifying complex algorithms. Much of its power is vested in the rich set of mathematical operators available, enabling easy manipulation of matrices and arrays of highest rank. This language is a real-time language developed primarily for scientific applications. It is usually used in an interpretive and interactive manner - an environment which greatly enhances its power. Since APL uses a rather large and unusual character set, a special keyboard and terminal is necessary for its implementation. Until recently, only certain IBM equipment supported APL. However, the advent of low-cost terminals capable of handling many different type of fonts and the growing number of APL users have brought it into wider, but still limited, usage.

ADA. ADA is a new general purpose programming language. It was developed in the year 1980 at the request of the U.S. Department of Defence (DOD) for use in military applications. The language was developed at Honeywell Computer Company by a group of scientists headed by Ichbiah. It is named in honour of Lord Byron's daughter Ada Augustha Lovelace. Ada was a close friend of Charles Babbage, who has made significant contributions in some of the earlier developments of computer systems. Ada used to work with Babbage. She is considered by many to be the first "programmer" because she wrote the first computer program for the mechanical computer (Analytical Engine) developed by Charles Babbage.

Ada is an extremely complicated language with a very large number of features. In addition to the normal types of statements and commands, ADA also allows the use of packages. A package allows for a collection of related computational procedures and resources. Packages are specified or written into the declarative part of the program, which is typically at the top of an ADA program. Then various procedural statements in the program can access the package and use it. Another important feature of ADA is the use of tasks. Tasks are used to allow concurrent programming which is very useful for military applications. Like packages, tasks have a specification part. Once specified, tasks can be used within the body of the program as and when needed. It is expected that most future programs written for the Department of Defense, U.S.A. will have to be in this language. As a result, this new language may become very popular in the field of defense.

LISP. LISP stands for LISt Processing. This language was designed by McCarthy and is suitable for nonnumeric applications. It is a powerful language for handling logical operations. Because of this feature, the language is extensively used in the areas of pattern recognition, artificial intelligence, and for simulation of games.

SNOBOL. SNOBOL (*S*triNg *O*riented *S*ymBOlic *L*anguage) is another language used for nonnumeric applications. As the name implies, the language was basically designed to manipulate strings of characters. SNOBOL has powerful string manipulation features that facilitate various types of operations on strings of characters such as string comparison, splitting of a string, combining two strings, etc. Thus, this language has been widely accepted for applications in the area-of text processing.

C. C is a relatively new language which is becoming very popular day-by-day. It was designed at Bell Telephone Laboratories, U.S.A. Like PASCAL and ALGOL, C is a block structured language and has several features that allow the use of the various concepts of structured programming. Moreover, a special feature of this language, that is normally absent in other high-level languages, is that it allows the manipulation of internal processor registers of the computer. Thus, the language also enjoys the advantage of having some of the powers of assembly language. Because of this feature, C language is now being extensively used for systems programming like design of compilers and operating systems. Most computer vendors of today, supply this language along with their computer systems.

PROLOG. PROLOG stands for *PRO*gramming in *LOG*ic. It is a very new programming language designed for handling complex logical operations. The language is being used to design intelligent computer systems and is expected to gain popularity in the near future.

CHARACTERISTICS OF A GOOD LANGUAGE

In the previous section, we have seen that there are some high-level languages which are very popular and there are others which are used only by a small group of programmers. Why do programmers prefer one language over another? One obvious reason is the area of application. However, another equally important reason is the characteristics of the language. Several properties believed to be important with respect to making a language good and usable by human beings are briefly outlined below.

1. *Simplicity*. Programming languages that are simple and easy to learn and use are liked by many programmers. For example, BASIC is used by many programmers only because of its simplicity. Thus, a language should provide a programmer with a clear, simple, and unified set of concepts which can be easily grasped. There should be a minimum number of different concepts, with the rules for their combination being as simple and regular as possible. However, the power needed for the language should not be sacrificed for simplicity.

2. *Naturalness*. A language should be natural for the application area it has been designed. In other words, it should be problem-oriented. It should provide appropriate operators, data structures, control structures, and a natural syntax in order to facilitate the users to code their problem easily and efficiently. Often if a major amount of programming in a particular area is required, it is extremely useful to develop a programming language just for that class of applications. FORTRAN and COBOL are good examples of scientific and business languages respectively that possess high degree of naturalness.

3. *Efficiency*. Efficiency is certainly a major element in the evaluation of any programming language. Thus, while designing a compiler or an interpreter for a particular language, system programmers must give due consideration to space and time efficiency. A programming language should be such that its programs are efficiently translated into machine code, are efficiently executed, and acquire as little space in the memory as possible.

4. *Structuredness*. Structuredness means that the language should have necessary features to allow its users to write their programs based on the concepts of structured programming. The main reason behind this is that, this property of a language greatly affects the ease with which a program may be written, tested, and maintained. Moreover, it forces a programmer to look at his problem in a logical way and hence he creates fewer errors while writing a program for his problem. PASCAL, PL/I, and ALGOL are some of the languages having this property.

5. *Compactness*. Users of a high-level, problem-oriented language should be able to express intended operations concisely, since this is one of the fundamental reasons for having it. A verbose language can tax the programmer's sheer writing stamina and thus reduce its usefulness. COBOL is generally not liked by many programmers because it is verbose in nature and lacks compactness.

6. *Locality*. A programming language should be such that while writing a program, a programmer need not jump around visually as the text of the program is prepared. COBOL lacks locality because data definitions are separated from processing statements, perhaps by many pages of code. On the other hand, APL is very local since it requires no declarations, thus permitting the programmer to concentrate almost solely on the part of the program around the statement currently being worked with.

7. *Extensibility*. A good programming language should also allow extension through simple, natural, and elegant mechanisms. Almost all

languages provide subprogram definition mechanisms for this purpose, but there are some languages that are rather weak in this aspect.

8. *Suitability to Environment*. Depending upon the type of application for which a programming language has been designed, the language must also be made suitable to its environment. For example, a language designed for real time applications must be interactive in nature. On the other hand, languages used for data processing jobs like pay-roll, stores accounting, etc., may be designed to be operative in batch mode. A language designed for interaction, say APL, is not as usable in batch mode. Likewise, a language designed for batch usage may prove quite frustrating if used interactively from a terminal.

SUBROUTINES

As we are going to abandon the chapter on program writing, it is felt essential to introduce the concept of subroutines which are very frequently used in programming.

Suppose we are writing a program for solving a trignometric problem. During the problem, suppose we need to calculate the square root of a number three times. Should we write the square root procedure into our program thrice? Why can we not write the square root procedure just once, and use it every time we need it?

We could of course write the steps required for the calculation of square root each time we need them in the program. But a much easier way of handling the problem is to write these steps once and then refer to them each time we need them.

Subroutines (also called subprograms) are programs written in such a way that they can be brought into use in other programs and used whenever needed, without rewriting. In other words, a subroutine is any standardized program written in such a way that it can be used as part of another program whenever necessary. A subroutine is normally invoked through other programs by the use of CALL statements.

There are many subroutines such as those for finding square roots, sines, cosines, logarithms, etc. which are used over and over by many programmers. Such subroutines are usually supplied by the computer manufacturers along with the language compiler and are referred to as *built-in functions*. Other subroutines can easily be written and used as and when we need them.

QUESTIONS

1. Discuss the analogy between a computer language and a natural language.
2. How does a computer language differ from a natural language ?
3. Name the three different categories of computer languages
4. What is a machine language ? Why is it required?
5. What are the advantages and limitations of machine language ?
6. When is a computer language called machine dependent ? What is the main disadvantage of such a language?
7. A machine language instruction has a two-part format. Identify these parts and discuss the function of each.
8. What is a mnemonic ? How is it helpful in case of computer languages ?
9. What is an assembly language ? What are its advantages over machine language ?
10. What is an assembler ?
11. What is the difference between a source program and an object program ?
12. What is a macro instruction ? How does it help in reducing a programmer's job ?
13. What are high-level languages ? Why are they known as problem-oriented languages ? Name some high-level languages.
14. Why are high-level languages easier to use ?
15. What is a compiler ? Why is it required ? A computer supports five high-level languages ? How many compilers will this computer have ?
16. Illustrate the machine independent characteristic of high-level languages.

17. What type of errors can be detected by a compiler ? What type of errors cannot be detected ?

18. What is an interpreter ? How does it differ from a compiler ?

19. List out the characteristics necessary for a programming language to be considered as a high-level language.

20. What are the advantages and limitations of high-level languages ?

21. Briefly describe the development and use of the following programming languages :

    (a) FORTRAN
    (b) COBOL
    (c) BASIC
    (d) PASCAL
    (e) PL/1
    (f) ADA
    (g) RPG
    (h) ALGOL
    (i) APL
    (j) SNOBOL
    (k) LISP

22. What characteristics are desirable for a good computer language ?

23. What is a subroutine ? How do subroutines help in program writing ?

24. Would you be equally likely to choose FORTRAN or COBOL for a given task ? Why ?

25. What is the purpose of a language processor ? Are language processors hardware or software ?

26. While writing a program, a programmer erroneously instructed the computer to calculate the area of a rectangle by adding the width to its length (that is, AREA = LENGTH + WIDTH) instead of multiplying the length and width. Would you expect the language processor to detect this error ? Explain.

27. A programmer eliminates all language processor errors from his program and then runs it to get printed results. The programmer therefore concludes that the program is complete. Comment.

28. It is said that an assembly language is "one-for-one" but a high-level language is "many-for-one". Explain what this means.

29. What is meant by standardization of a language? Why is it important ?

30. What is the role of comments in a program, and how are they treated by the language processor ?

31. What is a self-documenting language ? Illustrate with an example.

32. List out some of the program preparation techniques that are often included under the term "structured programming".

33. Give the full form of the following terms :
    (a) FORTRAN
    (b) COBOL
    (c) BASIC
    (d) PL/1
    (e) RPG
    (f) ALGOL
    (g) APL
    (h) LISP
    (i) SNOBOL

CHAPTER 13

13. SYSTEM IMPLEMENTATION AND OPERATION

In the previous two chapters, we have discussed the analysis, design, and coding phases of a computerised system. After the computer programs have been prepared, the computer information system enters the implementation and operation phase. The goal of this chapter is to describe the principal activities of the implementation and operation phase, which relate to testing and debugging of programs, complete documentation of the system, changeover to the new system, and system modification and maintenance.

TESTING AND DEBUGGING

So long as computers are programmed by human beings, computer programs will be subject to errors. Program errors are known as *bugs* and the process of detecting and correcting these errors is called *debugging*. In general, *testing* is the process of making sure that the program performs the intended task, and *debugging* is the process of locating and eliminating program errors. Testing and debugging are vital steps in developing computer programs. They are also time-consuming steps. In fact, the time spent in testing and debugging often equals or exceeds the time spent in program coding.

In general, there are two types of errors that occur in a computer program - *syntax errors* and *logical errors*. Syntax errors result when the rules or syntax of the programming language are not followed. Such program errors typically involve incorrect punctuation, incorrect word sequence, undefined terms, or misuse of terms. For example, the FORTRAN statement C = (A + B/2 has a syntax error. In this example, the problem is a missing closing parenthesis which should be placed in the appropriate place depending on the logic of the program.

All syntax errors must be found and corrected before there is any chance of running the program. Almost all language processors are designed to detect syntax errors. The language processors print error messages on the source listing that indicate the number of the statement with errors and give hints as to the nature of the error. These error messages are very useful and are used by the programmers to rectify all syntax errors in their programs. Thus, it is a relatively easy task to detect and correct syntax errors.

It should be noted that in high-level languages such as FORTRAN and COBOL a single error often causes multiple error messages to be generated. There are two reasons for multiple error messages. One is that high-level language instructions often require multiple machine steps. The other reason is that symbolic instructions are often dependent upon other instructions and if an instruction containing an error is one that defines a field name, all instructions in the program using that field name will be listed as errors. The error message will say that a field being used is not a defined name. In such a case, removal of the single error will result in the removal of all associated error messages.

The second type of error, a *logical error*, is an error in planning the program's logic. In this case, the language processor successfully translates the source code into machine code. The computer actually does not know that an error has been made. It follows the program instructions and outputs the results, but the output is not correct. The problem is that the logic being followed does not produce the results that were desired. When a logical error occurs, all you know is that the computer is not printing the correct output. The computer does not tell you what is wrong. For example, if a FORTRAN instruction should be "A = B*C" but has been coded as "A = B+C", this error will not be detected by the language processor since no language rules have been broken. However, the output produced will not be correct. Thus, it is an example of a logical error.

In order to determine whether or not a logical error exists, the program must be tested. The purpose of testing is to determine whether the results are correct. The testing procedure involves running the program to process input test data that will produce known results. By running the program and comparing the obtained answers to the known correct results, the accuracy of the program logic can be determined. Logic errors are typically due either to missing logic or to incorrect logic. If the logic is wrong, the answers generated from the test data will be wrong. These errors are the easiest of the logic errors to find. Errors caused by missing logic result from logical situations that the program was not designed to handle. As an example, suppose that a numeric field is to be used in an arithmetic process and that the data entry operator enters a value for the field that is not numeric. The program logic should determine that the data are not numeric prior to attempting the arithmetic process. If that logic is missing and nonnumeric data is used in an arithmetic operation, the program will fail. This type of bug can be difficult to find. The only way for this error to occur is for nonnumeric data to be entered into a numeric field. It is possible for the program to be used for weeks, months, or years before this happens and the error in program logic shows up.

In order to completely test the program logic, the test data must test each logical function of the program. Hence, the selection of proper test data is important in program testing. In general, the test data selected for testing a program should include :

1. Normal data, which will test the generally used program paths.

2. Unusual but valid data, which will test the program paths used to handle exceptions. Such data might be encountered occasionally in running the program.

3. Incorrect, incomplete, or inappropriate data, which will test the error-handling capabilities of the program. This is done to see how the program reacts in abnormal and unusual circumstances. Good programs should be able to handle abnormal data without blowing up or generating meaningless output.

If a program runs successfully with the test data and produces correct results, it is normally released for use. However, even at this stage errors may still remain in the program. In case of a complex problem, there may be thousands of different possible paths through the program and it may not be practical or even possible to trace through all these paths during testing. There are certain errors in complex systems that remain dormant for months and years together and hence it is impossible to certify that very complex systems are error-free.

There are several ways to locate and correct logical errors some of which are briefly described below :

1. One approach is to study the source code producing the incorrect results and try to determine the cause of problem. However, some errors are difficult to find, and simple inspection of the source program does not reveal the nature of the error.

2. Another approach is to put several print or write statements in the program that indicate the values of intermediate computations. These statements can also be used to tell you what is happening during the execution of the program. Once the errors have been found and corrected, these print or write statements are removed from the program.

3. The third approach involves the use of tracing routines which are software tools provided to the programmer to help in debugging the program logic. *Tracing routines* or *debug packages* assist the programmer in following the logic by printing out intermediate calculation results and field values that are used in making logical decisions in the program. Using these techniques, the programmer can follow the program's execution step by step in order to determine where the logic is in error.

4. In a few cases, the logical error can be so difficult to find, and the number of fields involved are so numerous, that the only way to uncover the source of the error is to look at a printout of the contents of memory. This printout is called a *storage dump* or *memory dump*. The memory dump lists the instructions and data held in the computer's memory in their raw form, that is, their binary or equivalent hexadecimal or octal form. The programmer can then study this listing for possible clues to the cause of the programming error(s). Most programmers of high level languages resort to the use of memory dumps only when all other methods of detecting the logical error fails.

