LOADER AND LINKER

Ques 1) What is loading and linking?
Or
Define terms:
i) Loading
ii) Relocation
iii) Linking

Ans: Loading and Linking
The Source Program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution. This conversion either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies addresses in primary memory where these items are to be loaded for execution. This contains the following three processes, and they are,

1) **Loading:** It allocates memory location and brings the object program into memory for execution - **(Loader)**
2) **Linking:** It combines two or more separate object programs and supplies the information needed to allow references between them - **(Linker)**
3) **Relocation:** It modifies the object program so that it can be loaded at an address different from the location originally specified - **(Linking Loader)**

Ques 2) What is loader? List the various steps of loader. Also list the various types of loaders.

Ans: Loader
A loader is a utility of an operating system. It copies programs from a storage device to a computer's main memory, where the program can then be executed. Loader performs the loading function. It is responsible for initiating the execution of the process.

Various Steps a Loader Performs
The various steps required for working of loader are as follows:

Step 1: Read executable file's header to determine the size of text and data segments.

Step 2: Create a new address space for the program.

Step 3: Copies instructions and data into address space.

Step 4: Copies arguments passed to the program on the stack.

Step 5: Initializes the machine registers including the stack pointer.

Step 6: Jumps to a start-up routine that copies the program's arguments from the stack to registers and calls the program's main routine.

Types of Loader
1) Assemble-and-Go Loader
2) Relocating Loader (Relative Loader)
3) Absolute Loader (Bootstrap Loader)
4) Direct Linking Loader

Ques 3) What are the basic functions of a loader? (2020[03]

Ans: Basic Loader Functions
A loader is a system program that performs the loading function. It brings object program into memory and starts its execution. The role of loader is as shown in the figure **4.1**:

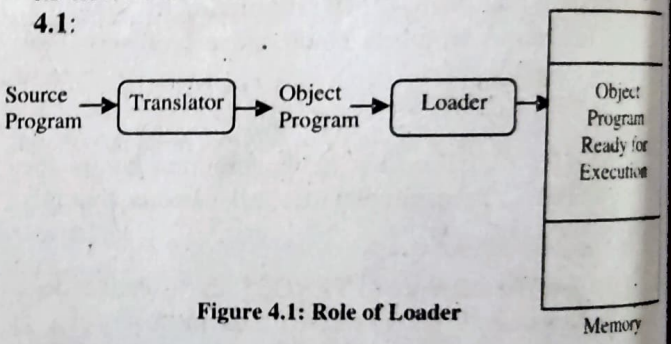

Figure 4.1: Role of Loader

Translator may be assembler/compiler, which generates the object program and later loaded to the memory by the loader for execution.

In **figure 4.2** the translator is specifically an assembler which generates the object loaded, which becomes input to the loader:

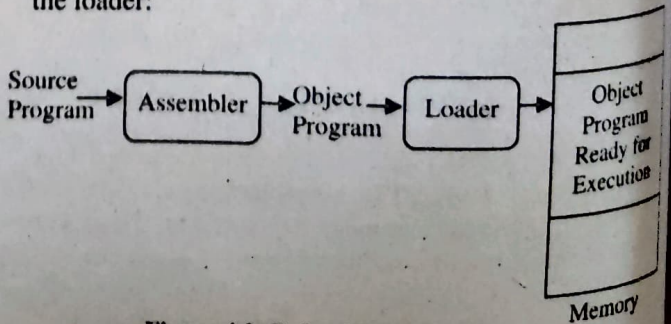

Figure 4.2: Role of Loader with

Figure 4.3 shows the role of both loader and linker:

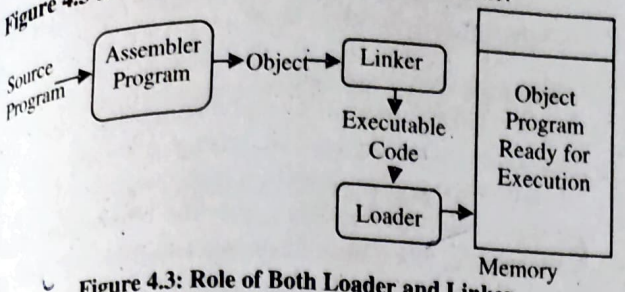

Figure 4.3: Role of Both Loader and Linker

Ques 4) Discuss assemble-and-go loader with their advantages and disadvantages.

Ans: Assemble-and-Go Loader

A assemble-and-go loader is one in which the assembler itself does the process of compiling and then places the assembled instructions in the designated memory locations. The assembling process is done first and then the assembler causes a transfer to the first instruction of the program. This loading scheme is also known as Assembler-and-Go. The loader here has just one instruction that directs the loading of the newly assembled code into the memory. **For example,** WATFOR FORTRAN compiler

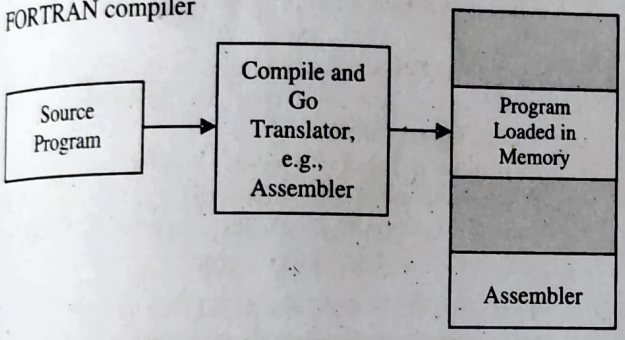

Figure 4.4: Compile and Go Loader

Advantages of Assemble-and-go loader

1) It is relatively easy to implement.
2) Involves no extra procedures.

Disadvantages of Assemble-and-go loader

1) A portion of memory is wasted.
2) It is necessary to re-translate the user's program every time it is run.
3) It is very difficult to handle multiple segments.
4) It is very difficult to produce orderly module programs.

Ques 5) Discuss relocating loader with their advantages and disadvantages.

Ans: Relocating Loader (Relative Loader)

The heart of a linker or loader's actions is **relocation** and **code modification**. The process of modifying the addresses used in the address dependent instructions of a program such that the program can execute from any allocated area of the memory is called **Program Relocation**. Loaders performing such a task is called **relocating loader**. As part of the loading process, the loader modifies the object code to reflect the actual addresses assigned. Relocating loaders are versatile when compared to the absolute loaders, when it comes to re-assembling routines every time when a single routine is changed.

Advantages of Relocating Loaders

1) Program can be loaded and executed in different parts of memory irrespective of the address.
2) Relocation is done by the usage of relocation bits.
3) Allocation is done by the availability of program length.
4) Linking is also done, as a part of the loading process itself.
5) Loader performs all the four functions given below:
   i) Memory allocation,
   ii) Subroutine linkages,
   iii) Relocation, and
   iv) Loading of code into the memory.

Disadvantages of Relocating Loaders

1) The complexity of the loader increases with the program size and capability to handle larger number of subroutines.
2) There is an increased overhead on memory as there is increase in number of subroutines.

Ques 6) Discuss bootstrap loader (absolute loader) with their advantages and disadvantages.

Ans: Absolute Loader (Bootstrap Loader)

The absolute loader is the simplest of all the loaders.

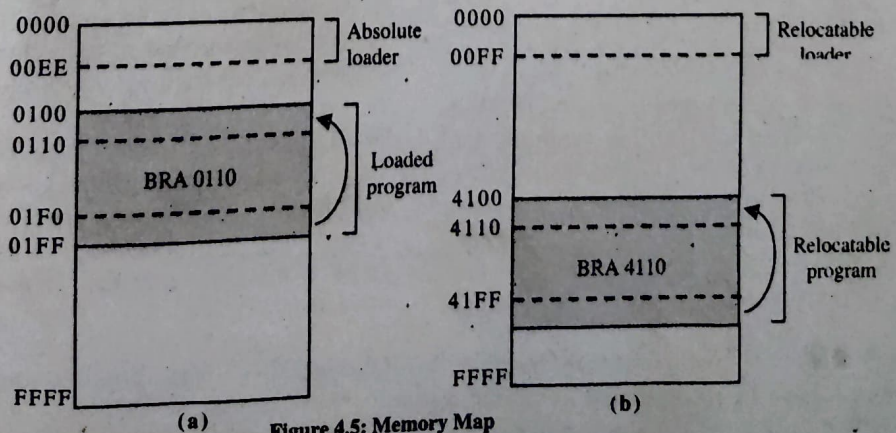

Figure 4.5: Memory Map

It can read a machine language program from the specified backup storage and place it in memory starting from a pre-determined address (**figure 4.5**). The machine language program so loaded will work correctly only if it is loaded starting from the specified address. Absolute type of loader is impractical, as there are lots of complications involved in loading the program. "Bootstrap loader" is an example of absolute loader.

