

## Unit 1

### What is System Programming?

System Programming consists of a variety of programs that support the operations of a computer. This software makes it possible for the user to focus on an application or the other problem to be solved. System programs (e.g. compiler, loader macro processors, and operating systems) were developed to make computer better adapted to the need of their users.

### Component of System Programming

Components of system programming are

1. Assemble
2. Loader
3. Compiler
4. Macros
5. Formal System

#### Assembler: -

- An assembler is a type of computer program that interprets software programs written in assembly language into machine language, code and instructions that can be executed by a computer.
- An assembler enables software and application developers to access, operate and manage a computer's hardware architecture and components.
- An assembler is sometimes referred to as the compiler of assembly language. It also provides the services of an interpreter.

#### Loader:-

- A *loader* is a program used by an operating system to load programs from a secondary to main memory so as to be executed.

#### Compiler:-

A *compiler* is a computer program (or a set of programs) that transforms source code written in a programming language (the source language) into another

computer language (the target language), with the latter often having a binary form known as object code.

### **Macro:-**

- A macro is a single line abbreviation for group of statement.
- A macro processor is a program that substitutes and specialized macro definitions for macro calls.

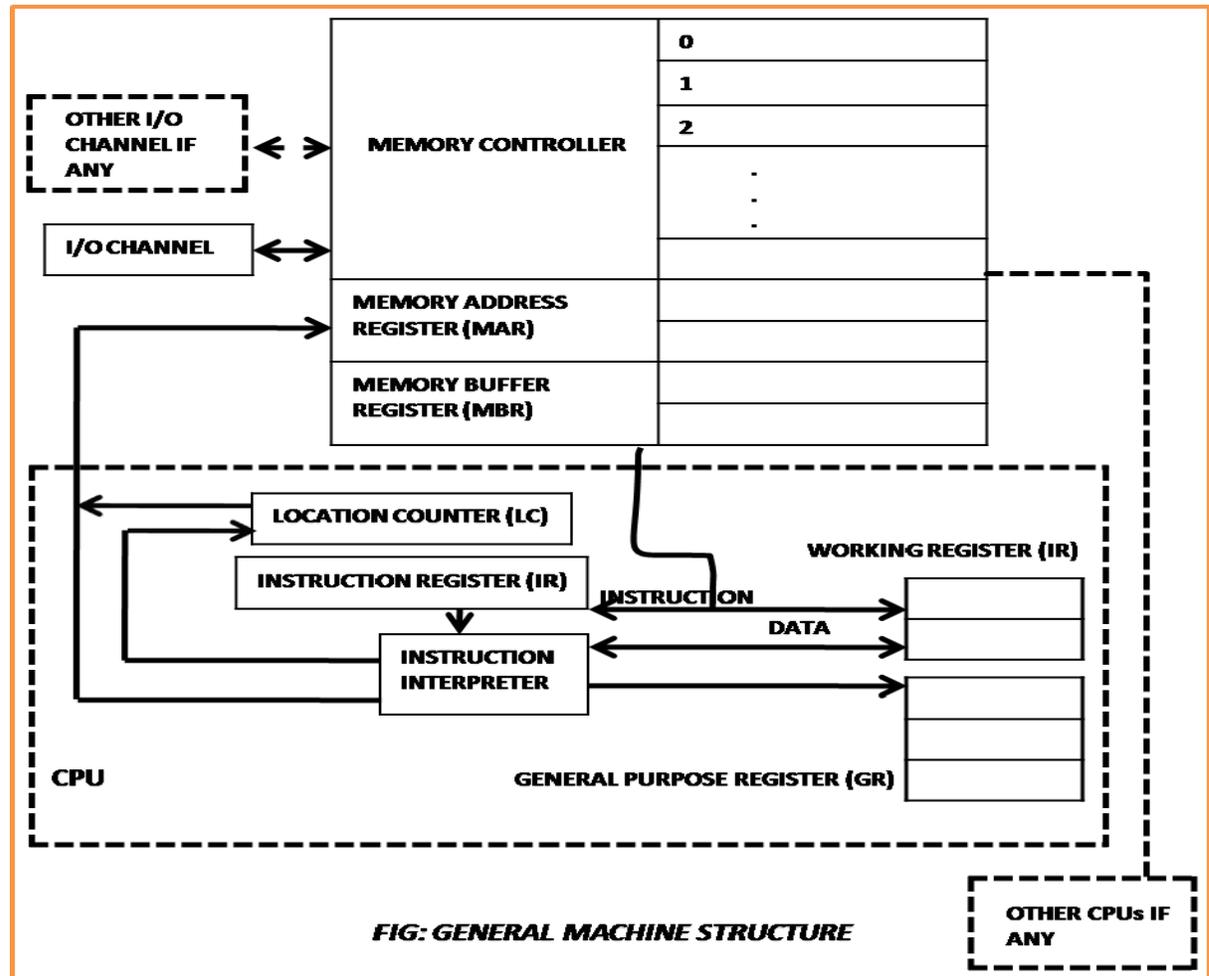
### **Formal System:-**

- A *formal system* consists of a **language** over some **alphabet** of **symbols** together with (**axioms** and) **inference rules** that distinguish some of the strings in the language as **theorems**.
- A formal system has the following components:
  - A **finite alphabet** of symbols.
  - A **syntax** that defines which strings of symbol are in the language of our formal system.
  - A **decidable** set of **axioms** and a finite set of **rules** from which the set of **theorems** of the system is generated. The rules must take a finite number of steps to apply.

----\*\*\*----

## General Machine Structure

All the conventional modern computers are based upon the concept of stored program computer, the model that was proposed by John von Neumann.



The components of a general machine are as follows:

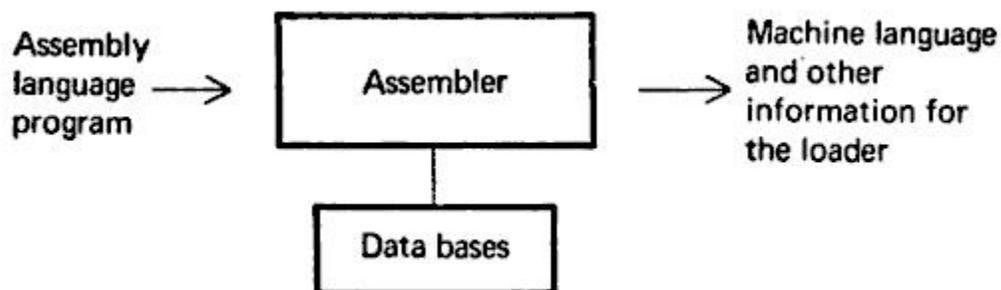
1. **Instruction interpreter:** A group of electronic circuits performs the intent of instruction of fetched from memory.
2. **Location counter:** LC otherwise called as program counter PC or instruction counter IC, is a hardware memory device which denotes the location of the current instruction being executed.
3. **Instruction register:** A copy of the content of the LC is stored in IR.
4. **Working register:** are the memory devices that serve as “scratch pad” for the instruction interpreter.

5. **General register:** are used by programmers as storage locations and for special functions.
6. **Memory address registers (MAR):** contains the address of the memory location that is to read from or stored into.
7. **Memory buffer register (MBR):** contain a copy of the content of the memory location whose address is stored in MAR. The primary interface between the memory and the CPU is through memory buffer register.
8. **Memory controller:** is a hardware device whose work is to transfer the content of the MBR to the core memory location whose address is stored in MAR.
9. **I/O channels:** may be thought of as separate computers which interpret special instructions for inputting and outputting information from the memory.

----\*\*\*----

### Assembly Language

- An assembler is a program that takes computer instruction and converts them into a pattern of bits that the computer processor can use to perform its basic operation.
- The assembler is responsible for translating the assembly language program into machine code. When the source program language is essentially a symbolic representation for a numerical machine language, the translator is called assembler and the source language is called an assembly language.



### Basic function of Assembler

- Translate mnemonics opcodes to machine language.

- Convert symbolic operands to their machine addresses.
- Build machine instructions in the proper format
- Convert data constants into machine representation.
- Error checking is provided.
- Changes can be quickly and easily incorporated with a reassembly.
- Variables are represented by symbolic names, not as memory locations.
- Assembly language statements are written one per line. A machine code program thus consists of a sequence of assembly language statements, where each statement contains a mnemonics.

### **Advantages**

- Reduced errors
- Faster translation times
- Changes could be made easier and faster.
- Addresses are symbolic, not absolute \
- Easy to remember

### **Disadvantages**

- Assembler language are unique to specific types of computer
- Program is not portable to the computer.
- Many instructions are required to achieve small tasks
- Programmer required knowledge of the processor architecture and instruction set.

### **Translation phase of Assembler**

The six steps that should be followed by the designer

1. Specify the problem
2. Specify data structure
3. Define format of data structure
4. Specify algorithm
5. Look for modularity
6. Repeat 1 through 5 on modules

### **Functions / Purpose of Assembler**

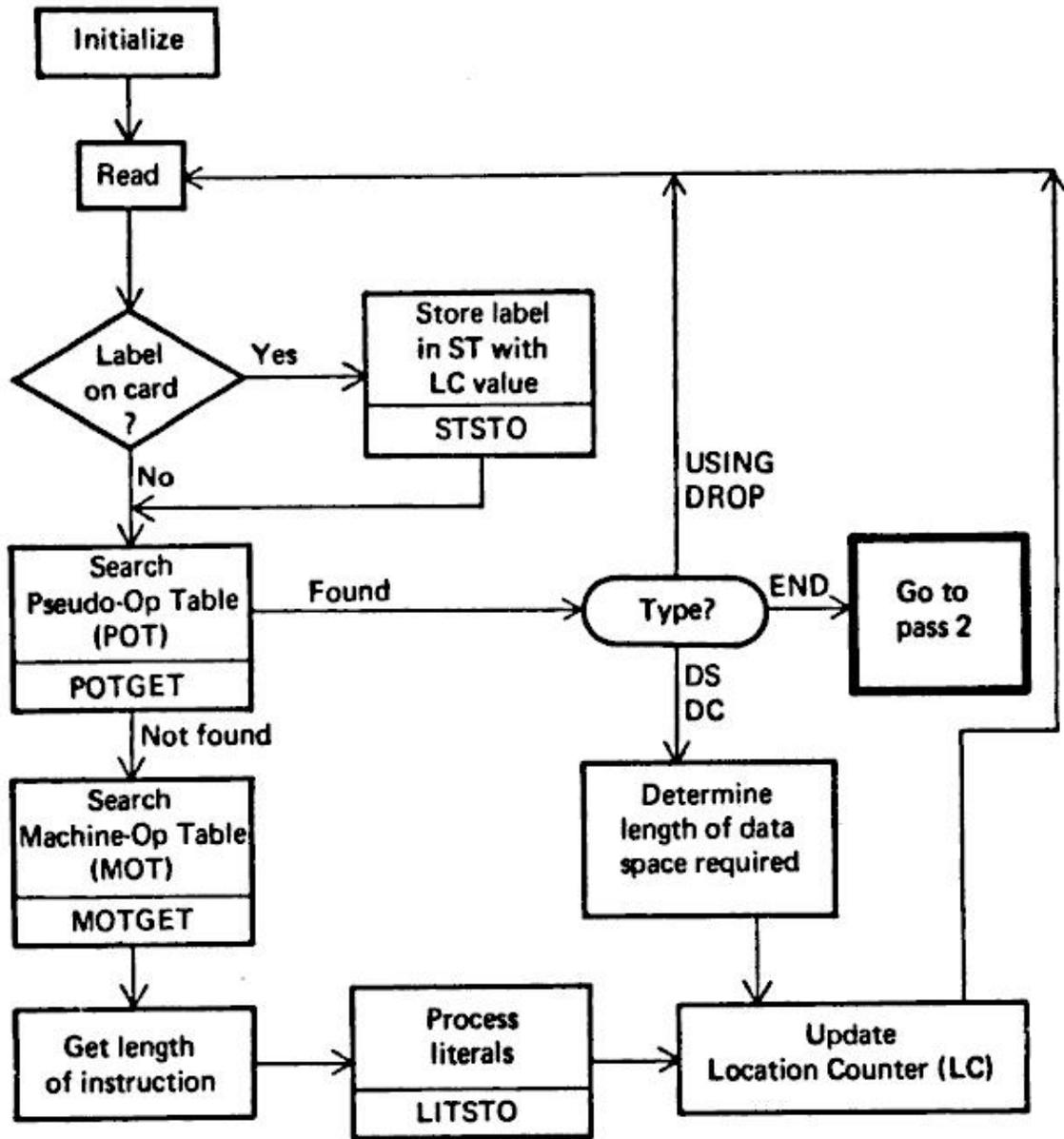
An assembler must do the following

1. Generate instruction
  - a. Evaluate the mnemonics in the operation field to produce the machine code
  - b. Evaluate the subfield-fine the value of each symbol. Process literals and assign addresses.
2. Process pseudo ops
  - a. Pass 1 (Define symbol and literals)
    - i. Determine length of machine instruction ( MOTGET)
    - ii. Keep track of location counter (LC)
    - iii. Remember value of symbol until pass 2 (STSTO)
    - iv. Process some pseudo ops(POTGET1)
    - v. Remember literal (LITSTO)
  - b. Pass 2 (Generate object Program)
    - i. Look up value of symbol (STGET)
    - ii. Generate instruction (MOTGET2)
    - iii. Generate data (for DS, DC and Literal)
    - iv. Process pseudo ops (POTGET2)

**Design data structure for assembler design in Pass-1 and Pass-2 with flow chart**

**Pass -1**

1. Input source program
2. A location counter used to keep track of each instruction location.
3. A table, the machine –operation table (MOT) that indicate the symbolic mnemonics for each instruction and its length (two, four or six bytes)
4. A table, the pseudo operation table (POT) that indicate the symbolic mnemonics and action to be taken for each pseudo-op in pass-1
5. A table, the literal table (LT) that is used to store each literal encounter and its corresponding assigned location.
6. A table, the symbol table (ST) that is used to store each label and its corresponding value.
7. A copy of the input to be used later by Pass-2. This may be stored in a secondary storage device

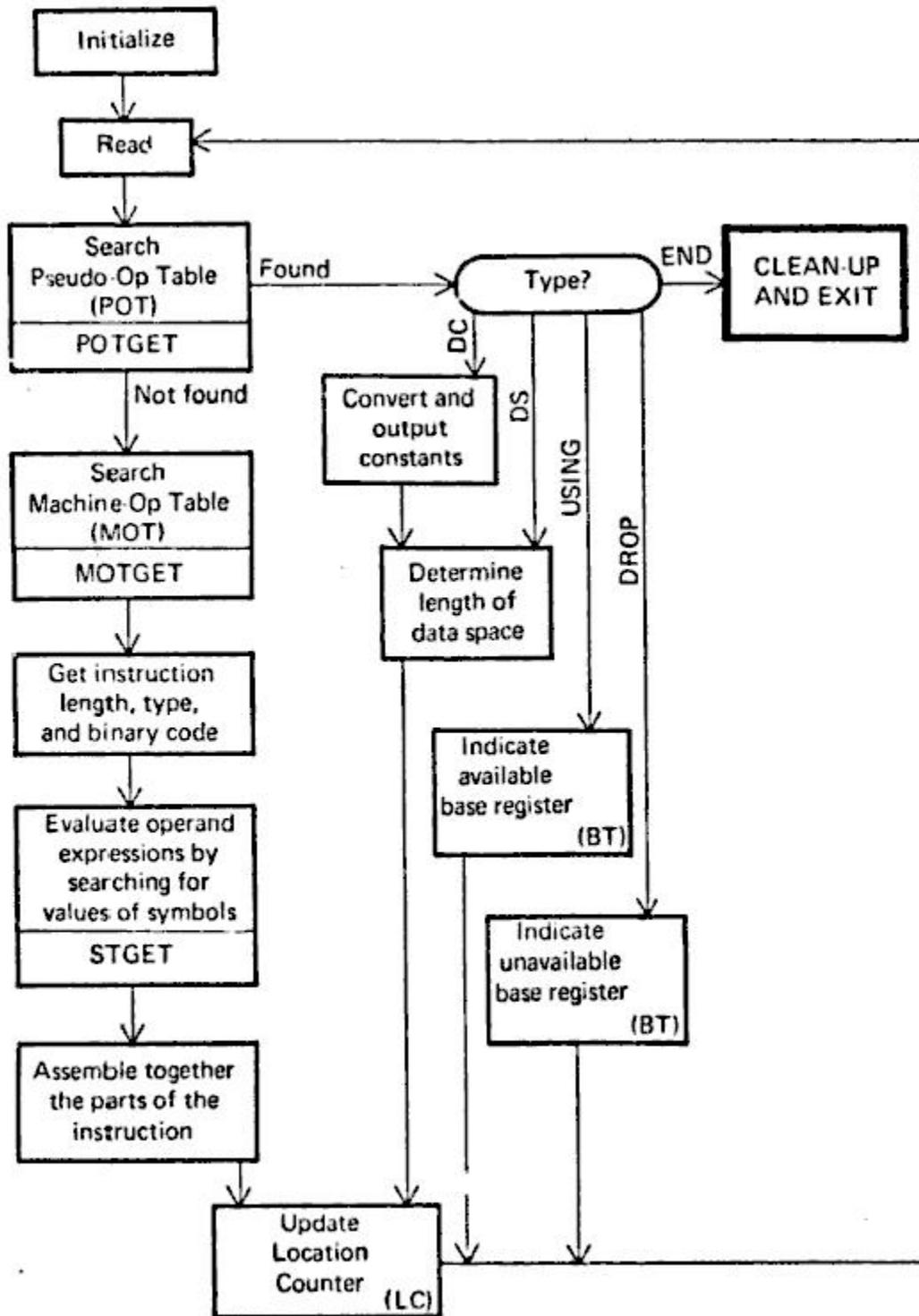


**FIGURE 3.3** Pass 1 overview: define symbols

### Pass-2

1. Copy of source program input to pass-1
2. Location counter
3. A table the MOT that indicates for each instruction
  - a. Symbolic
  - b. Mnemonics
  - c. Length
  - d. Binary machine op-code
  - e. Format (RR, RS, RX, SI, SS)

4. A table the POT that indicates for each pseudo-op the symbolic mnemonic and the action to be taken in Pass-2
5. The ST prepare by Pass-1, containing each label and its corresponding value.
6. A table, BT that indicate which register are currently specified by base register by USING pseudo-ops and what are the specified contents of these register.
7. A work space INSR that is to hold each instruction as its various parts are being assembled together.
8. A workspace PRINT LINE used to produce a printed listing
9. A workspace PUNCH CARD used prior to actual outputting for converting assembled instruction into the format needed by the loader.
10. An output deck of assembled instruction in the format needed by the loader.



**FIGURE 3.4** Pass 2 overview: evaluate fields and generate code

## 1. Machine Operation Table (MOT)

Fig machine op table for pass-1 and pass-2 the op code is the key and its value is the binary op-code equivalent which is stored for use in generating machine op-code. The instruction length is stored for use in updating the location counter, the instruction format for use in forming the machine language equivalent.

← 6-bytes per entry →				
Mnemonic op-code (4-bytes) (characters)	Binary op-code (1-byte) (hexadecimal)	Instruction length (2-bits) (binary)	Instruction format (3-bits) (binary)	Not used in this design (3-bits)
"Ab <b>bb</b> "	5A	10	001	
"AH <b>bb</b> "	4A	10	001	
"AL <b>bb</b> "	5E	10	001	
"ALR <b>b</b> "	1E	01	000	
"AR <b>bb</b> "	1A	01	000	
...	...	...	...	
"MVC <b>b</b> "	D2	11	100	
...	...	...	...	

*b* ~ represents the character "blank"

### Codes:

#### Instruction length

01 = 1 half-words = 2 bytes  
 10 = 2 half-words = 4 bytes  
 11 = 3 half-words = 6 bytes

#### Instruction format

000 = RR  
 001 = RX  
 010 = RS  
 011 = SI  
 100 = SS

## 2. Pseudo Operation Table (POT)

The table will actually contain the physical address. Fig POT for pass-1, each pseudo-op is listed with an associated pointer to the assembler routine for processing the pseudo-op.

← 8-bytes per entry →	
Pseudo-op (5-bytes) (character)	Address of routine to process pseudo-op (3-bytes = 24 bit address)
"DROP <b>b</b> "	P1DROP
"END <b>bb</b> "	P1END
"EQU <b>bb</b> "	P1EQU
"START"	P1START
"USING"	P1USING

↑  
 These are presumably labels of routines in pass 1; the table will actually contain the physical addresses.

FIGURE 3.7 Pseudo-Op Table (POT) for pass 1 (similar table for pass 2)

## 3. Symbol Table (ST)

Symbol table is usually hash-organized. It contains all relevant information about symbol defined and used in the source program. Information regarding all forward references in a symbol table, for each symbol and address is generated randomly and the information regarding that symbol get the same address conflict is resolve by using collision handling techniques.

The relative location indicator tells the assembler whether the value of the symbol is absolute or relative base of the program.

← 14-bytes per entry →			
Symbol (8-bytes) (characters)	Value (4-bytes) (hexadecimal)	Length (1-byte) (hexadecimal)	Relocation (1-byte) (character)
"JOHNbbb"	0000	01	"R"
"FOURbbb"	000C	04	"R"
"FIVEbbb"	0010	04	"R"
"TEMPbbb"	0014	04	"R"

**FIGURE 3.8** Symbol Table (ST) for pass 1 and pass 2

#### 4. Base Table (BT)

Base table is used by the assembler to generate the proper base register references in machine instruction and to compute the correct offsets, when generating an address, the assembler referencing the base register table to chose a base register that will contains a value close to the symbolic references. The address is the information using that base register.