Difference Between Testing and Debugging

Testing and debugging are two separate tasks. They should not be confused with each other. The differences between these two processes are outlined below :

TESTING	**DEBUGGING**
1. Testing is a process in which a program is validated.	Debugging is a process in which program errors are removed.
2. It is a positive activity that seeks to demonstrate that the program is correct and does, in fact, meets its design specifications.	It is a negative activity in that it is centered around elimination of known errors or bugs.
3. Testing is complete when all desired verifications against specifications have been performed.	Debugging is finished when there are no known errors. However, debugging is a process that ends only temporarily, because subsequent execution of a program may uncover other errors - thereby restarting the debugging process.
4. Testing, can and should be planned. It is a definable task in which the how and what to test can be specified. Testing can be scheduled to take place at a specific time in the development cycle.	Because debugging is a reactive procedure which stems from testing, it cannot be planned ahead of time. The best that can be done is to establish guidelines of how to debug and develop a list of "what to look for."
5. Testing can begin in the early stages of the development effort. Of course the tests themselves must be run near the end of a project, but the decisions of what to test, how to test, with what kind of data, can and should be completed before the coding is started.	Debugging, on the other hand, cannot begin until the end of the development cycle, because it requires an executable program.

DOCUMENTATION

A computerised system cannot be considered to be complete until it is properly documented. In fact, documentation is an on-going process that starts in the study phase of the system and continues till its implementation and operation phase. Moreover, documentation is a process that never ends throughout the life of the system. It has to be carried out from time to time as and when the system is modified during its maintenance phase.

System documentation involves collecting, organizing, storing, and otherwise maintaining a complete historical record of programs and other documents used or prepared during the different phases of the system. Proper documentation of a system is necessary due to the following reasons :

1. It solves the problem of indispensibility of an individual for an organisation. In large organisations, over the years, the designer and developer of a software system may not be in the same organisation. In such cases, even if the person who has designed or programmed the system, leaves the organisation, the documented knowledge remains with the organisation which can be used for the continuity of the system.

2. Maintainability of computer software poses a concern to major data processing installations. The key to maintenance is proper and dynamic documentation. A well documented system is easy to modify and maintain in the future. It is easier to understand the logic of a program from the documented records rather than its code. System flowcharts, program flowcharts, or comments used within the programs prove to be very helpful in this regard.

3. Documented records are quite helpful in restarting a project that was postponed due to some reason or the other. The job need not be started from scratch and the old ideas may still be easily recapitulated which saves lot of time and avoids duplication of work.

Documentation of a software system is normally provided in the following forms :

1. *Comments.* Comments are very useful aid in documenting a program. From maintenance point of view, comments have been considered to be a must. Comments are used within a program to assist anyone reading the source program listing. They are used to explain the logic of the program. They do not contain any program logic and are ignored (not translated) by the language processor.

Comments should be used intelligently to improve the quality and understandability of the program. They should not be redundant, incorrect, incomplete, or written in such a way that cannot be understood by anyone else. For example, a redundant comment for the statement N = N+1 would be "INCREMENT N BY 1". Useful comments are those which describe the meaning of a group of statements such as "READ AND ECHO PRINT THE INPUT DATA". In other words, comments should mediate between the program and the problem domain. Almost all high level languages provide the facility of writing comments along with the source code of a program and it is suggested that programmers should liberally use this facility for proper documentation of their programs.

2. *Systems Manual.* A good software system must be supported with a standard systems manual that contains the following information :

a. A statement of the problem clearly defining the objectives of the computerised system and its usefulness for various categories of users.

b. A broad description of the system specifying the scope of the problem, the environment in which the programs function, the system limitations, the form and type of input data to be used, and the form and type of output required.

c. Specific program names along with their description and purpose.

d. Detailed system flow charts and program flow charts cross referenced to the program listing.

e. Narrative description of the program listings, the processing and calculations performed and the control procedures.

f. A source listing of all the programs together with full details of any modifications made since.

g. Description and specifications of all input and output media required for the operation of various programs.

h. Specimen of all input forms and printed outputs.

i. File layout, that is, the detailed layout of input and output records.

j. The structure and description of test data and test results, storage dumps, trace program printouts, etc., used to debug the programs.

3. *Operation Manual.* A good software package must be supported with a good operation manual to ensure the smooth running of the package. It is the operator who will perform the regular processing after the system gets stabilised and not the programmer who has developed the package. Hence, the operation manual must contain the following information :

a. Set up and operational details of each program.

b. List of computer switches, their location, setting and purpose.

c. Loading and unloading procedures

d. Starting, running, and terminating procedures.

e. A description and example of any control statements that may be used.

f. All console commands along with errors and console messages that could arise, their meaning, reply and/or operation action.

g. List of error conditions with explanations for their re-entry into the system.

h. List of programs to be executed before and after execution of each program.

i. Special checks if any and security measures, etc.

The importance of program documentation cannot be over emphasized. There have been too many problems in the past with poorly documented systems. The result is usually errors and problems with the computer programs at a later date. It is very difficult to incorporate modifications in such systems and hence they are very expensive to maintain. Owing to these reasons, several computer installations develop strict documentation standards. These standards describe in detail how documentation is to be performed and what reports and outputs are necessary for documentation to be completed successfully.

CHANGEOVER TO THE NEW SYSTEM

Once the programs are tested and appear to be producing correct results, the system conversion and changeover begins. At this stage, the old system, if any, is phased out and the new system is phased in. The changeover process normally involves the following operations :

1. Training of personnel. Everyone who will be affected by the new system should receive some training to become familiar with the changes. This training should be both system training and user training. The overall purpose of system training is to train members of the data processing department on various technical aspects of the new system. On the other hand, user training is designed to allow managers, decision makers, and other users to become familiar with the new system so as to increase their involvement and participation in the new system.

2. Replacement of old forms and operation procedures by new ones.

3. Retirement of old input and output devices in favour of new hardware.

4. Incorporation of necessary changes in manual methods and assignment of new jobs to various personnel.

5. It is also necessary to convert all the data files from manual to computer files. That is, current files are changed into a form acceptable to the processor. Files should be consolidated and duplicate records eliminated. File inconsistencies or any errors in current files must be detected and removed.

There are three different methods normally followed to carry out the changeover process. These are immediate changeover, parallel run and phased conversion. Each of these methods has its own advantages and disadvantages.

No single method is suitable for converting all types of systems and the choice of a particular method largely depends upon the prevailing conversion circumstances. These methods are briefly described below :

IMMEDIATE CHANGEOVER

As illustrated in Figure 13.1(a), in this method of conversion, the operation of the old system is totally abandoned from the date of changeover and the complete new system starts operating from that day onwards. This method of changeover is very risky because it has been found that most systems pose some problem or the other during the changeover process. The inbetween failure of the new system due to any such problem may prove to be very harmful causing total breakdown, and the work cannot progress at all because the operation of the old system has already been abandoned. However, this method is preferred in situations where changeover time is very less, available manpower is also less, and the system is not a very critical one to business operations so that changeover problems would not trigger a disaster.

PARALLEL RUN

One of the best ways to implement a new system is called running in parallel. As shown in Figure 13.1(b), in this method, the old system is operated with the same data as the new system and on the same time schedule for the initial three or four cycles.

Confidence in the adequacy of the new system is normally established by comparing the data it produces with the data produced by the old system during the time of parallel operation. Some discrepencies may be discovered. Often these are due to inaccuracies in the old system that were not recognized before as inaccuracies. Unforeseen exceptions will appear for which no programming was provided. Some of the discrepencies that appear during the parallel operation stage will be due to oversights and mistakes in the programming itself. These must be corrected by further debugging before the conversion is complete.

The main advantage of parallel run is the availability of old system as a backup. If there are any problems with the new programs, they can be corrected while the existing system is still being used. Thus, there is no interruption of service if there are problems with the new programs. After the bugs are removed, the new system is slowly phased in, while the old system is slowly phased out.

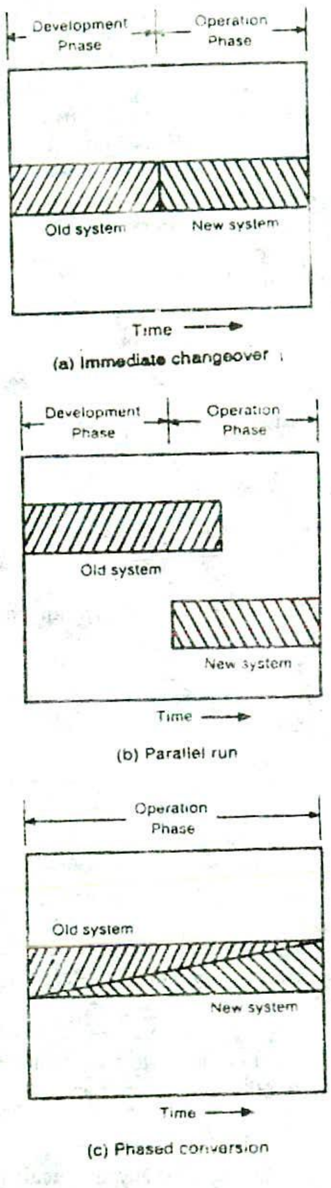


Figure 13.1. Methods of system changeover to the new system.

This method, however, is not preferred in some cases because it is very expensive and time consuming. Additional manpower, which is a scarce resource in most organizations, must be provided for the operation of two systems in parallel. Due to the requirement of additional manpower and equipment resources for parallel run, the organization is under considerable strain during the period of parallel operation, and, if it is long continued, organizational breakdowns tend to occur. To hold costs in line, experience indicates that parallel operation must not be carried on any longer than needed to establish confidence in the new system. Continuing them too long is a sign of weakness in the new system. Moreover, parallel run method of system conversion is also not used in situations where the new system differs to a great extent from the old system in the functions that it performs and its input and output.

PHASED CONVERSION

As depicted in Figure 13.1(c), in this method, the complete changeover to the new system takes place incrementally over a period of time. The new system is gradually implemented part by part and the old system is gradually phased out. The results produced by each part of the new system is compared against the results of the old system. Any discrepencies or errors found are checked and removed. Once confidence is developed in a particular part of the new system, that part of the new system is phased in and the corresponding part (operations) of the old system is phased out. This approach is continued for each and every part of the new system. Thus, over a period of time, the new system is slowly-slowly phased in, while the old system is slowly-slowly phased out.

This method of changeover enjoys several advantages. It is not as expensive as the parallel run method because the changeover process being gradual can usually be handled with existing manpower and equipment resources. There is no danger of interruption of service if there are problems with the new programs because the corresponding part of the old system is still in operation. Moreover, the users get sufficient time to become acquainted with the new system. Hence, they can confidently handle the new system when the complete system is handed over to them. However, the phased conversion method cannot be used in situations where the time period supplied for conversion process is very less or when the new system significantly differs from system.

SYSTEM EVALUATION

Once the new system is implemented and in operation, it is necessary to evaluate the system to verify whether or not it is meeting its performance objectives. These performance objectives of the system are clearly stated during its study phase. The post implementation system evaluation is normally carried out by people who have an independent view point and are not responsible for the development and maintenance of the system. While evaluating a system, the following points are considered :

1. *Evaluation of efficiency* : The new system is compared against the old system to evaluate its efficiency in comparison to the old one. In case of any slack, the reason is analysed and if possible, necessary modifications are incorporated in the system to rectify it.

2. *Cost/benefit analysis* : The key to evaluating a system is the costs and benefits to be derived from alternate systems. This can be a long and expensive process. Information can be assembled on the actual cost of using the new system, and this cost can be compared with the anticipated cost as outlined in the report of the feasibility study group. If discrepancies are found on the high side, action can be taken to find out the reasons and then to correct or offset the causes.

   Of course, benefits are not all tangible and measurable. However, those that are measurable should be expressed numerically. As a general rule, whenever we identify a tangible specific objective, we also should state the measurement that we will make in the operation phase to determine whether or not the system meets the objective. Mathematical measurements can also be made of the cost savings made possible by a computerised system, and hardware and software operating expenses can be monitored continually.

3. *Time schedule* : It should also be evaluated whether the various time schedules prepared during the study phase in the beginning have been met or not. This type of evaluation is quite helpful in preparing time schedules for the new systems that will be designed in future. Moreover, it should also be evaluated whether the planned processing procedures are being followed or not. Are all new procedures being processed on the computer? Have all old procedures been eliminated? If not, why not?

4. *Users satisfaction* : People are the final evaluators of information systems. Hence, it should be found out whether the users are satisfied with the new system or not. How useful is the system for them? How enthusiastic are they about the service they receive? Do they receive outputs in time to take necessary action? The morale of employees using or affected by a system is a good measure of the success of the project.

5. *Ease of modification* : Sooner or later, all systems must change in response to changes in their environment. New laws, changes in technology, and changes in the goals and objectives of the business are examples of causes of change. Thus, the ease with which a system can be modified to react to changes is also a significant measure of its effectiveness in achieving its objectives.

6. *Error rate* : The frequency of failure of the system should also be evaluated. This should be reduced to minimum.

SYSTEM MAINTENANCE

No matter how good the new system is, how well it was installed and how well it may be operating, changes in business operations will force changes in the system. Changing business conditions, revised user needs, new laws, changes in technology, etc. are some factors which require that production programs be continually maintained and modified. The major cause of program maintenance is due to user requests, normally for program enhancements. As a manager uses a computer program, there is a tendency to demand additional reports and outputs from the program. Changes in data storage and organization, program bugs, and emergency program repairs are other important causes for maintenance. The remaining program maintenance is due to hardware changes, system software changes, enhancing program documentation, and sheduled and routine debugging.

Several studies have shown that, on an average, application programmers and system analysis personnel spend over half their time on program maintenance. Hence, program maintenance is an important duty of programmers and may involve all steps from problem definition through analysis, design, and program preparation. In some installations there are programmers who do nothing but maintain production programs. In fact, in many organizations well over half the total programming effort is spent on maintenance. And it is estimated that over the life cycle of a typical application, the maintenance and enhancement costs that are incurred may be two to four times larger than the initial development costs.

Frequent change is disruptive and disturbing. Therefore, some control over changes is required. One method of achieving this control is to have all requests for change evaluated by a change control board. This board should be made up of the principal users of the system, a system analyst, and data processing personnel who are familiar with the system. Normal maintenance operations need not be approved by the change control board, but these operations should be recorded and summarized for periodic reporting to the board. Examples of maintenance activities are modifying the format of a report or rewriting a part of a computer program component to improve its efficiency. Major changes are those that significantly alter the system or require extensive personnel, hardware, or software. An example of a major change would be conversion of the system from batch processing to online terminals.

When programs are modified, it is important to make sure that program documentation is also changed accordingly. Without the existence of proper documentation that is consistent with the programs, future changes would be very difficult and costly to accomplish.

QUESTIONS

1. What are the two types of errors that can occur in a computer program ? Give an example of each to illustrate their nature.

2. How are syntax errors detected and corrected ?

3. How are logical errors detected and corrected ?

4. Is it easier to detect a syntax error or a logical error ? Give reasons for your answer.

5. Why should a program be tested ?

6. What are the different types of test data that should be selected for testing a program ?

7. Why is it not possible for a very complex system

to certify that it is error free ?

8. What is a memory dump ? How is it useful for a programmer ?

9. Differentiate between testing and debugging.

10. Why is system documentation necessary ?

11. Discuss the different types of system documentation normally used for documenting a system.

12. What type of operations are normally carried out in the system changeover process ?

13 Discuss the three different methods of system changeover along with their advantages and disadvantages.

14. What are the various factors that should be evaluated during the system evaluation process ?

15. Why is system maintenance required ? Why is it considered an important process ?

16. How can frequent program modifications be controlled ?

CHAPTER 14

14. OPERATING SYSTEMS

In the last few chapters we have dealt with the planning, coding, operation, and maintenance of software systems. In this chapter, you will learn about a very important and special type of software that falls under the category of systems software. This systems software is known as operating system. The goal of this chapter is to introduce the concepts related to operating systems and to show how this particular systems software is used to make the computer a useful, easy-to-use tool.

