MODULE IV

- **Linker and Loader**
  - **Basic Loader functions**
    - **Design of absolute loader**
    - **Simple bootstrap Loader**
  - **Machine dependent loader features**
    - **Relocation**
    - **Program Linking**
      - **Algorithm and data structures of two pass Linking Loader**
  - **Machine independent loader features**
  - **Loader Design Options.**

- **Linker and Loader**
  - **Loading**: Brings the object program into memory for execution.
  - **Relocation**: Modifies the object program so that it can be loaded at an address different from the location originally specified.
  - **Linking**: Combines two or more separate object programs and supplies the information needed to allow references between them.
  - **Loader** is a system program that performs the loading function. Many loaders also support relocation and linking.
  - **Linker (linkage editor)** is a system program that performs the linking operations and need a separate loader to handle relocation and loading.
  - **Linking Loader** is a system program that having both linking and loading capabilities.

  - **Basic Loader Functions**
    - Bringing an object program into memory.
    - Starting its execution.

  - **Basic Loader Functions**
    - **Allocation**: Allocates the space for program in the memory, by calculating the size of the program. Allocation is done by the programmer.
    - **Linking**: It resolves the symbolic references (code/data) between the object modules by assigning all the user subroutine and library subroutine addresses. Linking is done by the programmer.
    - **Relocation**: There are some address dependent locations in the program, such address constants must be adjusted according to allocated space. Relocation is done by the assembler.
    - **Loading**: Places all the machine instructions and data of corresponding programs and subroutines into the memory. Loading is done by the loader.

  - **Type of loaders**
    - Absolute loader
    - Bootstrap loader
    - Assemble-and-go loader
    - Relocating loader (Relative loader)
    - Direct linking loader

- **Absolute loader**
  - The object code is loaded to the specified location in the memory.
  - All functions are accomplished in a single pass as follows:
    - The Header record of object programs is checked to verify that the correct program has been presented for loading.
    - As each Text record is read, the object code it contains is moved to the indicated address in memory.
    - When the End record is encountered, the loader jumps to the specified address to begin execution of the loaded program.
  - No linking and relocation needed.

```
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
```

  - Example: Consider the following object program

```
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	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮
0FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
1000	14103348	20390010	36281030	30101548
1010	20613C10	0300102A	0C103900	102D0C10
1020	36482061	0810334C	0000454F	46000003
1030	000000xx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮
2030	xxxxxxxx	xxxxxxxx	xx041030	001030E0
2040	205D3020	3FD8205D	28103030	20575490
2050	392C205E	38203F10	10364C00	00F10010
2060	00041030	E0207930	20645090	39DC2079
2070	2C103638	20644C00	0005xxxx	xxxxxxxx
2080	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮

← COPY

(b) Program loaded in memory

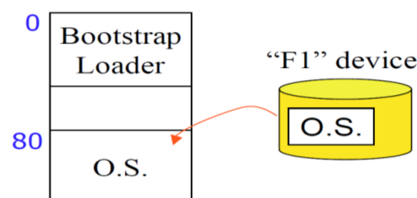
- Each pair of bytes from the object program record must be packed together into one byte during loading.
- Eg: Opcode for STL is 14. It is saved in object program as 2 bytes(2 characters). While loading it is converted to single byte(00010100).
- The content of the memory location for which there is no Text record are shown as xxxx.

▪ **Advantage**:
- Simple
- Efficient(less space and loading time)

▪ **Disadvantage**:
- Programmer should specify the actual address
  - If the system having small memory, only one program can run at a time. So it does not create much difficulty to specify the address.
  - On a larger system, we are supposed to run several independent programs. It is not easy to specify actual address while writing programs.
- Difficult to use subroutines libraries efficiently.
  - If there are multiple subroutines, the programmer must remember the address of each and use that absolute address explicitly in other subroutines to perform subroutine linkage.
  - **Solution**: Write relocatable programs instead of absolute ones.

Bootstrap Loader

- It is a special type of absolute loader.
- When a computer is first turned on or restarted, bootstrap loader is executed.
- This bootstrap loads the operating system into memory.
- The bootstrap itself begins at address 0
- It loads the OS(from device F1) starting at address 0x80.
- The object code having no header record, end record or control information.
- After loading the OS, the control is transferred to the instruction at address 0x80.



- Algorithm