Advantages of Absolute Loader

1) The absolute loader simply performs input and output operation to load a program into the main memory.
2) An absolute loader is coded in very few machine instructions.
3) Program is stored in the library in their ready-to-execute form. Such a library is called a Phase Library.

Disadvantages of Absolute Loader

1) The programmer must explicitly specify the assembler the memory where the program is to be loaded.
2) Handling multiple subroutines become difficult since the programmer must specify the address of the routines wherever they are referenced to perform subroutine linkages.
3) When dealing with lots of subroutines the manual shuffling and re-shuffling of memory address references in the routines become tedious and complex.

Ques 7) Given an idle computer with no program in memory, how do we get things started? (2020[03])

Ans: In this situation, with the machine empty and idle, there is no need for program relocation. We can simply specify the absolute address for whatever program is first loaded. Most often, this program will be the operating system, which occupies a predefined location in memory. This means that we need some means of accomplishing the functions of an absolute loader.

Some early computers required the operator to enter into memory the object code for an absolute loader, using switches on the computer console. However, this process is much too inconvenient and error-prone to be a good solution to the program.

On some computers, an absolute loader program is permanently resident in a read-only memory (ROM). When some hardware signal occurs (**for example,** the operator pressing a "system start" switch), the machine begins to execute this ROM program. On some computers, the program is executed directly in the ROM; on others the program is copied from ROM to main memory and executed there.

However, some machines do not have such read-only storage. In addition, it can be inconvenient to change a ROM program if modifications in the absolute loader are required.

Ques 8) Discuss direct linking loader with their advantages and disadvantages.

Ans: Direct Linking Loader

A direct linking loader is a general relocatable loader, and is perhaps the most popular scheme presently used. It allows programming multiple procedure segments and multiple data segment. This gives the programmer more convenience, since there is no need to reference data or instructions present in other segment. This provides flexible reference and accessing ability, while at the same time allowing independent accessing ability of the program. The assembler gives the loader the following information:

1) The length of the segment.
2) A list of all symbols in the segment that may be referenced by other segments and their relative locations within the segment.
3) A list of all symbols that are not defined in the segment but referenced in the segment.
4) Conveys the entry point of the program which is to know where the loader is to transfer the control and when the instructions are loaded.

The assembler does not provide its own symbol table to the loader. The assembler provides following types of cards in the object (deck).

1) **External Symbol Directory (ESD):** It provides information about all symbols defined and used in the source program. It is basically a symbol dictionary or look-up. This is the symbol table.
2) **Text (TXT):** It contains the actual translated object code, non-relocated data, initial values and so on.
3) **Relocatable and Linkage Directory (RLD):** It contains information about address sensitive instructions in the source program. They need to be relocated as performed in relocating loader.
4) **End (END):** It signifies the end of the object deck. This is the last card in an object deck.

Advantages of Direct Linking Loader

1) It provides the programmer with multiple procedure segments & multiple data segments.
2) It provides endorsement referencing & accessing ability.
3) It allows independent translations of programs along with above benefits.

Disadvantages of Direct Linking Loader

The linking and loading need to be postponed until the execution. During the execution if at all any subroutine is needed then the process of execution needs to be suspended until the required subroutine gets loaded in the main memory.

Ques 9) Discuss the design of absolute loader.
Or
Give the algorithm for an absolute loader. (2017 [03])
Or

Design an algorithm for an absolute loader. **(2019[03])**

Ans: Design of an Absolute Loader

The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program. The role of absolute loader is as shown in the **figure 4.6**:

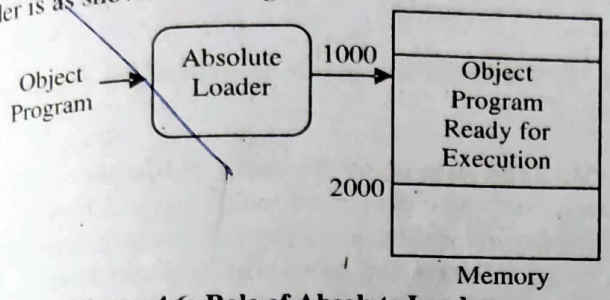

Figure 4.6: Role of Absolute Loader

The **advantage** of absolute loader is simple and efficient. But the **disadvantages** are, the need for programmer to specify the actual address, and, difficult to use subroutine libraries.

The algorithm for this type of loader is given below. The object program and, the object program loaded into memory by the absolute loader are also shown. Each byte of assembled code is given using its hexadecimal representation in character form which is easy to read by human beings.

Each byte of object code is stored as a single byte. Most machine store object programs in a binary form, and it is necessary that file and device conventions do not cause some of the program bytes to be interpreted as control characters.

Algorithm for an Absolute Loader

```
begin
  read Header record
  verify program name and length
  read first Text record
  while record type ≠ 'E' do
    begin
    {if object code is in character form, convert into internal representation} move object code to specified location in
    memory read next object program record
    end
    jump to address specified in End record
    end
```

One byte character

```
H␣COPY  ␣001000␣00107A
T␣001000␣1E␣141033␣482039␣001036␣281030␣301015␣482061␣3C1003␣00102A␣0C1039␣00102D
T␣00101E␣15␣0C1036␣482061␣081033␣4C0000␣454F46␣000003␣000000
T␣002039␣1E␣041030␣001030␣E0205D␣30203F␣D8205D␣281030␣302057␣549039␣2C205E␣38203F
T␣002057␣1C␣101036␣4C0000␣F1␣001000␣041030␣E02079␣302064␣509039␣DC2079␣2C1036
T␣002073␣07␣382064␣4C0000␣05
E␣001000
```

(a) Object Program

Memory address	Contents			
0000	x x x x x x x x	x x x x x x x x	x x x x x x x x	x x x x x x x x
0010	x x x x x x x x	x x x x x x x x	x x x x x x x x	x x x x x x x x
⋮	⋮	⋮	⋮	⋮
0FF0	x x x x x x x x	x x x x x x x x	x x x x x x x x	x x x x x x x x
1000	1 4 1 0 3 3 4 8	2 0 3 9 0 0 1 0	3 6 2 8 1 0 3 0	3 0 1 0 1 5 4 8
1010	2 0 6 1 3 C 1 0	0 3 0 0 1 0 2 A	0 C 1 0 3 9 0 0	1 0 2 D 0 0 C 1 0
1020	3 6 4 8 2 0 6 1	0 8 1 0 3 3 4 C	0 0 0 0 4 5 4 F	4 6 0 0 0 0 0 3
1030	0 0 0 0 0 0 x x	x x x x x x x x	x x x x x x x x	x x x x x x x x
⋮	⋮	⋮	⋮	⋮
2030	x x x x x x x x	x x x x x x x x	x x 0 4 1 0 3 0	0 0 1 0 3 0 E 0
2040	2 0 5 D 3 0 2 0	3 F D 8 2 0 5 D	2 8 1 0 3 0 3 0	2 0 5 7 5 4 9 0
2050	3 9 2 C 2 0 5 E	3 8 2 0 3 F 1 0	1 0 3 6 4 C 0 0	0 0 F 1 0 0 1 0
2060	0 0 0 4 1 0 3 0	E 0 2 0 7 9 3 0	2 0 6 4 5 0 9 0	3 9 D C 2 0 7 9
2070	2 C 1 0 3 6 3 8	2 0 6 4 4 C 0 0	0 0 0 5 x x x x	x x x x x x x x
2080	x x x x x x x x	x x x x x x x x	x x x x x x x x	x x x x x x x x
⋮	⋮	⋮	⋮	⋮

Half-byte

← COPY

(b) Program Loaded in Memory

Figure 4.7: Loading of an Absolute Program

The **Header record** is first checked. Then, each **Text record** is read to memory. When the **End record** is encountered, the loader jumps to the specified address.