DEFINITION AND FUNCTIONS

An *operating system* (OS) is an integrated set of programs that is used to manage the various resources and overall operations of a computer system. It is designed to support the activities of a computer installation. Its prime objective is to improve the performance and efficiency of a computer system and increase facility, the *ease* with which a system can be used. Thus, like a manager of a company, an operating system is responsible for the smooth and efficient operation of the entire computer system. Moreover, it makes the computer system *user friendly*. That is, it makes it easier for people to interface with and make use of the computer.

Operating systems go by many different names, depending on the manufacturer of the computer. Other terms used to describe the operating system *are monitor, executive, supervisor, controller*, and *master control programs*. No matter by which name they are called, today most operating systems perform the following functions :

1. Processor management, that is, assignment of processors to different tasks being performed by the computer system.

2. Memory management, that is, allocation of main memory and other storage areas to the system programs as well as user programs and data.

3. Input/Output management, that is coordination and assignment of the different input and output devices while one or more programs are being executed.

4. File management, that is, the storage of files on various storage devices and the transfer of these files from one storage device to another. It also allows all files to be easily changed and

modified through the use of text editors or some other file manipulation routines.

5. Establishment and enforcement of a job priority system. That is, it determines and maintains the order in which jobs are to be executed in the computer system.

6. Automatic transition from job to job as directed by special control statements.

7. Interpretation of commands and instructions.

8. Coordination and assignment of compilers, assemblers, utility programs, and other software to the various users of the computer system.

9. Establishment of data security and integrity. That is, it keeps different programs and data in such a manner that they do not interfere with each other. Moreover, it also protects itself from being destroyed by any user.

10. Production of dumps, traces, error messages, and other debugging and error-detecting aids.

11. Maintenance of internal time clock and log of system usage for all users.

12. Facilitates easy communication between the computer system and the computer (human) operator.

An operating system performs a wide variety of jobs. Each of these jobs are performed by one or more computer programs and all these computer programs are jointly known as an operating system. Out of the complete operating system, normally, one control program resides in the main memory of the computer system. This control program is known as the *resident program* or the *resident routine*. The other programs are stored on the disk and are called *transient programs* or *transient routines*. These programs include utility programs, compilers, assemblers, etc. The control program transfers these programs into the main memory and executes them as an when they are needed. It may be recalled here that the capacity of the main memory of any computer system is very small as compared to its secondary storage devices like disks. This is because main memory is very expensive as compared to secondary storage devices. This is the reason why only the control program is stored in main memory and the rest of the operating system is stored on disks.

In effect, besides the hardware, each computer system consists of an operating system that enables a user to effectively use the system. Thus, as shown in Figure 14.1, the OS tends to isolate the hardware from the user. The user communicates with the OS, supplies application programs and input data, and receives output results. However, it is interesting to known that all the tasks performed by the OS are performed automatically. The functions of the OS are transparent to the user - he really is not at all concerned with what the OS is doing or how the OS directs the hardware to handle certain tasks.

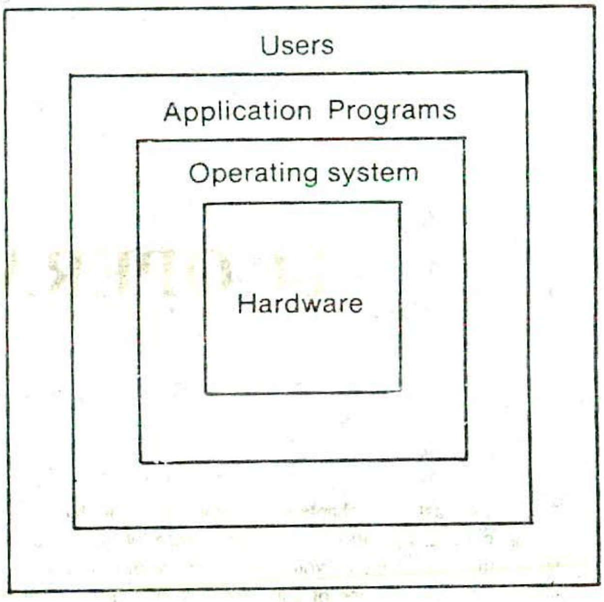


Figure 14.1. The in-between software layers isolate the hardware of a computer system from its users.

The efficiency of an operating system and the overall performance of a computer installation is judged by a combination of two main factors. They are :

1. *Throughput*. It is the total volume of work performed by the system over a given period of time.

2. *Turnaround Time*. It is also known as response time and is defined as the interval between the time a user submits his job to the system for processing and the time he receives results. Response time is especially important where

many different users share the use of the system and the overall progress of their work depends upon their receiving prompt results from the system.

EVOLUTION OF OPERATING SYSTEMS

It is believed that one of the first operating systems was developed in the early 1950s for the IBM 701 computers. This OS was elementary in nature and was not so powerful as the operating systems of todays computers. Since then, lot of research work has been carried out in this direction with the result that today we have very powerful operating systems which are machine independent and can execute several jobs at a time on the same machine. The main aim of all the researchers involved in the development of OS was to devise ways to minimise the idle time of the computer system and to use the computer system in the most efficient and economical way.

In the early days of computers, job-to-job transition was not automatic. For each and every job to be executed by the computer, the operator had to clear the main memory to remove any data remaining from the previous job, load the program and data of the current job from the input devices, set the appropriate switches, and finally run the job to obtain the results from the output devices. After the completion of one job, the same process had to be repeated for the next job by the computer operator. Because of the manual transition from one job to another, lot of computer time was wasted since the computer remained idle while the operator loaded and unloaded jobs. In order to reduce this idle time, a method of automatic job-to-job transition was devised. With this facility, when one job is finished, the system control is automatically transferred back to the operating system which automatically performs the housekeeping jobs needed to load and run the next job.

The automatic job-to-job transition facility provided by the OS reduced the idle time of the computer to a great extent. But still there was another scope for reducing the idle time of the CPU. If you remember properly, we have seen in Chapter 9 that the speed of CPU is much more as compared to the speed of I/O devices. Hence, the CPU was normally idle while a particular job was busy with some I/O operations. So the next attempt by OS developers was to overcome this speed mis-match by executing more than one program at the same time. In this method, while one program was busy with some I/O operation, the CPU time was utilised for processing another job.

In a similar manner, there have been many improvements in the operating systems of early days. A modern OS is very sophisticated and does much more than what we have discussed above. In the next few sections, we will discuss some of the common concepts and terms related to the operating systems of modern computers.

BATCH PROCESSING

Batch processing is one of the oldest methods of running programs that is still being employed by many data processing centres for processing their jobs. It is based on the idea of automatic job-to-job transition facility provided by almost all operating systems. In a batch mode, each user prepares his program off-line and submits it to the computer centre. A computer operator collects the programs which have been punched on cards and stacks one program or job on top of another. When a batch of programs have been collected, the operator loads this batch of programs into the computer at one time where they are executed one after another. Finally, the operator retrieves the printed outputs of all these jobs and returns them to the concerned users.

Batch processing is also known as serial, sequential, off line, or stacked job processing. When a computer is used in this way, the input data (and often the program) are introduced into the computer and processed automatically, generally without operator's intervention. Often many different jobs (or sets of data) are processed, one right after another, or even at the same time, but without any interaction from the users during program execution.

The method of batch processing reduces the idle time of a computer system because transition from one job to another does not require operator intervention. Moreover, it is the most appropriate method of processing for many types of applications such as payroll or preparation of customer statements where it is not necessary to update information (records) on daily basis. However, batch processing suffers from several disadvantages which are as follows :

1. It reduces timeliness in some cases. The time required to accumulate data into batches, in some instances, destroys much of the value of the data. The information that results from eventual processing is no longer timely.

2. Though efficient from the computer's point of view, batch processing makes each job wait in line at each step and often increases its turnaround time.

3. In batch processing, it is difficult to provide the desired priority scheduling. For example, if two high priority jobs were to be run but were in

separate batches, one would have to wait until the other's batch was completely processed.

JOB CONTROL LANGUAGE (JCL)

We have just now seen that in batch processing, a set of jobs are stacked together and fed to the computer system. But the obvious question that arises to ones mind is that how the computer separates each job for automatic job-to-job transition. Moreover, how does the system know which compiler or what hardware devices are to be used by a particular job ? In order that the operating system can identify a new job and determine what action should be taken for the job, some control information is necessary. These control statements are written in a language known as the *job control language* (JCL). Usually every program and data sets are preceded and followed by JCL statements.

When a program is prepared for a computer run, it is necessary to prepare job control statements and place them in proper order along with the program, before the program is fed to the computer system. Thus, each program has, besides the program itself, a set of instructions called JCL instructions which instruct the operating system on the identity and requirements of the job. JCL statements tell the OS such things as the name of the job, the user's name and account number, the I/O devices to be used during processing, the assembler or compiler to be used if language translation is required, and so on.

The job control language for one computer is different from that of another computer. Therefore, JCL statements differ from computer to computer and hence it is not possible to list here a set of statements that you can simply copy and use with your computer system. In order to know the JCL statements of your computer, it is suggested that you consult your instructor or supervisor, or an experienced programmer of your computer installation. However, a simple example is given below so that you can see what JCL statements look like. You will then have a better understanding of what the JCL statements used with your system do.

Suppose we wish to run a simple FORTRAN program on the Burroughs-6700 computer. The structure of the job deck in that case will be as follows :

```
<I> BEGIN JOB ONGC

<I> USER = SINHA/SNH

<I> COMPILE UPDATE FORTRAN GO
    FORTRAN source program cards

<I> DATA
    Data Cards

<I> END JOB
```

The above control statements are written in the work-flow language (WFL) which is the JCL for Burroughs-6700 system. Each control statement of WFL starts with the symbol <I> in column 1. The symbol <I> represents an invalid character (such as a combination of 1, 2 and 3 punched in the same column). This implies that no program or data card should have the symbol <I> punched on its first column. It should be noted that this symbol is typical and not universal. For example, IBM operating systems use two consecutive slashes to indicate a JCL statement and Honeywell-6000 uses a $ character in the first column for this purpose.

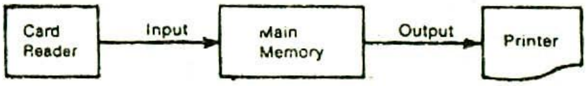
The BEGIN JOB statement signals the beginning of a new job to the operating system. A name must be assigned to the job which is ONGC in this example. The next WFL statement is used to specify the usercode and the password of the user. Each user of a Burroughs-6700 installation is assigned a usercode by the system manager. In addition to the usercode, each user is also allotted a password which can be subsequently changed by the user as and when he desires. Both the usercode and his password must be specified correctly in the USER statement of WFL. In our example, the usercode is SINHA and the password is SNH. The password provides an added security to the user so that another person cannot run a job using his usercode. The operating system first checks the usercode given by the user with the list of valid usercodes stored within the system. If the usercode is found to be valid, then his password is checked to see if it is valid for the given user. Unless a user quotes both the usercode and his password correctly, the job will not be accepted by the operating system. The third WFL statement is a COMPILE statement. The word FORTRAN in this statement indicates that the source program is written in FORTRAN and hence a FORTRAN compiler will be required for its translation. The compiler specified by the user (in our case FORTRAN) is fetched from the secondary storage device (normally disk) and placed in the main memory to facilitate translation of the source program. The user also specifies the name of the object program which is UPDATE for our example. Hence the object code obtained after the successful compilation of the source program will be stored in the system by the name UPDATE. Finally, the word GO in this statement indicates that the object program is to be executed. The operating system will automatically start executing the program as soon as the translation process is over. It may kindly be noted here that since we are compiling a program, so if any error is detected during the translation process, no object code will be created for this program and subsequently no execution will be done even if the user has specified GO in the COMPILE statement of WFL. The COMPILE statement is followed by the source program statements. The DATA statement which is again a WFL statement denotes the end of the source program and the beginning of the data that will be used by the program. Finally, the END JOB statement indicates the end of the job. A job deck prepared in the above fashion can be submitted to a Burroughs-6700 installation for compilation and execution of a FORTRAN program.

SPOOLING

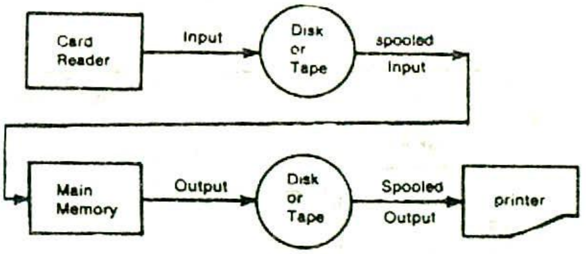
In batch mode of operation, the processing speed of a computer system can be further enhanced by a technique known as SPOOLing (*S*imultaneous *P*eripheral *O*utput *O*n *L*ine).

We know that dedicated I/O devices (devices which cannot be shared concurrently by several processes) like card readers and printers are considerably slower compared to the speed of the CPU. For example, a fast card reader can operate at a speed of 1500 cards/minute for an 80-column card. This is equivalent to reading 2000 bytes/second (1500 x 80 x 1/60). On the other hand, for a medium speed computer, the processing speed of the CPU is approximately $3x10^6$ bytes/second. Thus while a card reader is supplying just one character to the main memory, the CPU can perform many thousands of internal operations before it needs to become involved with next character from the card reader. In other words, during reading of information into memory the CPU had to wait because of slow reading. The speed mismatch in this case is of the order of 1500. Similar speed mismatch exists for a large number of peripheral devices such as printers, teletypes, etc.

Spooling is a technique that has been successfully used on a number of computer systems to reduce the above mentioned speed mismatch and in turn the idle time of the CPU. It is the process of placing all data that comes from an input device or goes to an output device on either a magnetic tape or disk. This is shown in Figure 14.2. A batch of program when fed to the card reader is read and temporarily stored on a magnetic tape or disk instead of being directly stored in the main memory. The programs stored on tape or disk are now fed to and processed by the main computer. The results obtained are again written on tape or disk instead of being directly printed on the printer. The contents of output tape or disk are later printed on the printer. The process of storing the input data and output results on tape or disk is known as spooling. The primary reason for spooling is to keep the programs and data readily available to the fast and expensive CPU on a high speed I/O medium such as tape or disk. Reading from tapes or disk is usually at the rate of 10^5 bytes/second in contrast to 2000 bytes/second for a card reader. Similarly, the speed of writing on tape or disk is of the order of 10^5 bytes/second contrasted to writing on even a fast line printer at a speed of 2000 bytes/second.



(a) Mode of data transfer without spooling facility.



(b) Mode of data transfer with spooling facility.

Figure 14.2. Illustrating the process of spooling.

Special spooling programs are executed by the operating system to transfer the data from the disk or tape to the main memory or an input or output device. In a sense, the disk or tape device acts as a buffer area between main storage, which is extremely fast, and I/O devices, which are relatively slow. Spooling programs are executed when the CPU is not too busy with other jobs. However, in most computer systems, special low cost I/O processors are used for spooling the input data from a card reader (or any other slow input device) on to the tape or disk or for printing the spooled results from the tape or disk on to the printer (or any other slow output device). These I/O processors function independent of the main CPU. This enables the main high speed expensive CPU to be fully devoted to main computing job.

The process of spooling is transparent to the user's program. In general, spooling makes better use of both the main memory and the CPU.