```
X = 0x80                    //initial location of the OS to be loaded
Loop
{   A = GETC                //read and convert from ASCII to hexadecimal digit
    Save the value in the higher order 4 bits of S
    A = GETC                //read and convert from ASCII to hexadecimal digit
    Combine the value to form one byte A = (A + S)
    Store the value of A to the address in register
    X = X + 1
}
```

Algorithm for GETC

```
A = read one character
If A = 0x04 then        jump to 0x80    //end of file
If A < 48 then GETC                     // ignore the character if it is < 0
A = A – 48                              // convert to hexadecimal
If A < 10 then return
A = A – 7
return
```

- Register X keeps the address of the next memory location to be loaded.
- GETC is used to read and convert a pair of characters from device F1.
- These 2 hexadecimal digit values are combined into a single byte by shifting the first one left 4 bit positions and adding the second to it.
- The resulting byte is stored at the address currently in register X.

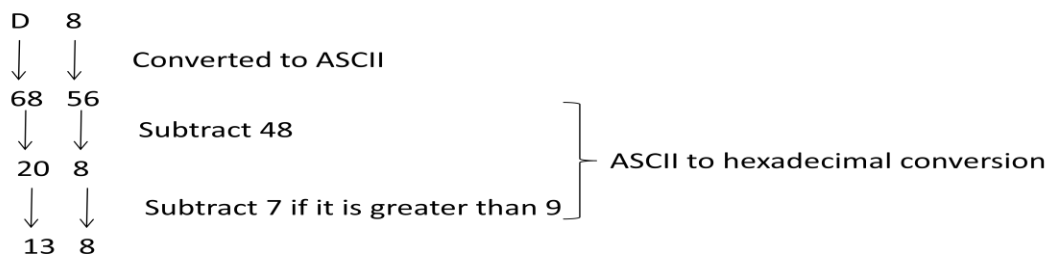
- Program

```
BOOT     START    0         BOOTSTRAP LOADER FOR SIC/XE

         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
         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
         STCH     0,X       STORE AT ADDRESS IN REGISTER X
         TIXR     X,X       ADD 1 TO MEMORY ADDRESS BEING LOADED
         J        LOOP      LOOP UNTIL END OF INPUT IS REACHED

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')
RETURN   RSUB               RETURN TO CALLER
INPUT    BYTE     X'F1'     CODE FOR INPUT DEVICE
         END      LOOP
```

- Eg: = C “D8”



- **Machine-Dependent Loader Feature**
  - Shortcoming of an absolute loader
    - Programmer needs to specify the actual address at which it will be loaded into memory.
    - It is difficult to run several programs concurrently, sharing memory between them.
    - It is difficult to use subroutine libraries.
  - Solution:
    - A more complex loader that provides
      - Program relocation
      - Program linking
  - The machine dependent loader features are
    - Relocation
    - Linking

  - **Program Relocation**
    - **Program Relocation**
      - 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.
    - **Relocating Loader / Relative Loader**: Loaders that allow program relocation.
    - **Two methods** for specifying relocation as part of the object program
      - Modification records
      - Relocation bits
    - **Modification records**
      - A Modification record is used to describe each part of the object code that must be changed when the program is relocated.
      - Used when a small number of relocations is required.
      - **Format**

Mod. Record	1	M
	2-7	Starting address of the field to be modified, relative to the beginning of the program (HEX)
	8-9	Length of the field to be modified, in half-bytes (HEX)
	10	Modification flag (+ or -)
	11-16	External symbol whose value is to be added to or subtracted from the indicated field

- Example

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
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	
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
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	

- Most of the instructions in this program use relative or immediate addressing
- Lines 15, 35 and 65 are the only items whose values are affected by program relocation.
- The object program should have one modification record for each value that must be changed during relocation.

- The object program is