← 4-bytes per entry →		↑ 15 entries ↓
Availability indicator (1-byte) (character)	Designated relative-address Contents of base register (3-bytes = 24-bit address) (hexadecimal)	
1	"N"	—
2	"N"	—
⋮	⋮	
14	"N"	—
15	"Y"	00 00 00

Code=

**Availability**

Y ~ register specified in USING pseudo-op

N ~ register never specified in USING pseudo-op or subsequently made unavailable by the DROP pseudo-op

**FIGURE 3.9** Base Table (BT) for pass 2

## Difference Between

### Difference between compiler and interpreter

Sr. No.	Compiler	Interpreter
1	Scans the entire program first and then translate it into machine code	Translate the program line-by-line.
2	Convert the entire program to machine code; when all the syntax errors have been removed execution takes place.	Each time the program is executed, every line is checked for syntax error and then converted to equivalent machine code
3	Execution time is less	Execution time is more
4	Machine code can be saved and used; source code and compiler no longer needed.	Machine code cannot be saved; interpreter is always required for translation.
5	Since source code is not required tampering with the source code is not possible	Source code can be easily modified and hence no security of programs.
6	Slow for debugging	Fast for debugging

### Difference between process and Programs

Sr. No.	Process	Programs
1	A process is an instance of program in execution	Program is a set of instructions written to carry out a particular task
2	Process is a dynamic concept	Program is a static concept
3	A process is termed as an 'active entity' since it is always stored in the main memory and disappears if the machine is power cycled. Several processes may be associated with a same program.	A program is an executable file residing on the disk (secondary storage) in a directory
4		It is read into the primary memory and executed by the kernel.
5	A <b>process is the actual execution</b> of those instructions.	A computer <b>program is a passive collection</b> of instructions;

### Difference between multiprogramming and multiprocessing

Sr. No.	Multiprocessing	Multiprogramming
1	Multiprocessing refers to processing of multiple processes at same time by multiple CPUs.	Multiprogramming keeps several programs in main memory at the same time and execute them concurrently utilizing single CPU.
2	It utilizes multiple CPUs.	It utilizes single CPU.
3	It permits parallel processing.	Context switching takes place.
4	Less time taken to process the jobs.	More Time taken to process the jobs.
5	It facilitates much efficient utilization of devices of the computer system.	Less efficient than multiprocessing.
6	Usually more expensive.	Such systems are less expensive.

### Difference between Open Subroutine and Closed Subroutine

Sr. No.	Open Subroutine	Closed Subroutine
1	Open subroutine is one whose code inserted into the main program	Close subroutine can be stored outside the main routine and control transfer to the subroutine
2	If some open subroutine where called four times. It would appear in four different places in the calling program	Close subroutine perform transfer of control and transfer of data
3	Arguments are passed in the registers that are given as arguments to the subroutine.	Arguments may be placed in registers or on the stack
4	Open Subroutines are very efficient with no wasted instructions	A subroutine also allows you to debug code once and then ensure that all future instantiations of the code

		will be correct
5	Open Subroutines are very flexible and can be as general as the program wishes to make them	Any register that the subroutine uses must first be <b>saved</b> and then <b>restored</b> after the subroutine completes execution

### Difference between Pure Procedure and Impure Procedure

Sr. No.	Pure Procedure	Impure Procedure
1	A pure procedure does not modify itself.	Procedures that modify themselves are called impure procedures.
2	It can be shared by multiple processors	Other program finds them difficult to read and moreover they cannot shard by multiple processors.
3	Pure procedures are readily reusable.	Impute procedures are not readily reusable.
4	To ensure that the instructions are the same each time a program is used.	Each processor executing an impure procedure modifies its contents.
5	Writing such procedure is a good programming practice	Writing such procedure is a poor programming practice

### Difference between user viewpoint and System viewpoint

Operating System is designed both by taking user view and system view into consideration.

1. The goal of the Operating System is to maximize the work and minimize the effort of the user.
2. Most of the systems are designed to be operated by single user, however in some systems multiple users can share resources, memory. In these cases Operating System is designed to handle available resources among multiple users and CPU efficiently.

3. Operating System must be designed by taking both usability and efficient resource utilization into view.
4. In embedded systems (Automated systems) user view is not present.
5. Operating System gives an effect to the user as if the processor is dealing only with the current task, but in background processor is dealing with several processes.

## System View

1. From the system point of view Operating System is a program involved with the hardware.
2. Operating System is allocator, which allocate memory, resources among various processes. It controls the sharing of resources among programs.
3. It prevents improper usage, error and handles deadlock conditions.
4. It is a program that runs all the time in the system in the form of Kernel.
5. It controls application programs that are not part of Kernel.

-----\*\*\*-----

## General Approaches to New Machines

In order to know a new machine we have a number of questions in mind. These questions can be categorized as follows.

- **Memory:** Basic unit, size and addressing scheme.
- **Registers** Number of registers, and size, functions, interrelation of each register.
- **Data:** Types of data and their storing scheme.
- **Instruction:** Classes of instructions, allowable operations and their storing scheme.
- **Special Features:** Additional features like interrupt and protections.

## The lists of opcodes used in the program are as follows

- **USING** is a pseudo code the indicates to the assembler which general purpose register to use as a base and what its contents will be. As we do not have any specific general register acting as the base register, so it becomes necessary to inform indicate a base register for the program. Because the address are relative so by the knowledge of base and offset the program can be easily be located and executed.

- **BALR** is machine opcode that load a register with the next address and branch to the address in the second field. Since second operand is 0 so the control will go to the next instruction.
- **START** is a pseudo opcode that tell the assembler where the beginning of the program is and allows the user to give a name to the program.
- **END** is a pseudo code that tells the assembler that the last card of the program has been reached.
- **BR 14**, the last machine opcode is used to branch to the location whose address is in general purpose register 14. By convention, calling programs leave their return address in register 14.

### **Literals**

The same program is repeated by using literals, that are mechanisms where by the assembler creates data areas for the programmer, containing constants he requests. =F'10', =F'49', =F'4' are the literal which would be result in the creation of a data area containing 10, 49 and 4 and replacement of the literal operand with the address of the data it describes. L 3, =F'10' is translated by the assembler to point to a full word that contains a 10. Generally the assembler keeps track of the literal with the help of a literal table. This table will contain all the constants that have been requested through the use of literal. A pseudo opcode LTOrg place the literal at an earlier location. This is required because, the program may be using 10000 data and it become difficult for the offset of the load instruction to reach the literal at the end of the program.

### **Data Format of IBM 360/370**

The 360 may store several different types of data as is depicted in the figure. The groups of bits stored in memory are interpreted by 360 processor in several ways. The list of different interpretation are shown in the figure are as follow.

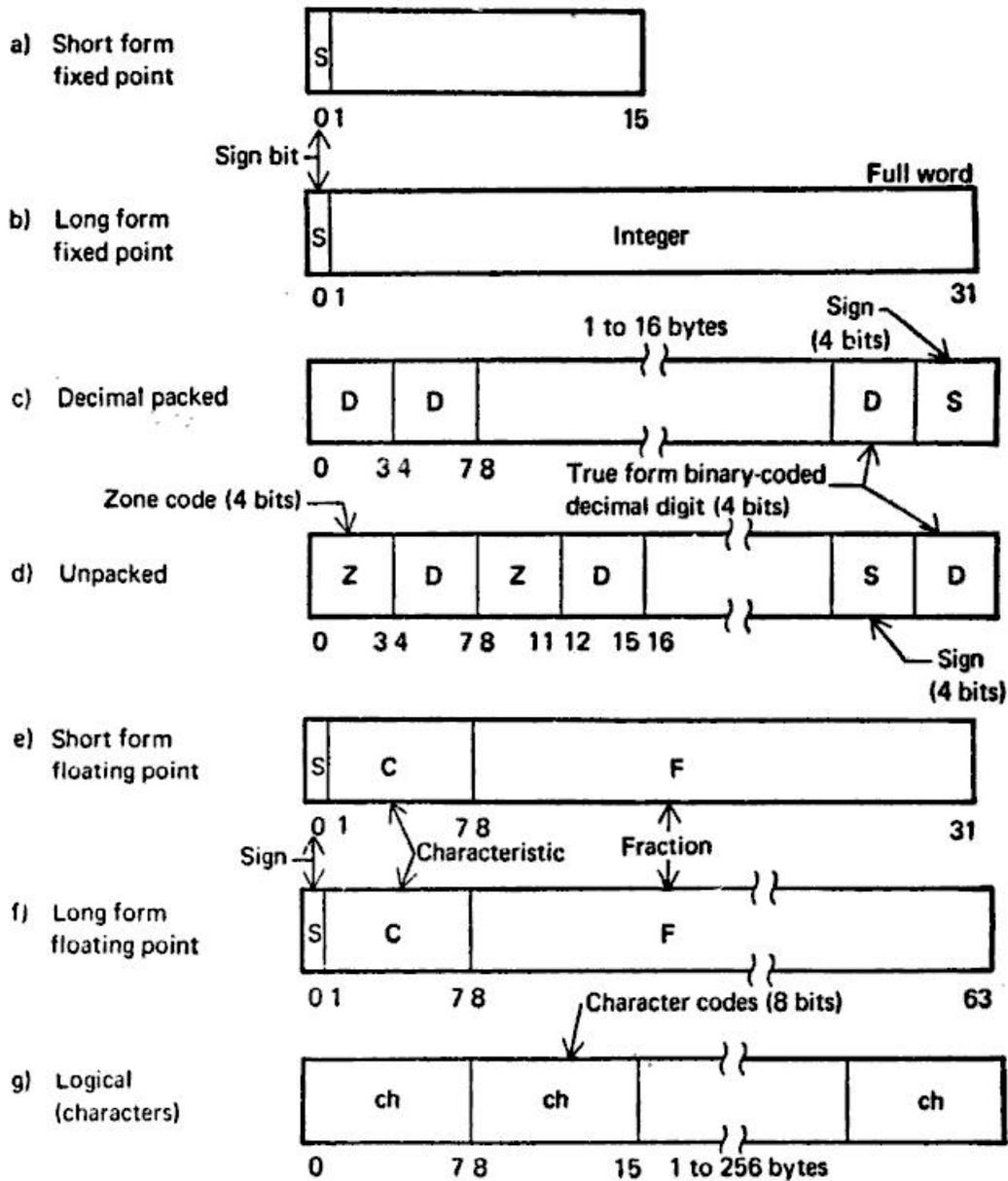
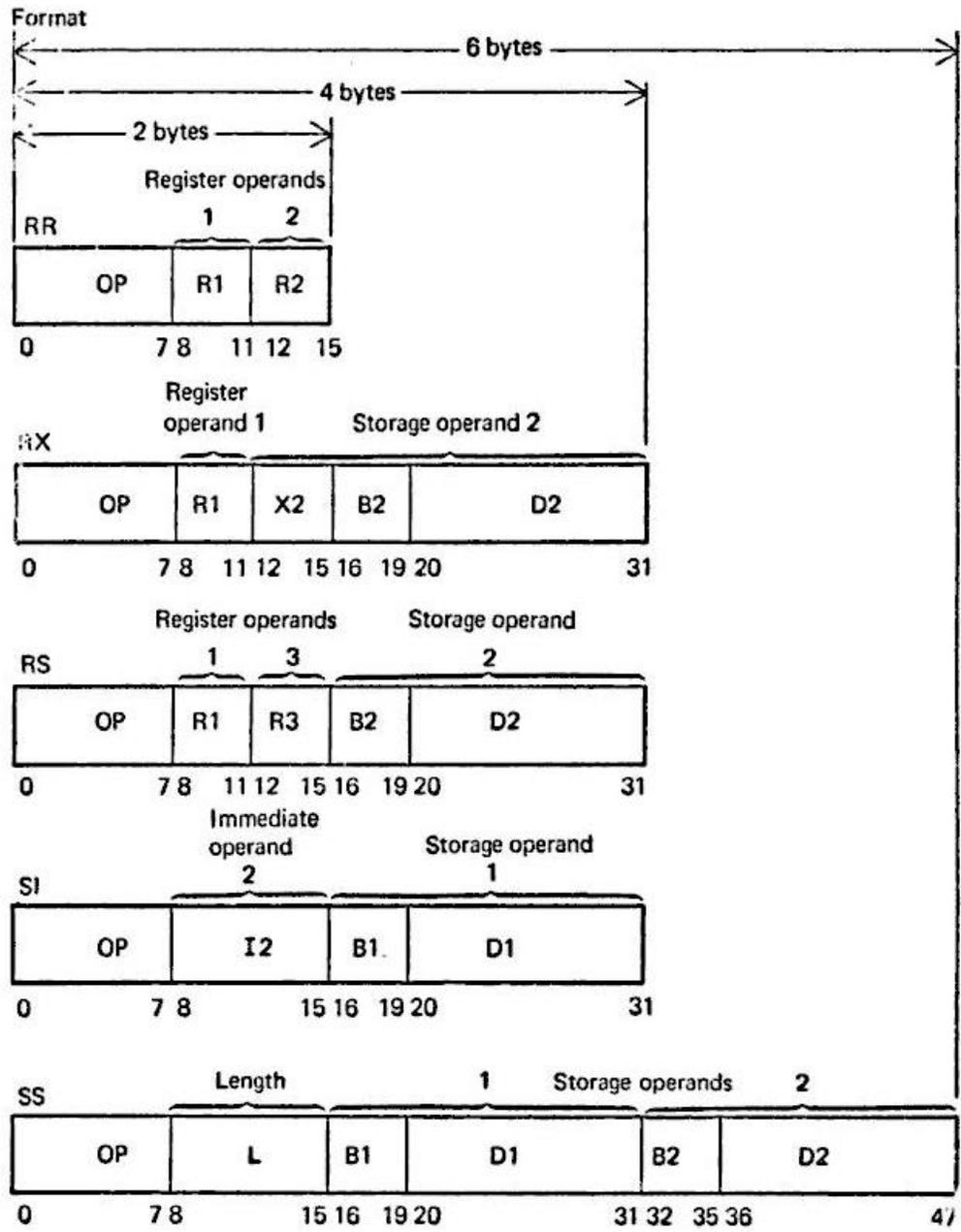


FIGURE 2.3 Data formats for the system/360 and 370

-----//-----

### Instruction Format

The instructions in 360 can be arithmetic, logical, control or transfer and special interrupt instructions. The format of 360 instructions is as in figure above.



Mnemonics used:

- OP ~ operation code
- Ri ~ contents of general register used as operand
- Xi ~ contents of general register used as index
- Bi ~ contents of general register used as base
- Di ~ displacement
- Ii ~ immediate data
- L ~ operand length

FIGURE 2.5 Basic 360 instruction formats

There are five types of instructions that differ in the type of operands they use.

Register operand refers to the data stored in the 16 general purpose registers (32 bits each). Registers being high-speed circuits provide faster access to data than the data in the core.

E.g. Add register 3, 4 causes the contents of the contents of the register 4 to be added to that of register 3 and stored back in the register 3. The instruction is represented as given in the diagram. This is called as RR format. A total of two bytes are required to represent the RR instruction 8 bits for opcode and 4 bits each of register (8+4+4=16 bits =2 bytes).

The address of ith storage operand is computed from the instruction in the following manner:

$$\text{Address} = c(B_i) + c(X_i) + D_i \quad (\text{RX format})$$

$$\text{or} = c(B_i) + D_i \quad (\text{RS, SI, SS format})$$

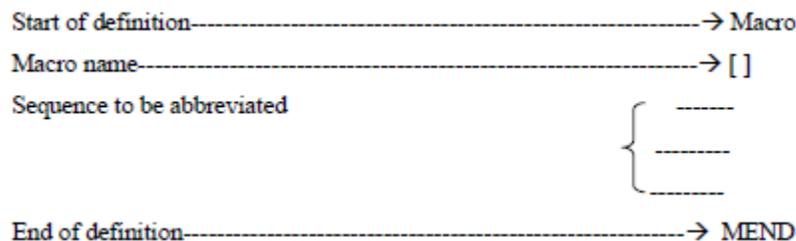
## Unit 2

### Definition

Macros are single-line abbreviations for a certain group of instructions. Once the macro is defined, these groups of instructions can be used anywhere in a program.

It is sometimes necessary for an assembly language programmer to repeat some blocks of code in the course of a program. The programmer needs to define a single machine instruction to represent a block of code for employing a macro in the program. The macro proves to be useful when instead of writing the entire block again and again, you can simply write the macro that you have already defined. An assembly language macro is an instruction that represents several other machine language instructions at once.

Macro facility permits you to attach a name to the sequence that is occurring several times in a program and then you can easily use this name when that sequence is encountered. All you need to do is to attach a name to a sequence with the help of the macro instruction definition. The following structure shows how to define a macro in a program:



This structure describes the macro definition in which the first line of the definition is the MACRO pseudo-op. Following line is the name line for macro, which identifies the macro

instruction name. The line following the macro name includes the sequence of instructions that are being abbreviated. Each instruction comprises of the actual macro instruction. The last statement in the macro definition is MEND pseudo-op. This pseudo-op denotes the end of the macro definition and terminates the definition of macro instruction.

## MACRO EXPANSION

Once a macro is being created, the interpreter or compiler automatically replaces the pattern, described in the macro, when it is encountered. The macro expansion always happens at the compile-time in compiled languages. The tool that performs the macro expansion is known as macro expander. Once a macro is defined, the macro name can be used instead of using the entire instruction sequence again and again.

As you need not write the entire program repeatedly while expanding macros, the overhead associated with macros is very less. This can be explained with the help of the following example.

Source	Expanded Source
MACRO	
INC	
A 1,DATA	
A 2,DATA	
A 3,DATA	
MEND	
⋮	
INC	A 1,DATA
⋮	A 2,DATA
⋮	A 3,DATA
⋮	⋮
INC	A 1,DATA
⋮	A 2,DATA
⋮	A 3,DATA
⋮	⋮
DATA DC F'2'	DC DATA

The macro processor replaces each macro call with the following lines:

```
A 1, DATA
A 2, DATA
A 3, DATA
```

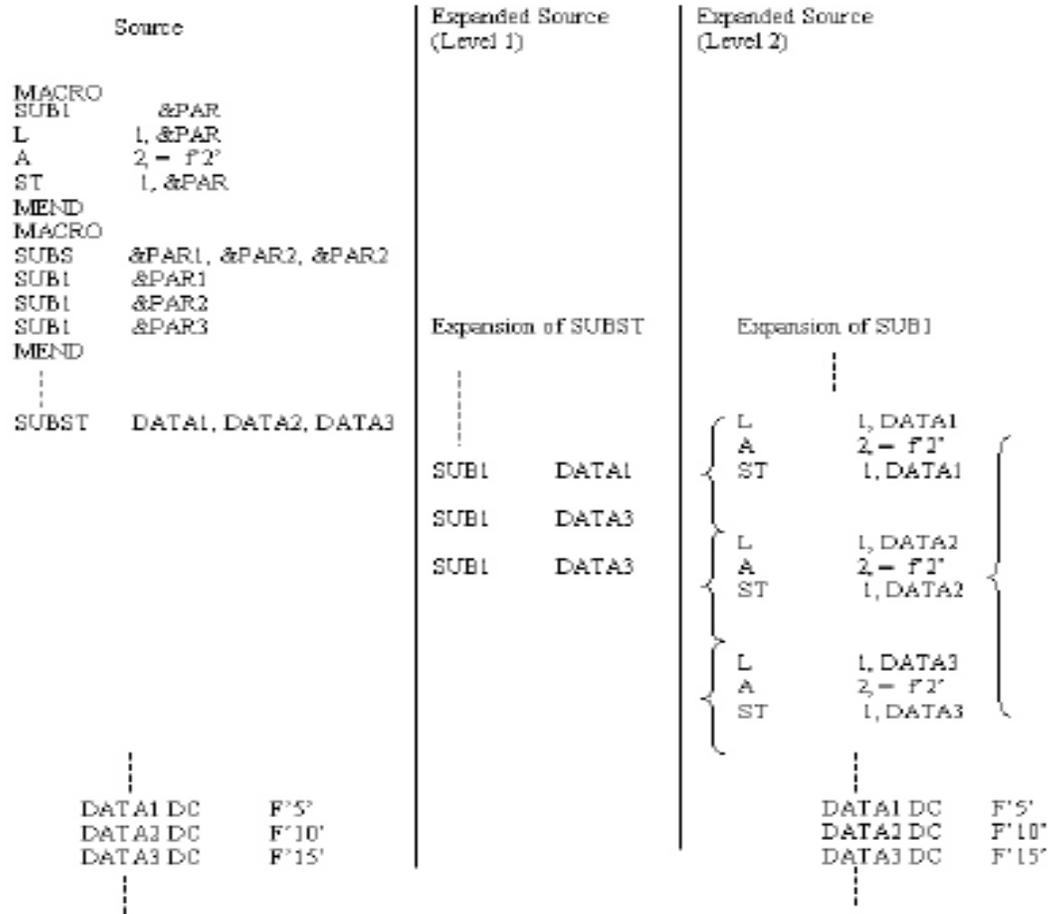
The process of such a replacement is known as expanding the macro. The macro definition itself does not appear in the expanded source code. This is because the macro processor saves the definition of the macro. In addition, the occurrence of the macro name in the source program refers to a macro call. When the macro is called in the program, the sequence of instructions corresponding to that macro name gets replaced in the expanded source.