Ques 10) Define the object program format of absolute loader.

Ans: Object Program Format
In our object program, each byte of assembled code is given using its hexadecimal representation in **character form.**

For example, the opcode for **STL** instruction would be represented by the pair of characters "1" and "4".

1) When they are read by the loader, they occupy two bytes of memory and must be stored in a singly byte with hexadecimal value 14.
2) Each pair of bytes from the object program record must be packed together into one byte during loading.

This method of representing an object program is inefficient in terms of **space** and **execution time**.

Therefore, most machines store object program in a **binary form** – Each byte of object code is stored as a single byte in the object program. The file and device conventions should not cause some of the object program bytes to be interpreted as control character.

For example, indicating the end of a record with a byte containing hexadecimal 00 would clearly be unsuitable for use with a binary object program. Object program stored in binary form do not lend themselves well to printing or to reading by human beings. Therefore, we continue to use character representations of object programs in this course.

Ques 11) Discuss the issues related with absolute loader.

Ans: Issues of Absolute Loaders
1) On a larger and more advanced machine, one does not know in advance where a program will be loaded.
2) Efficient sharing of the machine requires that one writes relocatable programs instead of absolute one.
3) Write absolute programs make it difficult to use subroutine libraries efficiently.
   i) Most such libraries contain many more subroutines than will be used by any one program.
   ii) To make efficient use of memory, it is important to be able to select and load exactly those routines that are needed.

Ques 12) Describe the machine dependent features of loader.

Ans: Machine Dependent Features of Loader
Features of machine dependent loader are as follows:
1) Program relocation is an indirect consequence of the change to larger and more powerful computers – The way relocation is implemented in a loader is also dependent upon machine characteristics.
2) Linking is not a machine-dependent function, but it has the same implementation techniques for loaders. The process of linking usually involves relocation of some of the routines being linked together.

Ques 13) What is relocation?

Ans: Relocation
The concept of program relocation is, the execution of the object program using any part of the available and sufficient memory. The object program is loaded into memory wherever there is room for it. The actual starting address of the object program is not known until load time.

Relocation provides the efficient sharing of the machine with larger memory and when several independent programs are to be run together.

It also supports the use of subroutine libraries efficiently. Loaders that allow for program relocation are called relocating loaders or relative loaders.

Ques 14) Explain various methods of specifying relocation.
Or
With the help of an Algorithm, explain how relocation is done using modification bits and direct addressing with relocation bits.
Or
What is the use of Bitmask in program relocation? **(2018[03])**

Ans: Methods for Specifying Relocation
Two methods for specifying relocation in object programs:
1) **Use Modification Records:** A Modification record is used to describe each part of the object code that must be changed when the program is relocated.

Figure 4.8 shows a SIC/XE program to illustrate this first method of specifying relocation:

The only portions of the assembled program that contain actual addresses are the extended format instructions on lines **15, 35,** and **65.** Thus these are the only items whose values are affected by relocation.

Figure 4.9 displays the object program corresponding to the source in **figure 4.8:**

Line Number	Machine Address	Label	Instruction	Operand	Object Code
5	0000	COPY	START	0	Relocatable program
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
20	000A		LDA	LENGTH	032026
25	000D		COMP	#0	290000
30	0010		JEQ	ENDFIL	332007
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
45	001A	ENDFIL	LDA	EOF	032010
50	001D		STA	BUFFER	0F2016
55	0020		LDA	#3	010003
60	0023		STA	LENGTH	0F200D
65	0026		+JSUB	WRREC	4B10105D
70	002A		J	@RETADR	3E2003
80	002D	EOF	BYTE	C'EOF'	454F46
95	0030	RETADR	RESW	1	
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
125	1036	RDREC	CLEAR	X	B410
130	1038		CLEAR	A	B400
132	103A		CLEAR	S	B440
133	103C		+LDT	#4096	75101000
135	1040	RLOOP	TD	INPUT	E32019
140	1043		JEQ	RLOOP	332FFA
145	1046		RD	INPUT	DB2013
150	1049		COMPR	A, S	A004
155	104B		JEQ	EXIT	332008
160	104E		STCH	BUFFER, X	57C003
165	1051		TIXR	T	B850
170	1053		JLT	RLOOP	3B2FEA
175	1056	EXIT	STX	LENGTH	134000
180	1059		RSUB		4F0000
185	105C	INPUT	BYTE	X'F1'	F1
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
210	105D	WRREC	CLEAR	X	B410
212	105F		LDT	LENGTH	774000
215	1062	WLOOP	TD	OUTPUT	E32011
220	1065		JEQ	WLOOP	332FFA
225	1068		LDCH	BUFFER, X	53C003
230	106B		WD	OUTPUT	DF2008
235	106E		TIXR	T	B850
240	1070		JLT	WLOOP	3B2FEF
245	1073		RSUB		4F0000
250	1076	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

Use Modification records

Figure 4.8

```
H,COPY  ,000000,001077
T,000000,1D,17202D,69202D,4B101036,032026,290000,332007,4B10105D,3F2FEC,032010
T,00001D,13,0F2016,010003,0F200D,4B10105D,3E2003,454F46
T,001036,1D,B410,B400,B440,75101000,E32019,332FFA,DB2013,A004,332008,57C003,B850
T,001053,1D,3B2FEA,134000,4F0000,F1,B410,774000,E32011,332FFA,53C003,DF2008,B85
T,001070,07,3B2FEF,4F0000,05
M,000007,05+COPY
M,000014,05+COPY
M,000027,05+COPY
E,000000
```

Modification records for + JSUB

Figure 4.9 object program corresponding to the source in figure 4.8

Each Modification record specifies the starting address and length of the field whose value is to be altered. It also describes the modification to be performed. In this example, all modifications add the value of the symbol COPY, which represents the starting address of the program (**figure 4.10**).

SIC/XE Relocation Loader Algorithm

```
begin
  get PROGADDR from operating system
  while not end of input do
    begin
      read next record
      while record type ≠ 'E' do
        begin
          read next input record
          while record type = 'T' then
            begin
              move object code from record to location
                  ADDR + specified address
            end
          while record type = 'M'
              add PROGADDR at the location PROGADDR +
                  specified address
        end
    end
end
```

Figure 4.10: SIC/XE Relocation Loader Algorithm

The Modification record scheme is a convenient means for specifying program relocation; however, it is not well-suited for use with all machine architectures. **For example,** consider the program in **figure 4.11**:

Line Number	Machine Address	Label	Instruction	Operand	Object Code
5	1000	COPY	START	1000 → Starting address	
10	1000	FIRST	STL	RETADR	141033
15	1003	CLOOP	JSUB	RDREC	482039
20	1006		LDA	LENGTH	001036
25	1009		COMP	ZERO	281030
30	100C		JEQ	ENDFIL	301015
35	100F		JSUB	WRREC	482061
40	1012		J	CLOOP	3C1003
45	1015	ENDFIL	LDA	EOF	00102A
50	1018		STA	BUFFER	0C1039
55	101B		LDA	THREE	00102D
60	101E		STA	LENGTH	0C1036
65	1021		JSUB	WRREC	482061
70	1024		LDL	RETADR	081033
75	1027		RSUB	PC ← (L)	4C0000
80	102A	EOF	BYTE	C'EOF'	454F46
85	102D	THREE	WORD	3	000003
90	1030	ZERO	WORD	0	000000
95	1033	RETADR	RESW	1	
100	1036	LENGTH	RESW	1	
105	1039	BUFFER	RESB	4096	
110					

```
115                 .       SUBROUTINE TO READ RECORD INTO BUFFER
120                 .
125  2039   RDREC   LDX     ZERO        041030
130  203C           LDA     ZERO        001030
135  203F   RLOOP   TD      INPUT       E0205D
140  2042           JEQ     RLOOP       30203F
145  2045           RD      INPUT       D8205D
150  2048           COMP    ZERO        281030
155  204B           JEQ     EXIT        302057
160  204E           STCH    BUFFER, X   549039
165  2051           TIX     MAXLEN      2C205E
170  2054           J/T     RLOOP       38203F
175  2057   EXIT    STX     LENGTH      101036
180  205A           RSUB                4C0000
185  205D   INPUT   BYTE    X'F1'       F1
Begin a new
Text record
190  205E   MAXLE   WORD    4096        001000
            N
195                 .
200                 .       SUBROUTINE TO WRITE RECORD FROM BUFFER
205                 .
210  2061   WRREC   LDX     ZERO        041030
215  2064   WLOOP   TD      OUTPUT      E02079
220  2067           JEQ     WLOOP       302064
225  206A           LDCH    BUFFER, X   509039
230  206D           WD      OUTPUT      DC2079
235  2070           TIX     LENGTH      2C1036
240  2073           JLT     WLOOP       382064
245  2076           RSUB                4C0000
250  2079   OUTPUT  BYTE    X'05'       05
255                 END     FIRST
```

Figure 4.11

This is a relocatable program written for the standard version of SIC. The important **difference** between this example and the one in **figure 4.8** is that the standard SIC machine does not use relative addressing. In this program the addresses in all the instructions except RSUB must be modified when the program is relocated. This would require 31 Modification records, which results in an object program more than twice as large as the one in **figure 4.9**.