```
HCOPY  000000001077
T0000001D17202D69202D4B1010360320262900003320074B10105D3F2FEC032010
T00001D130F20160100030F200D4B10105D3E2003454F46
T0010361DB410B400B44075101000E32019332FFADB2013A00433200857C003B850
T0010531D3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T001070073B2FEF4F000005
M00000705+COPY
M00001405+COPY
M00002705+COPY
E000000
```

One modification record for each relocation

- **Algorithm**

```
Get PROGADDR from Operating System
Read a record from input file
While record_type != 'E' do
{
        If record_type = 'T' do
        {
          Move object code from record to location PROGADDR + Specified address
        }
        If record_type = 'M' do
        {
            Add PROGADDR at the location (PROGADDR + Specified address)
        }
        Read next input record
}
```

- **Disadvantage**
  - It is not well suited for use with all machine architectures. It is not suited for standard SIC programs.

- **Relocation bits**
  - Each instruction is associated with one relocation bit
  - These relocation bits are gathered into bit masks in a Text record.
    - Relocation bit is 0: no modification is needed
    - Relocation bit is 1: modification is needed.
  - Used when a large number of relocations is required.
  - Format

    Text record
        col 1: T
        col 2-7: starting address
        col 8-9: length (byte)
        col 10-12: relocation bits
        col 13-72: object code

  - Twelve-bit mask is used in each Text record (col:10-12)
  - Each text record contains less than 12 words
  - Unused words are set to 0
  - For absolute loader, there are no relocation bits. Column 10-69 contains object code.

- Example: Relocatable program for standard SIC machine

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	140033
15	0003	CLOOP	JSUB	RDREC	481039
20	0006		LDA	LENGTH	000036
25	0009		COMP	ZERO	280030
30	000C		JEQ	ENDFIL	300015
35	000F		JSUB	WRREC	481061
40	0012		J	CLOOP	3C0003
45	0015	ENDFIL	LDA	EOF	00002A
50	0018		STA	BUFFER	0C0039
55	001B		LDA	THREE	00002D
60	001E		STA	LENGTH	0C0036
65	0021		JSUB	WRREC	481061
70	0024		LDL	RETADR	080033
75	0027		RSUB		4C0000
80	002A	EOF	BYTE	C'EOF'	454F46
85	002D	THREE	WORD	3	000003
90	0030	ZERO	WORD	0	000000
95	0033	RETADR	RESW	1	
100	0036	LENGTH	RESW	1	
105	0039	BUFFER	RESB	4096	
125	1039	RDREC	LDX	ZERO	040030
130	103C		LDA	ZERO	000030
135	103F	RLOOP	TD	INPUT	E0105D
140	1042		JEQ	RLOOP	30103F
145	1045		RD	INPUT	D8105D
150	1048		COMP	ZERO	280030
155	104B		JEQ	EXIT	301057
160	104E		STCH	BUFFER,X	548039
165	1051		TIX	MAXLEN	2C105E
170	1054		JLT	RLOOP	38103F
175	1057	EXIT	STX	LENGTH	100036
180	105A		RSUB		4C0000
185	105D	INPUT	BYTE	X'F1'	F1
190	105E	MAXLEN	WORD	4096	001000
210	1061	WRREC	LDX	ZERO	040030
215	1064	WLOOP	TD	OUTPUT	E01079
220	1067		JEQ	WLOOP	301064
225	106A		LDCH	BUFFER,X	508039
230	106D		WD	OUTPUT	DC1079
235	1070		TIX	LENGTH	2C0036
240	1073		JLT	LOOP	381064
245	1076		RSUB		4C0000
250	1079	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

- The standard SIC machine does not use relative addressing (*PC-relative*, *Base-relative*)
- All instructions expect RSUB need relocation
- Too many modification records are required if we adopt the 1st method. So we move to relocation bit method.
- Object program is

```
HCOPY  00000000107A
T0000001EFFC14003348103900003628003030000154810613C000300002A0C003900002D
T00001E15E000C00364810610800334C0000454F46000003000000
T0010391EFFC040030000030E0105D30103FD8105D2800303010575480392C105E38103F
T0010570A8001000364C0000F1001000
T00106119FE0040030E01079301064508039DC10792C0036381064 4C000005
E000000
```

  - The underlined hexadecimal digits in the object program represent the bit mask.
  - FFC
    - FFC → 111111111100
    - All ten words are to be modified
  - E00
    - E00 → 111000000000
    - First three records are to be modified.

- Algorithm