## NESTED MACRO CALLS

Nested macro calls refer to the macro calls within the macros. A macro is available within other macro definitions also. In the scenario where a macro call occurs, which contains another macro call; the macro processor generates the nested macro definition as text and places it on the input stack. The definition of the macro is then scanned and the macro processor compiles it. This is important to note that the macro call is nested and not the macro definition. If you nest the macro definition, the macro processor compiles the same macro repeatedly, whenever the section of the outer macro is executed. The following example can make you understand the nested macro calls:

```
MACRO
SUB1 &PAR
L    1, &PAR
A    1, =F'2'
ST   1, &PAR
MEND
MACRO
SUBST &PAR1, &PAR2, &PAR3
SUB1 &PAR1
SUB1 &PAR2
SUB1 &PAR3
2MEND
```

You can easily notice from this example that the definition of the macro 'SUBST' contains three separate calls to a previously defined macro 'SUB1'. The definition of the macro SUB1 has shortened the length of the definition of the macro 'SUBST'. Although this technique makes the program easier to understand, at the same time, it is considered as an inefficient technique. This technique uses several macros that result in macro expansions on multiple levels.



This is clear from the example that a macro call, SUBST, in the source is expanded in the expanded source (Level 1) with the help of SUB1, which is further expanded in the expanded source (Level 2).

## FEATURES OF MACRO FACILITY

The features of the macro facility are as follows:

- Macro instruction arguments
- Conditional macro expansion
- Macro instructions defining macros

### 1. Macro Instruction Arguments

The macro facility presented so far inserts block of instructions in place of macro calls. This facility is not at all flexible, in terms that you cannot modify the coding of the macro name for a specific macro call. An important extension of this facility consists of providing the arguments or parameters in the macro calls. Consider the following program.

```

:
:
:
A      1,DATA1
A      2,DATA1
A      3,DATA1
:
:
:
A      1,DATA2
A      2,DATA2
A      3,DATA2
:
:
:
A      1,DATA3
A      2,DATA3
A      3,DATA3
      DATA1 DC      F'5'
      DATA2 DC      F'10'
      DATA3 DC      F'15'

```

In this example, the instruction sequences are very much similar but these sequences are not identical. It is important to note that the first sequence performs an operation on an operand DATA1. On the other hand, in the second sequence the operation is being performed on operand DATA2. The third sequence performs operations on DATA3. They can be considered to perform the same operation with a variable parameter or argument. This parameter is known as a macro instruction argument or dummy argument.

Source			Expanded source
MACRO	Macro INC has one argument		
INC	&PAR		
A	1, &PAR		
A	2, &PAR		
A	3, &ARG		
MEND			
⋮			
INC	DATA1	Use DATA1 as operand	{ A 1,DATA1 A 2,DATA1 A 3,DATA1
⋮			
INC	DATA2	Use DATA2 as operand	
⋮			{ A 1,DATA2 A 2,DATA2 A 3,DATA3
INC	DATA3	Use DATA3 as operand	
⋮			
⋮			{ A 1,DATA3 A 2,DATA3 A 3,DATA3
DATA1	DC	F'5'	
DATA2	DC	F'10'	
DATA3	DC	F'15'	DATA1 DC F'5'
⋮			DATA2 DC F'10'
			DATA3 DC F'15'
⋮			

Notice that in this program, a dummy argument is specified on the macro name line and is distinguished by inserting an ampersand (&) symbol at the beginning of the name. There is no limitation on supplying arguments in a macro call. The important thing to understand about the macro instruction argument is that each argument must correspond to a definition or dummy argument on the macro name line of the macro definition. The supplied arguments are substituted for the respective dummy arguments in the macro definition whenever a macro call is processed.

## 2. Conditional Macro Expansion

These macro expansions permit conditional reordering of the sequence of macro expansion. They are responsible for the selection of the instructions that appear in the expansions of a macro call. These selections are based on the conditions specified in a program. Branches and tests in the macro instructions permit the use of macros that can be used for assembling the instructions. The facility for selective assembly of these macros is considered as the most powerful programming tool for the system software. The use of the conditional macro expansion can be explained with the help of an example.

Consider the following set of instructions:

```

LOOP 1      A      1,      DATA1
            A      2,      DATA2
            A      3,      DATA3
            ⋮
            ⋮
LOOP 2      A      1,      DATA3

```

```

A      2,    DATA2
      :
      :
DATA1   DC F'5'
DATA2   DC F'10'
DATA3   DC F'15'

```

In this example, the operands, labels and number of instructions generated are different in each sequence. Rewriting the set of instructions in a program might look like:

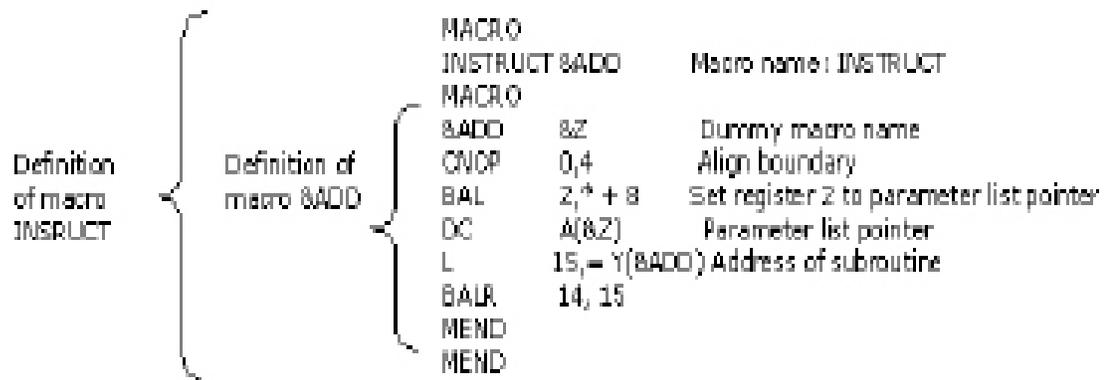
	:	MACRO		
&PAR0 VARY		&COUNT, &PAR1, &PAR2, &PAR3		
	A	1, &PAR1		
	AIF	(&COUNT EQ 1). FINI	Test if &COUNT = 1	
	A	2, &PAR2		
	AIF	(&COUNT EQ 2). FINI	Test if &COUNT = 2	
	ADD	3, &PAR3		
.FINI		MEND		
	:			Expanded source
LOOP1 VARY		3, DATA1, DATA2, DATA3		LOOP1 A 1, DATA1
	:			A 2, DATA2
	:			A 3, DATA3
LOOP2 VARY		2, DATA3, DATA2		LOOP2 A 1, DATA3
	:			A 2, DATA2
	:			
DATA1 DC		F'5'		
DATA2 DC		F'10'		
DATA3 DC		F'15'		

The labels starting with a period (.) such as .FINI are macro labels. These macro labels do not appear in the output of the macro processor. The statement AIF (& COUNT EQ 1).FINI directs the macro processor to skip to the statement labeled .FINI, if the parameter corresponding to &COUNT is one. Otherwise, the macro processor continues with the statement that follows the AIF pseudo-op. AIF pseudo-op performs an arithmetic test and since it is a conditional branch pseudo-op, it branches only if the tested condition is true. Another pseudo-op used in this program is AGO, which is an unconditional branch pseudo-op and works as a GOTO statement. This is the label in the macro instruction definition that specifies the sequential processing of instructions from the location where it appears in the instruction. These statements are indications or directives to the macro processor that do not appear in the macro expansions.

### 3. Macro Instructions Defining Macros

A single macro instruction can also simplify the process of defining a group of similar macros. The considerable idea while using macro instructions defining macros is that the inner macro definition should not be defined until the outer macro has been called once. Consider a macro instruction INSTRUCT in which another subroutine &ADD is also defined.

This is explained in the following macro instruction.



In this code, first the macro INSTRUCT has been defined and then within INSTRUCT, a new macro &ADD is being defined. Macro definitions within macros are also known as “macro definitions within macro definitions”.

## DESIGN OF A MACRO PRE-PROCESSOR

A Macro pre-processor effectively constitutes a separate language processor with its own language. A macro pre-processor is not really a macro processor, but is considered as a macro translator. The approach of using macro pre-processor simplifies the design and implementation of macro pre-processor. Moreover, this approach can also use the features of macros such as macro calls within macros and recursive macros. Macro pre-processor recognises only the macro definitions that are provided within macros. The macro calls are not considered here because the macro pre-processor does not perform any macro expansion.

The macro preprocessor generally works in two modes: passive and active. The passive mode looks for the macro definitions in the input and copies macro definitions found in the input to the output. By default, the macro pre-processor works in the passive mode. The macro pre-processor switches over to the active mode whenever it finds a macro definition in the input. In this mode, the macro preprocessor is responsible for storing the macro definitions in the internal data structures. When the macro definition is completed and the macros get translated, then the macro pre-processor switches back to the passive mode.

Four basic tasks that are required while specifying the problem in the macro pre-processor are as follows:

1. **Recognising macro definitions:** A macro pre-processor must recognize macro definitions that are identified by the MACRO and MEND pseudo-ops. The macro definitions can be easily recognised, but this task is complicated in cases where the macro definitions appear within macros. In such situations, the macro pre-processor must recognise the nesting and correctly matches the last MEND with the first MACRO.
2. **Saving the definitions:** The pre-processor must save the macro instructions definitions that can be later required for expanding macro calls.

3. **Recognising macro calls:** The pre-processor must recognise macro calls along with the macro definitions. The macro calls appear as operation mnemonics in a program.
4. **Replacing macro definitions with macro calls:** The pre-processor needs to expand macro calls and substitute arguments when any macro call is encountered. The pre-processor must substitute macro definition arguments within a macro call.

## Implementation of Two-Pass Algorithm

The two-pass algorithm to design macro pre-processor processes input data into two passes. In first pass, algorithm handles the definition of the macro and in second pass; it handles various calls for macro. Both the passes of two-pass algorithm in detail are:

### 1. First Pass

The first pass processes the definition of the macro by checking each operation code of the macro. In first pass, each operation code is saved in a table called Macro Definition Table (MDT). Another table is also maintained in first pass called Macro Name Table (MNT). First pass uses various other databases such as Macro Name Table Counter (MNTC) and Macro Name Table Counter (MDTC). The various databases used by first pass are:

1. The input macro source deck.
2. The output macro source deck copies that can be used by pass 2.
3. The Macro Definition Table (MDT), which can be used to store the body of the macro definitions. MDT contains text lines and every line of each macro definition, except the MACRO line gets stored in this table. For example, consider the code described in macro expansion section where macro INC used the macro definition of INC in MDT. Table 2.1 shows the MDT entry for INC macro:

**Table 2.1: MDT**

Macro Definition Table (MDT)		
&LAB	INC	&ARG1, &ARG2, &ARG3
#0	A	1, #1
	A	2, #2
	A	3, #3
	MEND	

4. The Macro Name Table (MNT), which can be used to store the names of defined macros. Each MNT entry consists of a character string such as the macro name and a pointer such as index to the entry in MDT that corresponds to the beginning of the macro definition. Table 2.2 shows the MNT entry for INCR macro:

**Table 2.2: MNT**

Macro Definition Table (MDT)		
&LAB	INC	&ARG1, &ARG2, &ARG3
#0	A	1, #1
	A	2, #2
	A	3, #3
	MEND	

5. The Macro Definition Table Counter (MDTC) that indicates the next available entry in the MDT.
6. The Macro Name Table Counter (MNTC) that indicates the next available entry in the MNT.
7. The Argument List Array (ALA) that can be used to substitute index markers for dummy arguments prior to store a macro definition. ALA is used during both the passes of the macro pre-processor. During Pass 1, dummy arguments in the macro definition are replaced with positional indicators when the macro definition is stored. These positional indicators are used to refer to the memory address in the macro expansion. It is done in order to simplify the later argument replacement during macro expansion. The  $i^{\text{th}}$  dummy argument on the macro name card is represented in the body of the macro by the index marker symbol #. The # symbol is a symbol reserved for the use of macro pre-processor.

**Table 2.3: ALA**

Macro Definition Table (MDT)		
&LAB	INC	&ARG1, &ARG2, &ARG3
#0	A	1, #1
	A	2, #2
	A	3, #3
	MEND	

## 2. Second Pass

Second pass of two-pass algorithm examine each operation mnemonic such that it replaces macro name with the macro definition. The various data-bases used by second pass are:

1. The copy of the input macro source deck.
2. The output expanded source deck that can be used as an input to then assembler.
3. The MDT that was created by pass 1.
4. The MNT that was created by pass 1.
5. The MDTP for indicating the next line of text that is to be used during macro expansion.
6. The ALA that is used to substitute macro calls arguments for the index markers in the stored macro definition.

## 3 .Two-Pass Algorithm

In two-pass macro-preprocessor, you have two algorithms to implement, first pass and second pass. Both the algorithms examines line by line over the input data available. Two algorithms to implement two-pass macro-preprocessor are:

- Pass 1 Macro Definition
- Pass 2 Macro Calls and Expansion

### **Pass 1 - Macro Definition**

Pass 1 algorithm examines each line of the input data for macro pseudo opcode. Following are the steps that are performed during Pass 1 algorithm:

1. Initialize MDTC and MNTC with value one, so that previous value of MDTC and MNTC is set to value one.
2. Read the first input data.
3. If this data contains MACRO pseudo opcode then
  - A. Read the next data input.
  - B. Enter the name of the macro and current value of MDTC in MNT.
  - C. Increase the counter value of MNT by value one.
  - D. Prepare that argument list array respective to the macro found.
  - E. Enter the macro definition into MDT. Increase the counter of MDT by value one.
  - F. Read next line of the input data.
  - G. Substitute the index notations for dummy arguments passed in macro.
  - H. Increase the counter of the MDT by value one.
  - I. If mend pseudo opcode is encountered then next source of input data is read.
  - J. Else expands data input.
4. If macro pseudo opcode is not encountered in data input then
  - A. A copy of input data is created.
  - B. If end pseudo opcode is found then go to Pass 2.
  - C. Otherwise read next source of input data.

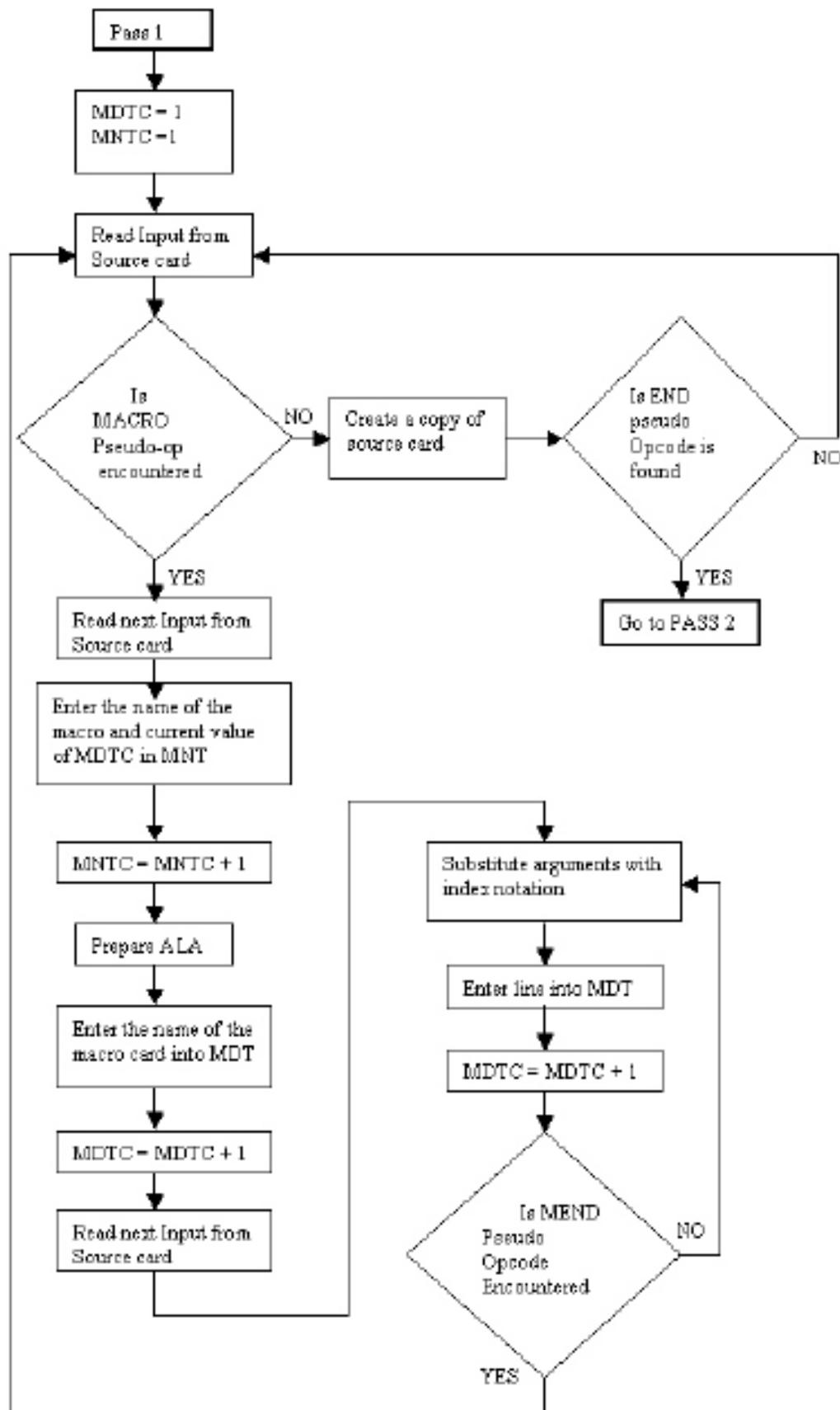


Figure 2.1: Flow Chart for First Pass

## Pass 2 - Macro Calls and Expansion

Pass two algorithm examines the operation code of every input line to check whether it exist in MNT or not. Following are the steps that are performed during second pass algorithm:

1. Read the input data received from Pass 1.
2. Examine each operation code for finding respective entry in the MNT.
3. If name of the macro is encountered then
  - A. A Pointer is set to the MNT entry where name of the macro is found. This pointer is called  
Macro Definition Table Pointer (MDTP).
  - B. Prepare argument list array containing a table of dummy arguments.
  - C. Increase the value of MDTP by value one.
  - D. Read next line from MDT.
  - E. Substitute the values from the arguments list of the macro for dummy arguments.
  - F. If end pseudo opcode is found then next source of input data is read.
  - G. Else expands data input.
4. When macro name is not found then create expanded data file.
5. If end pseudo opcode is encountered then feed the expanded source file to assembler for processing.
6. Else read next source of data input.

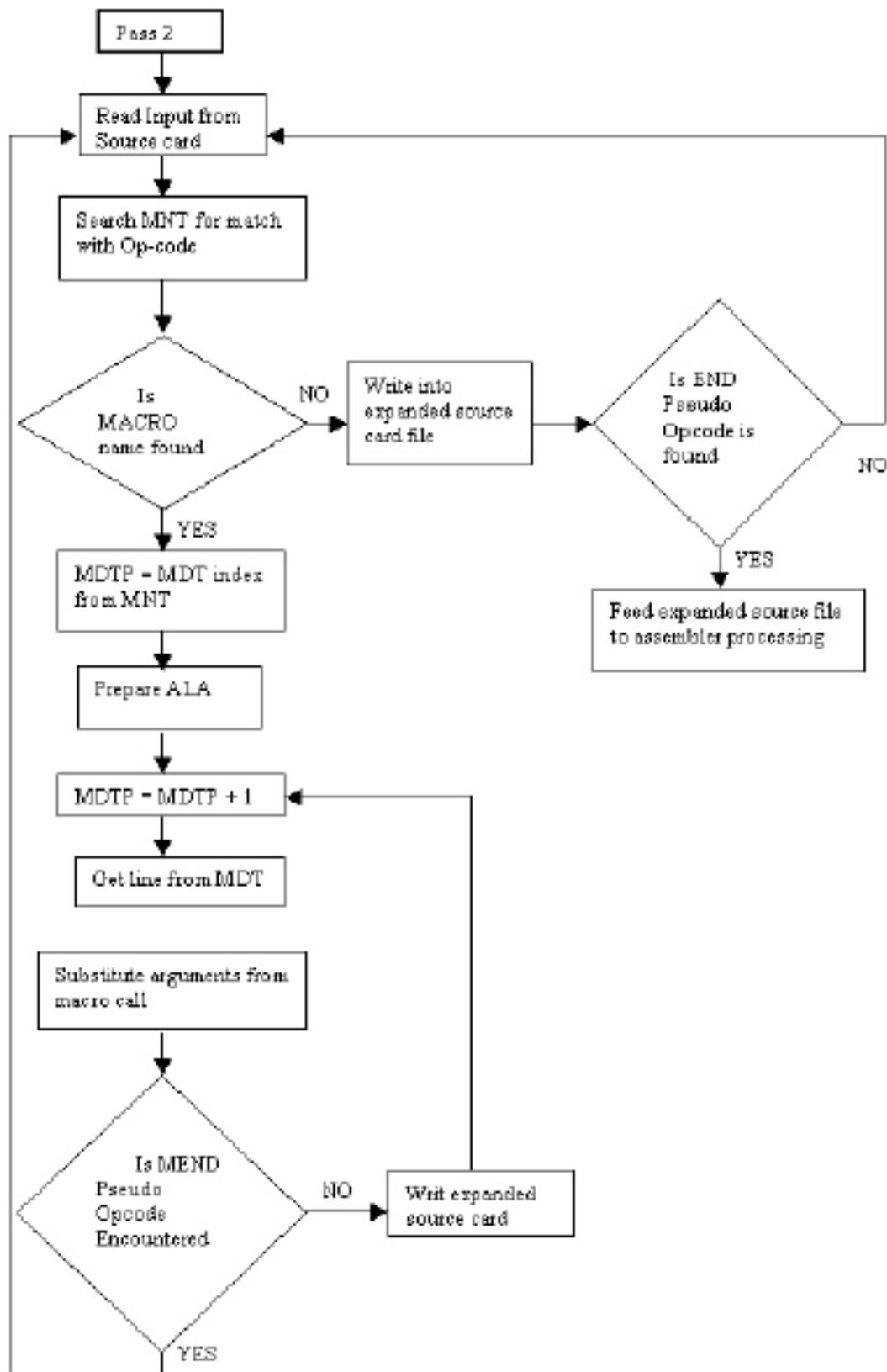


Figure 2.2: Flow Chart for Second Pass

### Implementation of Single-Pass Algorithm

The single-pass algorithm allows you to define macro within the macro but not supports macro calls within the macro. In the single-pass algorithm two additional Level Counter (MDLC). Following is the usage of MDI and MDLC in single-pass algorithm:

- **MDI indicator:** Allows you to keep track of macro calls and macro definitions. During expansion of macro call, MDI indicator has value ON and retains value OFF otherwise. If MDI indicator is on, then input data lines are read from MDT until mend pseudo opcode is not encountered. When MDI is off, then data input is read from data source instead of MDI.
- **MDLC indicator:** MDLC ensures you that macro definition is stored in MDT. MDLC is counters that keeps track of the numbers of macro1 and mend pseudo opcode found. Single-pass algorithm combines both the algorithms defined above to implement two-pass macro pre-processor. Following are the steps that are followed during single-pass algorithm:

1. Initialize MDTC and MNTC to value one and MDLC to zero.
2. Set MDI to value OFF.
3. Performs read operation.
4. Examine MNT to get the match with operation code.
5. If macro name is found then
  - A. MDI is set to ON.
  - B. Prepare argument list array containing a table of dummy arguments.
  - C. Performs read operation.
6. Else it examines that macro pseudo opcode is encountered. If macro pseudo opcode is found then
  - A. Enter the name of the macro and current value of MDTC in MNT at entry number MNTC.
  - B. Increment the MNTC to value one.
  - C. Prepare argument list array containing a table of dummy arguments..
  - D. Enter the macro card into MDT.
  - E. Increment the MDTC to value one.
  - F. Increment the MDLC to value one.
  - G. Performs read operation.
  - H. Substitute the index notations for the arguments list of the macro for dummy arguments.
  - I. Enter data input line into MDT.
  - J. Increment the MDTC to value one.
  - K. If macro pseudo opcode is found then increments the MDLC to value one and performs read operation.