MULTIPROGRAMMING

In case of batch processing, the batched programs are loaded one after another in sequence into the main memory for processing. Once loaded, a program will remain in the main memory until its execution is completed. Thus, the program which is currently being executed will be the sole occupant of the users' area of the main memory (remember that the supervisor always resides in a part of the main memory) and it will have the CPU exclusively available to itself. When there is only one program in main memory, two of the system's most powerful resources may be under-utilized, its expensive memory and the full capabilities of the CPU. As shown in Figure 14.3, every program will not be large enough to occupy the full users' area of the main memory. Similarly, all programs will not be highly computational to utilize the full processing capability of the CPU. Basically there are two types of programs :

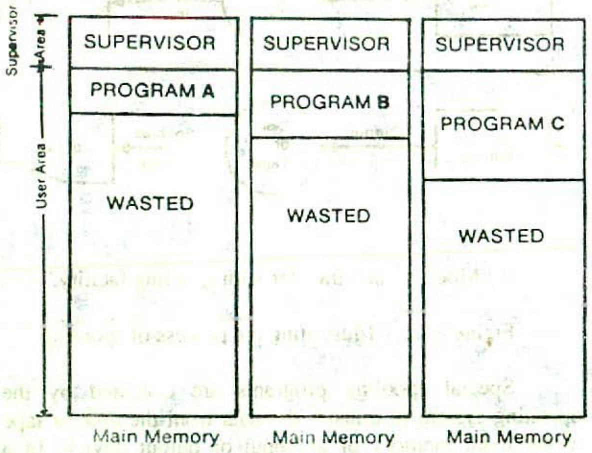


Figure 14.3. Illustrating the under-utilization of memory when only one program occupies the whole memory.

I/O-bound programs and CPU-bound programs. Programs used for commercial data processing normally read in vast amount of data, perform very little computation and output large amount of information. Such programs are known as *I/O-bound programs*, since the majority of work they perform is input-output. On the other hand, programs used for scientific and engineering applications need very little I/O but require enormous computation. These programs are called *CPU-bound programs* because more of CPU-time is required for processing such programs.

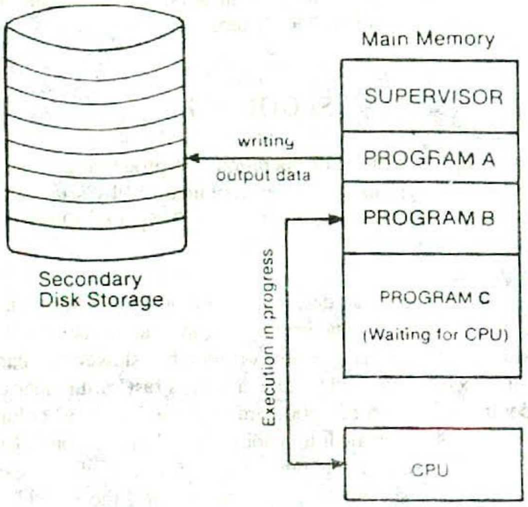


Figure 14.4. Illustrating the operation of multiprogramming.

In order to overcome the problem of under-utilization of main memory and the CPU, the concept of multiprogramming was introduced in operating systems. "*Multiprogramming* is the name given to the interleaved execution of two or more different and independent programs by the same computer." With the storage-resident supervisor concept (see Figure 14.3) we have been introduced to the notion of having two programs in the main memory at the same time : the supervisor for overall system control and the user program for performing user's task. In multiprogramming, this concept is carried one step further by placing two or more user's programs in main memory and executing them concurrently. The CPU switches from one program to another almost instantaneously. Since the operating speed of CPU is much faster than that of I/O operations, the CPU can allocate time to several programs instead of remaining idle when one is busy with I/O operations. In multiprogramming system, when one program is waiting for I/O transfer, there is another program ready to utilize the CPU, thus it is possible for several users to share the time of the CPU. However, it is important to note that multiprogramming is not defined to be the execution of instructions from several programs at the same instant of time. Rather, it does mean that there are a number of programs available to the CPU (stored in main memory) and that a portion of one is executed, then a segment of another and so on. Although two or more users' programs reside in the main storage simultaneously, the CPU is capable of executing only one instruction at a time. Hence at any given time, only one of the programs has control of the CPU and is executing instructions. Simultaneous execution of more than one program with a single CPU is impossible. In some multiprogramming systems, only a fixed number of jobs can be processed concurrently (*m*ultiprogramming with *f*ixed *t*asks) (MFT), while in others the number of jobs can vary (*m*ultiprogramming with *v*ariable *t*asks) (MVT).

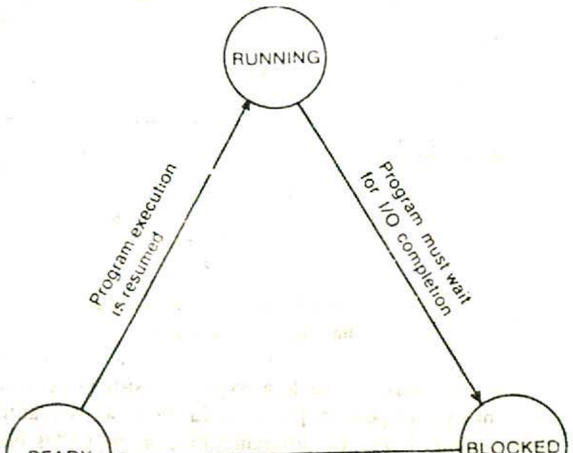

RUNNING
READY
BLOCKED
Program execution is resumed
Program must wait for I/O completion
I/O Completed

Figure 14.5. The three different states of a program residing in main memory in case of multiprogramming.

A simple example of multiprogramming is given in Figure 14.4. At the particular time instance shown in the figure, program A is not utilizing the CPU since it is busy writing output data on to the disk (I/O operation). Hence the CPU is being utilized to execute program B which is also present in the main memory. Another program C, residing in the main memory, is waiting for the CPU to become free. Actually, as shown in Figure 14.5, in case of multiprogramming all the programs residing in the main memory will be in one of the following three states : running (CPU is being used), blocked (I/O operation is being done) and ready (waiting for CPU). For our example, programs A, B and C are in blocked, running and ready states respectively. Since program C is in the ready state, as soon as the execution of program B is completed or program B requires to do I/O operation, the CPU will start executing program C. In the meanwhile, if A completes its output operation, it will be in the ready state waiting for the CPU. Thus in multiprogramming, the CPU is almost always busy. When program A is reading data or outputting results (I/O operations), program B's instructions can then be executed, and if both programs are involved in I/O activity, then program C can be executed. The area occupied by each program residing simultaneously in the main memory is known as a *memory partition*. The actual number of memory partitions and hence programs allowed in the main memory at any given time varies depending upon the operating system in use at a particular installation. Moreover, those jobs awaiting entry into the main memory are queued on a fast secondary storage device such as a magnetic disk. The first job from this queue will be loaded into the main memory as soon as any one of the jobs already occupying the main memory is completed and the corresponding memory partition becomes free.

Requirements of Multiprogramming Systems

Multiprogramming has two main advantages : increased throughput and lowered response time. Throughput is increased by utilizing the idle time of the CPU for running other programs that are already residing in the main memory. Response time is lowered by recognizing the priority of a job as it enters the system and by processing jobs on a priority basis.

On the other hand, the incorporation of multiprogamming in the operating system has, of course, complicated matters. For a computer to work simultaneously on many programs, the following additional hardware and software features are required :

1. *Large memory*. For multiprogramming to work satisfactorily, large main memory is required (of

the order of 128K or more), together with fast secondary storage devices like disk and fast CPU. The main memory should be large enough to accommodate a good number of users' programs along with the supervisor.

2. *Memory protection*. Computers designed for multiprogramming must provide some type of memory protection mechanism to prevent a program in one memory partition from changing information or instruction of a program in another memory partition. For example, in Figure 14.4 we would not want program A to inadvertently destroy something in the completely independent program B or program C. In a multiprogramming environment this is achieved by the *memory protection feature*, a combination of hardware and software, which prevents one program from addressing beyond the limits of its own allocated storage area.

3. *Program status preservation*. We have seen that in multiprogramming, a portion of one program is executed, then a segment of another, and so on. This requires the stopping of a program execution and then restarting its execution after some time. In order to restart a program, all the values that were stored in memory and the CPU registers that were being used at the time of its stopping should be restored. A new program would however need all CPU registers for its use and would clear them. Thus, before a program is suspended and the control is passed to another program, the values of all CPU registers (pc, accumulator, etc.) should be stored in the memory area of that program and then restored when the control is ultimately returned to the first program. This is known as program status preservation.

4. *Proper job mix*. A proper mix of I/O-bound and CPU-bound jobs is required to effectively overlap the operations of the CPU and I/O devices. It is necessary that when a program is waiting for I/O operation, another program must have enough computation to keep the CPU busy. If all programs need I/O at the same time, the CPU will again be idle. Hence the main memory should contain some CPU-bound programs and some I/O-bound programs in its various partitions so that at least one of the program which does not need I/O is always available to the CPU for processing.

MULTIPROCESSING

Upto this point we have considered systems with a single CPU. However, we have already seen that the use of I/O channels or I/O processors improves the efficiency of the computer system by making possible concurrent input, processing, and output operations. The CPU can perform arithmetic and logical operations on parts of one or more programs while I/O operations are concurrently carried out by I/O processors on other parts of programs. The architecture of a computer having I/O processors is shown in Figure 14.6.

The idea of use of I/O processors to improve the performance of a computer system was carried one step further by designing systems that make use of more than one CPU. Such systems are called multiprocessing systems. The term *multiprocessing* is used to describe interconnected computer configurations or computers with two or more independent CPUs that have the ability to simultaneously execute

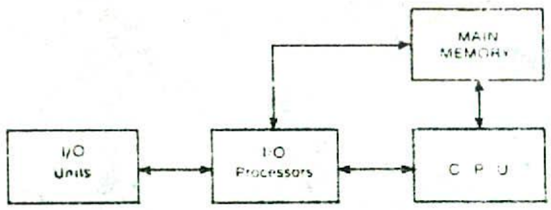


Figure 14.6. Architecture of a computer system having I/O processors.

several programs. In such a system, instructions from different and independent programs can be processed at the same instant of time by different CPUs or the CPUs may simultaneously execute different instructions from the same program. The basic organisation of a typical multiprocessing system is shown in Figure 14.7.

There are almost limitless number of possible multiprocessing systems. In some systems, several small CPUs are linked together to perform the major processing. If one of the small CPUs breaks down, the other CPUs will automatically take over its job. In other systems, CPUs are connected into elaborate computer networks. Distributed data processing (discussed in next chapter) is an example. In these networks, small CPUs called front-end processors are used for scheduling and controlling all jobs entering the system from remote terminals and other input devices. Thus, the main CPU or CPUs called host computers or back-end processors are used only for major processing jobs and not for data communication. In some multiprocessing systems, each CPU performs only specific

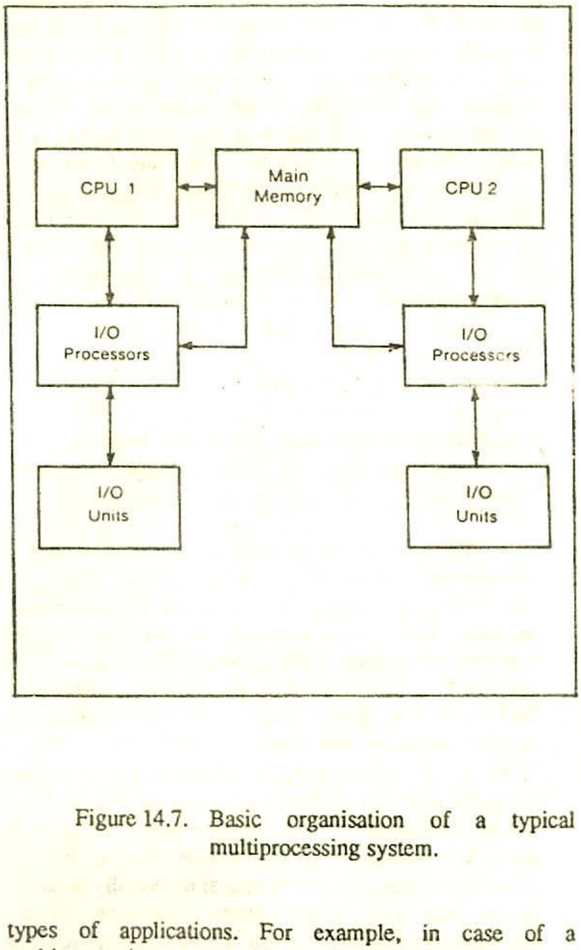


Figure 14.7. Basic organisation of a typical multiprocessing system.

types of applications. For example, in case of a multiprocessing system with two CPUs, one may be used to process only on-line jobs while another one may be meant for processing only batch applications. However, these systems are so designed that in case of breakdown of one CPU, the other CPU takes over the complete workload until repairs are made. Moreover, different multiprocessing systems use different types of memory configurations. In some systems each CPU has its own main memory, in others all the CPUs may share a common memory, while in some others each CPU may have access to both separate and common memories.

Difference Between Multiprogramming and Multiprocessing

Multiprogramming is the interleaved execution of two or more processes by a single CPU computer system. On the other hand, multiprocessing is the simultaneous execution of two or more processes by a computer system having more than one CPU. To be more specific, we may point out here that multiprogramming involves executing a portion of one program, then a segment of another, etc., in brief consecutive time periods. Multiprocess design, however, makes it possible for the system to simultaneously work on several program segments of one or more programs.

Advantages and Limitations of Multiprocessing

There are numerous advantages of multiprocessing some of which are listed here :

1. It improves the performance of computer systems by allowing parallel processing of segments of programs. Better performance is directly reflected by increased throughput and lowered turnaround time of such systems.
2. In addition to the CPUs, it also facilitates more efficient utilization of all the other devices of the computer system.
3. It provides a built-in backup. If one of the CPUs breaks down, the other CPU(s) automatically takes over the complete workload until repairs are made. Thus, a complete breakdown of such system is very-very rare.

Multiprocessing, however, is not an easy task because of the following reasons :

1. A very sophisticated operating system is required to schedule, balance and coordinate the input, output and processing activities of multiple CPUs. The design of such an OS is a time taking job and requires highly skilled computer professionals.
2. A large main memory is required for accommodating the sophisticated operating system along with several users programs.
3. Such systems are very expensive. In addition to the high charges paid initially, the regular operation and maintenance of these systems is also a costly affair.

It is expected that multiprocessing systems will soon become commonplace. In the future, all large computer systems will use multiple, parallel processors to share high-speed and complex operations and to enhance processing throughout. There will be computers within computers, because some of these processors will be complete microcomputers. The entire system will be under the control of a complex, powerful operating system.

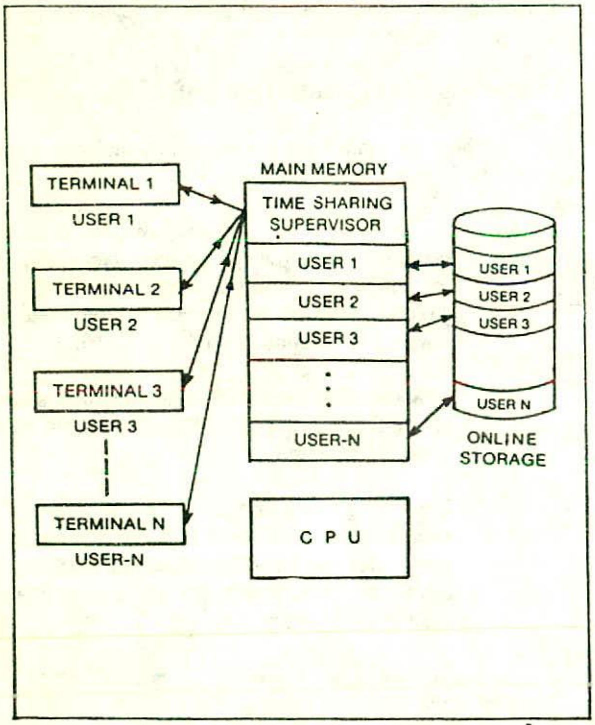


Figure 14.8. Concept of a timesharing system.

TIME-SHARING

Timesharing is a term used to describe a processing system with a number of independent, relatively low speed, online, simultaneously usable stations. Each station provides direct access to the CPU.