```
Get PROGADDR from Operating System
Read a record from the input file
While record_type != 'E' do
{
    If record_type = 'T' do
    {
        length = second data
        mask bits (M) = third data
        for i=0 to length do
        {
          If Mi = 1 then
             Add PROGADDR to object code and move that data to the location
             (PROGADDR + Specified address)
          else
            Move object code from record to location PROGADDR + Specified address
        }
    }
    Read next input record
}
```

- **Program Linking**
  - The goal of program linking is to resolve the problems with external references (EXTREF) and external definitions (EXTDEF) from different control sections.
    - **EXTDEF (external definition)** - The EXTDEF statement in a control section names symbols that are defined in present control section and may be used by other sections.
      - Eg: EXTDEF LISTA, ENDA
      - The Define Record is used to represent symbols in the object program
      - Syntax:

Col. 1	D
Col. 2-7	Name of external symbol defined in this control section
Col. 8-13	Relative address within this control section (hexadecimal)
Col.14-73	Repeat information in Col. 2-13 for other external symbols

        - Eg: D^LISTA ^000040^ENDA ^000054
    - **EXTREF (external reference)** - The EXTREF statement names symbols used in present control section and are defined elsewhere.
      - Eg: EXTREF LISTB,ENDB,LISTC,ENDC
      - The Refer Record is used to represent symbols in the object program
      - Syntax:

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

        - Eg: R^ LISTB ^ENDB ^LISTC ^ENDC
  - Example: 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.
  - Each program contains
    - Instruction operands (REF1, REF2, REF3).
    - Values of data words (REF4 through REF8).
    - Consider REF1 in all control sections: +LDA LISTA
      - PROGA
        - Here LISTA is the label in the current control section.
        - The instruction is PC relative and will get the correct object code (03201D).
      - PROGB, PROGMC
        - Here LISTA is a reference to an external symbol.
        - We will not get the correct object code. Simply set the address field to 0.
        - Also place a modification record in the object program. It instructs the loader to add the value of the symbol LISTA to this address field when the program is linked.

(PROGA)

Loc		Source statement		Object code
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-LISTB)	000014
0060	REF8	WORD	LISTB-LISTA	FFFFC0
		END	REF1	

Object programs (PROGA)

```
H^PROGA ^000000^000063
D^LISTA ^000040^ENDA  ^000054
R^LISTB ^ENDB  ^LISTC ^ENDC
:
T^000020^0A^03201D^77100004^050014
:
T^000054^0F^000014^FFFFF6^00003F^000014^FFFFC0
M^000024^05^+LISTB
M^000054^06^+LISTC
M^000057^06^+ENDC
M^000057^06^-LISTC
M^00005A^06^+ENDC
M^00005A^06^-LISTC
M^00005A^06^+PROGA
M^00005D^06^-ENDB
M^00005D^06^+LISTB
M^000060^06^+LISTB
M^000060^06^-PROGA
E^000020
```

(PROGB)

Loc		Source statement		Object code
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-LISTB)	FFFFF0
007C	REF8	WORD	LISTB-LISTA	000060
		END		

Object programs (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
```

(PROGC)