- L. Else it checks for mend pseudo opcode if not found then performs read operation.
  - M. If mend pseudo opcode is found then decrement the MDLC to value one.
  - N. If MDLC is equal to zero then it goes to step 2. Otherwise, it performs read operation.
7. In case macro pseudo opcode is not found, then write it into expanded source card file.

If end pseudo opcode is found, then it feeds expanded source file to assembler for processing, otherwise performs read operation at step 2.

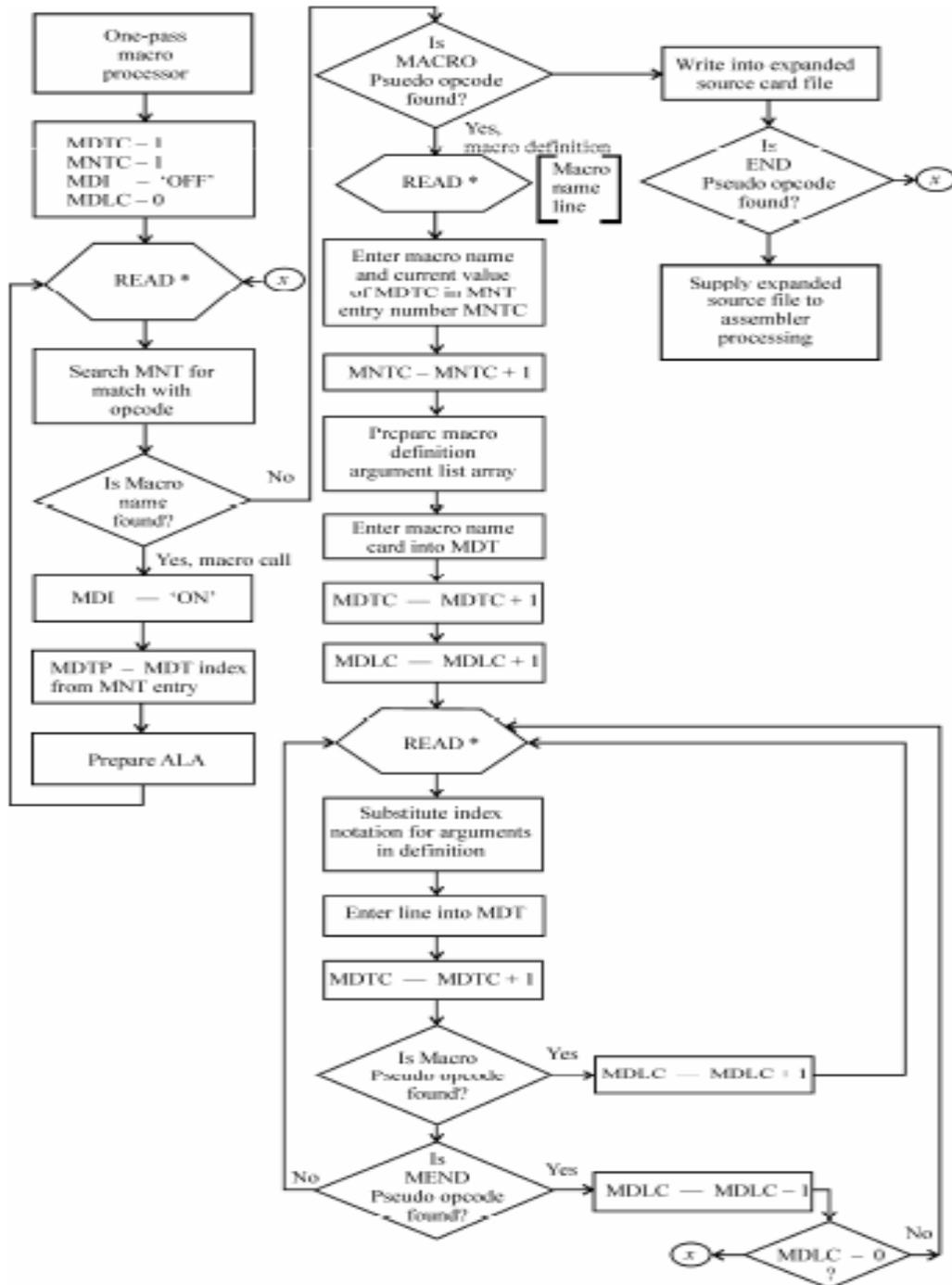


Figure 2.3: Single Pass

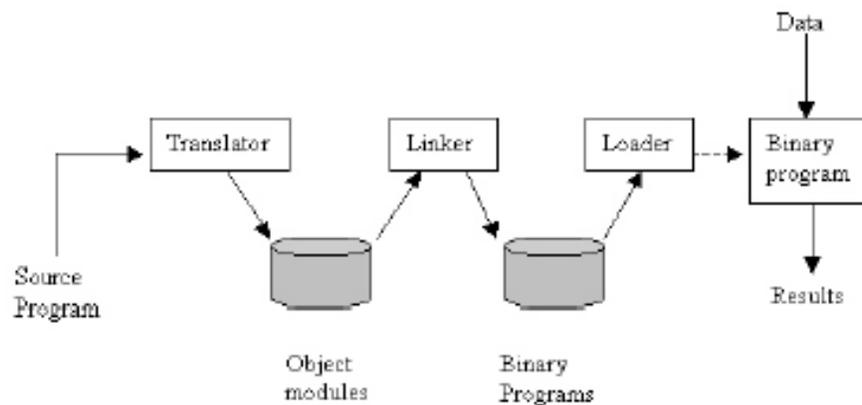
## Unit 3

### INTRODUCTION

Earlier programmers used loaders that could take program routines stored in tapes and combine and relocate them in one program. Later, these loaders evolved into linkage editor used for linking the program routines but the program memory remained expensive and computers were slow. A little progress in linking technology helped computers become faster and disks larger, thus program linking became easier. For the easier use of memory space and efficiency in speed, you need to use linkers and loaders.

Loaders and linker's helps you to have a schematic flow of steps that you need to follow while creating a program. Following are the steps that you need to perform when you write a program in language:

1. Translation of the program, which is performed by a processor called translator.
2. Linking of the program with other programs for execution, this is performed by a separate processor known as linker.
3. Relocation of the program to execute from the memory location allocated to it, which is performed by a processor called loader.
4. Loading of the program in the memory for its execution, this is performed by a loader.



*Figure 4.1: Schematic Flow of Program Execution*

### LOADERS

A loader is a program that performs the functions of a linker program and then immediately schedules the resulting executable program for some kind of action. In other words, a loader accepts the object program, prepares these programs for execution by the computer and then initiates the execution. It is not necessary for the loader to save a program as an executable file.

The functions performed by a loader are as follows:

1. **Memory Allocation** allocates space in memory for the program.

2. **Linking: Resolves** symbolic references between the different objects.
3. **Relocation** adjusts all the address dependent locations such as address constants, in order to correspond to the allocated space.
4. **Loading** places the instructions and data into memory.

Functions of Loader:

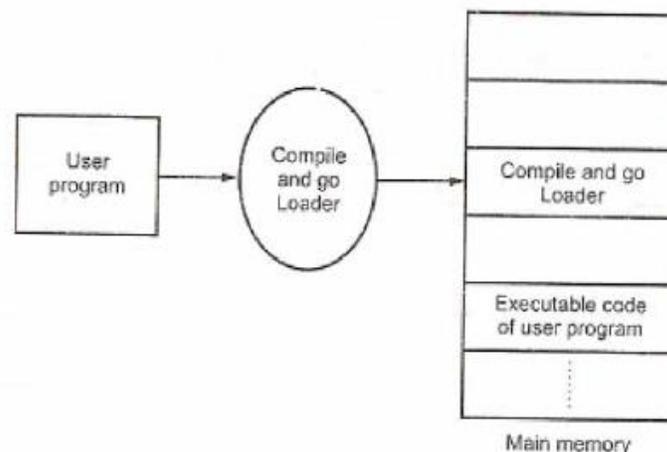
The loader is responsible for the activities such as allocation, linking, relocation and loading

1. It allocates the space for program in the memory, by calculating the size of the program. This activity is called allocation.
2. It resolves the symbolic references (code/data) between the object modules by assigning all the user subroutine and library subroutine addresses. This activity is called linking.
3. There are some address dependent locations in the program, such address constants must be adjusted according to allocated space, such activity done by loader is called relocation.
4. Finally it places all the machine instructions and data of corresponding programs and subroutines into the memory. Thus program now becomes ready for execution, this activity is called loading.

### 1. Compile and Go Loader

Compile and go loader is also known as “assembler-and-go”. It is required to introduce the term “segment” to understand the different loader schemes. A segment is a unit of information such as a program or data that is treated as an entity and corresponds to a single source or object deck. A figure shows the compile and go loader.

The compile and go loader executes the assembler program in one part of memory and places the assembled machine instructions and data directly into their assigned memory locations. Once the assembly is completed, the assembler transfers the control to the starting instruction of the program.



Compile and go loading scheme

Advantages

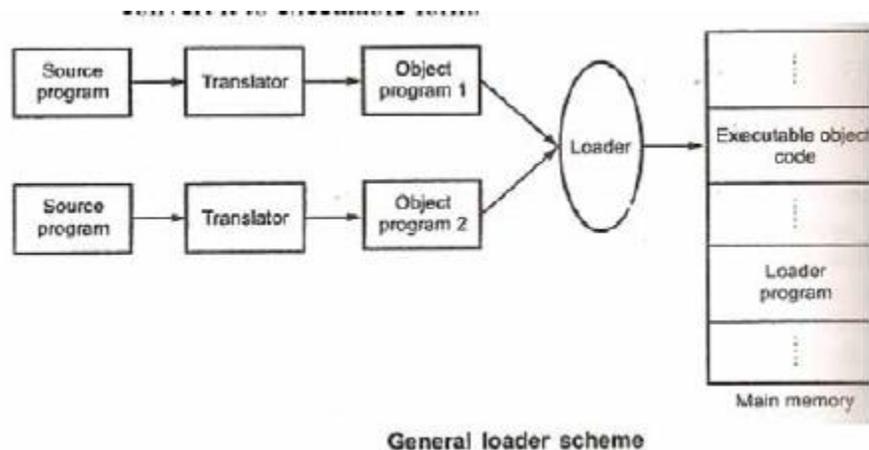
1. This scheme is easy to implement,
2. the assembler simply places the code into core and the loader, which consists of one instruction, transfers control to the starting instruction of the newly assembled program.

#### Disadvantages

1. In this scheme, a portion of memory is wasted. This is mainly because the core occupied by the assembler is not available to the object program.
2. It is essential to assemble the user's program deck every time it is executed.
3. It is quite difficult to handle multiple segments, if the source programs are in different languages. This disadvantage makes it difficult to produce orderly modular programs.

## 2. General Loader Scheme.

The concept of loaders can be well understood if one knows the general loader scheme. It is recommended for the general loader scheme that the instructions and data should be produced in the output, as they were assembled. This strategy, if followed, prevents the problem of wasting core for an assembler. When the code is required to be executed, the output is saved and loaded in the memory. The assembled program is loaded into the same area in core that it occupied earlier. The output that contains a coded form of the instructions is called the object deck. The object deck is used as intermediate data to avoid the circumstances in which the addition of a new program to a system is required. The loader accepts the assembled machine instructions, data and other information present in the object. The loader places the machine instructions and data in core in an executable computer form. More memory can be made available to a user, since in this scheme, the loader is assumed to be smaller than the assembler. Figure shows the general loader scheme.

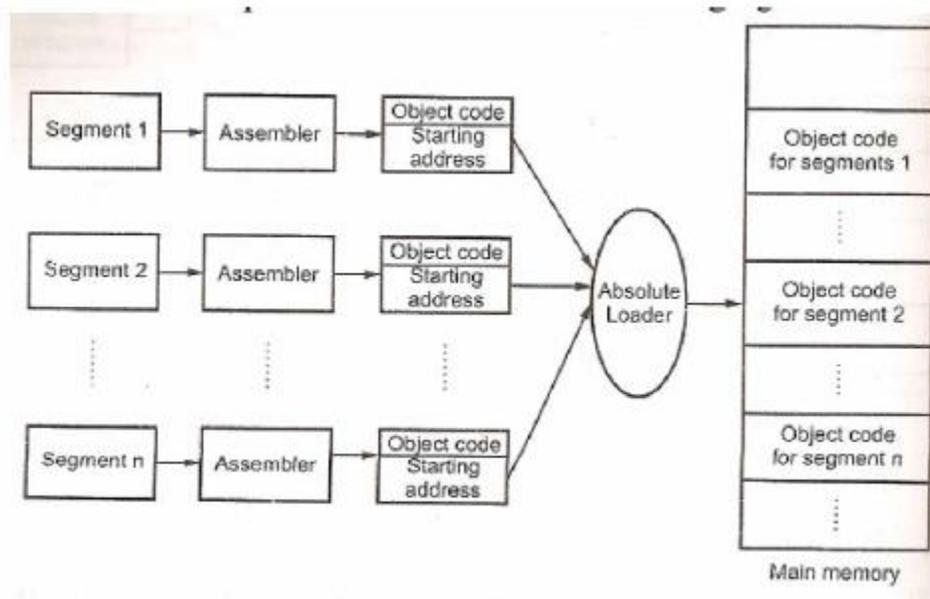


#### Advantages:

- The program need not be retranslated each time while running it. This is because initially when source program gets executed an object program gets generated. Of program is not modified, and then loader can make use of this object program to convert it to executable form.
- There is no wastage of memory, because assembler is not placed in the memory, instead of it, loader occupies some portion of the memory. And size of loader is smaller than assembler, so more memory is available to the user.
- It is possible to write source program with multiple programs and multiple languages, because the source programs are first converted to object programs always, and loader accepts these object modules to convert it to executable form.

### 3. Absolute Loader

An absolute loader is the simplest type of loader scheme that fits the general model of loaders. The assembler produces the output in the same way as in the “compile and go loader”. The assembler outputs the machine language translation of the source program. The difference lies in the form of data, i.e., the data in the absolute loader is punched on cards or you can say that it uses object deck as an intermediate data. The loader in turn simply accepts the machine language text and places it at the location prescribed by the assembler. When the text is being placed into the core, it can be noticed that much core is still available to the user. This is because, within this scheme, the assembler is not in the memory at the load time.



Process of absolute loading

In the figure, the MAIN program is assigned to locations 1000-2470 and the SQRT subroutine is assigned locations 4000-4770. This means the length of MAIN has increased to more than 3000 bytes, as it can be noticed from figure 4.4. If the modifications are required to be made in MAIN subroutine, then the end of MAIN subroutine, i.e.,  $1000+3000=4000$ , gets overlapped with the start of SQRT, i.e., with 4000. Therefore, it is necessary to assign a new location to SQRT. This can be made possible by changing the START pseudo-op card and

reassembling it. It is then quite obvious to modify all other subroutines that refer to address of SQRT.

#### Advantages

1. Absolute loaders are simple to implement.
2. This scheme allows multiple programs or the source programs written different languages. If there are multiple programs written in different languages then the respective language assembler will convert it to the language and a common object file can be prepared with all the address resolution.
3. The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code in the main memory.
4. The process of execution is efficient.

#### Disadvantages.

1. It is desirable for the programmer to specify the address in core where the program is to be loaded.
2. A programmer needs to remember the address of each subroutine, if there are multiple subroutines in the program.
3. Additionally, each absolute address is to be used by the programmer explicitly in the other subroutines such that subroutine linkage can be maintained.

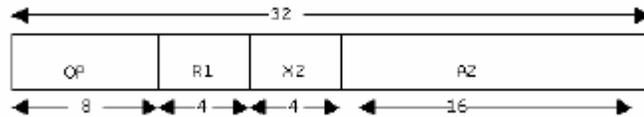
## **4. Relocating Loaders**

Relocating loaders were introduced in order to avoid possible reassembling of all subroutines when a single subroutine is changed. It also allows you to perform the tasks of allocation and linking for the programmer. The example of relocating loaders includes the Binary Symbolic Subroutine (BSS) loader. Although the BSS loader allows only one common data segment, it allows several procedure segments. The assembler in this type of loader assembles each procedure segment independently and passes the text and information to relocation and intersegment references.

In this scheme, the assembler produces an output in the form of text for each source program. A transfer vector that contains addresses, which includes names of the subroutines referenced by the source program, prefixes the output text. The assembler would also provide the loader with additional information such as the length of the entire program and also the length of the transfer vector portion. Once this information is provided, the text and the transfer vector get loaded into the core. Followed by this, the loader would load each subroutine, which is being identified in the transfer vector. A transfer instruction would then be placed to the corresponding subroutine for each entry in the transfer vector.

The output of the relocating assembler is the object program and information about all the programs to which it references. Additionally, it also provides relocation information for the locations that need to be changed if it is to be loaded in the core. This location may be arbitrary in

the core, let us say the locations, which are dependent on the core allocation. The BSS loader scheme is mostly used in computers with a fixed-length direct-address instruction format. Consider an example in which the 360 RX instruction format is as follows:



In this format, A2 is the 16-bit absolute address of the operand, which is the direct address instruction format. It is desirable to relocate the address portion of every instruction. As a result, the computers with a direct-address instruction format have much severe problems than the computers having 360-type base registers. The 360-type base registers solve the problem using relocation bits. The relocation bits are included in the object deck and the assembler associates a bit with each instruction or address field. The corresponding address field to each instruction must be relocated if the associated bit is equal to one; otherwise this field is not relocated.

## 5. Direct-Linking Loaders

A direct-linking loader is a general relocating loader and is the most popular loading scheme presently used. This scheme has an advantage that it allows the programmer to use multiple procedure and multiple data segments. In addition, the programmer is free to reference data or instructions that are contained in other segments. The direct linking loaders provide flexible intersegment referencing and accessing ability. An assembler provides the following information to the loader along with each procedure or data segment.

This information includes:

- Length of segment.
- List of all the symbols and their relative location in the segment that are referred by other segments.
- Information regarding the address constant which includes location in segment and description about the revising their values.
- Machine code translation of the source program and the relative addresses assigned.