In other words, timesharing refers to the allocation of computer resources in a time-dependent fashion to several programs simultaneously. The principal notion of a timesharing system is to provide a large number of users direct access to the computer for problem solving. This is accomplished by providing a separate terminal to each user.. All these terminals are connected to the main computer system. Thus, a timesharing system has many, even hundreds, of terminals linked up to the same computer at the same time. This is shown in Figure 14.8. Unlike multiprogramming, where programs are executed on a priority basis, in timesharing the CPU time is divided among all the users on a scheduled basis. The basic idea behind timesharing systems is to allow all user programs to have a brief share of the CPU time in turn. Each user program, beginning from the first program and proceeding through the last, is allocated a very short period of CPU time one by one. This short period of time during which a user gets the attention of the CPU is known as a *time slice*, *time slot* or *quantum* and is typically of the order of 10 to 20 milliseconds. The processing speed of the system and the use of multiprogramming in conjunction with timesharing allows the CPU to switch from one user station to another and to do a part of each job in the allocated time slice until the job is completed. The speed is frequently such that the user has the illusion that he alone is using the computer. It is somewhat like viewing a motion-picture film made up of individual frames, the switching is so fast that the processing at any given terminal appears to be continuous.

For example, let us assume that the time slice for a timesharing system is 10 milliseconds. That is, the timesharing operating system allocates 10 milliseconds to each user during which a program belonging to this user is executed. An average speed computer whose speed is of the order of 1 million instructions per second can execute 10 x 10^{-3} x 10^{-6} = 10,000 instructions in 10 milliseconds. Suppose there are 100 users for this timesharing system. Then if 10 milliseconds is allocated to each user, a particular user will get the CPU's attention once in every 10 x 100 milliseconds = 1 second. As human reaction times are a few seconds, a particular user will not notice any delay in executing his commands and normally feels that he is the sole user of the system. Moreover, it is not economically feasible to allow a single user to use a large computer interactively because his speed of thinking and typing is much slower than the processing speed of a computer. While a particular user is engaged in thinking or is busy typing his input, a timesharing system can service many other users.

Even though it may appear that several users are using the computer system at the same time, a single CPU system can only execute one instruction at a time. Thus like a multiprogramming system, even with a timesharing system, only one program can be in control of the CPU at any given time. As a result, at any instant, all the users who are using a timesharing system will fall in one of the following three status groups.

1. *Active* : the user's program currently has control of the CPU. Obviously only one user will be active at a time.

2. *Ready* : the user's program is ready to continue but is waiting for its turn to get the attention of CPU. More than one user can be in ready state at a time.

3. *Wait* : the user has made no request for execution of his job or the user's program is waiting for some I/O operation (for instance, the user is sitting at the terminal and is thinking what should be the next step). Again, more than one user can be in wait state at a time.

The process of switching from one status to another is illustrated in Figure 14.9. Moreover, the concepts of a timesharing system and user status are illustrated in Figure 14.10. In Figure 14.10(a) user 2 is active, users 1, 3, and 4 are in wait status and users 5 and 6 are in ready status. As soon as the time slice of user 2 is completed, the timesharing supervisor moves on to the next ready user

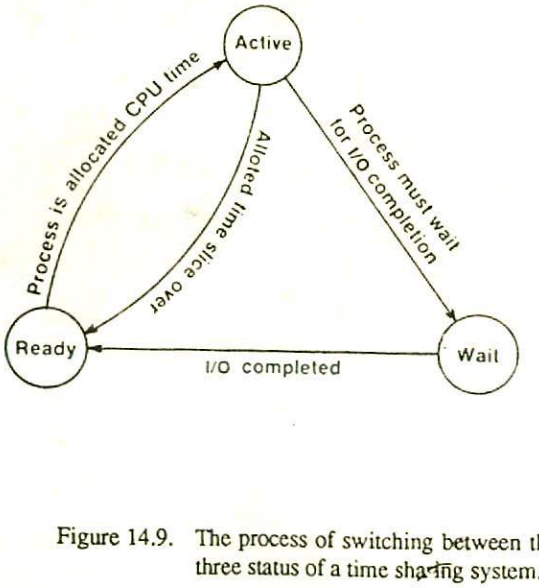

Active
Process is allocated CPU time
Alloted time slice over
Process must wait for I/O completion
Ready
Wait
I/O completed

Figure 14.9. The process of switching between the three status of a time sharing system.

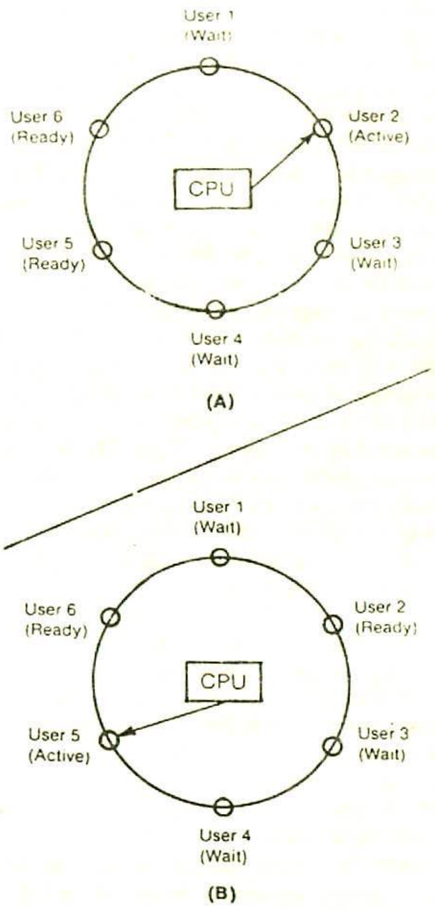
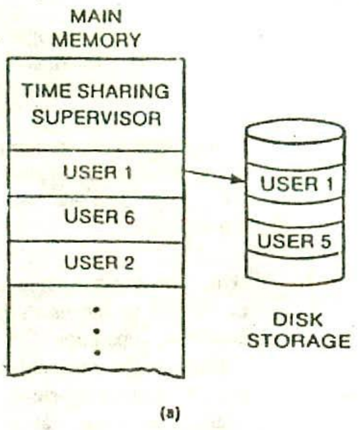


Figure 14.10. User status in a timesharing system.

(those in wait status are skipped since they are making no demand for the CPU). The next ready user in the queue is user 5 which now becomes active as shown in Figure 14.10(b). User 5 will remain active until the allotted time slice expires, or until the program needs I/O operation, or if the program execution is over during this time period. At that time, control is passed on to the next ready user in the queue which is user 6 for our example. Whenever I/O operation is completed for a wait user, that user's status will be changed to ready and serviced the next time around

In a typical timesharing system, hundreds of users may be using the system simultaneously. As the total main memory available in a computer is limited, it is not possible to keep the programs of all the users of a timesharing system simultaneously in the main memory. Thus at any instant, the timesharing operating system keeps only a few programs in the main memory and the rest are stored on the disk storage. At a particular instance of time, the memory resident programs include the active program and some of the ready programs which will get CPU's attention very shortly. A wait program of the main memory is normally replaced by a ready program on the disk storage. As and when a program is to be executed, it is brought back to the main memory from the disk and the inactive program is sent to the disk. The operation of transferring programs from the main memory to the disk storage and back is known as *swapping*. Referring to Figure 14.10, as user 2 is executing, the system will be ensuring that the next ready job is in main memory. If it is not, then one of the wait programs is swapped out (onto disk) as shown in Figure 14.11(a) and the next ready user, user 5 in this case, is swapped in as shown in Figure 14.11(b). This swapping process, sometimes know as *roll-in roll-out*, is repeated many times within a few seconds. In this case, disks are the only feasible secondary storage devices since they have a much faster rate of information transfer than magnetic tapes and provide direct access.

In timesharing systems, the user often carries a dialogue or conversation with the central system. Hence it is also known as *conversational* or *interactive* computing. The computer can be programmed to interrogate the user as required, to respond to requests, replies, and even to mistakes. A user can proceed step-by-step, testing portions of his procedure or trying out various approaches to a problem solution. This is the reason why timesharing systems have been found to be most suitable for program development and testing. In fact, the BASIC language was designed specifically for timesharing systems. Systems which are fully interactive inspect each statement of a new program as it is entered into the computer via a terminal. Any syntax errors in the usage of the language are detected and immediately displayed on the video screen so that the user can make appropriate corrections. The user corrects his program with an editing system. When all the syntax errors are corrected, the program can be run and tested to ensure its validity. The requisite data is fed from the terminal during program execution. The data may also be stored in a data file on disk and fed to the program when it needs it. Errors encountered during execution of the program are displayed on the terminal. These run time errors can be immediately corrected by the user and another test run can be made. The greatest benefit of such a system is that errors can be encountered, corrected, and work can continue immediately. This is in contrast to a batch system in which errors are corrected offline and the job is submitted for another run. The time delay between job submission and return of the output in a batch system is often measured in hours. Overnight turnaround is also very common.



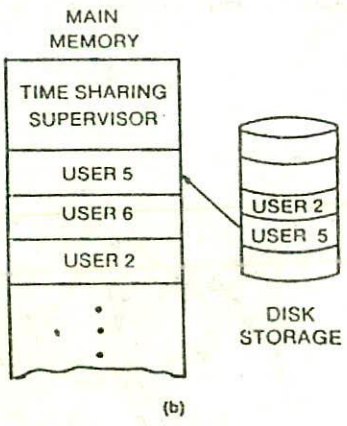


Figure 14.11. Swapping of programs in a timesharing system.

Obviously this time lapse does not contribute well to the thinking efficiency of a programmer. The interactive programming and debugging capability of BASIC has proven to be so effective in improving programmer efficiency, that interactive versions of the batch-oriented FORTRAN and COBOL are also available now.

Advantages of Timesharing

1. *Reduces CPU idle time.* It is wasteful and expensive for the CPU to be effectively utilized less than 30% of the time. Yet this is what happens in a conventional batch processing installation as the CPU waits during set-up times and during I/O operations. Timesharing significantly increases CPU utilization by switching from one program to another in rapid succession. Thus the throughput of the installation increases to a great extent.

2. *Offers computing facility to small users.* Small users can gain direct access to much more sophisticated hardware and software than they could otherwise justify or afford. In a timesharing system they merely pay a fee for resources used and are relieved of the hardware, software, and personnel problems associated with acquiring and maintaining their own installation.

3. *Provides advantages of quick response.* The turnaround time or the response time is negligible in case of a timesharing system. Thus, timesharing allows managers to react more rapidly. Furthermore, it permits them to interact or converse with the system in seeking solutions to unusual problems and answers to poorly defined questions. Timesharing may also reduce waste in the use of business resources and it can permit quick follow-up on creative ideas. In short, it helps in improving the users' efficiency to a great extent.

4. *Reduces the output of paper.* If a manager can retrieve at any time the specific information he needs from an online file, he does not need a bulky report that contains much of the file information.

5. *Avoids duplication of software.* There are several programs which are frequently used by many users. In a time-sharing system, such programs are stored in the system library. A user need not write his own program instructions when performing such processing tasks. He need only call up the needed program stored online at the computer site and supply the data.

Disadvantages of Timesharing

1. *Question of security.* Since hundreds of users use a timesharing system simultaneously, provision must be made to protect the security and integrity of user programs and data. The programs and data of different users should not get mixed up. This is currently being accomplished by such methods as (a) assigning and (b) requiring hierarchies of passwords or lockwords from users prior to file access. However, in spite of such precautions, skilled penetrators succeed in bypassing the programmed controls of current timesharing systems.

2. *Problem of reliability.* A time sharing system should be highly reliable as it caters to the needs of several users. Hence provisions must be made to provide dependable and continuous service. The self-repairing computer or some sort of standby arrangement may ultimately help to overcome the reliability problem. But troubles that occur are often with the online peripheral devices or software and not with the main CPU.

3. *Problem of data communications.* In a timesharing system, the users interact with the main computer system through remote terminals that require data communication facilities. The cost of data communication has been declining but not so rapidly as the cost of data processing. Thus, data transmission charges make up an increasing portion of the total timesharing cost package. In addition, telephone lines were designed for voice communication rather than data communication, with the result that current transmission facilities are not considered adequate by many timesharing spokesmen.

4. *Question of overhead involved.* The reader can probably appreciate that the timesharing system with its control functions such as switching from user to user and swapping programs in and out takes up an appreciable amount of CPU time. This is termed overhead and must be minimized in the overall hardware-software design. If a system is properly balanced, then the overhead is manageable and the computer response time to a user request will be small. However, if the system is overloaded with too many users, then the overhead can get out of hand, resulting in very poor response.

ON-LINE PROCESSING

On-line processing (also called direct-access or random-access processing) permits transaction data to be fed under CPU control directly into secondary on-line storage devices from the point where data originates without first being sorted (sorting of data is required in case of sequential processing). These data may be keyed in by the use of a typewriter like terminal, or they may be produced by a variety of other data collection and transaction recording devices. The CPU can make programmed input control checks during this process. Using these input data, appropriate records (which are normally organized in the secondary storage unit in random fashion) may be quickly updated. The access to, and retrieval of, any record is quick and direct. Information contained in any record is accessible to the user without the necessity of a sequential search of the file and within a fraction of a second after the enquiry message has been transmitted. Thus, on-line processing systems feature random and rapid input of transactions and immediate and direct access to record contents as and when needed. A simplified concept of on-line processing is depicted in Figure 14.12.

A timesharing system is a typical example of on-line processing. However, it should be noted here that on-line processing systems may differ considerably in level of complexity. Some systems may have only a few terminals, and the volume of transactions to be processed may thus be low; these transactions may be processed on a first-come, first-served basis with no attempt being made to use timesharing or multiprogramming and the system may employ relatively simple data communication facilities. At

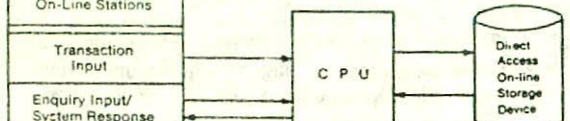

On-Line Stations
Transaction Input
Enquiry Input/ System Response
C P U
Direct Access On-line Storage Device

Figure 14.12. On-line processing.

the other extreme are on-line systems that have hundreds of remote stations and communication lines; they use multiprogramming or timesharing to keep the response time within acceptable range so that users do not get irritated due to delays in response.

On-line processing and direct access to records require unique hardware and software. For example, the capacity of the primary storage unit of the system must be adequate to accommodate the complex on-line operating system supervisor along with other users' programs. Also, since many on-line users may have access to stored records, software security provisions are necessary to prevent confidential information from falling into unauthorized hands and prevent deliberate or accidental tampering of data and program files. Furthermore, in many cases, CPU must be fast enough to respond to multiple on-line stations operating simultaneously in a multiprogramming mode; and large capacity peripheral on-line storage units are required to store additional operating system elements, user data and programs. Finally, data transmission facilities must be provided to communicate with on-line terminals located in the next room, on the next block, or thousands of miles away.

REAL-TIME PROCESSING

There are many applications that require an immediate response from the computer. Getting a stock market quotation, finding the current level of product inventory, and searching a criminal data file for a possible suspect may all be actions that need to be done without delay. In these cases, a real-time processing system is needed. Real-time means immediate response from the computer. A system in which a transaction accesses and updates a file quickly enough to affect the original decision making is called a *real-time system*. The essential feature is that the input data must be processed quickly enough so that further action can then be promptly taken on the results.

In other words, a real-time processing system may be described as an on-line processing system with severe time limitations. It may be noted here that a real-time system uses on-line processing, but an on-line system need not necessarily operate in real-time mode.

Real-time processing requires immediate (not periodic) transaction input from all input-originating terminals. Many remote stations are tied directly by high-speed communications equipment into one or more CPUs. Several stations may be operating simultaneously. Files are updated each minute, and enquiries are answered by split-second access to up-to-the-minute records. The system processes input data and presents the result in such a form that human judgement can immediately be brought into decisive action.