2) **Use Direct Addressing with Relocation Bits:** On a machine that primarily uses direct addressing and has a fixed instruction format, it is often more efficient to specify relocation using a different technique. **Figure 4.12** show this method applied to SIC program example:

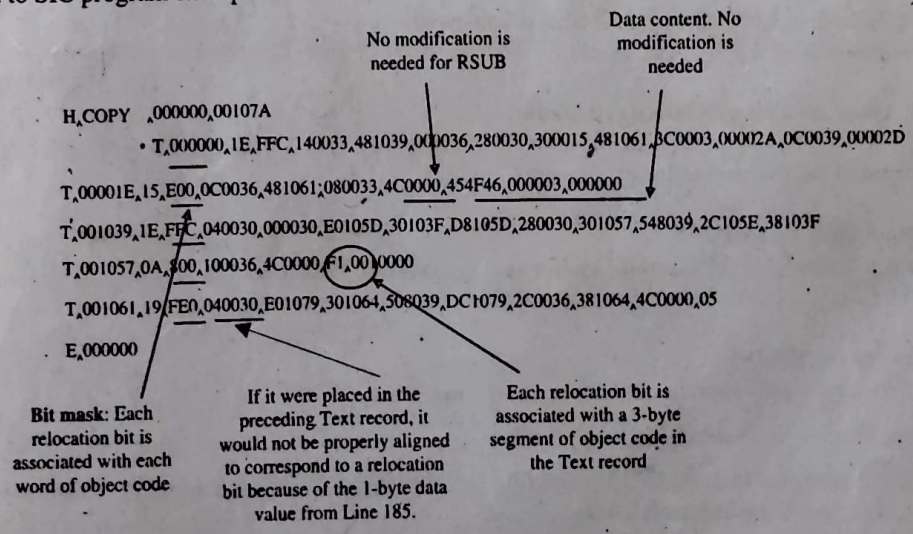

Figure 4.12: Object Program with Relocation Bits

There are no Modification records. The Text records are the same as before except that there is a relocation bit associated with each word of object code. Since all SIC instructions occupy one word, this means that there is one relocation bit for each possible instruction. The relocation bits are gathered together into a bit mask following the

length indicator in each Text record. In **figure 4.12** this mask is represented (in character form) as three hexadecimal digits. These characters are underlined for easier identification in the **figure 4.12**.

If the relocation bit corresponding to a word of **object code is set to 1**, the program's starting address is to be added to this word when the program is relocated. A **bit value of 0** indicates that no modification is necessary.

If a Text record contains fewer than 12 words of object code, the bits corresponding to unused words are set to 0. Thus the bit mask FFC (representing the bit string 111111111100) in the first Text record specifies that all 10 words of object code are to be modified during relocation.

These words contain the instructions corresponding to lines 10 through 55 in **figure 4.11**.

SIC Relocation Loader Algorithm

SIC relocation loader algorithm is shown in **figure 4.13**:

```
begin
  get PROGADDR from operating system
  while not end of input do
    begin
      read next record
      while record type ≠ 'E' do
      while record type = 'T'
        begin
          get length = second data
          mask bits(M) as third data
            For(i = 0, i < length, i++)
                  if Mi = 1 then
                    add PROGADDR at the location PROGADDR + specified address
                  else
                    move object code from record to location
        PROGADDR + specified address read next record
        end
    end
end
```

Figure 4.13: SIC Relocation Loader Algorithm

Note:
1: Program's starting address needs to be added to this word.
0: No need to be added.

Ques 15) Write a note on a simple bootstrap loader. **Or**

Write the algorithm of bootstrap loader.

Ans: A Simple Bootstrap Loader

Bootstrap loader is a special type of **absolute loader** that is executed when a computer is first turned on or restarted. This bootstrap loads the first program to be run by the computer – usually by operating systems.

The bootstrap loader for SIC/XE:

1) The bootstrap begins at address 0 in the memory of the machine.
2) It loads the operating system starting at address 80.
3) Because this loader is used in a unique situation (the initial program load or the system), the program to be loaded can be represented in a very simple format:
   i) Each byte of object code to be loaded is represented on device **F1** as two hexadecimal digits. (No Header record, End record, or control info.)
   ii) The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80.
   iii) After loading, the bootstrap jumps to address 80 to execute loaded program.

```
BOOT    START   0       BOOTSTRAP   LOADER  FOR     SIC/XE
.   THIS BOOTSTRAP READS OBJECT CODE FROM DEVICE F1 AND ENTERS IT
.   INTO MEMORY STARTING AT ADDRESS 80 (HEXADECIMAL). AFTER ALL OF
.   THE CODE FROM DEVF1 HAS BEEN SEEN ENTERED INTO MEMORY, THE
.   BOOTSTRAP EXECUTES A JUMP TO ADDRESS 80 TO BEGIN EXECUTION OF
.   THE PROGRAM JUST LOADED. REGISTER X CONTAINS THE NEXT ADDRESS
.   TO BE LOADED.
        CLEAR   A       CLEAR REGISTER A TO ZERO
        LDX     #128    INITIALIZE REGISTER X TO HEX 80
LOOP    JSUB    GETC    READ HEX DIGIT FROM PROGRAM BEING LOADED
        RMO     A, S    SAVE IN REGISTER S                    S ← (A)
        SHIFTL  S, 4    MOVE TO HIGH-ORDER 4 BITS OF BYTE
        JSUB    GETC    GET NEXT HEX DIGIT
        ADDR    S, A    COMBINE DIGITS TO FORM ONE BYTE       A ← (S)+(A)
        STCH    0, X    STORE AT ADDRESS IN REGISTER X
        TIXR    X, X    ADD 1 TO MEMORY ADDRESS BEING LOADED  X ← (X)+1;
        J       LOOP    LOOP UNTIL END OF INPUT IS REACHED    (X):(X)
.
.   SUBROUTINE TO READ ONE CHARACTER FROM INPUT DEVICE AND
.   CONVERT IT FROM ASCII CODE TO HEXADECIMAL DIGIT VALUE. THE
.   CONVERTED DIGIT VALUE IS RETURNED IN REGISTER A. WHEN AN
.   END-OF-FILE IS READ, CONTROL IS TRANSFERRED TO THE STARTING
.   ADDRESS (HEX 80).
.
GETC    TD      INPUT   TEST INPUT DEVICE
        JEQ     GETC    LOOP UNTIL READY
        RD      INPUT   READ CHARACTER
        COMP    #4      IF CHARACTER IS HEX 04 (END OF FILE),
        JEQ     #80         JUMP TO START OF PROGRAM JUST LOADED
        COMP    48      COMPARE TO HEX 30 (CHARACTER '0')
        JLT     #GETC   SKIP CHARACTERS LESS THAN '0'
        SUB     48      SUBTRACT HEX 30 FROM ASCII CODE
        COMP    #10     IF RESULT IS LESS THAN 10, CONVERSION IS
        JLT     #RETURN     COMPLETE. OTHERWISE, SUBTRACT 7 MORE
        SUB     #7          (FOR HEX DIGITS 'A' THROUGH 'F')
        RSUB            RETURN TO CALLER
        BYTE    #X'F1'  CODE FOR INPUT DEVICE
RETURN  END     LOOP
INPUT                   X 'F1'
                        LOOP
```

Figure 4.14: Bootstrap Loader for SIC/XE

Algorithm for the Bootstrap Loader

The algorithm for the bootstrap loader is as follows:

Begin

X = 0x80 (the address of the next memory location to be loaded.)