## LINKAGE EDITOR

Supply information needed to allow references between them. A linkage editor is also known as linker. To allow linking in a program, you need to perform:

- Program relocation
- Program linking

### 1. Program relocation

Program relocation is the process of modifying the addresses containing instructions of a program. You need to use program relocation to allocate a new memory address to the instruction. Instructions are fetched from the memory address and are followed sequentially to execute a program. The relocation of a program is performed by a linker and for performing relocation you need to calculate the relocation\_factor that helps specify the translation time address in every instruction. Let the translated and linked origins of a program P be  $t\_origin$  and  $l$

The algorithm that you use for program relocation is:

1. `program_linked_origin := <link origin> from linker command;`
2. For each object module
3. `t_origin := translated origin of the object module;`  
`OM_size := size of the object module;`
4. `relocation_factor := program_linked_origin - t_origin;`
5. Read the machine language program in work\_area;
6. Read RELOCTAB of the object module
7. For each entry in RELOCTAB
  - A. `translated_addr := address in the RELOCTAB entry;`
  - B. `address_in_work := address of work_area + translated_address - t_origin;`
  - C. add relocation\_factor to the operand in the wrd with the address address\_in\_work\_area.
  - D. `program_linked_origin := program_linked_origin + OM_size;`

## 2. Program Linking

Linking in a program is a process of binding an external reference to a correct link address. You need to perform linking to resolve external reference that helps in the execution of a program. All the external references in a program are maintained in a table called name table (NTAB), which contains the symbolic name for external references or an object module. The information specified in NTAB is derived from

LINKTAB entries having type=PD. The algorithm that you use for program linking is:

### Algorithm (Program Linking)

1. `program_linked_origin` := <link origin> from linker command.
2. for each object module
  - A. `t_origin` := translated origin of the object module;  
`OM_size` := size of the object module;
  - B. `relocation_factor` := `program_linked_origin` - `t_origin`;
  - C. read the machine language program in `work_area`.
  - D. Read LINKTAB of the object module.
  - E. For each LINKTAB entry with type=PD  
`name` := symbol;  
`linked_address` := translated address + `relocation_factor`;  
Enter (`name`, `linked_address`) in NTAB.
  - F. Enter (`object module name`, `program_linked_origin`) in NTAB.
  - G. `Program_linked_origin` := `program_linked_origin` + `OM_size`;
3. for each object module
  - A. `t_origin` := translated origin of the object module;  
`program_linked_origin` := load\_address from NTAB;
  - B. for each LINKTAB entry with type=EXT
    - `address_in_work_area` := address of `work_area` + `program_linked_origin` - <link origin> + translated address - `t_origin`
    - search symbol in NTAB and copy its linked address. Add the linked address to the operand address in the word with the address `address_in_work_area`.

### DYNAMIC LINKING

Sophisticated operating systems, such as Windows allow you to link executable object modules to be linked to a program while a program is running. This is known as dynamic linking. The operating system contains a linker that determines functions, which are not specified in a program. A linker searches through the specified libraries for the missing function and helps extract the object modules containing the missing functions from the libraries. The libraries are constructed in a way to be able to work with dynamic linkers. Such libraries are known as dynamic link libraries (DLLs). Technically, dynamic linking is not like static linking, which is done at build time. DLLs contain functions or routines, which are loaded and executed when needed by a program. The advantages of DLLs are:

- **Code sharing:** Programs in dynamic linking can share an identical code instead of creating an individual copy of a same library. Sharing allows executable functions and routines to be shared by many application programs. For example, the object linking and embedding (OLE) functions of OLE2.DLL can be invoked to allow the execution of functions or routines in any program.

- **Automatic updating:** Whenever you install a new version of dynamic link library, the older version is automatically overridden. When you run a program the updated version of the dynamic link library is automatically picked.
- **Securing:** Splitting the program you create into several linkage units makes it harder for crackers to read an executable file.

## Unit 4

### Unit -4

#### Common Object File Format (COFF)

This chapter describes the Common Object File Format (COFF). COFF is the format of the output file produced by the assembler and the link editor.

The following are some key features of COFF:

- applications can add system-dependent information to the object file without causing access utilities to become obsolete
- space is provided for symbolic information used by debuggers and other applications
- programmers can modify the way the object file is constructed by providing directives at compile time

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file contains:

# Object file format

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information
--------------------	--------------	--------------	------------------------	--------------	-----------------------

- Object file header describes the size and position of the other pieces of the file
- Text segment contains the machine instructions
- Data segment contains binary representation of data in assembly file
- Relocation info identifies instructions and data that depend on absolute addresses
- Symbol table associates addresses with external labels and lists unresolved references
- Debugging info

## Object file format

FILE HEADER
Optional Information
Section 1 Header
...
Section <i>n</i> Header
Raw Data for Section 1
...
Raw Data for Section <i>n</i>
Relocation Info for Sect. 1
...
Relocation Info for Sect. <i>n</i>
Line Numbers for Sect. 1
...
Line Numbers for Sect. <i>n</i>
SYMBOL TABLE
STRING TABLE

The last four sections (relocation, line numbers, symbol table, and the string table) may be missing if the program is linked with the **-s** option of the **ld** command, or if the line number information, symbol table, and string table are removed by the command. The line number information does not appear unless the program is compiled with `-g`. Also, if there are no unresolved external references after linking, the relocation information is no longer needed and is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters.

### File header

The file header contains the 20 bytes of information shown in. The last 2 bytes are flags that are used by **ld** and object file utilities.

### File header contents

Bytes	Declaration	Name	Description
0-1	unsigned short	f_magic	<a href="#">Magic number</a>
2-3	unsigned short	f_nscns	Number of sections
4-7	long int	f_timdat	Time and date stamp indicating when the file was created, expressed as the number of elapsed seconds since 00:00:00 GMT, January 1, 1970
8-11	long int	f_symptr	File pointer containing the starting address of the symbol table

12-15	long int	f_nsyms	Number of entries in the symbol table
16-17	unsigned short	f_opthdr	Number of bytes in the optional header
18-19	unsigned short	f_flags	Flags (see <a href="#">"File header flags"</a> )

### Magic numbers

The magic number specifies the target machine on which the object file is executable.

### Flags

The last 2 bytes of the file header are flags that describe the type of the object file. Currently defined flags are found in the header file *filehdr.h* and are shown in ["File header flags"](#).

### File header flags

Mnemonic	Flag	Meaning
F_RELFLG	00001	Relocation information stripped from the file
F_EXEC	00002	File is executable (i.e., no unresolved external references)
F_LNNO	00004	Line numbers stripped from the file
F_LSYMS	00010	Local symbols stripped from the file
F_AR16WR	0000200	16-bit byte reversed word
F_AR32WR	0000400	32-bit byte reversed word

## File header declaration

The C structure declaration for the file header is shown below. This declaration may be found in the header file *filehdr.h*.

## Section headers

Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in.

## Section header contents

Bytes	Declaration	Name	Description
0-7	char	s_name	8-character null padded section name
8-11	long int	s_paddr	Physical address of section
12-15	long int	s_vaddr	Virtual address of section
16-19	long int	s_size	Section size in bytes
20-23	long int	s_scnptr	File pointer to raw data
24-27	long int	s_relptr	File pointer to relocation entries
28-31	long int	s_lnnoptr	File pointer to line number entries
32-33	unsigned short	s_nreloc	Number of relocation entries
34-35	unsigned short	s_nlnno	Number of line number entries
36-39	long int	s_flags	Flags (see <a href="#">"Section header flags"</a> )

The size of a section is padded to a multiple of 4 bytes. File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. They can be readily used with the function [fseek\(S\)](#).

## Line numbers

When invoked with [cc-g](#) the compiler causes an entry in the object file for every source line where a breakpoint can be inserted. You can then reference line numbers when using a software debugger like [sdb\(CP\)](#). All line numbers in a section are grouped by function as shown in ["Line number grouping"](#).

### Line number grouping

symbol index	0
physical address	line number
physical address	line number
.	.
.	.
.	.
symbol index	0
physical address	line number
physical address	line number

The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries are relative to the beginning of the function and appear in increasing order of address.

### Line number declaration

The structure declaration currently used for line number entries is shown below.

```

struct lineno
{
    union
    {
        long l_symndx; /* symtbl index of func name */
        long l_paddr; /* paddr of line number */
    } l_addr;
    unsigned short l_lno; /* line number */
};

```

## Symbol table

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the sequence shown in ["COFF symbol table"](#).

### COFF symbol table

filename 1
function 1
C_WEAKEXT aliases
for function 1
function 1b (alias)
...
local symbols for function 1
function 2
C_WEAKEXT aliases
for function 2

...
local symbols for function 2
...
statics
...
filename 2
function 1
C_WEAKEXT aliases
for function 1
...
local symbols for function 1
...
statics
...
defined global symbols
undefined global symbols

The word "statics" in "[COFF symbol table](#)" means symbols defined with the C language storage class static outside any function. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

### String table

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; offsets into the string table, therefore, are greater than or equal to 4. For example, given a file containing two symbols (with names longer than eight characters, *long\_name\_1* and *another\_one*) the string table has the format as shown in "[String table](#)":

## String table

`l`	`o`	`n`	`g`
`_`	`n`	`a`	`m`
`e`	`_`	`l`	`\0`
`a`	`n`	`o`	`t`
`h`	`e`	`r`	`_`
`o`	`n`	`e`	`\0`

The index of *long\_name\_1* in the string table is 4 and the index of *another\_one* is 16.

## Debugger

A **debugger** or **debugging tool** is a computer program that is used to test and debug other programs (the "target" program). The code to be examined might alternatively be running on an *instruction set simulator* (ISS), a technique that allows great power in its ability to halt when specific conditions are encountered but which will typically be somewhat slower than executing the code directly on the appropriate (or the same) processor. Some debuggers offer two modes of operation—full or partial simulation—to limit this impact.

A "trap" occurs when the program cannot normally continue because of a programming bug or invalid data. For example, the program might have tried to use an instruction not available on the current version of the CPU or attempted to access unavailable or protected memory. When the program "traps" or reaches a preset condition, the debugger typically shows the location in the original code if it is a **source-level debugger** or **symbolic debugger**, commonly now seen in integrated development environments. If it is a **low-level debugger** or a **machine-language debugger** it shows the line in the disassembly

A debugger is a program that runs other programs, allowing the user to exercise control over these programs, and to examine variables when problems arise.

GNU Debugger which is called **gdb** is the most popular debugger for UNIX systems to debug C and C++ Programs.

GNU Debugger helps you in finding out followings:

- If a core dump happened then what statement or expression did the program crash on?
- If an error occurs while executing a function, what line of the program contains the call to that function, and what are the parameters?
- What are the values of program variables at a particular point during execution of the program?
- What is the result of a particular expression in a program?

### How GDB Debugs?

GDB allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.

GDB uses a simple command line interface.

### Note the Followings:

- Even though GDB can help you in finding out memory leakage related bugs but it is not a tool to detect memory leakages
- GDB cannot be used for programs that do not compile without errors and it does not help in fixing those errors.

## Unit 5

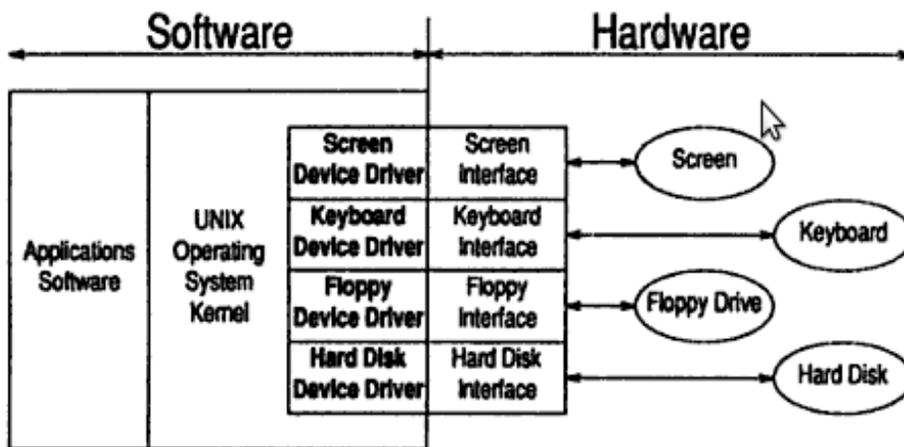
### Unit -5

**Device driver** (commonly referred to as a *driver*) is a computer program that operates or controls a particular type of device that is attached to a computer. A driver provides a software interface to hardware devices, enabling operating systems and other computer

programs to access hardware functions without needing to know precise details of the hardware being used.

A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

**FIGURE 1-1**



### Types of Device Drivers

A device driver is a software module that resides within the Digital UNIX kernel and is the software interface to a hardware device or devices. A hardware device is a peripheral, such as a disk controller, tape controller, or network controller device. In general, there is one device driver for each type of hardware device. Device drivers can be classified as:

- Block device drivers
- Character device drivers (including terminal drivers)
- Network device drivers
- Pseudodevice drivers

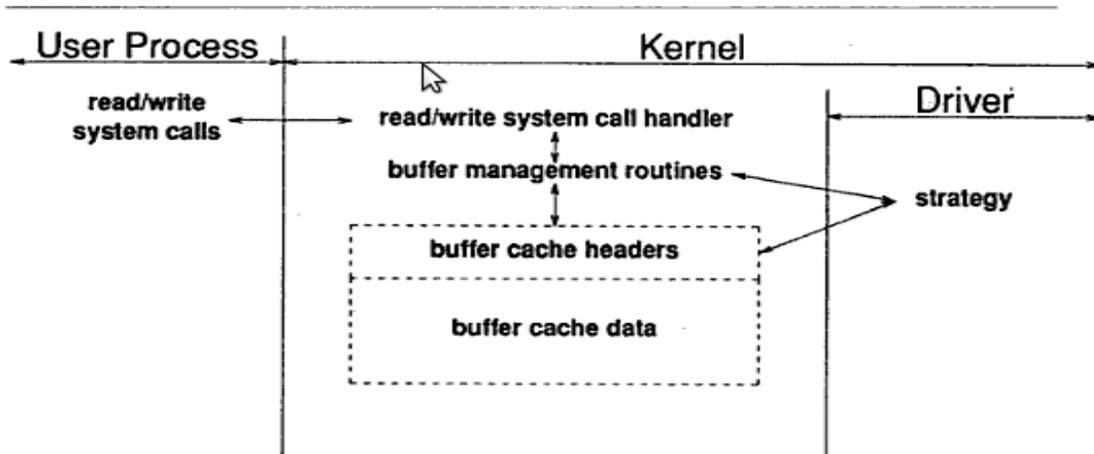
The following sections briefly discuss each type.

## Block Device Driver

A block device driver is a driver that performs I/O by using file system block-sized buffers from a buffer cache supplied by the kernel. The kernel also provides for the device driver support interfaces that copy data between the buffer cache and the address space of a process.

Block device drivers are particularly well-suited for disk drives, the most common block devices. For block devices, all I/O occurs through the buffer cache.

*FIGURE 1-2*



## Character Device Driver

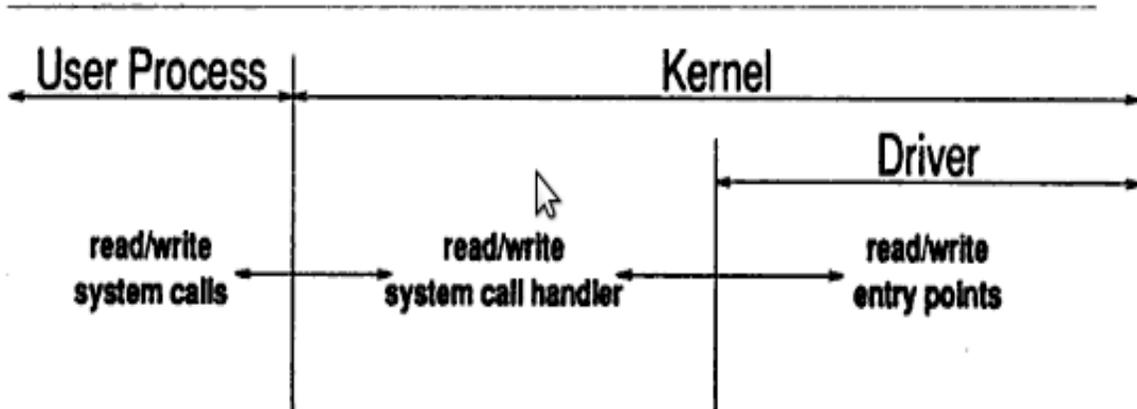
A character device driver does not handle I/O through the buffer cache, so it is not tied to a single approach for handling I/O. You can use a character device driver for a device such as a line printer that handles one character at a time. However, character drivers are not limited to performing I/O one character at a time (despite the name "character" driver). For example, tape drivers frequently perform I/O in 10K chunks. You can also use a character device driver when it is necessary to copy data directly to or from a user process.

Because of their flexibility in handling I/O, many drivers are character drivers. Line printers, interactive terminals, and graphics displays are examples of devices that require character device drivers.

A terminal device driver is actually a character device driver that handles I/O character processing for a variety of terminal devices. Like any character device, a terminal device

can accept or supply a stream of data based on a request from a user process. It cannot be mounted as a file system and, therefore, does not use data caching.

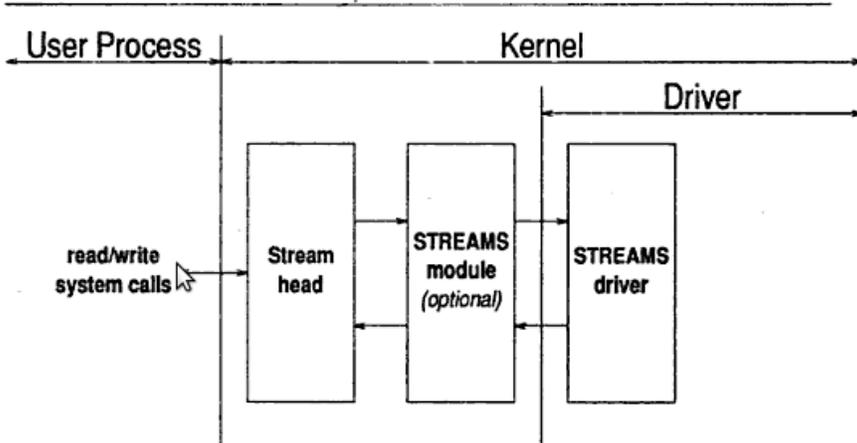
**FIGURE 1-3**



### Network Device Driver

A network device driver attaches a network subsystem to a network interface, prepares the network interface for operation, and governs the transmission and reception of network frames over the network interface.

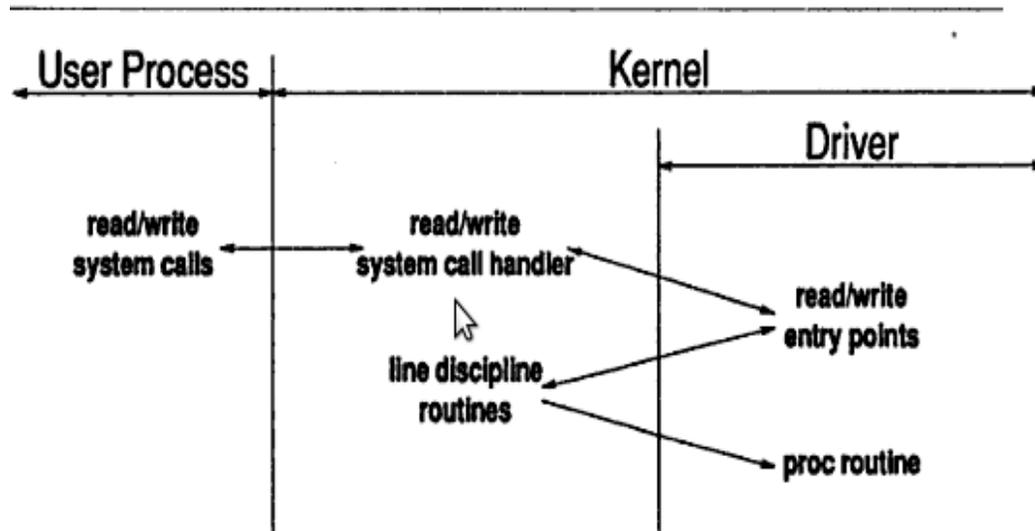
**FIGURE 1-5**



### Pseudo device Driver

Not all device drivers control physical hardware. Such device drivers are called "pseudodevice" drivers. Like block and character device drivers, pseudodevice drivers make use of the device driver interfaces. Unlike block and character device drivers, pseudodevice drivers do not operate on a bus. One example of a pseudodevice driver is the pseudo terminal or `pty` terminal driver, which simulates a terminal device. The `pty` terminal driver is a character device driver typically used for remote logins.

**FIGURE 1-4**

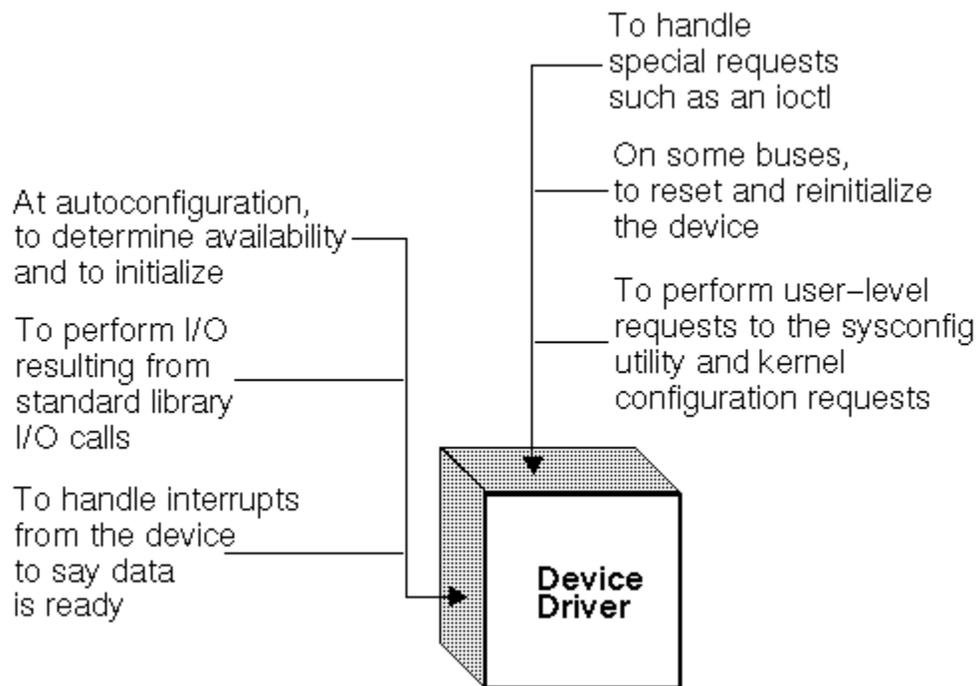


### When a Device Driver Is Called

- Auto configuration  
The kernel calls a device driver (specifically, the driver's `probe` interface) at auto configuration time to determine what devices are available and to initialize them.
- I/O operations  
The kernel calls a device driver to perform I/O operations on the device. These operations include opening the device to perform reads and writes and closing the device.
- Interrupt handling  
The kernel calls a device driver to handle interrupts from devices capable of generating them.
- Special requests  
The kernel calls a device driver to handle special requests through `ioctl` calls.