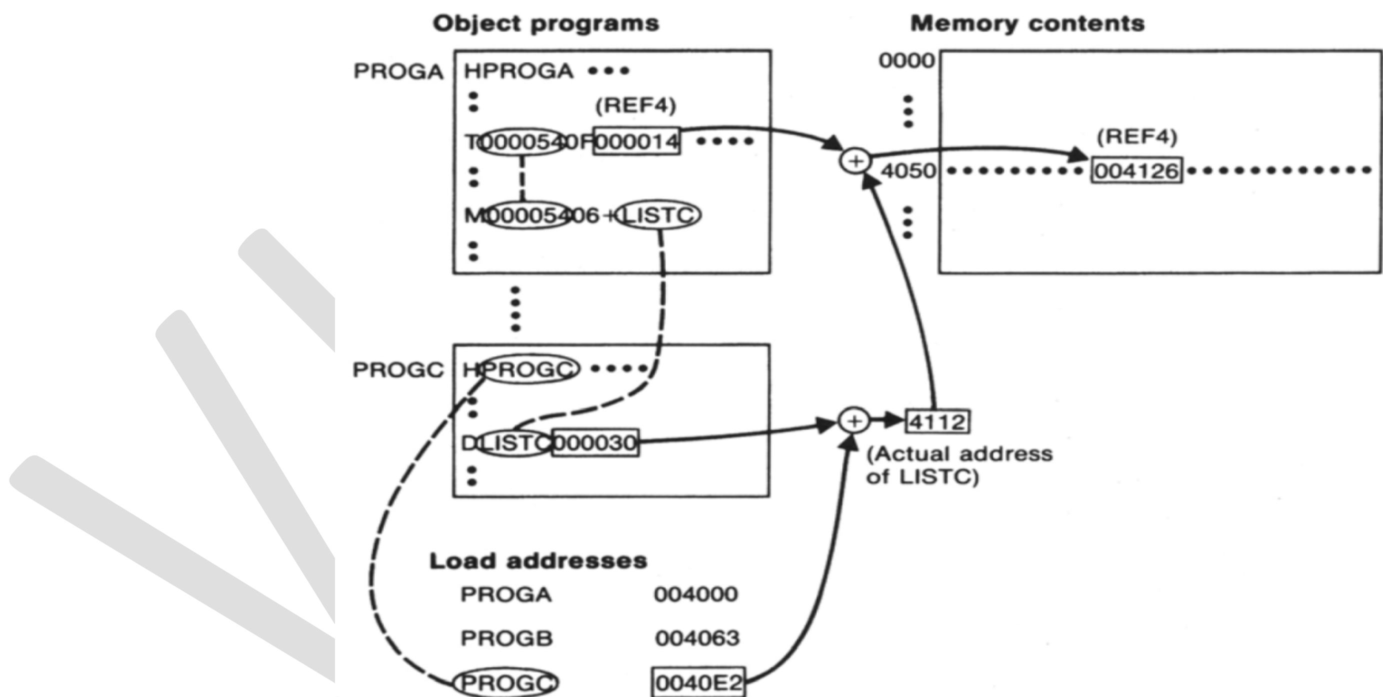
```
Loc     Source statement                              Object code

0000    PROGC   START   0
                EXTDEF  LISTC,ENDC
                EXTREF  LISTA,ENDA,LISTB,ENDB
                .
                .
                .
0018    REF1    +LDA    LISTA                         03100000
001C    REF2    +LDT    LISTB+4                       77100004
0020    REF3    +LDX    #ENDA-LISTA                   05100000
                .
                .
                .
0030    LISTC   EQU     *
                .
                .
0042    ENDC    EQU     *
0042    REF4    WORD    ENDA-LISTA+LISTC              000030
0045    REF5    WORD    ENDC-LISTC-10                 000008
0048    REF6    WORD    ENDC-LISTC+LISTA-1            000011
004B    REF7    WORD    ENDA-LISTA-(ENDB-LISTB)       000000
004E    REF8    WORD    LISTB-LISTA                   000000
                END
```