Loop

A ← GETC (and convert it from the ASCII character code to the value of the hexadecimal digit) save the value in the high-order 4 bits of S

A ← GETC

combine the value to form one byte A ← (A+S) store the value (in A) to the address in register X

X ← X + 1.

End

It uses a subroutine GETC, which is

GETC A ← read one character
if A = 0x04 then jump to 0x80
if A < 48 then GETC
A ← A-48 (0x30)
if A < 10 then return
A ← A-7 return

Ques 16) Discuss the meaning of program linking with the help of an example. Or Define external references and external definitions. Also implement them.

Ans: Program Linking

The goal of program linking is to resolve the problems with external references (EXTREF) and external definitions (EXTDEF) from different control sections.

1) **EXTDEF (External Definition):** The EXTDEF statement in a control section names symbols, called external symbols, that are defined in this (present) control section and may be used by other sections.

For example,
EXTDEF BUFFER, BUFFEND, LENGTH
EXTDEF LISTA, ENDA

2) **EXTREF (External Reference):** The EXTREF statement names symbols used in this (present) control section and are defined elsewhere. **For example,** EXTREF RDREC, WRREC
EXTREF LISTB, ENDB, LISTC, ENDC

Implementation of EXTDEF and EXTREF

The assembler must include information in the object program that will cause the loader to insert proper values where they are required – in the form of Define record (D) and, Refer record(R).

1) **Define Record:** The format of the Define record (D) along with examples is as shown here:
Col. 1 D
Col. 2-7 Name of external symbol defined in this control section
Col. 8-13Relative address within this control section (hexadecimal)
Col. 14-73 Repeat information in Col. 2-13 for other external symbols

Example Records

D LISTA	000040	ENDA	000054
D LISTB	000060	ENDB	000070

2) **Refer Record:** The format of the Refer record (R) along with examples is as shown here:
Col. 1 R
Col.2-7 Name of external symbol referred to in this control section
Col. 8-73 Name of other external reference symbols

Example Records

R LISTB	ENDB	LISTC	ENDC
R LISTA	ENDA	LISTC	ENDC
R LISTA	ENDA	LISTB	ENDB

Here are the three programs named as PROGA, PROGB and PROGC, which are separately assembled and each of which consists of a single control section. LISTA, ENDA in PROGA, LISTB, ENDB in PROGB and LISTC, ENDC in PROGC are external definitions in each of the control sections. Similarly LISTB, ENDB, LISTC, ENDC in PROGA, LISTA, ENDA, LISTC, ENDC in PROGB, and LISTA, ENDA, LISTB, ENDB in PROGC, are external references. These sample programs given here are used to illustrate linking and relocation. The **following figures** give the sample programs and their corresponding object programs, which contain D and R records along with other records.

Program Linking Example (PROGA)

```
0000  PROGA  START   0
              EXTDEF  LISTA, ENDA
              EXTREF  LISTB , ENDB, LISTC,
                      ENDC
              ..........
              ..........
0020  REF1    LDA     LISTA                03201D
0023  REF2    +LDT    LISTB+4              77100004
0027  REF3    LDX     #ENDA-LISTA          050014
0040  LISTA   EQU     *
0054  ENDA    EQU     *
0054  REF4    WORD    ENDA-LISTA+LISTC     000014
0057  REF5    WORD    ENDC-LISTC-10        FFFFF6
005A  REF6    WORD    ENDC-LISTC+LISTA-1   00003F
005D  REF7    WORD    ENDA-LISTA-(ENDB-    000014
                      LISTB)
0060  REF8    WORD    LISTB-LISTA          FFFFC0
              END     REF1
```

Program Linking Example (PROGB)

```
0000  PROGB  START   0
              EXTDEF  LISTB, ENDB
              EXTREF  LISTA , ENDA, LISTC,
                      ENDC
              ..........
              ..........
0036  REF1    +LDA    LISTA                03100000
003A  REF2    LDT     LISTB+4              772027
003D  REF3    +LDX    #ENDA-LISTA          05100000
0060  LISTB   EQU     *
0070  ENDB    EQU     *
0070  REF4    WORD    ENDA-LISTA+LISTC     000000
0073  REF5    WORD    ENDC-LISTC-10        FFFFF6
0076  REF6    WORD    ENDC-LISTC+LISTA-1   FFFFFF
0079  REF7    WORD    ENDA-LISTA-(ENDB-    FFFFF0
                      LISTB)
007C  REF8    WORD    LISTB-LISTA          000060
              END
```

Program Linking Example (PROGC)

```
0000  PROGC  START   0
              EXTDEF  LISTC,
                      ENDC
              EXTREF  LISTA.  ENDB,  LISTB,
                      ENDB
              ..........
              ..........
0018  REF1    +LDA    LISTA                03100000
001C  REF2    +LDT    LISTB+4              77100004
0020  REF3    +LDX    #ENDA-               05100000
                      LISTA
0030  LISTC   EQU     *
0042  ENDC    EQU     *
0042  REF4    WORD    ENDA-LISTA+LISTC     000030
0045  REF5    WORD    ENDC-LISTC-10        000008
0045  REF6    WORD    ENDC-LISTC+LISTA-1   000011
004B  REF7    WORD    ENDA-LISTA-(ENDB-LISTB)  000000
004E  REF8    WORD    LISTB-LISTA          000000
              END
```

Ques 17) Define the data structure for linking loader.

Or

What is ESTAB?

Or

Describe the data structure used for the linking loads algorithm. (2017 [2.5])

Or

Justify the need for having two passes in a linking loader. Illustrate the data structures used for a linking loader, showing how they are used in each pass. (2019[04])

Ans: Need for Two Passes in Linking Loader

The algorithm for a linking loader is considerably more complicated than the absolute loader algorithm. The input to such a loader consists of a set of object programs (i.e., control sections) that are to be linked together. It is possible (and common) for a control section to make an external reference to a symbol whose definition does not appear until later in this input stream.

In such a case the required linking operation cannot be performed until an address is assigned to the external symbol (that is, until the later control section is read). Thus a linking loader usually makes two passes over its input, just as an assembler does. In terms of general function, the two passes of a linking loader are quite similar to the two passes of an assembler:

1) Pass 1 assigns addresses to all external symbols
2) Pass 2 performs the actual loading, relocation and linking.

Data Structures for a Linking Loader

The algorithm for a linking loader is considerably more complicated than the absolute loader program. The concept given in the program linking section is used for developing the algorithm for linking loader. The modification records are used for relocation so that the linking and relocation functions are performed using the same mechanism. Linking loader uses two-passes logic:

Pass 1: Assign addresses to all external symbols.

Pass 2: Perform the actual loading, relocation, and linking.

ESTAB (External Symbol Table) is the main data structure for a linking loader:

1) It is to store the **name** and **address** of each external symbol.
2) It is similar to SYMTAB in the assembler.
3) It indicates in which control section the symbol is defined.
4) A **hash organisation** is typically used for this table.
5) Two variables are defined:
   i) **PROGADDR (Program Load Address):** Indicate the beginning address in memory where the linked program is to be loaded. Its value is supplied to loader by the operating system.
   ii) **CSADDR (Control Section Address):** Contain the starting address assigned to the control section currently being scanned by the loader.

For example, ESTAB (refer three programs PROGA, PROGB, and PROGC) given is as shown below. The ESTAB has four entries in it; they are name of the control section, the symbol appearing in the control section, its address and length of the control section.