- Reinitialization  
The kernel calls a device driver to reinitialize the driver, the device, or both when the bus (the path from the CPU to the device) is reset.
- User-level requests to the `sysconfig` utility  
The kernel calls a device driver (specifically, the driver's `configure` interface) to handle requests that result from use of the `sysconfig` utility. The `sysconfig` utility allows a system manager to dynamically configure, unconfigure, query, and reconfigure a device. These requests cause the kernel to call the device driver's `configure` interface. In addition, the driver's `configure` interface performs one-time initializations when called by the boot software or by the `sysconfig` utility.

**Figure 1-1: When the Kernel Calls a Device Driver**



Some of these requests, such as input or output, result directly or indirectly from corresponding system calls in a user program. Other requests, such as the calls at auto configuration time, do not result from system calls but from activities that occur at boot time.

## Device Driver Configuration

Device driver configuration consists of the tasks necessary to incorporate device drivers into the kernel to make them available to system management and other utilities. After you write your device driver you need to create a single binary module (a file with a `.mod` extension) from the driver source file (a file with a `.c` extension). After you create the single binary module, you need to configure it into the kernel so that you can test it on a running system. There are two methods of device driver configuration: static configuration and dynamic configuration. Static configuration consists of the tasks and tools necessary to link a device driver (single binary module) directly into the kernel at kernel build time. Dynamic configuration consists of the tasks and tools necessary to link a device driver (single binary module) directly into the kernel at any point in time. This section describes how to create a single binary module and then how to statically and dynamically configure the single binary module (the device driver) into the kernel.

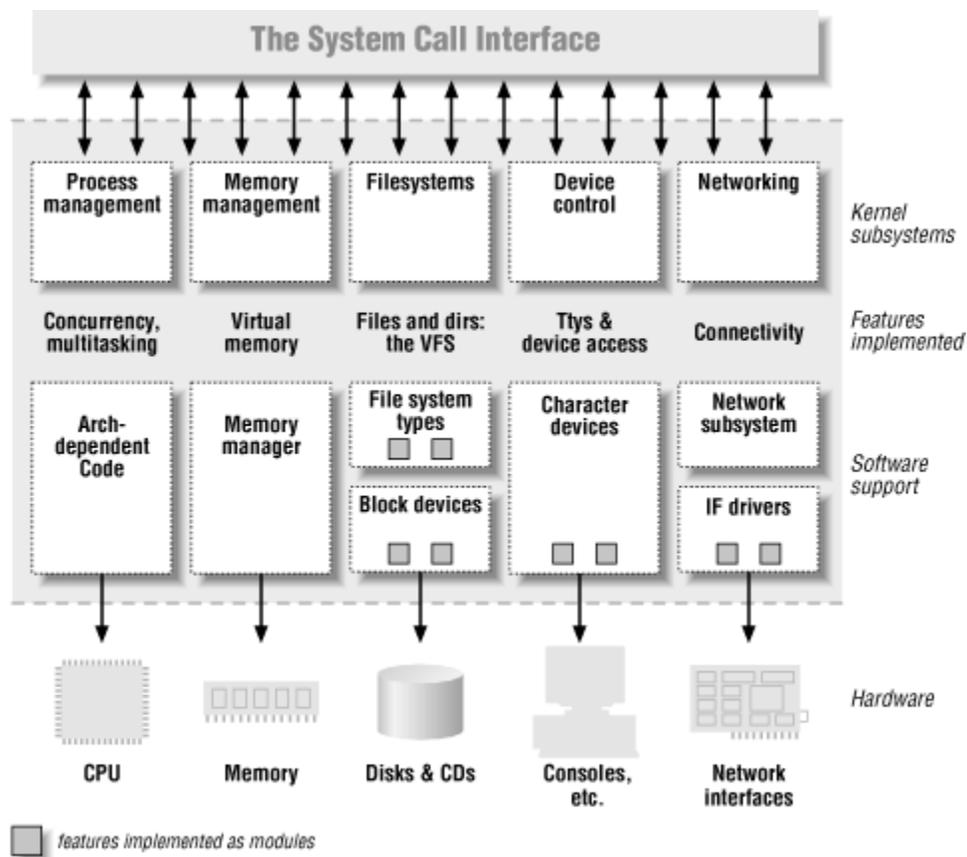
Do not confuse device driver configuration (static configuration and dynamic configuration), which encompasses the tools and steps for configuring the driver into the kernel, with auto configuration and configuration. Auto configuration is a process that determines what hardware actually exists during the current instance of the running kernel at static configuration time. The auto configuration software (specifically, the bus's `conf11` interface) calls the driver's `probe`, `attach`, and `slave` interfaces. Thus, the driver's `probe`, `attach`, and `slave` interfaces cooperate with the bus's `conf11` interface to determine if devices exist and are functional on a given system.

Configuration is a process associated with handling user-level requests to the `sysconfig` utility to dynamically configure, unconfigure, query, and reconfigure devices. The `cfgmgr` framework calls the driver's `configure` interface as a result of these `sysconfig` utility requests. The `cfgmgr` framework also calls the driver's `configure` interface as a result of static configuration requests. Thus, the driver's `configure` interface cooperates with the `cfgmgr` framework to statically configure and to dynamically configure, unconfigure, query, and reconfigure devices. The driver's `configure` interface also cooperates with the `cfgmgr` framework to perform one-time initialization tasks such as allocating memory, initializing data structures and variables, and adding driver entry

points to the `sent` table. A driver's `configure` interface should be implemented to handle static and dynamic configuration.

### The Role of the Device Driver

In a Unix system, several concurrent *processes* attend to different tasks. Each process asks for system resources, be it computing power, memory, network connectivity, or some other resource. The *kernel* is the big chunk of executable code in charge of handling all such requests. Though the distinction between the different kernel tasks isn't always clearly marked, the kernel's role can be split, as shown in Figure 1-1, into the following parts:



### Process management

The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output). Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel. In addition, the scheduler, which controls how processes share the CPU, is

part of process management. More generally, the kernel's process management activity implements the abstraction of several processes on top of a single CPU or a few of them.

### ***Memory management***

The computer's memory is a major resource, and the policy used to deal with it is a critical one for system performance. The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple *malloc free* pair to much more exotic functionalities.

### ***File systems***

Unix is heavily based on the file system concept; almost everything in Unix can be treated as a file. The kernel builds a structured file system on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. In addition, Linux supports multiple file system types, that is, different ways of organizing data on the physical medium. For example, diskettes may be formatted with either the Linux-standard ext2 file system or with the commonly used FAT file system.

### ***Device control***

Almost every system operation eventually maps to a physical device. With the exception of the processor, memory, and a very few other entities, any and all device control operations are performed by code that is specific to the device being addressed. That code is called a *device driver*. The kernel must have embedded in it a device driver for every peripheral present on a system, from the hard drive to the keyboard and the tape streamer. This aspect of the kernel's functions is our primary interest in this book.

### ***Networking***

Networking must be managed by the operating system because most network operations are not specific to a process: incoming packets are asynchronous events. The packets must be collected, identified, and dispatched before a process takes care of them. The system is in charge of delivering data packets across program and network interfaces, and it must control the execution of programs according to their network activity. Additionally, all the routing and address resolution issues are implemented within the kernel.

## **Comparison of Windows and UNIX Environments**

This section compares the Windows and UNIX architectures, emphasizing the areas that directly affect software development in a migration project.

### *Kernels and APIs*

As with most operating systems, Windows and UNIX both have kernels. The kernel provides the base functionality of the operating system. The major functionality of the kernel includes process management, memory management, thread management, scheduling, I/O management, and power management.

In UNIX, the API functions are called system calls. System calls are a programming interface common to all implementations of UNIX. The kernel is a set of functions that are used by processes through system calls.

Windows has an API for programming calls to the executive. In addition to this, each subsystem provides a higher-level API. This approach allows Windows operating systems to provide different APIs, some of which mimic the APIs provided by the kernels of other operating systems. The standard subsystem APIs include the Windows API (the Windows native API) and the POSIX API (the standards-based UNIX API).

### *Windows Subsystems*

A subsystem is a portion of the Windows operating system that provides a service to application programs through a callable API.

Subsystems come in two varieties, depending upon the application program that finally handles the request:

- **Environment subsystems.** These subsystems run in a user mode and provide functions through a published API. The Windows API subsystem provides an API for operating system services, GUI capabilities, and functions to control all user input and output. The Win32 subsystem and POSIX subsystem are part of the environment subsystems and are described as follows:

- **Win32 subsystem.** The Win32 environment subsystem allows applications to benefit from the complete power of the Windows family of operating systems. The Win32 subsystem has a vast collection of functions, including those required for advanced operating systems, such as security, synchronization, virtual memory management, and threads. You can use Windows APIs to write 32-bit and 64-bit applications that run on all versions of Windows.
- **POSIX subsystem and Windows Services for UNIX.** To provide more comprehensive support for UNIX programs, Windows uses the Interix subsystem. Interix is a multiuser UNIX environment for a Windows-based computer. Interix conforms to the POSIX.1 and POSIX.2 standards. It provides all features of a traditional UNIX operating system, including pipes, hard links, symbolic links, UNIX networking, and UNIX graphical support through the X Windows system. It also includes case-sensitive file names, job control tools, compilation tools, and more than 300 UNIX commands and tools, such as KornShell, C Shell, **awk**, and **vi**.

Because it is layered on top of the Windows kernel, the Interix subsystem is not an emulation. Instead, it is a native environment subsystem that integrates with the Windows kernel, just as the Win32 subsystem does. Shell scripts and other scripted applications that use UNIX and POSIX.2 tools run under Interix.

- **Integral subsystems.** These subsystems perform key operating system functions and run as a part of the executive or kernel. Examples are the user-mode

subsystems, Local Security Authority subsystem (LSASS), and Remote Procedure Call subsystem (RPCSS).

### *Kernel Objects and Handles*

Kernel objects are used to manage and manipulate resources—such as files, synchronization objects, and pipes—across the processes. These kernel objects are owned by the kernel and not by the process. Handles are the opaque numbers or the data type used to represent the objects and to uniquely identify the objects. To interact with an object, you must obtain a handle to the object.

In UNIX, the kernel object can be created using the system calls and it returns an unsigned integer. There is no exact equivalent of handles in UNIX.

In Windows, Windows APIs are used to create the kernel object and it returns a Windows-specific data type called HANDLE.

### *Hardware Drivers*

Hardware drivers are system software used to interact the hardware devices with the operating system.

In UNIX, there are several different ways to manage drivers. Some UNIX implementations allow dynamic loading and unloading of drivers, whereas other implementations do not. The UNIX vendor usually provides drivers. The range of Intel hardware supported for UNIX is typically smaller than that for Windows.

In Windows, the Windows driver model provides a platform for developing drivers for industry-standard hardware devices attached to a Windows-based system. The key to developing a good driver package is to provide reliable setup and installation procedures and interactive GUI tools for configuring devices after the installation. In addition, the hardware must be compatible with Windows Plug and Play technology in order to provide a user-friendly hardware installation.

## *Process Management*

A process is usually defined as the instance of the running program. Process management describes how the operating systems manage the multiple processes running at a particular instance of time. Multitasking operating systems such as Windows and UNIX must manage and control many processes simultaneously. Both Windows and UNIX operating systems support processes and threads.

The following sections provide more details on process management in both UNIX and Windows:

- **Multitasking.** UNIX is a multiprocessing, multiuser system. At any given point, you can have many processes running on UNIX. Consequently, UNIX is very efficient at creating processes.

Windows has evolved greatly from its predecessors, such as Digital Equipment Corporation's VAX/VMS. It is now a preemptive multitasking operating system. As a result, Windows relies more heavily on threads than processes. (A thread is a construct that enables parallel processing within a single process.) Creating a new process is a relatively expensive operation while creating a new thread is not as expensive in terms of system resources like memory and time. Hence, multiprocess-oriented applications on UNIX typically translate to multithreaded applications on the Windows platform, thus saving such system resources as memory and time.

- **Multiple users.** One key difference between UNIX and Windows is in the creation of multiple user accounts on a single computer.

When you log on to a computer running UNIX, a shell process is started to service your commands. The UNIX operating system keeps track of the users and their processes and prevents processes from interfering with one another. The operating system does not come with any default interface for user interaction. However, the shell process on the computer running UNIX can connect to other computers to load third-party UIs.

When a user logs on interactively to a computer running Windows, the Win32 subsystem's Graphical Identification and Authentication dynamic-link library (GINA) creates the initial process for that user, which is known as the user desktop, where all user interaction or activity takes place. The desktop on the user's computer is loaded from the server. Only the user who is logged on has access to the desktop. Other users are not allowed to log on to that computer at the same time. However, if a user employs Terminal Services or Citrix, Windows can operate in a server-centric mode just as UNIX does.

The Windows operating system supports multiple users simultaneously through the command line and a GUI. The latter requires the use of Windows Terminal Services. The UNIX operating system supports multiple simultaneous users through the command line and through a GUI. The latter requires the use of X Windows. Windows comes with a default command shell (cmd.exe); UNIX typically includes several shells and encourages each user to choose a shell as the user's default shell. Both operating systems provide complete isolation between simultaneous users. There are some key differences between the two: Windows comes with a "single user" version that allows one user at a time (Windows XP) as well as a multiuser server version (Windows Server). It is rare for a Windows Server system to have multiple simultaneous command-line users.

- **Multithreading.** Most new UNIX kernels are multithreaded to take advantage of symmetric multiprocessing (SMP) computers. Initially, UNIX did not expose threads to programmers. However, POSIX has user-programmable threads. There is a POSIX standard for threads (called Pthreads) that all current versions of UNIX support.

In Windows, creating a new thread is very efficient. Windows applications are capable of using threads to take advantage of SMP computers and to maintain interactive capabilities when some threads take a long time to execute.

- **Process hierarchy.** When a UNIX-based application creates a new process, the new process becomes a child of the process that created it. This process hierarchy is often important, and there are system calls for manipulating child processes.

Unlike UNIX, Windows processes do not share a hierarchical relationship. The creating process receives the process handle and ID of the process it created, so a hierarchical relationship can be maintained or simulated if required by the application. However, the operating system treats all processes like they belong to the same generation. Windows provides a feature called Job Objects, which allows disparate processes to be grouped together and adhere to one set of rules.

### *Signals, Exceptions, and Events*

In both operating systems, events are signaled by software interrupts. A signal is a notification mechanism used to notify a process that some event has taken place or to handle the interrupt information from the operating system. An event is used to communicate between the processes. Exceptions occur when a program executes abnormally because of conditions outside the program's control.

In UNIX, signals are used for typical events, simple interprocess communication (IPC), and abnormal conditions such as floating point exceptions.

Windows has the following mechanisms:

- Windows supports some UNIX signals and others can be implemented using Windows API and messages.
- An event mechanism that handles expected events, such as communication between two processes.
- An exception mechanism that handles nonstandard events, such as the termination of a process by the user, page faults, and other execution violations.

### *Daemons and Services*

A daemon is a process that detaches itself from the terminal and runs disconnected in the background, waiting for requests and responding to them. A service is a special type of application that is available on Windows and runs in the background with special privileges.

In UNIX, a daemon is a process that the system starts to provide a service to other applications. Typically, the daemon does not interact with users. UNIX daemons are

started at boot time from init or rc scripts. To modify such a script, it needs to be opened in a text editor and the values of the variables in the script need to be physically changed. On UNIX, a daemon runs with an appropriate user name for the service that it provides or as a root user.

A Windows service is the equivalent of a UNIX daemon. It is a process that provides one or more facilities to client processes. Typically, a service is a long-running, Windows-based application that does not interact with users and, consequently, does not include a UI. Services may start when the system restarts and then continue running across logon sessions. Windows has a registry that stores the values of the variables used in the services. Control Panel provides a UI that allows users to set the variables with the valid values in the registry. The security context of that user determines the capabilities of the service. Most services run as either Local Service or Network Service. The latter is required if the service needs to access network resources and must run as a domain user with enough privileges to perform the required tasks.

### *Virtual Memory Management*

Virtual memory is a method of extending the available physical memory or RAM on a computer. In a virtual memory system, the operating system creates a pagefile, or swapfile, and divides memory into units called pages. Virtual memory management implements virtual addresses and each application is capable of referencing a physical chunk of memory, at a specific virtual address, throughout the life of the application.

Both UNIX and Windows use virtual memory to extend the memory available to an application beyond the actual physical memory installed on the computer. For both operating systems, on 32-bit architecture, each process gets a private 2 GB of virtual address space. This is called user address space or process address space. The operating system uses 2 GB of virtual address space, called system address space or operating system memory. On 64-bit architecture, each process gets 8 terabytes of user address space.

## *File Systems and Networked File Systems*

This section describes the file system characteristics of UNIX and Windows. Both UNIX and Windows support many different types of file system implementations. Some UNIX implementations support Windows file system types, and there are products that provide Windows support for some UNIX file system types.

File system characteristics and interoperability of file names between UNIX and Windows are discussed as follows:

- **File names and path names.** Everything in the file system is either a file or a directory. UNIX and Windows file systems are both hierarchical, and both operating systems support long file names of up to 255 characters. Almost any character is valid in a file name, except the following:
  - In UNIX: /
  - In Windows: ?, ", /, \, >, <, \*, |, and :

UNIX file names are case sensitive while Windows file names are not.

In UNIX, a single directory known as the root is at the top of the hierarchy. You locate all files by specifying a path from the root. UNIX makes no distinction between files on a local hard drive partition, CD-ROM, floppy disk, or network file system (NFS). All files appear in one tree under the same root.

The Windows file system can have many hierarchies, for example, one for each partition and one for each network drive. A UNIX system provides a single file system tree, with a single root, to the applications it hosts. Mounted volumes (whether local devices or network shares) are "spliced" into that tree at locations determined by the system administrator. The Windows operating system exposes a forest of file system trees, each with its own root, to the applications it hosts. Mounted volumes appear as separate trees ("drive letters") as determined by the administrator or user. Both UNIX and Windows provide a tree view of all network-accessible shares. UNIX provides an administrator-defined view of these shares through an automounter mechanism, while Windows provides a full view through the Universal Naming Convention (UNC) pathname syntax.

- **Server message block (SMB) and Common Internet File System (CIFS).** One of the earliest implementations of network resource sharing for the Microsoft MS-DOS® platform was network basic input/output system (NetBIOS). The features of NetBIOS allow it to accept disk I/O requests and direct them to file shares on other computers. The protocol used for this was named server message block (SMB). Later, additions were made to SMB to apply it to the Internet, and the protocol is now known as Common Internet File System (CIFS).

UNIX supports this through a software option called Samba. Samba is an open-source, freeware, server-side implementation of a UNIX CIFS server.

In Windows, the server shares a directory, and the client then connects to the Universal Naming Convention (UNC) to connect to the shared directory. Each network drive usually appears with its own drive letter, such as X.

- **Windows and UNIX NFS interoperability.** UNIX and Windows can interoperate using NFS on Windows or CIFS on UNIX. There are a number of commercial NFS products for Windows. For UNIX systems, Samba is an alternative to installing NFS client software on Windows-based computers for interoperability with UNIX-based computers. It also implements NetBIOS-style name resolution and browsing. Microsoft offers a freely downloadable NFS Server, Client, and Gateway as part of Windows Services for UNIX 3.5 installation. Windows Services for UNIX also provide a number of services for interoperability between Windows-based and UNIX-based computers.

### *Security*

This section describes some of the security implementation details and differences between UNIX and Windows:

- **User authentication.** A user can log on to a computer running UNIX by entering a valid user name and password. A UNIX user can be local to the computer or known on a Network Information System (NIS) domain (a group of cooperating computers). In most cases, the NIS database contains little more than the user name, password, and group.

A user can log on to a computer running Windows by entering a valid user name and password. A Windows user can be local to the computer, can be known on a Windows domain, or be known in the Microsoft Active Directory® directory service. The Windows domain contains only a user name, the associated password, and some user groups. Active Directory contains the same information as the Windows domain and may contain the contact information of the user, organizational data, and certificates.

- **UNIX versus Windows security.** UNIX uses a simple security model. The operating system applies security by assigning permissions to files. This model works because UNIX uses files to represent devices, memory, and even processes. When a user logs on to the system with a user name and a password, UNIX starts a shell with the UID and GID of that user. From then on, the permissions assigned to the UID and GID, or the process, control all access to files and other resources. Most UNIX vendors can provide Kerberos support, which raises their sophistication to about that of Windows.