Object programs (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
```

- Consider REF2: REF2 +LDT LISTB+4
  - PROGA, PROGC
    - Store the value of constant in the address field(00004)
    - There should be one modification record which instructs the loader to add to this field the value of LISTB.
  - PROGB
    - LISTB is the label in the current control section.
    - The instruction is PC relative and the will get the object code (772027).
- Consider REF4: REF4 WORD ENDA-LISTA+LISTC
  - PROGA
    - Here LISTA and ENDA are the labels in the current control section.
    - LISTC is a reference to an external symbol.
    - The object code is the value of ENDA-LISTA (000014), which is a temporary value.
    - Place a modification record for LISTC



- While loading the starting address of
  - PROGA is 4000
  - PROGB is 4063
  - PROGC is 40E2
- Starting address of REF4 in PROGA is 4054(beginning address of PROGA + 0054)

- PROGB
  - Here LISTA, ENDA and LISTC are the references to external symbols

- Put 000000 as object code.
- Place modification records for LISTA, ENDA and LISTC

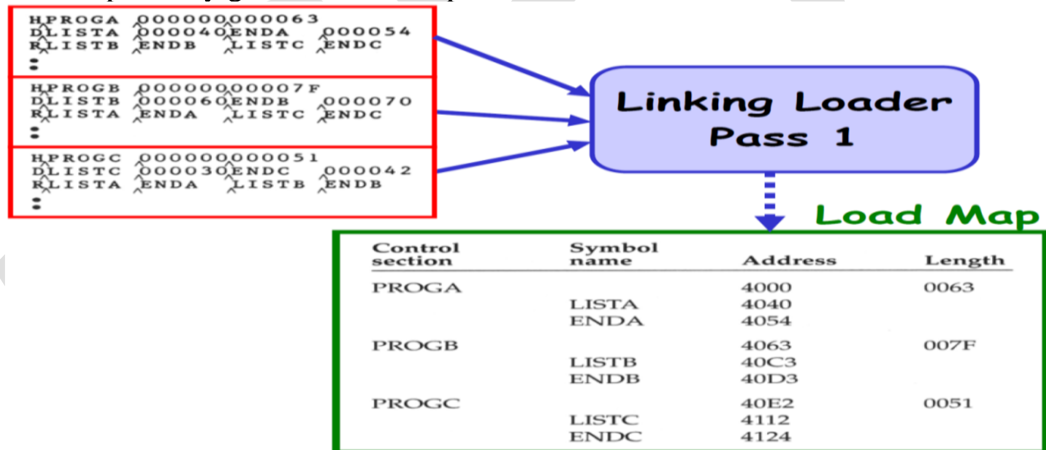
▪ PROGC
- Here LISTA and ENDA are the references to external symbols
- LISTC is the labels in the current control section.
- The object code is the value of LISTC (000030). While loading PROGC the address of LISTC may change. So place a modification record for PROGC.
- Place two more modification records for LISTA and ENDA

- The program in memory after relocation, linking and loading

Memory address	Contents				
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	
3FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
4000	....	....	....	....	
4010	....	....	....	....	
4020	03201D77	1040C705	0014....	....	← PROGA
4030	....	....	....	....	
4040	....	....	....	....	
4050	....	00412600	00080040	51000004	
4060	000083..	....	....	....	
4070	....	....	....	....	
4080	....	....	....	....	
4090	....	....	..031040	40772027	← PROGB
40A0	05100014	....	....	....	
40B0	....	....	....	....	
40C0	....	....	....	....	
40D0	00	41260000	08004051	00000400	
40E0	0083....	....	....	....	
40F0	....	....	0310	40407710	
4100	40C70510	0014....	....	....	← PROGC
4110	....	....	....	....	
4120	....	00412600	00080040	51000004	
4130	000083xx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
4140	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	

- **Algorithm and Data Structures for a Linking Loader**
  - ▪ Input: set of object programs (control sections) that are to be linked together.
  - ▪ A linking loader usually makes two passes over its input
    - Pass 1: Assigns addresses to all external symbols.
    - Pass 2: Performs the actual relocation, linking and loading.
  - ▪ Data structure used is External Symbol Table (ESTAB). The fields are
    - Control sections name
    - Name each external symbol
    - Address of each external symbol
  - ▪ Two other important variables are:
    - PROGADDR (program load address)
      - The beginning address in memory where the linked program is to be loaded.
      - Its value is supplied to the loader by the OS.
    - CSADDR (control section address)

  - The starting address assigned to the control section currently being scanned by the loader
- A hashed organization is used for this table
- **Pass 1**
  - The loader is concerned only with Header and Define records in the control sections
  - The beginning load address for the linked program (PROGADDR) is obtained from the OS. This becomes the starting address (CSADDR) for the first control section in the input sequence.
  - The control section name from 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 is added to CSADDR. This calculation gives the starting address for the next control section in sequence.
  - 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.
  - Pass 1 optionally generate load map



Control section	Symbol name	Address	Length
PROGA		4000	0063
	LISTA	4040	
	ENDA	4054	
PROGB		4063	007F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	0051
	LISTC	4112	
	ENDC	4124	

- **Pass 2**
  - As each Text record is read, the object code is moved to the specified address + 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.
  - The last step performed by the loader is usually the transferring of control to the loaded program to begin execution.
    - The End record for each control section may contain the address of the first instruction in that control section to be executed. Our loader takes this as the transfer point to begin execution.
    - If more than one control section specifies a transfer address, the loader arbitrarily uses the last one encountered.

- If no control section contains a transfer address, the loader uses the beginning of the linked program (i.e., PROGADDR) as the transfer point.
- Normally, a transfer address would be placed in the End record for a main program, but not for a subroutine.

Pass 1 Linking Loader Algorithm