Control Section	Symbol	Address	Length
PROGA		4000	63
	LISTA	4040	
	ENDA	4054	
PROGB		4063	7F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	51
	LISTC	4112	
	ENDC	4124	

Use of Data Structure in Pass 1

During the first pass, the loader is concerned only with Header and Define record types in the control sections. The beginning load address for the linked program (PROGADDR) is obtained from the operating System. This becomes the starting address (CSADDR) for the first control section in the input sequence. The control section name from the Header record is entered into ESTAB, with value given by CSADDR. All external symbols appearing in the Define record for the control section are also entered into ESTAB. Their addresses are obtained by adding the value specified in the Define record to CSADDR. When the End record is read, the control section length CSLTH (which was saved from the Header record) is added to CSADDR. This calculation gives the starting address for the next control section in sequence. At the end of Pass I, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each. Many loaders include as an option the ability to print a load map that shows these symbols and their addresses. This information is often useful in program debugging.

Use of Data Structure in Pass 2

Pass 2 of our loader performs the actual loading, relocation, and linking of the program. CSADDR is used in the same way it was in Pass 1– it always contains the actual starting address of the control section currently being loaded. As each Text record is read, the object code is moved to the specified address (plus the current value of CSADDR). When a Modification record is encountered, the symbol whose value is to be used for modification is looked up in ESTAB. This value is then added to or subtracted from the indicated location in memory.

Ques 18) Describe the program logic for pass 1 of linking loader. Also write its algorithm.

Or

Give the algorithm for pass 1 of the linking loader. (2020[04]) (2017 [2.5])

Or

Explain the algorithm for Pass 1 of Linking Loader. (2018 [06])

Ans: Program Logic for Pass 1

Pass 1 assign addresses to all external symbols. The variables and Data structures used during pass 1 are,

PROGADDR (Program Load Address) from OS, CSADDR (Control Section Address), CSLTH (Control Section Length) and ESTAB. The pass 1 processes the Define Record. In Pass 1, the loader is concerned only with **Header** and **Define** record types in the control sections:

1) **Initialisation**
   i) The beginning load address for the linked program (PROGADDR) is obtained from the operating system.
   ii) This becomes the starting address (CSADDR) for the first control section in the input sequence.
2) **Record Scanning**
   i) The control section name from the Header record is entered into ESTAB, with value given by CSADDR.
   ii) All external symbols appearing in the Define record for the control section are also entered into ESTAB.
   iii) External symbols' addresses are obtained by adding the value specified in the Define record to CSADDR. (Specified address + CSADDR).
   iv) When End record is read, the control section length CSLTH (obtained from Header record) is added to CSADDR to calculate the starting address for the next control section. (CSADDR = CSAADR + CSLTH)

At the end of Pass 1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each. Many loaders include as an option the ability to print a **load map** that shows these symbols and their addresses. This information is useful in program debugging.

Algorithm for Pass 1 of Linking Loader

The algorithm for Pass 1 of Linking Loader is given below:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR (for first control section)

while not end of input do //Each iteration processes one control section
    begin
        read next input record {Header record for control section}
        set CSLTH to control section length //Header record
        search ESTAB for control section name
        if found then
            set error flag {duplicate external symbol}
        else
            enter control section name into ESTAB with value CSADDR
        while record type ≠ 'E' do
            begin
                read next input record
                if record type = 'D' then    //Define record
                    for each symbol in the record do
                        begin
                        search ESTAB for symbol name
                            if found then
                                set error flag (duplicate external symbol)
                            else
                                enter symbol into ESTAB with value
                                    (CSADDR + indicated address)
                        end {for}
            end {while ≠ 'E'}    //reach End record
        add CSLTH to CSADDR {starting address for next control section}
    end {while not EOF}
end {Pass 1}
```

Ques 19) Describe the program logic for pass 2 of linking loader. Also write its algorithm.

Or

Give the algorithm for pass 2 of a linking loader.

(2019[04])

Ans: Program Logic for Pass 2

Pass 2 of linking loader perform the actual loading, relocation, and linking. It uses modification record and lookup the symbol in ESTAB to obtain its address. Finally it uses end record of a main program to obtain transfer address, which is a starting address needed for the execution of the program. The pass 2 process Text record and Modification record of the object programs:

1) CSADDR is used in the same way as it was in Pass 1 – It always contains the actual starting address of the control section currently being loaded.
2) As each **Text record** is read, the object code is moved to the specified address plus the current value of CSADDR. (specified address + CSADDR)
3) When a **Modification record** is encountered, the symbol whose value is to be used for modification is looked-up in ESTAB – This value is then added to or subtracted from the indicated location in memory.

The last step of Pass 2 is to transfer control to the loaded program to begin execution:

1) The **End record** for each control section may contain the address of the first instruction in that control section to be executed.
2) Two scenarios could be encountered:
   i) If more than one control section specifies a transfer address, the loader arbitrarily uses the last one encountered.
   ii) If no control section contains a transfer address, the loader uses the beginning of the linked program (i.e., PROGADDR) as the transfer point.
3) Normally, a transfer address would be placed in the **End record** for a **main program**, but not for a subroutine.

Algorithm for Pass 2 of Linking Loader

The algorithm for Pass 2 of Linking Loader is given below:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
    begin    //Each iteration processes one control section
        read next input record {Header record}
        set CSLTH to control section length
        while record type ≠ 'E' do
            begin
                read next input record
                if record type = 'T' then    //Text record
                  begin
                      {if object code is in character
                       form, convert into internal
                       representation}
                      move object code from record to
                     location
                        (CSADDR + specified
                     address)
                  end {if 'T'}
                else if record type = 'M' then
//Modification record
                      begin
                          search ESTAB for modifying
                         symbol name
                          if found then
                             add or subtract symbol value
                         at location
                                   {CSADDR + specified
                                  address}
                          else
                             set error flag (undefined
                         external symbol)
                      end {if 'M'}
            end {while ≠ 'E'}
        if an address is specified {in End record} then
      set EXECADDR to (CSADDR + specified address)
            //End record: transfer address is specified
        add CSLTH to CSADDR          //Move to next
      CS
    end {while not EOF}
    jump to location given by EXECADDR{start execution of
    loaded program}
end {Pass 2}
```

Ques 20) Describe the advanced method for external symbol.

Ans: Advanced Method for External Symbols

One can improve the efficiency of the linking loader. Also observe that, even though he/she has defined Refer record (R), he/she has not made use of it. The efficiency can be improved by the use of local searching instead of multiple searches of ESTAB for the same symbol. For implementing this one assigns a reference number to each external symbol in the Refer record:

1) One can assign a reference number to each external symbol referred to in a control section.
2) This reference number is used in Modification records.

For example:

i) Control section name with reference number 01.
ii) The other external reference symbols are assigned numbers as part of the **Refer record** for the control section.

3) The main **advantage** of this reference-number mechanism is that it avoids multiple searches of ESTAB for the same symbol during the loading of a control section:
   i) An external reference symbol can be looked up in ESTAB once for each control section that uses it.
   ii) The value for code modification can then be obtained by simply indexing into an array of these values.

The object programs for PROGA, PROGB and PROGC are shown below, with above modification to Refer record:

Advanced Method for External Symbols (PROGA)