Windows uses a unified security model that protects all objects from unauthorized access. The system maintains security information for:

- **Users.** System users are people who log on to the system, either interactively by entering a set of credentials (typically user name and password) or remotely through the network. Each user's security context is represented by a logon session.
- **Objects.** These are the secured resources (for example, files, computers, synchronization objects, and applications) that a user can access.
- **Active Directory.** Windows Server 2003 uses the Active Directory directory service to store information about objects. These objects include users, computers, printers, and every domain on one or more wide area networks (WANs). Active Directory can scale from a single computer to many large computer networks. It provides a store for all the domain security policy and account information.

## *Networking*

Networking basically provides the communication between two or more computers. It defines various sets of protocols, configures the domains, IP addresses, and ports, and communicates with the external devices like telephones or modems and data transfer methods. It also provides the standard set of API calls to allow applications to access the networking features.

The primary networking protocol for UNIX and Windows is TCP/IP. The standard programming API for TCP/IP is called sockets. The Windows implementation of sockets is known as Winsock (formally known as Windows Sockets). Winsock conforms well to the Berkeley implementation, even at the API level. The key difference between UNIX sockets and Winsock exists in asynchronous network event notification. There is also a difference in the remote procedure calls (RPC) implementation in UNIX and Windows.

## *User Interfaces*

The user interface (UI) provides a flexible way of communicating between the users, applications, and the computer.

The UNIX UI was originally based on a character-oriented command line, whereas the Windows UI was based on GUI. UNIX originated when graphic terminals were not available. However, the current versions of UNIX support the graphical user interface using the X Windows system. Motif is the most common windowing system, library, and user-interface style built on X Windows. This allows the building of graphical user interface applications on UNIX.

The Windows user interface was designed to take advantage of advancements in the graphics capabilities of computers. It can be used by all applications—including both client side and server side—and can also be used for tasks such as service administration. Windows contains the Graphics Device Interface (GDI) engine to support the graphical user interface.

## *System Configuration*

UNIX users generally configure a system by editing the configuration files with any of the available text editors. The advantage of this mechanism is that the user does not need to learn how to use a large set of configuration tools, but must only be familiar with an editor and possibly a scripting language. The disadvantage is that the information in the files comes in various formats; hence the user must learn the various formats in order to change the settings. UNIX users often employ scripts to reduce the possibility of repetition and error. In addition, they can also use NIS to centralize the management of many standard configuration files. Although different versions of UNIX have GUI management tools, such tools are usually specific to each version of UNIX.

Windows has GUI tools for configuring the system. The advantage of these tools is that they can offer capabilities depending on what is being configured. In recent years, Microsoft Management Console (MMC) has provided a common tool and UI for creating configuration tools. Windows provides a scripting interface for most configuration needs through the Windows Script Host (WSH). Windows provides WMI (Windows Management Instrumentation), which can be used from scripts. Windows also includes extensive command-line tools for controlling system configuration. In Windows Server 2003, anything that can be done to manage a system through a GUI can be done through a command-line tool as well.

## *Interprocess Communication*

An operating system designed for multitasking or multiprocessing must provide mechanisms for communicating and sharing data between applications. These mechanisms are called interprocess communication (IPC).

UNIX has several IPC mechanisms that have different characteristics and which are appropriate for different situations. Shared memory, pipes, and message queues are all suitable for processes running on a single computer. Shared memory and message queues are suitable for communicating among unrelated processes. Pipes are usually chosen for communicating with a child process through standard input and output. For communications across the network, sockets are usually the chosen technique.

Windows also has many IPC mechanisms. Like UNIX, Windows has shared memory, pipes, and events (equivalent to signals). These are appropriate for processes that are local to a computer. In addition to these mechanisms, Windows supports clipboard/Dynamic Data Exchange (DDE), and Component Object Model (COM). Winsock and Microsoft Message Queuing are good choices for cross-network tasks. Other cross-network IPC mechanisms for Windows include remote procedure calls (RPCs) and mail slots. RPC has several specifications and implementations, developed by third-party vendors, which support client server applications in distributed environments. The most prevalent RPC specifications are Open Network Computing (ONC) by Sun Microsystems and Distributed Computing Environment (DCE) by Open Software Foundation. UNIX systems support interoperability with Windows RPC. UNIX does not implement mailslots.

### *Synchronization*

In a multithreaded environment, threads may need to share data between them and perform various actions. These operations require a mechanism to synchronize the activity of the threads. These synchronization techniques are used to avoid race conditions and to wait for signals when resources are available.

UNIX environments use several techniques in the Pthreads implementation to achieve synchronization. They are:

- Mutexes
- Condition variables
- Semaphores
- Interlocked exchange

Similarly, the synchronization techniques available in the Windows environment are:

- Spinlocks
- Events
- Critical sections
- Semaphores
- Mutexes

- Interlocked exchange

### *DLLs and Shared Libraries*

Windows and UNIX both have a facility that allows the application developer to put common functionality in a separate code module. This feature is called a shared library in UNIX and a dynamic-link library (DLL) in Windows. Both allow application developers to link together object files from different compilations and to specify which symbols will be exported from the library for use by external programs. The result is the capability to reuse code across applications. The Windows operating system and most Windows programs use many DLLs.

Windows DLLs do not need to be compiled to position-independent code (PIC), while UNIX shared objects must be compiled to PIC. However, the exact UNIX behavior can be emulated in Windows by pre-mapping different DLLs at different fixed addresses.

### *Component-based Development*

Component-based development is an extension to the conventional software development where the software is assembled by integrating several components. The components themselves may be written in different technologies and languages, but each has a unique identity, and each of them exposes common interfaces so that they can interact with other components.

UNIX supports CORBA as the main component-based development tool. However, it is not a standard component of the UNIX system; you have to obtain a CORBA implementation from another source.

On the other hand, the Windows environment offers a wide range of component-based development tools and technologies. This includes:

- COM
- COM+
- Distributed COM (DCOM)
- .NET components

## *Middleware*

This section compares the various middleware solutions available for UNIX-based and Windows-based applications. Middleware technologies are mostly used to connect the presentation layer with the back-end business layers or data sources. One of the most prominent middleware technologies used in applications is a message queuing system.

Message queuing is provided as a feature in AT&T System V UNIX and can be achieved through sockets in Berkeley UNIX versions. These types of memory queues are most often used for IPC and do not meet the requirements for persistent store and forward messaging.

To meet these requirements, versions of the IBM MQSeries and the BEA Systems MessageQ (formally the DEC MessageQ) are available for UNIX. Microsoft provides similar functionality with Message Queuing for Windows.

## *Transaction Processing Monitors*

A transaction processing monitor (TP monitor) is a subsystem that groups sets of related database updates and submits them to a database as a transaction. These transactions, often monitored and implemented by the TP monitors, are known as online transaction processing (OLTP). OLTP is a group of programs that allow real-time updating, recording, and retrieval of data to and from a networked system. OLTP systems have been implemented in the UNIX environments for many years. These systems perform such functions as resource management, threading, and distributed transaction management.

Although OLTP was originally developed for UNIX, many OLTP systems have Windows versions. Windows also ships with its own transaction manager. In addition, gateways exist to integrate systems that use different transaction monitors.

## *Shells and Scripting*

A shell is a command-line interpreter that accepts typed commands from a user and executes the resulting request. Every shell has its own set of programming features

known as scripting languages. Programs written through the programming features of a shell are called shell scripts. As with shell scripts, scripting languages are interpreted.

Windows and UNIX support a number of shells and scripting languages, some of which are common to both operating systems. Examples are: Rexx, Python, and Tcl/Tk.

### Command-Line Shells

A number of standard shells are provided in the UNIX environment as part of the standard installation. They are:

- Bourne shell (sh)
- C shell (csh)
- Korn shell (ksh)
- Bourne Again Shell (bash)

On the Windows platform, Cmd.exe is the command prompt or the shell.

Windows versions of the Korn shell and the C shell are delivered with the Windows Services for UNIX 3.5, MKS, and Cygwin products.

### Scripting Languages

In UNIX, there are three main scripting languages that correspond to the three main shells: the Bourne shell, the C shell, and the Korn shell. Although all the scripting languages are developed from a common core, they have certain shell-specific features to make them slightly different from each other. These scripting languages mainly use a group of UNIX commands for execution without the need for prior compilation. Some of the external scripting languages that are also supported on the UNIX environment are Perl, Rexx, and Python.

On the Windows environment, WSH is a language-independent environment for running scripts and is compatible with the standard command shell. It is often used to automate administrative tasks and logon scripts. WSH provides objects and services for scripts, establishes security, and invokes the appropriate script engine, depending on the script language. Objects and services supplied allow the script to perform such tasks as

displaying messages on the screen, creating objects, accessing network resources, and modifying environment variables and registry keys. WSH natively supports VBScript and JScript. Other languages that are available for this environment are Perl, Rexx, and Python. WSH is built-in to all current versions of Windows and can be downloaded or upgraded from Microsoft.

### *Development Environments*

The generic UNIX development environment uses a set of command-line tools. In addition, there are many third- party integrated development environments (IDEs) available for UNIX. Most of the character-based and visual IDEs provide the necessary tools, libraries, and headers needed for application development

The Windows development environment can be of two types: a standard Windows development environment or a UNIX-like development environment such as Windows Services for UNIX.

The standard Windows development environment uses the Microsoft Platform Software Development Kit (SDK) and Microsoft Visual Studio® .NET. The platform SDK delivers documentation for developing Windows-based applications, libraries, headers, and definitions needed by language compilers. It also includes a rich set of command-line and stand-alone tools for building, debugging, and testing applications. It is available at no cost from the MSDN Web site.

The Microsoft Visual Studio .NET environment delivers a complete set of tools for application development, including the development of multitier components, user interface design, and database programming and design. It also provides several language tools, editing tools, debugging tools, performance analysis tools, and application installation tools.

The development environment of Windows Services for UNIX contains documentation, tools, API libraries, and headers needed by language compilers. It also comes with a UNIX development environment, with tools such as GNU gcc, g++, g77 compilers, and a gdb debugger, which makes compilation of UNIX applications possible on the Windows environment.

## *Application Architectures*

The following sections introduce and discuss different application architectures and how these applications are implemented on UNIX and Windows platforms.

### *Distributed Applications*

Distributed applications are logically partitioned into multiple tiers: the view or the presentation tier, the business tier, and the data storage and access tier. A simple distributed application model consists of a client that communicates with the middle tier, which consists of the application server and an application containing the business logic. The application, in turn, communicates with a database that stores and supplies the data.

The most commonly used databases in UNIX applications are Oracle and DB2. You can either run the application's current database (for example, Oracle) on Windows (that is, migrate the existing database from UNIX to Windows) or migrate the database to Microsoft SQL Server™. In some cases, the best migration decision is a conversion to SQL Server.

Another option available is Oracle. It offers a range of features available to both Windows Server 2003 and UNIX. You may choose to use these features with the current Oracle database. By separating the platform migration from the database migration, you have greater flexibility in migrating database applications.

Next is the presentation tier, which provides either a thick or a thin client interface to the application. UNIX applications may use XMotif to provide a thick client interface. In Windows, a thick client can be developed using the Win32 API, GDI+. The .NET Framework provides Windows Forms that help in rapid development of thick clients. Either one of these two methods can be used while migrating the thick client from UNIX to Windows.

### *Workstation Applications*

Workstation-based applications run at the UNIX workstation (desktop) computer and access data that resides on network file shares or database servers. Workstation-based applications have the following architectural characteristics:

- They can be single-process applications or multiple-process applications.
- They use character-based or GUI-based (for example, X Windows or OpenGL) UIs.
- They access a file server (through NFS) or a database server for data resources.
- They access a computer server for computer-intensive services (for example, finite element models for structural analysis).

The Windows environment supports a similar workstation application using client/server technology.

### Web Applications

Web applications from a UNIX Web server are normally one of the following types:

- **Common Gateway Interface (CGI).** CGI is a standard for connecting external applications with information servers, such as Hypertext Transfer Protocol (HTTP) or Web servers. CGI has long been out of favor because of performance problems.
- **Java Server Page (JSP).** Java Server Page (JSP) technology also allows for the development of dynamic Web pages. JSP technology uses Extensible Markup Language (XML)-like tags and scriptlets written in the Java programming language to encapsulate the logic that generates the content for the page. The application logic can reside in server-based resources (for example, the JavaBean component architecture) that the page accesses by using these tags and scriptlets. All HTML or XML formatting tags are passed directly back to the response page.
- **HTML.** Hypertext Markup Language is the authoring language used to create documents on the World Wide Web.
- **PHP.** PHP is a widely used, general-purpose scripting language that is especially suited for Web development and can be embedded into HTML.
- **JavaScript.** JavaScript is an extension of HTML. JavaScript is a script language (a system of programming codes) created by Netscape that can be embedded into the HTML of a Web page to add functionality.

Web applications on UNIX can be used on Windows with minor configuration changes on the Web server. You can also migrate Java-based Web applications on UNIX to Microsoft Web technologies on Windows.

### Graphics-Intensive Applications

Graphics-intensive applications may support additional UI standards, such as OpenGL. OpenGL has become a widely used and supported two-dimensional and three-dimensional graphics API. OpenGL fosters innovation and provides for faster application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can take advantage of the capabilities of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment.

OpenGL runs on every major operating system, including Macintosh OS, OS/2, UNIX, Windows Server 2003, Windows XP, Linux, OPENStep, and BeOS. OpenGL also works with every major windowing system, including Win32, Macintosh OS, Presentation Manager, and X-Windows. OpenGL includes a full complement of graphics functions. The OpenGL standard has language bindings for C, C++, Fortran, Ada, and Java; therefore, applications that use OpenGL functions are typically portable across a wide array of platforms.

### System-Level Programs or Device Drivers

The existence of an appropriate device driver on the Windows platform is often a gating factor for migrating applications that make use of nonstandard device drivers. Typically, customers do not have access to device-driver source code and are therefore unable to migrate a UNIX device driver to Windows. The topic of migrating device drivers is highly specialized and is not covered in this guide. The Windows Driver Development Kit (DDK) can be used to develop drivers on the Windows environment.

**Q8 b) What are the steps involved in installing a device driver in UNIX system? Explain the steps in details.**

**Ans-**

Installing the device driver in a UNIX system involves the following steps:

- 1) compiling the device driver
- 2) Modifying the kernel configuration tables and files
- 3) Linking the device driver with the kernel object files to produce a new kernel
- 4) Creating the necessary entries in the /dev directory
- 5) Rebooting the system with new kernel

### **1) Compiling the device driver**

- Compiling the driver is similar to compiling any other C program
- Since the driver is not a stand-alone program it is always compiled to produce an object rather than a linked executable file.
- System special flags are needed to ensure the correct part of include files are used and that compiler generates a module appropriate for linking into the kernel
- The information must be obtained from system's documentation.

### **2) Modifying or configuring the kernel**

- kernel configuration tables are used to reflect the addition of new driver
- Methods varies between different version of Unix
- there are two config files used by the system to generate a kernel are the mdevice and sdevice file in /etc/conf/cd.d directory contains entry for each device driver that exists on the system
- Sdevice contains information for each device driver that is to be incorporated into the kernel
- Mdevice contains entry for each device driver that exist on the system

### **3) Linking the device driver with the kernel object to produce a new kernel**

- system utility to run is idbuild
- once the kernel configuration tables have been modified then we can generate the new kernel
- The command to build a new kernel varies between different UNIX systems

- On SVR 3.2 system the utility to run is idbuild. This program will generate a new UNIX kernel.
- On some SVR 3.2 systems the idbuild utility will ask us if we want the kernel to boot by default
- On others, the new kernel will automatically be used during the next boot

#### **4) Creating the necessary entries in the /dev directory**

- dev directory reference our device driver
- It can be done with mknod command
- On most system it will be automatic
- Node files has four fields
  - Device Name
  - Node Name
  - Node type
  - Minor Device Number

#### **Q6 b) Explain the following terms in Direct linking loader**

- 1) Global external symbol table (GEST)**
- 2) Local external symbol array (LESA)**

**Ans-**

#### **3) Global external symbol table (GEST)**

It is used to store the external definition by means of a segment definition (SD) or Local Definition (LD) entry on an External Symbol Director (ESD) card. When these symbols are encountered during pass 1, they are assigned an absolute core address; this is stored along with the symbol in the GEST.

#### **4) Local external symbol array (LESA)**

In pass 2 of the loader the GEST and ESD information for each individual object deck is merged to produce the Local external symbol array (LESA) that directly relates the ID number and value it is necessary to create a separate LESA for each segment.

# Unit 6

## Unit -6

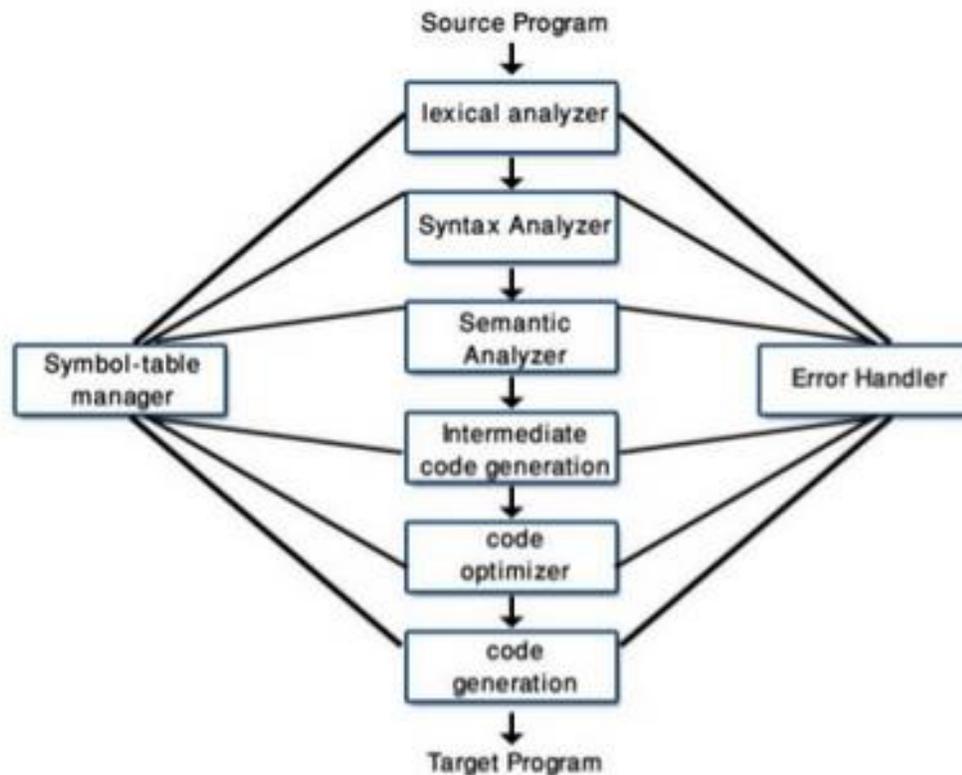
### Compiler

The word *compilation* is used to denote the task of translating *high level language* (HLL) programs into machine language programs. Though the objective of this task of translation is similar to that of an *assembler*, the problem of compilation is much more complex than that of an assembler. A *compiler* is a program that does the compilation task. A compiler recognizes programs in a particular HLL and produces equivalent output programs appropriate for some particular computer configuration (hardware and OS). Thus, an HLL program is to a great extent independent of the configuration of the machine it will eventually run on, as long as it is ensured that the program is compiled by a compiler that recognizes that HLL and produces output for the required machine configuration. It is common for a machine to have compilers that would translate programs to produce executables for that machine (*hosts*). But there also are compilers that runs on one type of machine but the output of which are programs that shall run on some other machine configuration, such as generating an MS-DOS executable program by compiling an HLL program in UNIX. Such a compiler is called a *cross compiler*. Another kind of translator that accepts programs in HLL are known as *interpreters*. An interpreter translates an input HLL program and *also runs the program on the same machine*. Hence the output of running an interpreter is actually the output of the program that it translates.

### Important phases in Compilation

The following is a typical breakdown of the overall task of a compiler in an approximate sequence -

- Lexical analysis
- Syntax analysis
- Intermediate code generation
- Code optimization
- Code generation.



Like an assembler, a compiler usually performs the above tasks by making multiple passes over the input or some intermediate representation of the same. The compilation task calls for intensive processing of information extracted from the input programs, and hence data structures for representing such information need to be carefully selected. During the process of translation a compiler also detects certain kinds of errors in the input, and may try to take some recovery steps for these.

### Lexical Analysis

Lexical analysis in a compiler can be performed in the same way as in an assembler. Generally in an HLL there are more number of tokens to be recognized - various keywords (such as, *for*, *while*, *if*, *else*, etc.), punctuation symbols (such as, comma, semi-colon, braces, etc.), operators (such as arithmetic operators, logical operators, etc.), identifiers, etc. Tools like *lex* or *flex* are used to create lexical analysers.

### Syntax Analysis

Syntax analysis deals with recognizing the structure of input programs according to known set of *syntax rules* defined for the HLL. This is the most important aspect in which HLLs are significantly different from lower level languages such as assembly language. In assembly languages the syntax rules are simple which roughly requires that a program should be a sequence of statements, and each statement should essentially contain a mnemonic followed by zero or more operands depending on the mnemonic. Optionally, there can be also being an identifier preceding the mnemonic. In case of HLLs, the syntax rules are much more complicated. In most HLLs the notion of a

statement itself is very flexible, and often allows *recursion*, making nested constructs valid. These languages usually support multiple data types and often allow programmers to define abstract data types to be used in the programs. These and many other such features make the process of creating software easier and less error prone compared to assembly language programming. But, on the other hand, these features make the process of compilation complicated.