One of the early and very sophisticated commercial real-time systems was the American Airlines SABRE reservation system. The following factors justify the use of real-time processing for an airline reservation system :

1. There are hundreds of flights daily.

2. Each flight may have as many as 300 seats or more "in inventory".

3. As soon as a seat is reserved/cancelled, the concerned files must be updated before the next transaction can be processed.

4. The response time should be very short because a customer's reservation is to be done while he waits.

5. Seats may be sold for only a portion of flight. For example, Mr. XYZ may book a seat to Baroda on a Delhi to Bombay flight which stops in Baroda. That seat will then be available for the Baroda to Bombay leg.

6. Hundreds of agents throughout the country are selling seats from the inventory.

7. An airline seat is a very perishable item. If it is not sold, it is lost once a flight is made.

Owing to the above mentioned reasons, efficient operation of present-day major airlines would be very difficult and almost impossible without a real-time processing system. Few more examples of business real-time processing are :

1. Air traffic control system.

2. Reservation systems used by hotels and car rental agencies. These systems keep track of the availability of hotel rooms or cars at any instance of time.

3. Systems that provide immediate updating of customer accounts in saving banks.

4. Systems that provide up-to-the-minute information on stock prices.

5. Process control systems as in nucler reactor plants and steel mills.

Similarly, there are many applications that require real-time processing. It would be a mistake, however, to assume that real-time processing should be universally applied to all data processing applications. A quick-response system can be designed to fit the needs of the business. Some applications can be processed on a lower priority or background basis using batch methods (e.g. payroll); some can be on-line with periodic (not immediate) updating real-time methods.

Real-time systems are required to be highly reliable because even minimal downtime in many critical applications may be hazardous causing danger to several lives and substantial financial loss. For example, in case of a computerised air traffic control system, the radar and computers that keep track of air traffic in each region, must operate constantly with minimum maintenance. An unscheduled downtime in this real-time system may cost lives of several human beings. Similarly, in case of a chemical process-control system, high degree of reliability is essential, since even minimal downtime can lead to a ruined batch of product or damage to expensive equipment with substantial financial loss. In order to achieve the desired degree of reliability, real-time systems are normally duplicated so that, in the event of a break down, back-up facilities are immediately available for continuous operation of the system. This makes some systems very expensive but, in the environment to which real-time systems are applied, a fail-proof system is essential.

VIRTUAL STORAGE

Virtual storage may be described as a hierarchy of two memory systems - one of them is a low cost, large capacity, low speed system (on-line disk storage) and the other is a high cost, small capacity, high speed system (main memory). The operating system manages the two memory systems in such a way that a user feels that he has access to a single, large, directly addressable, and fast, main memory.

A virtual memory system facilitates its users to use a large addressable memory space without worrying about the size limitations of the physical main memory. Moreover, in case of multiprogramming or timesharing systems, it also permits the sharing of memory space among several users efficiently and economically.

In order to implement a virtual memory system, the main memory is divided into fixed size contiguous areas, called *page frames*. In addition, all users programs, residing on the on-line disk storage, are also divided into pieces of the same size, called either *pages* or *segments*. Now, only those program pages or segments that are actually required at a particular time in the processing, need be in the primary (or real) storage. The remaining pages or segments may be kept temporarily in online (or virtual) storage, from where they can be rapidly retrieved as and when needed following program interruption (see Figure 14.13). The operating system handles the swapping of program pages or segments between the main memory and the on-line disk storage.

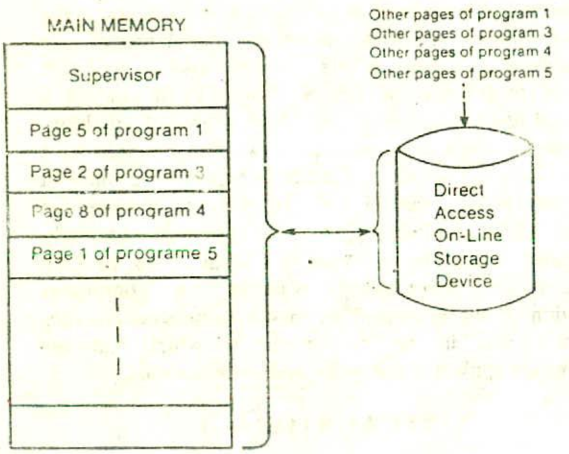


Figure 14.13. A virtual storage system.

Thus, from the applications programmer's point of view, the effective (or virtual) size of the available primary storage may appear to be unlimited.

In previous sections, we have seen that when a computer system is operated in timesharing or real-time mode, only a few instructions are executed for one user at any given moment. Then the computer executes a few instructions for another user and so on. In a few seconds, the computer could process a few instructions for over a hundred users. Because the computer is only executing a few instructions of one user's program at one time, it will be wastage of main memory if the complete program of that user is stored in main memory, instead, the concept of virtual storage can be effectively used in this case to simultaneously accommodate program segments of a large number of users in the main memory. With more program segments of different users residing simultaneously in main memory, the CPU is less likely to have to wait for programs to be transferred from the disk to main storage. This reduces CPU idle time and increases the number of jobs that can be run in a given time span. It is important to note that even in this particular application of virtual storage, the physical size of main memory remains the same. It only appears to be larger because more gets done in less time.

OS-CONTROLLED SOFTWARE

A computer can do nothing with a program of instructions, and each job required must have its own special program. However there are many tasks of a routine nature that all computer users require their machine to perform from time to time. It would clearly be wasteful if each user spent a lot of time writing programs for these tasks and it is normal practice for the computer manufacturers to supply programs for these tasks along with the operating system and the hardware of the machine. These OS-controlled *softwares* reduce the time and expense of preparing applications programs and are normally grouped in three categories - translating programs, library programs, and utility programs. They are briefly discussed below.

TRANSLATING PROGRAMS

Translating programs, also known as language processors, are system programs that translate a source program written by the user to an object program which is meaningful to the hardware of the computer. These include the assembler and the various compilers and interpreters available with the system. A translating program is usually called up from a direct-access storage device only after the job control program of the operating system interprets a job control statement and informs the operating system supervisor of what is needed.

LIBRARY PROGRAMS

Library programs consist of frequently used subroutines supplied by users and computer vendors. These standard routines are stored in a direct-access storage device and are called up by the operating system whenever they are required in the processing of other programs. This eliminates the need for a programmer to rewrite these modules every time they are used. A *librarian program* controls the storage and use of these programs in the system library. It maintains a program directory for this purpose and also facilitates the addition of new programs to the library or deletion of unwanted programs from the library. In the area of scientific applications, the usual types of library routines available are the mathematical functions such as square root and exponential functions. Other operations of various types are also encountered such as matrix inversion, statistical analysis, conversion of numbers from one base to another (binary to decimal and decimal to binary), etc. All these and many other frequently used routines are normally available as library programs.

UTILITY PROGRAMS

Utility programs, also known as service programs, are routines that perform needed services such as editing texts or programs, debugging programs to correct logical mistakes, sorting records into a particular sequence for processing, or transferring data from one I/O device to another. These routines are also available for call-up by the operating system and once again the job control statements of a particular user tell the operating system supervisor which utility programs are needed by the user. A few examples of utility programs commonly available in a computer system are briefly described below.

1. *Text editor.* A text editor is a program that facilitates the creation and correction of texts. The text being edited could be an English language letter, but most often, it is a symbolic language program typed by the user. The text editor program does not interpret the meaning of the text but has the capability of changing it when special commands are issued by user. For example, when a symbolic language program is being entered into a computer memory via a video-terminal, the programmer may use the facility of a text editor program to correct his typing errors by issuing commands to insert, delete, or replace characters in his source programs. Thus, with the help of a text editor, the user can prepare programs and correct them with relative ease.

2. *Debugging tool.* Debugging tool is a program that helps the user to locate and correct logical mistakes in his program. A dynamic debugging tool allows the programmer to control program execution using a video terminal while his program is being executed in the computer. While his program is being executed, the user can stop the execution of the program at any desired point, he can examine contents of various registers, change contents of registers and memory, make alterations to his binary programs, and other similar functions. Thus, by using the facilities of a dynamic debugging tool, the user can easily detect and correct logical errors in his program.

3. *Sort and merge.* In the area of commercial data processing applications, the most widely used routines are sort and merge. Sort programs are used to arrange data into a specified sequence. For example, business transactions may be stored in computer in the order in which they occur. The transactions may have to be sorted by different items such as by account number to identify the customer or by salesman's name to calculate the commission to be paid. The sort program reads the unsequenced input file and by means of various copying techniques ultimately produces as output a copy of the input file in the required sequence. Merge programs, on the other hand, are used to combine two or more sets of sorted data into one file containing all the items of all the original sets in sorted order.

4. *Memory dump program.* A memory dump program allows the user to print the contents of specified locations in main memory at some particular point during the program execution. A memory dump typically shows both the program and operand data. By inspecting both program and data, and comparing it with what it should have been if the program had run correctly, the programmer is able to find the mistakes in his program.

5. *Trace routine.* A trace routine allows the user to trace the flow of his program while it is executed. He can request, for example, that the contents of certain registers or memory locations be printed every time a branch statement is executed or when the value of certain variables are changed. This allows the user to get a clear picture of what his program is doing and thus be able to correct mistakes in his program.

6. *Peripheral interchange programs.* These utility programs facilitate transfer of data from one I/O device to another. They make possible the copying of data from one unit, for instance, magnetic tape, to another unit, for instance, magnetic disk. It is also possible to copy data from one tape unit to another tape unit or from one disk unit to another disk unit. This results in a more efficient utilization of the data preparation equipments.

QUESTIONS

1. What is an operating system ? Why is it necessary for a computer system ?
2. List out the various functions normally performed by an operating system.
3. "The operating system tends to isolate the hardware from the users". Discuss this statement.
4. What is a supervisor ? How does it differ from the transient routines of an operating system ?
5. Differentiate between the terms throughput and turnaround time.
6. Explain how jobs are processed in batch mode.
7. What are the advantages and disadvantages of batch processing ?
8. What are some of the reasons that JCL

statements are needed when you submit your job for computer processing ?

9. Would you say that all computers use the same types of JCL statements ? Why ?

10. You want to compile and execute a COBOL program. In plain English, list out the necessary JCL statements you will prepare for this job.

11. What is spooling ? How does it help in improving the efficiency of a computer system ?

12. Define multiprogramming. Explain how multiprograming ensures effective utilization of main memory and CPU.

13. Differentiate between I/O-bound and CPU-bound jobs.

14. List out some of the hardware and software facilities required for a multiprogramming system to work satisfactorily.

15. What is multiprocessing ? Give the basic organization of a multiprocessing system.

16. How is multiprocessing different from mutiprogramming ?

17. Discuss the advantages and limitations of mutiprocessing systems.

18. What is a time-slice ? In a timesharing system, explain how each and every user feels that he is the sole user of the computer system.

19. What are the three different states in which all users of a timesharing system fall ? Illustrate how a particular user switches from one state to another.

20. What is swapping ? How does it help an operating system in memory management ?

21. Why are timesharing systems considered to be most suitable for program development and testing ?

22. What type of hardware facilities are required for a timesharing computer system ?

23. Discuss the advantages and disadvantages of a timesharing system.

24. Mutiprogramming and timesharing both involve multiple users in the computer concurrently. What is the basic difference between the two concepts ?

25. What is meant by real-time processing ? Give some examples of real-time applications.

26. "A real-time system uses on-line processing but an on-line system need not necessarily operate in real-time mode." Explain.

27. Why is a high degree of reliability necessary for real-time systems ? How is this achieved ?

28. Will it be practical to use magnetic tape files in conjunction with a real-time system? Give reasons for your answer.

29. What do you understand by the term "response-time" ? Why is response time critical in a real-time system ?

30. What is a virtual memory ? How is it implemented ?

31. What are the two main advantages of a virtual memory system ?

32. What are library routines ? Why are they normally supplied by the computer manufacturers ?

33. What is a librarian? What are its functions ?

34. What is a text editor ?

35. How is a dynamic debugging tool used by a programmer ?

36. Explain how memory dump programs and trace routines help a programmer in finding out mistakes in his program.

37. What is the use of peripheral interchange programs ?

38. One of your friends wishes to use your account in the computer to enter and test his program. What information you must provide for him/her to proceed ?

CHAPTER 15

15. BUSINESS DATA PROCESSING CONCEPTS

This chapter deals with the basic concepts of business data processing. In this chapter, you will first learn the difference between data and information and the hierarchy of data storage. Then you will learn about the various types of file organizations and file utilities commonly used in business data processing applications. Finally, this chapter also introduces the basic concepts of data base systems and its advantages and limitations.

WHAT IS DATA PROCESSING

Data are a collection of facts - unorganized but able to be organized into useful information. A collection of sales orders, time sheets, and class registration cards are a few examples. Data are manipulated to produce output, such as bills and paychecks. When this output can be used to help people make decisions, it is called *information*.

Processing is a series of actions or operations that convert inputs into outputs. When we speak of data processing, the input is data, and the output is useful information. Hence, data processing is defined as series of actions or operations that converts data into useful information. The *data processing* system is used to include the resources such as people, procedures, and devices that are used to accomplish the processing of data for producing desirable output.

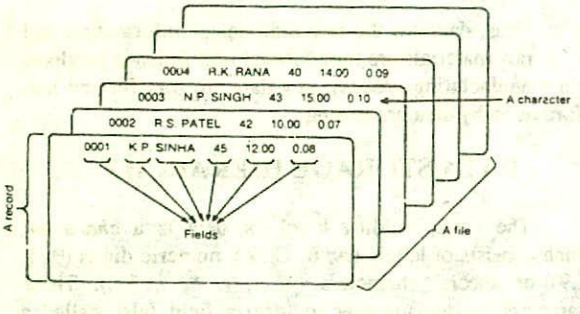
Thus, data are the raw material of information and just as raw materials are transformed into finished products by a manufacturing process, raw data are transformed into information by data processing.

DATA STORAGE-HIERARCHY

The basic building block of data is a *character*, which consists of letters (A, B, C...Z), numeric digits (0, 1, 2...9) or special characters (+, -, /, *, ., $...). These characters are put together to form a field (also called a fact, data item, or data element). A *field* is a meaningful collection of related characters. It is the smallest logical data entity that is treated as a single unit in data processing.

For example, if we are processing employees data of a company, we may have an employee code field, an employee name field, an hours worked field, an hourly-pay-rate field, a tax-rate-deduction field, etc. Fields are normally grouped together to form a record. A *record*, then, is a collection of related fields that are treated as a single unit. An employee record would be a collection of fields of one employee. These fields would include the employee's code, name, hours-worked, pay-rate, tax-rate-deduction, and so forth. Records are then grouped to form a file. A *file* is a number of related records that are treated as a unit. For example, a collection of all employee records for one company would be an employee file. Similarly, a collection of all inventory records for a particular company forms an inventory file. Figure 15.1 reveals these data relationships.

It is customary to set up a *master file* of permanent (and, usually, the latest) data, and to use *transaction files* containing data of a temporary nature. For example, the master payroll file will contain not only all the permanent details about each employee, his name and code, pay-rate, income tax rate and so forth, but it will also include the current gross-pay-to-date total and the tax paid-to-date total. The transaction payroll file will contain details of hours worked this week, normal and overtime, and, if piecework is involved, the quantity of goods made. When the payroll program is processed, both files will have to be consulted to generate this week's payslips, and the master file updated in readiness for the following week.



0004 R.K. RANA 40 14.00 0.09
0003 N.P. SINGH 43 15.00 0.10
A character
0002 R.S. PATEL 42 10.00 0.07
0001 K.P. SINHA 45 12.00 0.08
A record
Fields
A file

Figure 15.1. Relationship between character, field, record, and file.