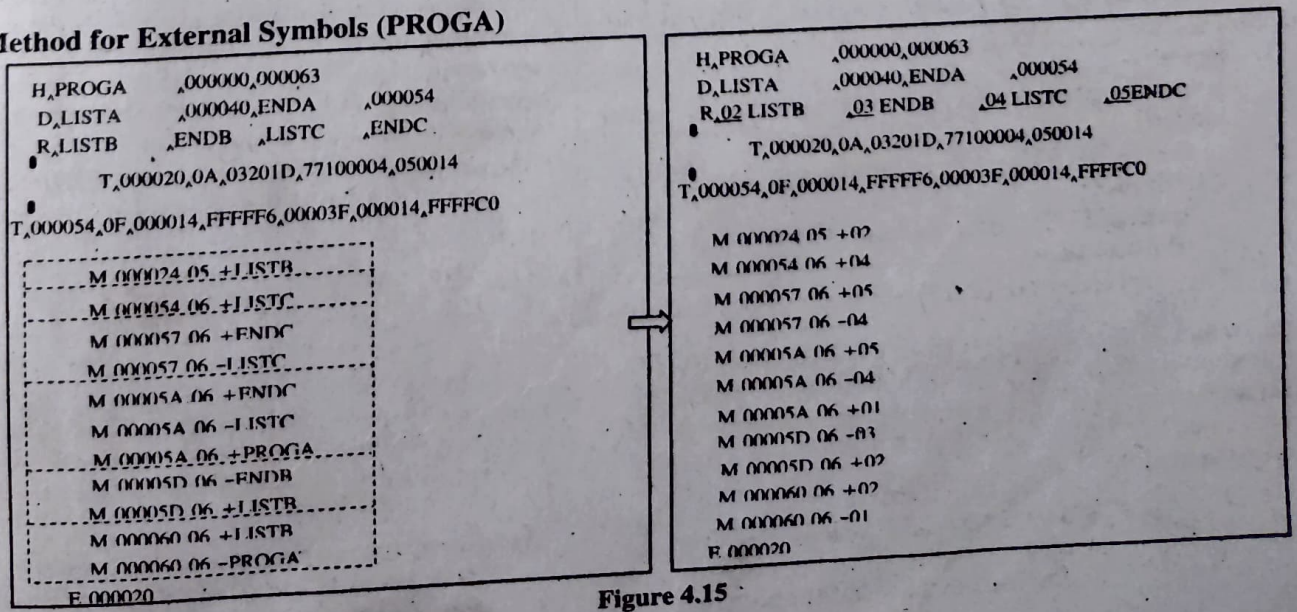

Figure 4.15

Advanced Method for External Symbols (PROGB)

```
H,PROGB  ,000000,00007F
D,LISTB  ,000060,ENDB  ,000070
R,LISTA  ,ENDA ,LISTC  ,ENDC
.
T,000036,0B,03100000,772027,05100000
.
T,000070,0F,000000,FFFFF6,FFFFFF,FFFFF0,000060
M,000037,05,+LISTA
M,00003E,05,+ENDA
M,00003E,05,-LISTA
M,000070,06,+ENDA
M,000070,06,-LISTA
M,000070,06,+LISTC
M,000073,06,+ENDC
M,000073,06,-LISTC
M,000076,06,+ENDC
M,000076,06,- LISTC
M,000076,06,+LISTA
M,000079,06,+ENDA
M,000079,06,-LISTA
M,00007C,06,+PROGB
M,00007C,06,-LISTA
E
```

⇨

```
H,PROGB  ,000000,00007F
D,LISTB  ,000060,ENDB  ,000070
R,02LISTA  ,03ENDA  ,04LISTC  ,05ENDC
.
T,000036,0B,03100000,772027,05100000
.
T,000070,0F,000000,FFFFF6,FFFFFF,FFFFF0,000060
M,000037,05,+02
M,00003E,05,+03
M,00003E,05,-02
M,000070,06,+03
M,000070,06,-02
M,000070,06,+04
M,000073,06,+05
M,000073,06,-04
M,000076,06,+05
M,000076,06,-04
M,000076,06,+02
M,000079,06,+03
M,00007C,09,-02
M,00007C,06,-01
M,00007C,06,-02
E
```

Figure 4.16

Advanced Method for External Symbols (PROGC)

```
H,PROGC  ,000000,000051
D,LISTC  ,000030,ENDC  ,000042
R,LISTA  ,ENDA ,LISTB  ,ENDB
.
T,000018,0C,03100000,77100004,05100000
.
T,000042,0F,000030,000008,000011,000000,
000000
M,000019,05,+LISTA
M,00001D,05,+LISTB
M,000021,05,+ENDA
M,000021,05,-LISTA
M,000042,06,+ENDA
M,000042,06,-LISTA
M,000042,06,+PROGC
M,000048,06,+LISTA
M,00004B,06,+ENDA
M,00004B,06,-LISTA
M,00004B,06,-ENDB
M,00004B,06,+LISTB
M,00004E,06,+LISTB
M,00004E,06,-LISTA
E
```

⇨

```
H,PROGC  ,000000,000051
D,LISTC  ,000030,ENDC  ,000042
         ,03ENDA  ,04LISTB  ,05ENDB
.
T,000018,0C,03100000,77100004,05100000
.
T,000042,0F,000030,000008,000011,000000,
000000
M,000019,05,+02
M,00001D,05,+04
M,000021,05,+03
M,000042,06,+03
M,000042,06,-02
M,000042,06,+01
M,000048,06,+02
M,00004B,06,+03
M,00004B,06,-02
M,00004B,06,-05
M,00004B,06,+04
M,00004E,06,+04
M,00004E,06,-02
E
```

Figure 4.17

Symbol and Addresses in PROGA, PROGB and PROGC are as **shown below**. These are the entries of ESTAB. The main **advantage** of reference number mechanism is that it avoids multiple searches of ESTAB for the same symbol during the loading of a control section.

PROGA

Reference No.	Symbol	Address
1	PROGA	4000
2	LISTB	40C3
3	ENDB	40D3
4	LISTC	4112
5	ENDC	4124

PROGB

Reference No.	Symbol	Address
1	PROGB	4063
2	LISTA	4040
3	ENDA	4054
4	LISTC	4112
5	ENDC	4124

PROGC

Reference No.	Symbol	Address
1	PROGC	4063
2	LISTA	4040
3	ENDA	4054
4	LISTB	40C3
5	ENDB	40D3

Ques 21) Discuss the machine independent features of loader. Also list the loader design options.

Or

Write notes on machine independent loader features.

(2017 [04])

Or

Write notes on the different loader design options.

(2017 [04])

Or

List and explain the different machine independent features of loaders.

(2018[03])

Or

Write short notes on the following:

i) Automatic Library Search

ii) Loader Options

Or

List and explain the different machine independent loader features.

(2019[04])

Or

Explain the concept of automatic library search.

(2020[03])

Ans: Machine Independent Loader Features

Some loader features are not directly related to machine architecture and design. Automatic Library Search and Loader Options are such Machine-independent Loader Features.

1) **Automatic Library Search:** This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded. The routines are automatically retrieved from a library as they are needed during linking. This allows programmer to use subroutines from one or more libraries. The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program and loaded. The loader searches the library or libraries specified for routines that contain the definitions of these symbols in the main program. Automatic library search allows programmer to use subroutines from one or more libraries.

2) **Loader Options:** Many loaders have a special command language that is used to specify options. Sometimes there is a separate input file to the loader that contains such control statements. Sometimes these same statements can also be embedded in the primary input stream between object programs. Loader options allow the user to specify options that modify the standard processing. The options may be specified in three different ways. They are:
   1) Specified using a command language,
   2) Specified as a part of job control language that is processed by the operating system, and
   3) Be specified using loader control statements in the source program.

Loader Design Options

There are some common alternatives for organising the loading functions, including relocation and linking. s

1) **Linking Loaders:** It Perform all linking and relocation at load time. The Other Alternatives are Linkage editors, which perform linking prior to load time

2) **Dynamic Linking:** In this linking function is performed at execution time

Ques 22) Write a note on linkage editors.

Or

What is linkage editor? Compare it with linking loader.

Or

Differentiate between linking loaders and linkage editors.

(2017, 2019 [03])

Ans: Linkage Editors

It performs linking prior to the load time. The linkage editors are found in many computing systems instead of linking loader and the concept of dynamic linking loads subprograms at the time of first call. A linkage editor performs linking and relocation, and the linked program is written on a file or library instead of loading it immediately into the memory.

It produces a linked version of the program which is written onto a file or library for execution. When the user is ready to run the linked program, a simple relocation loader is used to load the program into memory.

In contrast, a linking loader performs all the linking and relocation operations, including automatic library search, if specified and loads the linked program directly into the memory for execution. The **figure 4.18** illustrates the processing of linking loader and linkage editor:

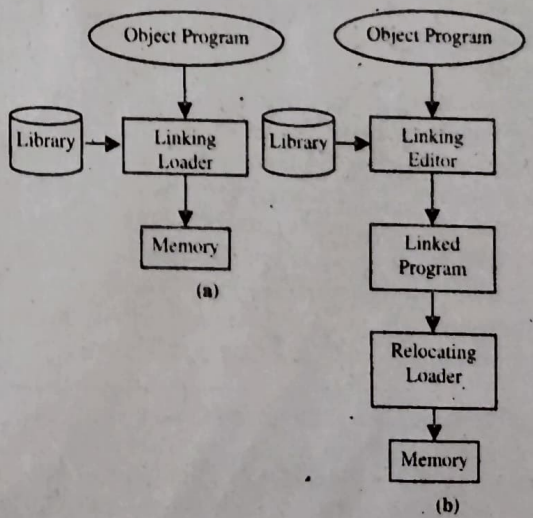

Figure 4.18: Processing of an Object Program using (a) Linking Loader and (b) Linkage Editor

If a program is executed many times without being re-assembled, the use of a linkage editor substantially reduces the overhead. Resolution of external references and library searching are performed only once. In contrast, a linking loader searches library and resolves external references

every time the program is executed. The linked program produced by the linkage editor is generally in a form that is suitable for processing by a Relocating Loader. All external references are resolved and relocation is indicated by certain mechanism such as Modification Records or a Bit Mask. Even though all the linking has been performed, the information concerning external references is often retained in the linked program. Therefore subsequent re-linking of the program is done to replace the control section and modify external references.