```
Get PROGADDR from OS
CSADDR = PROGADDR               \\ for the 1st control section
While not end of input do
{
    Read the next input record      \\ header record for the control section
    CSLTH = control section length
    Search ESTAB for the control section name
    If found then
        Set error flag
    Else
        Enter control section name and CSADDR into ESTAB
    While record_type != 'E' do
    {
        Read the next input record
        If record_type = 'D' then
        {
            For each symbol in the record do
            {
                Search ESTAB for symbol name
                If found then
                    Set error flag
                Else
                    Enter symbol and (CSADDR + indicated address) into ESTAB
            }
        }
    }
    CSADDR = CSADDR + CSLTH        //starting address of the next control section
}
```

Pass 2 Linking Loader Algorithm

```
CSADDR = PROGADDR               \\ for the 1st control section
EXECADDR = PROGADDR
While not end of input do
{
    Read the next input record      \\ header record for the control section
    CSLTH = control section length
    While record_type != 'E' do
    {
        Read the next input record
        If record_type = 'T' then
        {
```


VJCET

```
            If the object code is in character form then
                Convert it into internal representation
            Move the object code from record to location (CSADDR + specified address)
        }
        Else if record_type = 'M' then
        {
            Search ESTAB for modification symbol name
            If found then
                Add or subtract the symbol value at location (CSADDR + specified address)
            Else
                Set error flag
        }
    }
    If an address is specified in End record then
        EXECADDR = CSADDR + specified address
    CSADDR = CSADDR + CSLTH
}
Jump to location given by EXECADDR to start execution of the program
```

- We can make the linking loader algorithm more efficient by
  - Assigning a *reference number* to each external symbol referred to in a control section
    - 01: control section name
    - 02~: external reference symbols
  - Use this reference number (instead of the symbol name) in Modification records
  - Advantage of this reference-number mechanism:
    - It avoids multiple searches of ESTAB for the same symbol during the loading of a control section.
      - Search of ESTAB for each external symbol can be performed once and the result is stored in a table indexed by the reference number.
      - The values for code modification can then be obtained by simply indexing into the table.

PROGA

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

PROGB

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

PROGC

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

- The object program will be as follows

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
R02LISTB 03ENDB  04LISTC 05ENDC
.
.
T0000200A03201D77100004050014
.
.
T0000540F000014FFFFF600003F000014FFFFC0
M00002405+02
M00005406+04
M00005706+05
M00005706-04
M00005A06+05
M00005A06-04
M00005A06+01
M00005D06-03
M00005D06+02
M00006006+02
M00006006-01
E000020
```

```
HPROGB 00000000007F
DLISTB 000060ENDB  000070
R02LISTA 03ENDA  04LISTC 05ENDC
.
.
T0000360B0310000077202705100000
.
.
T0000700F000000FFFFF6FFFFFFFFFFF0000060
M00003705+02
M00003E05+03
M00003E05-02
M00007006+03
M00007006-02
M00007006+04
M00007306+05
M00007306-04
M00007606+05
M00007606-04
M00007606+02
M00007906+03
M00007906-02
M00007C06+01
M00007C06-02
E
```

```
HPROGC 000000000051
DLISTC 000030ENDC  000042
R02LISTA 03ENDA  04LISTB 05ENDB
.
.
T0000180C0310000077100004051000000
.
.
T0000420F000030000008000011000000000000
M00001905+02
M00001D05+04
M00002105+03
M00002105-02
M00004206+03
M00004206-02
M00004206+01
M00004806+02
M00004B06+03
M00004B06-02
M00004B06-05
M00004B06+04
M00004E06+04
M00004E06-02
E
```

- **<u>MACHINE-INDEPENDENT LOADER FEATURES</u>**
  - Loading and linking are OS service functions.
  - Following are the machine-independent loader features
    - Automatic Library Search
    - Loader Options
  - **<u>Automatic Library Search(Automatic Library Call)</u>**
    - This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded.
    - This feature allows the programmer to use subroutines from one or more libraries (eg: mathematical or statistical routines).
    - These subroutines are automatically retrieved from a library, linked with main program and loaded as they are needed during linking.