A *data base* is a collection of integrated and related master files. It is a collection of logically related data elements that may be structured in various ways to meet the multiple processing and retrieval needs of organizations and individuals. Characters, fields, records, files, and data bases form a hierarchy of data storage. Figure 15.2 summarizes the data storage hierarchy used by computer-based processing systems. Characters are combined to make a field, fields are combined to make a record, records are combined to make a file, and files are combined to make a data base.

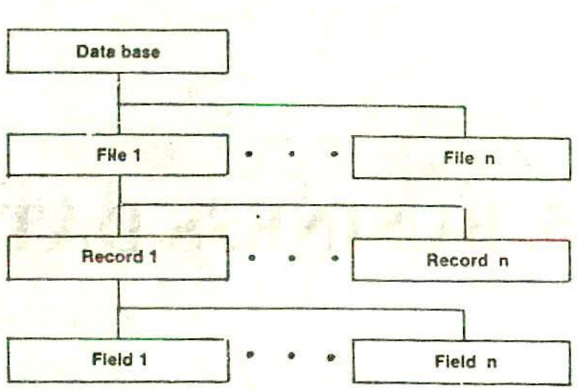


Figure 15.2. A data storage hierarchy.

FILE ORGANIZATIONS

System designers choose to organize, access, and process records and files in different ways depending on the type of application and the needs of users. The three commonly used file organizations used in business data processing applications are - sequential, direct and indexed sequential organizations. The selection of a particular file organization depends upon the type of application. The best organization to use in a given application is the one that happens to meet the user's needs in the most effective and economical manner. In making the choice for an application, designers must evaluate the distinct strengths and weaknesses of each file organization. File organization requires the use of some *key field* or unique identifying value that is found in every record in the file. The key value must be unique for each record of the file because duplications would cause serious problems. In the payroll example, the employee code field may be used as the key field.

SEQUENTIAL FILES

In a sequential file, records are stored one after another in an ascending or descending order determined by the key field of the records. In payroll example, the records of the employee file may be organized sequentially by employee code sequence. Sequentially organized files that are processed by computer systems are normally stored on storage media such as magnetic tape, punched paper tape, punched cards, or magnetic disks. To access these records, the computer must read the file in sequence from the beginning. The first record is read and processed first, then the second record in the file sequence, and so on. To locate a particular record, the computer program must read in each record in sequence and compare its key field to the one that is needed. The retrieval search ends only when the desired key matches with the key field of the currently read record. On an average, about half the file has to be searched to retrieve the desired record from a sequential file.

Advantages of sequential files

1. Easy to organize, maintain, and understand.

2. There is no overhead in address generation. Locating a particular record requires only the specification of the key field.

3. Relatively inexpensive I/O media and devices can be used for the storage and processing of such files.

4. It is the most efficient and economical file organization in case of applications in which there are a large number of file records to be updated at regularly scheduled intervals. That is, when the *activity ratio* (the ratio of the total number of records in transaction file and the total number of records in master file) is very high. Applications such as payroll processing, billing and statement preparation, and bank cheque processing meet these conditions.

Disadvantages of sequential files

1. It proves to be very inefficient and uneconomical for applications in which the activity ratio is very low.

2. Since an entire sequential file may need to be read just to retrieve and update few records, accumulation of transactions into batches is required before processing them.

3. Transactions must be sorted and placed in sequence prior to processing.

4. Timeliness of data in the file deteriorates while batches are being accumulated.

5. Data redundancy is typically high since the same data may be stored in several files sequenced on different keys.

DIRECT FILES

A direct file (also called a random or relative file) consists of records organized in such a way that it is possible for the computer to directly locate the key of the desired record without having to search through a sequence of other records. This means that the time required for online enquiry and updating of a few records is much faster than when batch techniques are used. However, a direct-access storage device (DASD) such as a drum, disk, strip file, or mass core is essential for storing a direct file.

A record is stored in a direct file by its key field. Although it might be possible to directly use the storage location numbers in DASD as the keys for the records stored in those locations, this is seldom done. Instead, an arithmetic procedure called *hashing* is frequently used. In this method, an address generating function is used to convert the record key number into a DASD storage address. The address generating function is selected in such a manner that the generated addresses should be distributed uniformly over the entire range of the file area and a unique address should be generated for each record key. However, in practice, the above constraints are usually not satisfied and the address generating function often maps a large number of records to the same storage address. Several methods are followed to overcome this problem of *collision* when it occurs. One approach is to include a pointer field at the location calculated by the hashing function. This field points to the DASD location of another record that has the same calculated address value. When the computer is given the key of a record to be processed at a later date, it reuses the hashing function to locate the stored record. If the record is found at the location calculated by the hashing function, the search is over and the record is directly accessed for processing. On the other hand, if the record at the calculated address does not have the correct key, the computer looks at the pointer field to continue the search.

Advantages of direct files

1. The access to, and retrieval of a record is quick and direct. Any record can be located and retrieved directly in a fraction of a second without the need for a sequential search of the file.

2. Transactions need not be sorted and placed in sequence prior to processing.

3. Accumulation of transactions into batches is not required before processing them. They may be processed as and when generated.

4. It can also provide up-to-the minute information in response to inquiries from simultaneously usable online stations.

5. If required, it is also possible to process direct file records sequentially in a record key sequence.

6. A direct file organization is most suitable for interactive online applications such as airline or railway reservation systems, teller facility in banking applications, etc.

Disadvantages of direct files

1. These files must be stored on a direct-access storage device. Hence, relatively expensive hardware and software resources are required.

2. File updation (addition and deletion of records) is more difficult as compared to sequential files.

3. Address generation overhead is involved for accessing each record due to hashing function.

4. May be less efficient in the use of storage space than sequentially organized files.

5. Special security measures are necessary for online direct files that are accessible from several stations.

INDEXED SEQUENTIAL FILES

We are all familiar with the concept of an index. For example, the directory in a large multistoried building is an index that helps us to locate a particular person's room within the building. For instance, to find the room of Dr. Sharma within the building, we would look up his name in the directory (index) and read the corresponding floor number and room number. This idea of scanning a logically sequenced table is preferable to searching door by door for the particular name. Similarly, if we wished to read the section in this book about printers, we would not begin on page 1 and read every page until we came across the topic of interest. Rather, we would find the subject in the contents (which serves as an index) to locate the page number, and then turn directly to that page to begin reading.

Indexed sequential files use exactly the same principle. The records in this type of file are organized in sequence and an index table is used to speed up access to the records without requiring a search of the entire file. The records of the file can be stored in random sequence but the index table is in sorted sequence on the key value. This provides the user with a very powerful tool. Not only can the file be processed randomly, but it can also be processed sequentially. Since the index table is in a sorted sequence on the key value, the file management system simply accesses the data records in the order of the index values. Thus indexed sequential files provide the user sequential access, even though the file management system is accessing the data records in a physically random order.

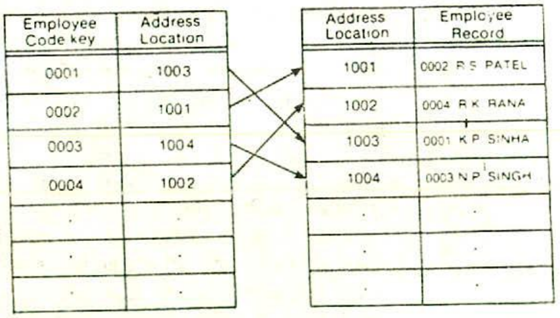


Figure 15.3. Organization of an indexed sequential file.

This concept is illustrated in Figure 15.3. This technique of file management is commonly referred to as the *Indexed Sequential Access Method (ISAM)*. Files of this type are called *ISAM files*.

Advantages of indexed sequential files

1. Permits the efficient and economical use of sequential processing techniques when the activity ratio is high.

2. Permits direct access processing of records in a relatively efficient way when the activity ratio is low.

Disadvantages of indexed sequential files

1. These files must be stored on a direct-access storage device. Hence, relatively expensive hardware and software resources are required.

2. Access to records may be slower than direct files.

3. Less efficient in the use of storage space than some other alternatives.

FILE UTILITIES

File utilities consist of routines which perform a variety of generalised operations on data files. Normally, file utilities are data independent. This means that the routines are written quite generally, and will operate on any data formats and even on data held on different types of storage medium. Some of the commonly used file utilities are discussed below.

SORTING

The purpose of sorting a file is to arrange records within a file in some defined sequence. This sequence is determined by the ordering of certain specified fields within the record. Fields whose ordering determine the sequence of a file in the sorting process are known as *keys*. The simplest case is an ordering on a single key. For example, a file of personnel records may be sequenced by ascending order of employee code as shown in Figure 15.4. A more complex ordering may be produced by introducing a further key in the sorting process. For example, suppose each record of the personnel file also contains a field for department number to which the employee belongs. Now the order of sorting may be employee code within department number. This means that all records for the lowest department number are presented first, eacn in

Employee code	Department Number
101	2
123	3
124	1
176	2
178	1
202	3
213	1

Figure 15.4. Sorting on one key in ascending employee code sequence.

Employee Code	Department Number
124	1
178	1
213	1
101	2
176	2
123	3
202	3

Figure 15.5. Sorting on two keys. Ascending employee code within ascending department number. Department number is primary key and employee code is secondary key.

ascending sequence of employee code : then all records for the next department number and so on. This is indicated in Figure 15.5. In this example, two keys have been used in the sorting process - department number is called the *primary key* and employee code is known as the *secondary key*.

According to the extent and sophistication of the sort utility available, the size and number of keys which can be specified, and the type of ordering (e.g. ascending, descending, alphabetical) will vary.

Since sorting is a very common data-processing requirement, manufacturers provide sort utility software which enables users to specify their particular sequencing requirements by means of simple parameters. Software is usually available for sorting files held on all types of storage devices. The user specifies the sort keys, and also details about the type of file such as storage device, file labels, record structure. The sort utility program reads the unsequenced input file, and by means of various copying techniques ultimately produces as output a copy of the input file in the required sequence.

SEARCHING

Searching is the process of scanning a file to find a particular record. The efficiency of a search algorithm depends on the file organisation. For example, to search a particular record in a sequential file, the file is scanned sequentially beginning with the first record and the desired key is compared one-by-one with the key field of each record. The search process terminates when the matching key is found. On the other hand, direct file organisation enables the program to have immediate access to the desired record. The program need only inform the file management system which record is needed, and the management system then searches through the filing system and produces the desired record. Normally, the time required to search a particular record from a direct file is much less as compared to the time required to search it from a sequential file.

MERGING

Merging of files involves the combining of records from two or more ordered files into a single ordered file. Each of the constituent files must be in the same order, although the record layout of files need not be identical. The output file will be in the same order as the input files, placing records from each in their correct relative order.

For example, in Figure 15.6, files A and B are merged to produce an output file C.

COPYING

File copying routines are provided for producing an exact copy of a file, either from one unit of a storage device onto another similar unit, e.g., from one tape reel to another, or from one storage medium to another, e.g., copying a card or disk file onto tape.

These routines are normally used for taking back-up copies of useful files. For example, a disk file may be copied on a tape or floppy for back-up purpose. File copying routines are also known as peripheral interchange programs since they are used to copy a file from one peripheral device onto another peripheral device.

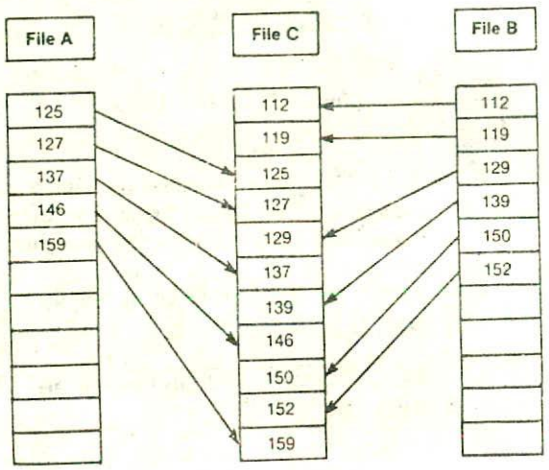


Figure 15.6. Merging of files A and B to produce file C.

PRINTING

Printing routines are used to print file contents upon a printer. File contents may be reproduced in different formats on the printer if there is a difference in the internal and external data formats, e.g. binary to character conversion. Special printing may be provided if the file contains program instructions rather than data. Some selection and editing facilities are normally provided with printing routines to enable parts of files to be output, e.g, specified number of records or blocks and certain portions only of records to be printed.

MAINTENANCE

File maintenance is the term given to any system of reorganisation of data items within a file, where the reorganisation is independent of the information content of the file. File maintenance software in effect is a form of selective copying. Extra facilities include the combining of data from more than one file, the deletion of records indentified by record key or record count within a file, and the selection of specific portions of records to be copied.

LABELLING

Files held on a storage device are identified by a special block of data held as the first block on the file. This block, called the file label, contains certain control information enabling the file contents to be identified, and may also contain additional information about the storage unit itself, such as the date when the unit was last written to, the number of times written to, and the serial number of the unit.

SCRATCHING

In systems where file labelling is used, it is generally not possible to write to a file unless the control information in the label block indicates that the data on the file is no longer valid. In any system, however, certain files may hold data which has become out of the date, or is invalid for other reasons, and thus is available for writing to. A routine must thus be available to alter the label so as to indicate that this state has been reached, even though the purge (make pure and clean) date has not yet been reached, or the file label is unchanged. The scratch routine will enable a new label to be written to a file with an already valid label, and must therefore be used with care, as it is possible to destroy valid data if wrongly used.

DATA BASE SYSTEMS

With the reduction in the cost of computer systems, data processing activities have increased drastically. Nowadays, computers are being used not only by almost all organizations, but also by the various departments of an organization. Thus, an organization may use computers for handling a variety of applications for its different departments. In a conventional data processing system, each seperate application had its own master file organized in a sequential, direct, or indexed sequential fashion. The records in each file were organized according to a single key field. Associated with each of these files was a set of programs for preparing required reports. Each file was processed virtually independently of the other files, and each file would include its own set of processing programs for processing that file.

This method of file processing was used on virtually all early computers and, indeed, is still commonly used today. However, for most modern processing needs, this type of system has a number of shortcomings, which are as follows:

1. *Data redundancy.* The same basic data fields are included in many different files. For example, suppose a file containing employee details is organized according to the employee code as key field. If we wish to get the names of employees with a certain educational qualification from this file, we need to search all file records. If the need for such an information became a routine, then a new file structured on an educational background key would be created and a new program would be written to process this file. Of course, this second file would duplicate much of the data stored in the first personnel file.

2. *Data inconsistency.* When changes occur in a data item, every file which contains that field should be updated to reflect the change. Confusion can result when one file is updated while another file containing the same field is not updated. This is a very common occurrence and leads to the problem of inconsistency. If one of the files is inadvertently not updated correctly, then the same entries in different files will be different (inconsistent) resulting in frequent discrepencies among reports produced from different files.

3. *Lack of program/data independence.* The programs used with file oriented applications usually contain "picture," "format," or "data" statements that precisely define each data field to be processed. Anytime if there is a need to add delete, or change data formats, the application program must also be changed. Likewise, a significant revision in a program may require a restructuring of the data file processed by the program. Changing programs to accommodate data format changes is a major maintenance activity in many data processing installations today.

4. *Wastage of resources.* Redundancy of data between files also results in wasted storage and wasted processing time in updating all files. If the same data item is stored in several files, obviously wasted storage will result. Moreover, the cost of entering and storing the same data in many files can be quite expensive.

THE DATA BASE CONCEPT

In order to overcome the problems of a conventional file-oriented data processing system, the data base concept was introduced Although there are differences of opinion about what constitutes a data base system, the most prevelant view is that such systems possess the following characteristics :

1. It is a centralized and integrated shared data file which consists of all data used by a company
2. It is organized and structured in a different manner than the conventional sequential file organizations.
3. Its organization permits access to any or all data quantities by all applications with equal ease.
4. Its organization is such that duplication of data is minimised if not eliminated entirely.
5. It emphasizes the independence of programs and data. It involves the concept of separating data definition from the applications programs and including it as part of the database.
6. It provides for the definition of logical relationships which exist between various records in the data base.
7. It is stored on a direct-access storage device.