The non-trivial syntax rules of HLLs need to be cleverly specified using some suitable notation, so that these can be encoded in the compiler program. One commonly used formalism for this purpose is the *Context Free Grammar (CFG)*. CFG is a formalism that is more powerful than *regular grammars* (used to write regular expressions to describe *tokens* in a lexical analyser). Recursion, which is a common feature in most constructs of HLLs, can be defined using a CFG in a concise way, whereas a regular grammar is incapable of doing so. It needs to be noted that there are certain constructs that cannot be adequately described using CFG, and may require other more powerful formalisms, such as *Context Sensitive Grammars (CSG)*. A common notation used to write the rules of CFG or CSG is the *BNF (Backus Naur Form)*.

During syntax analysis, the compiler tries to apply the rules of the grammar of the input HLL given using BNF, to recognise the structure of the input program. This is called *parsing* and the module that performs this task is called a *parser*. From a somewhat abstract point of view, the output of this phase is a *parse tree* that depicts how various rules of the grammar can be repetitively applied to recognise the input program. If the parser cannot create a parse tree for some given input program, then the input program is not valid according to the syntax of the HLL.

The soundness of the CFG formalism and the BNF notation makes it possible to create different types of efficient parsers to recognise input according to a given language. These parsers can be broadly classified as *top-down parsers* and *bottom-up parsers*. *Recursive descent parsers* and *Predictive parsers* are two examples of top-down parsers. *SLR parsers* and *LALR parser* are two examples of bottom-up parsers. For certain simple context free languages (languages that can be defined using CFG) simpler bottom-up parsers can be written. For example, for recognising mathematical expressions, an *operator precedence parser* can be created.

In creating a compiler, a parser is often built using tools such as *yacc* and *bison*. To do so the CFG of the input language is written in BNF notation, and given as input to the tool (along with other details).

### **Intermediate Code Generation**

Having recognized a given input program as valid, a compiler tries to create the equivalent program in the language of the target environment. In case of an assembler this translation was somewhat simpler since the operation implied by the mnemonic opcode in each statement in the input program, there is some equivalent machine opcode. The number of operands applicable for each operation in the machine language is the same as allowed for the corresponding assembly language mnemonic opcodes. Thus for the assembly language the translation for each statement can be done for each statement almost independently of the rest of the program. But, in case of an HLL, it is futile to try to associate a single machine opcode for each statement of the input language. One of the

reasons for this is, as stated above, the extent of a statement is not always fixed and may contain recursion. Moreover, data references in HLL programs can assume significant levels of abstractions in comparison to what the target execution environment may directly support. The task of associating meanings (in terms of primitive operations that can be supported by a machine) to programs or segments of a program is called *semantic processing*.

Though it is not entirely straightforward to associate target language operations to statements in the HLL programs, the CFG for the HLL allows one to associate *semantic actions* (or implications) for the various syntactic rules. Hence in the broad task of translation, when the input program is parsed, a compiler also tries to perform certain semantic actions corresponding to the various syntactic rules that are eventually applied. However, most HLLs contain certain syntactic features for which the semantic actions are to be determined using some additional information, such as the contents of the symbol table. Hence, building and usage of data-structures such as the symbol table are an important part of the semantic action that are performed by the compiler.

Upon carrying out the semantic processing a more manageable equivalent form of the input program is obtained. This is stored (represented) using some *Intermediate code* representation that makes further processing easy. In this representation, the compiler often has to introduce several temporary variables to store intermediate results of various operations. The language used for the intermediate code is generally not any particular machine language, but is such which can be efficiently converted to a required machine language (some form of assembly language can be considered for such use).

### **Code Optimisation**

The programs represented in the intermediate code form usually contains much scope for optimization both in terms of storage space as well as run time efficiency of the intended output program. Sometimes the input program itself contains such scope. Besides that, the process of generating the intermediate code representation usually leaves much room for such optimization. Hence, compilers usually implement explicit steps to optimise the intermediate code.

### **Code Generation**

Finally, the compiler converts the (optimised) program in the intermediate code representation to the required machine language. It needs to be noted that if the program being translated by the compiler actually has dependencies on some external modules, then *linking* has to be performed to the output of the compiler. These activities are independent of whether the input program was in HLL or assembly language.

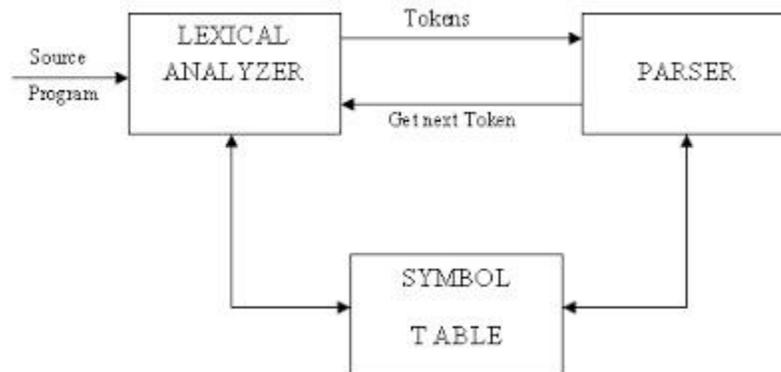
### **Lexical analysis**

**Lexical analysis** is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called

a **lexical analyzer**, **lexer**, or **scanner**. A lexer often exists as a single function which is called by a **parser** or another function.

### Role of the lexical analyzer

The main task is to read the input characters and produce as output sequence of tokens that the parser uses for syntax analysis.



**Fig 2.1 role of the lexical analyzer diagram**

Up on receiving a “get next token” command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Its secondary tasks are,

- One task is stripping out from the source program comments and white space in the form of blank, tab, new line characters.
- Another task is correlating error messages from the compiler with the source program.

Sometimes lexical analyzer is divided in to cascade of two phases.

- 1) Scanning
- 2) lexical analysis.

The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

## Recognition of tokens:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

```
Stmt → if expr
      then stmt
      | If expr then else stmt
      | ε
Expr → term relop term
      | term
Term → id
      | number
```

For relop ,we use the comparison operations of languages like Pascal or SQL where = is “equals” and <> is “not equals” because it presents an interesting structure of lexemes. The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

```
digit    -->[0,9]
digits   --
>digit+
number   -->digit(.digit)?(e.[+-]?digits)?
letter   -->[A-Z,a-z]
id       --
>letter(letter/digit)*
if       -->
if
then     -->then
else     -->else
relop    --></>/<=/>/=//< >
```

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the “token” we defined by:

$ws \rightarrow (\text{blank/tab/newline})^+$

Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

Lexeme	Token Name	Attribute Value
Any ws	–	–
if	if	–
then	then	–
else	else	–
Any id	id	pointer to table entry
Any number	number	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	ET
<>	relop	NE

### Transition Diagram:

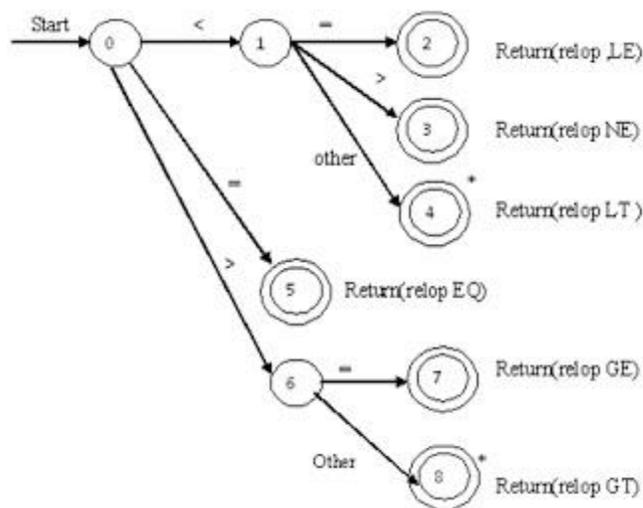
Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns.

Edges are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or set of symbols.

If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

### Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.
2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a \* near that accepting state.
3. One state is designed the state ,or initial state ., it is indicated by an edge labeled “start” entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.



### Theory

During the first phase the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to lex so it can generate code that will allow it to scan and match strings in the input. Each pattern specified in the input to lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.

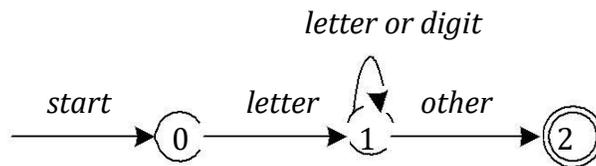
The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

**letter(letter|digit)\***

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

repetition, expressed by the “\*”  
operator alternation, expressed by  
the “|” operator concatenation

Any regular expression expressions may be expressed as a finite state automaton (FSA). We can represent an FSA using states, and transitions between states. There is one start state and one or more final or accepting states.



**Figure 3:** Finite State Automaton

In Figure 3 state 0 is the start state and state 2 is the accepting state. As characters are read we make a transition from one state to another. When the first letter is read we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit we transition to accepting state 2. Any FSA may be expressed as a computer program. For example, our 3-state machine is easily programmed:

```
start: goto state0
state0: read c
       if c = letter goto state1
       goto state0
state1: read c
       if c = letter goto state1
       if c = digit goto state1
       goto state2
state2: accept string
```

This is the technique used by lex. Regular expressions are translated by lex to a computer program that mimics an FSA. Using the next *input* character and *current state* the next state is easily determined by indexing into a computer-generated state table.

Now we can easily understand some of lex's limitations. For example, lex cannot be used to recognize nested structures such as parentheses. Nested structures are handled by incorporating a stack. Whenever we encounter a "(" we push it on the stack. When a ")" is encountered we match it with the top of the stack and pop the stack. However lex only has states and transitions between states. Since it has no stack it is not well suited for parsing nested structures. Yacc augments an FSA with a stack and can process constructs such as parentheses with ease. The important thing is to use the right tool for the job. Lex is good at pattern matching. Yacc is appropriate for more challenging tasks.

### Practice

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
\$	end of line
a b	a or b
(ab)+	one or more copies of <b>ab</b> (grouping)
"a+b"	literal " " (C escapes still work)
	<b>a+b</b>
[]	character class

**Table 1:** Pattern Matching Primitives

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abc bc abc bc bc ...
a(bc)?	a abc
[abc]	a b c
	<b>a b c</b>
[a-z]	any letter, a-z
[a\ -z]	a - z
[-az]	one a - z

	of:
	- a z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a b
[a^b]	one a b of:
[a b]	one a ^ b of:
a b	one a   b of:
	a b

**Table 2:** Pattern Matching Examples

Regular expressions in lex are composed of metacharacters (Table 1). Pattern-matching examples are shown in Table 2. Within a character class normal operators lose their meaning.

Two operators allowed in a character class are the hyphen (“-”) and circumflex (“^”). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string, the longest match wins. In case both matches are the same length, then the first pattern listed is used.

... **definitions** ...

%%

... **rules** ...

%%

... **subroutines** ...

Input to Lex is divided into three sections with %% dividing the sections. This is best illustrated by example. The first example is the shortest possible lex file:

```
%%
```

Input is copied to output one character at a time. The first %% is always required, as there must always be a rules section. However if we don't specify any rules then the default action is to match everything and copy it to output. Defaults for input and output are **stdin** and **stdout**, respectively. Here is the same example with defaults explicitly coded:

```
%%
    /* match everything except newline */
    . ECHO;
    /* match newline */
    \n ECHO;
%%

int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

Two patterns have been specified in the rules section. Each pattern must begin in column one. This is followed by whitespace (space, tab or newline) and an optional action associated with the pattern. The action may be a single C statement, or multiple C statements, enclosed in braces. Anything not starting in column one is copied verbatim to the generated C file. We may take advantage of this behavior to specify comments in our lex file. In this example there are two patterns, “.” and “\n”, with an **ECHO** action associated for each pattern. Several macros and variables are predefined by lex. **ECHO** is a macro that writes code matched by the pattern. This is the default action for any unmatched strings. Typically, **ECHO** is defined as:

```
#define ECHO fwrite(vvtext, vvleng, 1, vvout)
```

Variable **yytext** is a pointer to the matched string (NULL-terminated) and **yylen** is the length of the matched string. Variable **yyout** is the output file and defaults to stdout. Function **yywrap** is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required. Every C program requires a **main** function. In this case we simply call **yylex** that is the main entry-point for lex. Some implementations of lex include copies of **main** and **yywrap** in a library thus eliminating the need to code them explicitly. This is why our first example, the shortest lex program, functioned properly.

Name	Function
<b>int yylex(void)</b>	call to invoke lexer, returns token
<b>char *yytext</b>	pointer to matched string
<b>yyleng</b>	length of matched string
<b>yylval</b>	value associated with token
<b>int yywrap(void)</b>	wrapup, return 1 if done, 0 if not done
<b>FILE *yyout</b>	output file
<b>FILE *yyin</b>	input file
<b>INITIAL</b>	initial start condition
<b>BEGIN</b>	condition switch start condition
<b>ECHO</b>	write matched string

**Table 3:** Lex Predefined Variables

Here is a program that does nothing at all. All input is matched but no action is associated with any pattern so there will be no output.

```
%%
.
```

The following example prepends line numbers to each line in a file. Some implementations of lex predefine and calculate **yylineno**. The input file for lex is **yyin** and defaults to **stdin**.

```
%{
    int yylineno;
}%
%%
^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) { yyin
    = fopen(argv[1], "r"); yylex();
    fclose(yyin);
}
```



The definitions section is composed of substitutions, code, and start states. Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with “%{” and “%}” markers. Substitutions simplify pattern-matching rules. For example, we may define digits and letters:

```
digit [0-9] letter
{[A-Za-z] %{
    int count;
%}
%%
    /* match identifier */
{letter}{{letter}|{digit))* count++;
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

Whitespace must separate the defining term and the associated expression. References to substitutions in the rules section are surrounded by braces (**{letter}**) to distinguish them from literals. When we have a match in the rules section the associated C code is executed. Here is a scanner that counts the number of characters, words, and lines in a file (similar to Unix wc):

```
%{
    int nchar, nword, nline;
%}
%%
\n    { nline++; nchar++; }
[\t\n]+ { nword++, nchar += yyleng; }
.    { nchar++; }
%%
int main(void) {
    yylex();
    printf("%d\t%d\t%d\n", nchar, nword, nline);
    return 0;
}
```

1  
0

## Yacc

### Theory

Grammars for yacc are described using a variant of Backus Naur Form (BNF). This technique, pioneered by John Backus and Peter Naur, was used to describe ALGOL60. A BNF grammar can be used to express *context-free* languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is

```
E -> E + E  
E -> E * E  
E -> id
```

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as **E** (expression) are nonterminals. Terms such as **id** (identifier) are terminals (tokens returned by lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

```
E -> E * E           (r2)  
  -> E * z           (r3)  
  -> E + E * z       (r1)  
  -> E + y * z       (r3)  
  -> x + y * z       (r3)
```

At each step we expanded a term and replace the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression we need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar we need to reduce an expression to a single nonterminal. This is known as *bottom-up* or *shift-reduce* parsing and uses a stack for storing terms. Here is the same derivation but in reverse order:

<b>1</b>	<b>. x + y * z</b>	<b>shift</b>	
<b>2</b>	<b>x . + y * z</b>	<b>reduce(r3)</b>	
<b>3</b>	<b>E . + y * z</b>	<b>shift</b>	
<b>4</b>	<b>E + . y * z</b>	<b>shift</b>	
<b>5</b>	<b>E + y . * z</b>	<b>reduce(r3)</b>	
<b>6</b>	<b>E + E . * z</b>	<b>shift</b>	
<b>7</b>	<b>E + E * . z</b>	<b>shift</b>	
<b>8</b>	<b>E + E * z .</b>	<b>reduce(r3)</b>	
<b>9</b>	<b>E + E * E .</b>	<b>reduce(r2)</b>	<b>emit multiply</b>
<b>10</b>	<b>E + E .</b>	<b>reduce(r1)</b>	<b>emit add</b>
<b>11</b>	<b>E .</b>	<b>accept</b>	

Terms to the left of the dot are on the stack while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production we replace the matched tokens on the stack with the lhs of the production. In other words the matched tokens of the rhs are popped

off the stack, and the lhs of the production is pushed on the stack. The matched tokens are known as a *handle* and we are *reducing* the handle to the lhs of the production. This process continues until we have shifted all input to the stack and only the starting nonterminal remains on the stack. In step 1 we shift the **x** to the stack. Step 2 applies rule r3 to the stack to change **x** to **E**. We continue shifting and reducing until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply

1

1

instruction. Similarly the add instruction is emitted in step 10. Consequently multiply has a higher precedence than addition.

Consider the shift at step 6. Instead of shifting we could have reduced and apply rule r1. This would result in addition having a higher precedence than multiplication. This is known as a *shift-reduce* conflict. Our grammar is *ambiguous* because there is more than one possible derivation that will yield the expression. In this case operator precedence is affected. As another example, associativity in the rule

**E -> E + E**

is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar or supply yacc with directives that indicate which operator has precedence. The latter method is simpler and will be demonstrated in the practice section.

The following grammar has a *reduce-reduce* conflict. With an **id** on the stack we may reduce to **T**, or **E**.

**E -> T**

**E -> id**

**T -> id**

Yacc takes a default action when there is a conflict. For shift-reduce conflicts yacc will shift. For reduce-reduce conflicts it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous. Several methods for removing ambiguity will be presented in subsequent sections.

### **Practice, Part I**

**... definitions ...**

**%%**

**... rules ...**

**%%**

**... subroutines ...**

Input to yacc is divided into three sections. The definitions section consists of token declarations and C code bracketed by “**%{**” and “**%}**”. The BNF grammar is placed in the rules section and user subroutines are added in the subroutines section.

This is best illustrated by constructing a small calculator that can add and subtract numbers. We'll begin by examining the linkage between lex and yacc. Here is the definitions section for the yacc input file:

**%token INTEGER**

This definition declares an **INTEGER** token. Yacc generates a parser in file **y.tab.c** and an include file, **y.tab.h**:

**#ifndef YYSTYPE**

```
#define YYSTYPE int  
#endif  
#define INTEGER 258  
extern YYSTYPE yylval;
```

1  
2

Lex includes this file and utilizes the definitions for token values. To obtain tokens yacc calls **yylex**. Function **yylex** has a return type of `int` that returns a token. Values associated with the token are returned by lex in variable **yylval**. For example,

```
[0-9]+    {
           yylval = atoi(yytext);
           return INTEGER;
           }
```

would store the value of the integer in **yylval**, and return token **INTEGER** to yacc. The type of **yylval** is determined by **YYSTYPE**. Since the default type is integer this works well in this case. Token values 0-255 are reserved for character values. For example, if you had a rule such as

```
[-+]      return *yytext;      /* return operator */
```

the character value for minus or plus is returned. Note that we placed the minus sign first so that it wouldn't be mistaken for a range designator. Generated token values typically start around 258 because lex reserves several values for end-of-file and error processing. Here is the complete lex input specification for our calculator:

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h" %}
%%

[0-9]+    {
           yylval = atoi(yytext);
           return INTEGER;
           }

[-+\n]    return *yytext;

[\t]      ; /* skip whitespace */

.         yyerror("invalid character");

%%

int yywrap(void) {
    return 1;
}
```

Internally yacc maintains two stacks in memory; a parse stack and a value stack. The parse stack contains terminals and nonterminals that represent the current parsing state. The value stack is an array of **YYSTYPE** elements and associates a value with each element in the parse stack. For example when lex returns an **INTEGER** token yacc shifts this token to the parse stack. At the same time the corresponding **yylval** is shifted to the value stack. The parse and value stacks are always synchronized so

finding a value related to a token on the stack is easily accomplished. Here is the yacc input specification for our calculator:

1  
3

```

%{
    #include <stdio.h> int
    yylex(void); void
    yyerror(char *);
%}

%token INTEGER

%%

program:
    program expr '\n'          { printf("%d\n", $2); }
    |
    ;

expr:
    INTEGER                    { $$ = $1; }
    | expr '+' expr            { $$ = $1 + $3; }
    | expr '-' expr            { $$ = $1 - $3; }
    ;

%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}

```

The rules section resembles the BNF grammar discussed earlier. The left-hand side of a production, or nonterminal, is entered left-justified and followed by a colon. This is followed by the right-hand side of the production. Actions associated with a rule are entered in braces.

With left-recursion, we have specified that a program consists of zero or more expressions. Each expression terminates with a newline. When a newline is detected we print the value of the expression. When we apply the rule

```

expr: expr '+' expr          { $$ = $1 + $3; }

```

we replace the right-hand side of the production in the parse stack with the left-hand side of the same production. In this case we pop “**expr '+' expr**” and push “**expr**”. We have reduced the stack by popping three terms off the stack and pushing back one term. We may reference positions in the value stack in our C code by specifying “**\$1**” for the first term on the right-hand side of the production, “**\$2**” for the second, and so on. “**\$\$**” designates the top of the stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack, and pushes back a single sum. As a consequence the parse and value stacks remain synchronized.

1  
4

Numeric values are initially entered on the stack when we reduce from **INTEGER** to **expr**. After **INTEGER** is shifted to the stack we apply the rule

**expr: INTEGER            { \$\$ = \$1; }**

The **INTEGER** token is popped off the parse stack followed by a push of **expr**. For the value stack we pop the integer value off the stack and then push it back on again. In other words we do nothing. In fact this is the default action and need not be specified. Finally, when a newline is encountered, the value associated with **expr** is printed.

In the event of syntax errors yacc calls the user-supplied function **yyerror**. If you need to modify the interface to **yyerror** then alter the canned file that yacc includes to fit your needs. The last function in our yacc specification is **main** ... in case you were wondering where it was. This example still has an ambiguous grammar. Although yacc will issue shift-reduce warnings it will still process the grammar using shift as the default operation.