Comparison of Linking Loader and Linkage Editor

Linking Loader	Linkage Editor
The linking loader performs all the linking and relocation operations including automatic library search. It then loads the program directly into the memory for execution.	The linkage editor produces a linked version of the program. Such a linked version is also called as Load Module or Executable Image. This load module is generally written in file or library for later execution.
There is no need for relocating loader.	The relocating loader loads the load module into the memory.
The linking loader searches the libraries and resolves the external references every time the program is executed.	If the program is executed many times without being re-assembled then linkage editor is the best choice.
The loading requires two passes.	The loading is accomplished in one pass.

Ques 23) Illustrate the process of dynamic linking. (2020[06])

Or

Discuss the dynamic linking with their advantages.

Or

Write short note on Dynamic Linking. (2018[03])

Ans: Dynamic Linking

In this the linking function is performed at execution time. Dynamic linking allows a module to include only the information the system needs at load time or runtime. Dynamic linking differs from static linking, in which the linker copies a library function's code into each module that calls it, which is not done in the case of dynamically linked modules. If the program is completely linked before execution, the subroutines need to be loaded and linked every time the program is run. Dynamic linking provides the ability to load the routines only when they are in need. If subroutines are large or have any external references, saving in memory space and time is achieved.

Dynamic linking offers the ability to load the routines only when (and if) they are required. The actual loading and linking can be accomplished using operating system service request.

1) Linkage editors perform linking before the program is loaded for execution.
2) Linking loaders perform these same operations at load time.
3) Dynamic linking postpones the linking function until execution time. A subroutine is loaded and linked to the rest of the program when it is first called.

Dynamic Linking Advantage

1) The subroutines that diagnose errors may never need to be called at all. However, without using dynamic linking, these subroutines must be loaded and linked every time the program is run.
2) Using dynamic linking can save both space for storing the object program on disk and in memory, and time for loading the bigger object program
3) Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library, ex. run-time support routines for a high-level language like C. For example, a single copy of the standard C library can be loaded into memory.
4) All C programs currently in execution can be linked to this one copy, instead of linking a separate copy into each object program.
5) In an object-oriented system, dynamic linking is often used for references to software object.
6) This allows the implementation of the object and its method to be determined at run time.
7) The implementation can be changed at any time, without affecting the program that makes use of the object

Ques 24) Define the implementation of dynamic linking.

Ans: Dynamic Linking Implementation

A subroutine that is to be dynamically loaded must be called via an operating system service request. This method can also be thought of as a request to a part of the loader that is kept in memory during execution of the program. Instead of executing a JSUB instruction to an external symbol, the program makes a load-and-call service request to the OS. The parameter of this request is the symbolic name of the routine to be called. The OS examines its internal tables to determine whether the subroutine is already loaded. If needed, the subroutine is loaded from the library. Then control is passed from the OS to the subroutine being called. When the called subroutine completes its processing, it returns to its caller (operating system). The OS then returns control to the program that issues the request. After the subroutine is completed, the memory that was allocated to it may be released.

However, often this is not done immediately. If the subroutine is retained in memory, it can be used by later calls to the same subroutine without loading the same subroutine multiple times. Control can simply pass from the dynamic loader to the called routine directly. Figure 4.19 illustrates a method in which routines that are to be dynamically loaded must be called via an OS service request. Instead of executing a JSUB instruction referring to an external symbol, the program makes a load-and-call service request to OS. The parameter of this request is the symbolic name of the routine to be called (Figure 4.19(a)).

In figure 4.19(b), OS examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded from the specified user or system libraries. Control is then passed from OS to the routine being called (FIGURE 4.19(c)). When the called subroutine completes it processing, it returns to its caller (i.e., OS). OS then returns control to the program that issued the request (figure 4.19(d)).

If a subroutine is still in memory, a second call to it may not require another load operation. Control may simply be passed from the dynamic loader to the called routine (Figure 4.19(e)).

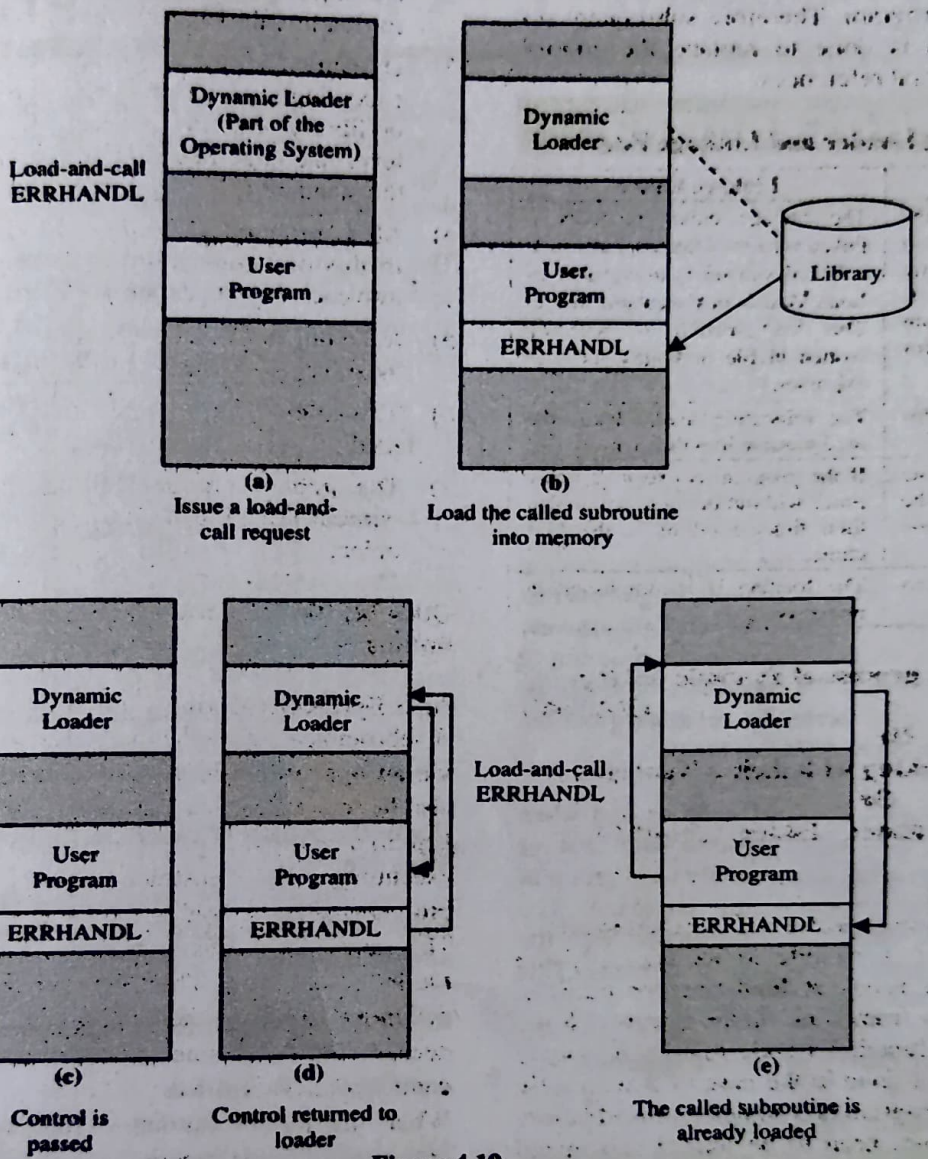

Figure 4.19