The first step in moving from ordinary file management to a data base system is to separate all data definitions from the applications programs and to consolidate them into a separate entity called a *schema*, as illustrated in Figure 15.7. In addition to data definition, the schema also includes an indication of the logical relationships between various components of the data base. This is represented in Figure 15.7 by the data structure definitions. In other words, virtually everything there is to know about the data base and its structure is included in the schema.

The schema then becomes a component of the overall data base itself. From the schema the installation can generate dictionaries containing a complete description of the data base. These will, in turn, be used by systems analysts in defining new applications.

Data base systems are typically installed and coordinated by an individual called the *data base administrator*. He has the overall authority to establish and control data definitions and standards. He is responsible for determining the relationships among data elements, and for designing the data base security system to guard against unauthorised use. He also trains and assists applications programmers in the use of data base. A data dictionary is developed and used in a data base to document and maintain the data definitions.

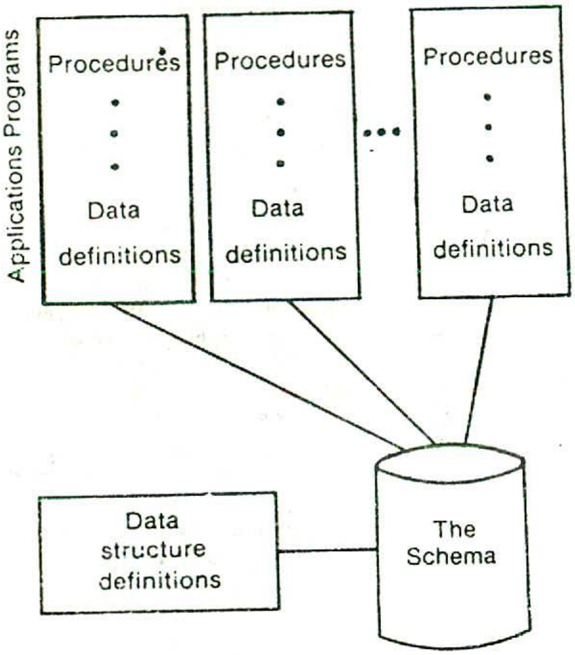


Figure 15.7. Illustrating the use of schema in data base for separating data definitions from programs.

DATA BASE MANAGEMENT SYSTEM

A collection of programs required to store and retrieve data from a data base is called a *data base management system (DBMS)*. As shown in Figure 15.8, the principal components of a DBMS are a data description module and a data manipulation module. The data description module of the DBMS analyzes the data requirements of applications programs and transfers control to the data manipulation module, which retrieves the needed data elements from the data base.

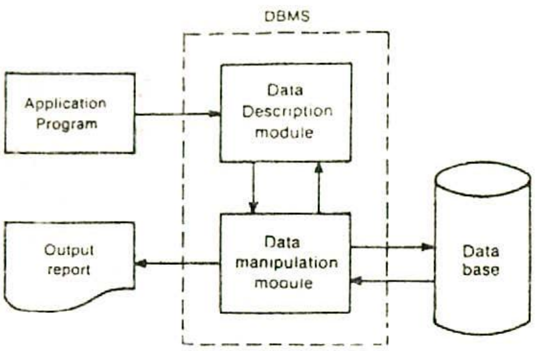


Figure 15.8. Principal components of a DBMS.

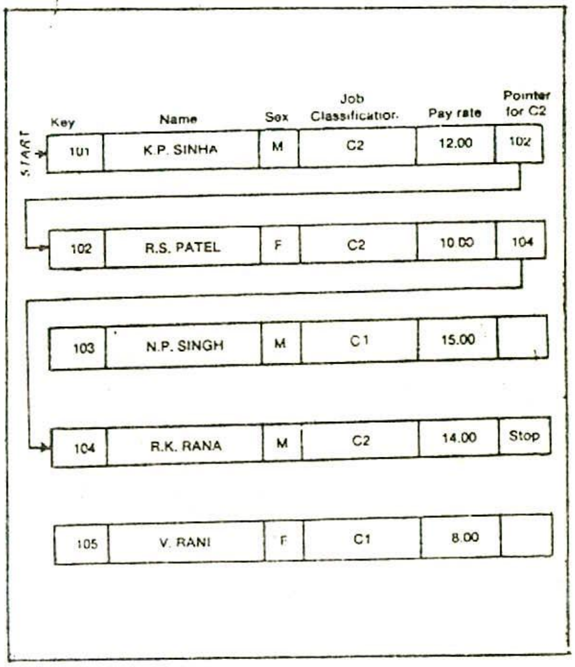
A DBMS can organize, process, and present selected data elements from the data base. This capability enables decision makers to search, probe, and query data base contents in order to extract answers that are not available in regular reports. For example, a query of the form "list out all male employees who are more than 45 years old and less than 50 years old and whose basic salary is more than Rs 3000/- per month" can easily be answered from an employee data base.

Data base management systems free the programmer from the need to worry about the organization and location of data. All of the data needed by an application program can be accessed, regardless of access method, record location, or record content. Programming is speeded up because the programmer can concentrate upon the logic of the application. Most DBMS are designed to interact with the commonly used programming languages such as COBOL. Many DBMS include special, user-friendly *query languages*. These languages can be easily learned by nonprogramming users of the system, enabling them to access the data base for information as needed without the help of any programmer.

DATA BASE STRUCTURING TECHNIQUES

We have seen how sequential, direct and indexed sequential approaches are used to organize and structure the data in *single* files. However a DBMS is able to integrate data elements from several files to answer specific user inquiries for information. This means that the DBMS is able to access and retrieve data from nonkey record fields. That is, the DBMS is able to structure and tie together the logically related data from several large files. Identifying these logical relationships is a job of the data base administrator. A *data definition language* is used for this purpose. The DBMS may then employ one of the following structuring techniques for the efficient storage, access, and retrieval operations of data.

List structures. In this method, records are linked together by the use of pointers. A *pointer* is a data item in a record that identifies the storage location of another logically related record. For example, let us assume that an employee file contains five records. There is one record for each employee. Now assume that we wish to find out all employees with a job classification of C2. One way to do this is to add a pointer at the end of each record for job classification C2. The pointer simply indicates (or points to) the address or location of next record containing an employee with a job classification of C2. Pointers for job classification C2 are shown in Figure 15.9. When a list of records are tied together using pointers, the list is called a *linked list*.



Key	Name	Sex	Job Classification	Pay rate	Pointer for C2
101	K.P. SINHA	M	C2	12.00	102
102	R.S. PATEL	F	C2	10.00	104
103	N.P. SINGH	M	C1	15.00	
104	R.K. RANA	M	C2	14.00	Stop
105	V. RANI	F	C1	8.00	

Figure 15.9. A list structure using a pointer for C2.

With pointer, the addresses and locations of related records are actually placed in the records themselves. An alternative to pointers is the index or inverted list. In this method, the relationship between various data elements is placed in a separate file or table called an *index* or an *inverted list*. It is referred to as an inverted list because the attribute values, such as sex, job classification, and pay rate are inverted with the keys used for direct access. In other words, the key for the record and the actual contents or attributes of the record are reversed. This allows us to start with an attribute, such as job classification, and determine which records contain employees with that particular attribute. Inverted lists for sex, job classification, and pay rates of employees of Figure 15.9 are shown in Figure 15.10.

Suppose if there is a query to determine all employees belonging to job class C1, then the computer goes to the job classification index and reads the pointer values for job classification C1. This allows the computer to directly access the records of employees 103 and 105, the employees belonging to job classification C1. Similarly, the index for sex can be used to list out all the male or all the female employees directly. Also the pay rate index can be used to find employees with certain pay rate ranges.

SEX INDEX	
Key Values	Pointer Values
M	101, 103, 104
F	102, 105

JOB CLASSIFICATION INDEX	
Key Values	Pointer Values
C1	103, 105
C2	101, 102, 104

PAY RATE INDEX	
Key Values	Pointer Values
5.01—10.00	102, 105
10.01—15.00	101, 103, 104

Fig 15.10. Example of indexes and inverted lists

Hierarchical or tree structures. In this method, data units are structured in multiple levels that graphically resemble an inverted tree with the root at the top and branches formed below. Below the single-root data component are subordinate elements or nodes, each of which, in turn, has one or more other elements. There is a parent-child relationship in a hierarchical structure. A parent node is one that has one or more subordinate elements or nodes. The data and records that are below the parent node are its children nodes. There may be numerous children nodes under each parent node, but there can be only one parent node for any one child node. Note that the branches in a tree structure are not connected. A typical hierarchical structure is shown in Figure 15.11.

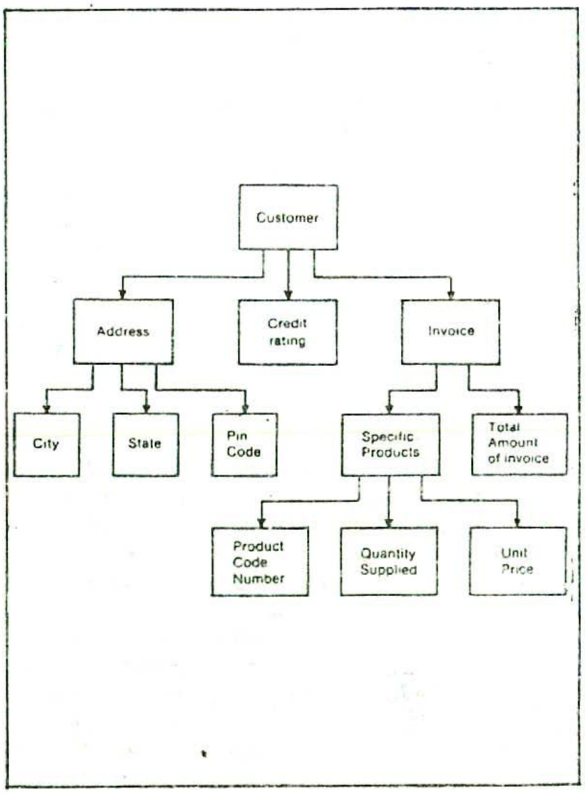


Figure 15.11. A typical hierarchical (tree) structure.

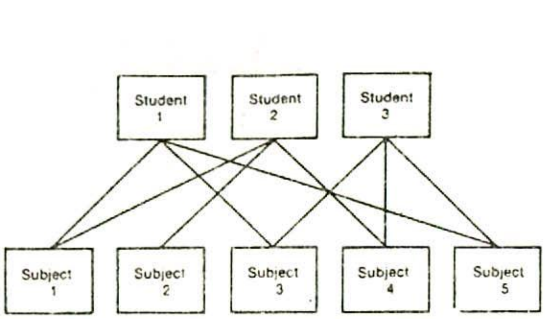


Figure 15.12. A typical network structure.

The hierarchical structure for data bases in very popular. A large number of computer programs have been written that use this structure. On the other hand, there are also disadvantages. While it is possible to force most data into a hierarchical structure, it can be difficult and awkward. In addition, a hierarchical structure can be more complex and more difficult to understand for managers and executives. Even then, the hierarchical structure continues to be a dominant one for large organizations, such as airline companies and credit card companies.

Network structures. A network structure is an extension of the hierarchical or tree structure. Instead of having only one parent node, however, the network structure can have multiple parent nodes for a child node. Thus, each node may have several owners and may, in turn, own any number of other data items. Data management software permits the extraction of the needed information from such a structure by beginning with any record in a file.

A typical network structure illustrating the relationship between students and the subjects taken by them is shown in Figure 15.12. There are three students and five subjects. The lines drawn between the students and the subjects show which students have taken which subjects. It can be easily seen that student 1 has taken subjects 1, 3, and 5, that student 2 has taken subjects 1, 2, and 4, and that student 3 has taken subjects 3, 4, and 5. Of course, there could be thousands of students and many more subjects.

Relational structures. The overall purpose of this model is to relate data records using a standard tabular format. It uses normal two-dimensional tables to describe all relationships between data. For example, relational tables could be established to link a college course with the instructor of the course, and with the location of the class as shown in Figure 15.13. To find the name of the instructor and the location of the Hindi class, the course/instructor relation is searched to get the name of the instructor (R. Pandey), and the course/location relation table is searched to get the class location (Room 210). Many other relations are, of course, possible.

COURSE/INSTRUCTOR RELATION TABLE

Course	Instructor
English	S.K. Ray
Hindi	R. Pandey
Physics	P.K. Sen
Chemistry	R.S. Gupta
Maths	N.P. Singh
•	•
•	•

COURSE/ LOCATION RELATION TABLE

Course	Location
English	Room 105
Maths	Room 203
Chemistry	Room 115
Physics	Room 108
Hindi	Room 201
•	•
•	•

OTHER RELATION TABLES

For example tables relating courses with time of meeting, days of meeting, hours of credit, etc.

Figure 15.13. Illustrating relational structure.

The relational model is a relatively new data base structuring technique. This model does not force us to use a structure, such as the hierarchical structure or the network structure. This is one of the advantages of the relational approach. The data in the relational model, in most cases, can be identical to the actual relationships that exist between the records and various data items. This is an important advantage over hierarchical and network models. As a result, there has been a considerable amount of interest in developing relational data base systems. Today relational models exist for large computers as well as small microcomputers.

ADVANTAGES AND LIMITATIONS OF DATA BASE SYSTEMS

In comparison to a conventional file-oriented data processing system, a data base system enjoys the following advantages :

1. The integration and sharing of data files minimizes the duplication and redundancy of data to a great extent.

2. Integration of data files also results in a considerable saving of storage space and in data entry and data storage costs.

3. Fewer applications programs need to be developed for obtaining various reports due to independence of programs and data.

4. The query language facility helps non-programming personnel to access the data base for information as needed without the help of any programmer.

5. Faster preparation of information to support nonrecurring tasks and changing conditions is possible.

6. Updation of data becomes easier due to integration of data files. Fewer errors may result when several records may be updated simultaneously.

On the other hand, the following are some of the limitations of a data base system :

1. More complex and expensive hardware and software resources are needed.

2. Sophisticated security measures must be implemented to prevent unauthorised access of sensitive data in online storage.

3. Hardware or software failures might result in the destruction of vital data base contents.

4. A lengthy conversion period may be needed, higher personnel training costs may be incurred, and more sophisticated skills are needed by those responsible for the data base system.

QUESTIONS

1. What is the difference between data and information ?

2. What is meant by data processing ?

3. What is a data processing system ?

4. Describe the data storage hierarchy.

5. Give an example to illustrate the relationship between a character, a field, a record, and a file.

6. What is the difference between a master file and a transaction file ?

7. How are data organized in business data processing systems ?

8. What is a key field ? What role does it play in file creation ?

9. How is a sequential file organized ? How are records in a sequential file accessed ? How are these records processed ?

10. What conditions support the use of sequential file ?

11. How is a record stored in a direct file ? How is it retrieved and processed ?

12. What conditions support the use of direct files and direct-access processing ?

13. How are records stored in an indexed sequential file ? How are they retrieved and processed ?

14. Discuss the advantages and limitations of the sequential, direct, and indexed sequential file approaches.

15. What are file utilities ?

16. Differentiate between the process of sorting and merging of files.

17. Explain the use of the following file utilities : copying, printing, labelling, and scratching.

18. What are the shortcomings of a conventional file oriented data processing system ?

19. What are the characteristics of a data base system ?

20. What is a data base schema ?

21. What are the jobs and responsibilities of a data base administrator ?

22. What is a data dictionary ?

23. What is a data base management system ? How is it used ?

24. Identify and discuss the structuring techniques used by data base management systems.

25. What is an inverted list ? How does it compare to a pointer ?

26. What is the difference between a hierarchical and a network structure ?

27. What are the advantages and limitations of a data base system ?