- The programmer needs to mention these subroutine names as external references in the source program.
- Steps:
  - Linking loaders that support automatic library search must keep track of external symbols that are referred to in the primary input to the loader.
  - At the end of Pass 1, the symbols in ESTAB that remain undefined represent unresolved external references.
  - The loader searches the library or libraries specified for routines that contain the definitions of these symbols, and processes the subroutines found by this search exactly as if they had been part of the primary input stream.
  - The subroutines fetched from a library in this way may themselves contain external references. It is therefore necessary to repeat the library search process until all references are resolved.
  - If unresolved external references remain after the library search is completed, these must be treated as errors.
- It also allows the programmer to override standard subroutines.
  - Example
    - Suppose the main program refers to a standard subroutine SQRT.
    - A programmer wanted to use a different version of SQRT by including it as input to the loader.
    - By the end of Pass1 of the loader, SQRT would already be defined, so it would not be included in any library search.
- The libraries to be searched by the loader ordinarily contain assembled or compiled versions of the subroutines (i.e., object programs)
- In most cases a special file structure is used for the libraries. This structure contains a directory that gives the name of each routine and a pointer to its address within the file.
- Some operating systems can keep the directory for commonly used libraries permanently in memory.
- The same technique applies equally well to the resolution of external references to data items

- **Loader Options**
  - Many loaders allow the user to specify options that modify the standard processing.
  - Many loaders have a special command language that is used to specify loader options.
  - Different ways to provide these control statements
    - A separate input file to the loader that contains such control statements.
    - These statements are embedded in the primary input stream between object programs
    - Include these statements in source program, and the assembler retains these commands as a part of the object program.

VJCET

- **Loader option 1**: Allows the selection of alternative sources of input.
  - Command: INCLUDE program-name (library-name)
    Direct the loader to read the designated object program from a library and treat it as if it were part of the primary loader input.
- **Loader option 2**: Allows the user to delete or change external symbols or entire control sections.
  - Command: DELETE csect-name
    Instruct the loader to delete the named control section(s) from the set of programs being loaded.
  - Command: CHANGE name1, name2
    The external symbol name1 to be changed to name2.
  - Example: Suppose we have a main program COPY and two subroutines RDREC and WRREC, each of these are separate control sections. Suppose that READ and WRITE are the two utility subroutines which perform the functions as RDREC and WRREC. The following commands allow the main program COPY to use these utility subroutines.

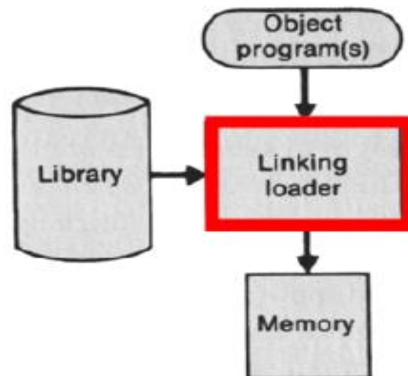
```
INCLUDE READ(UTLIB)
INCLUDE WRITE(UTLIB)
DELETE RDREC, WRREC
CHANGE RDREC, READ
CHANGE WRREC, WRITE
```

    - It direct the loader to include control sections READ and WRITE from the library UTLIB
    - Delete the control section RDREC and WRREC
    - All external references to RDREC is changed to refer to symbol READ
    - All external references to WRREC is changed to refer to symbol WRITE
- **Loader option 3**: Involves the automatic inclusion of library routines to satisfy external references.
  - Command: LIBRARY MYLIB
    Such user-specified libraries are normally searched before the standard system libraries. This allows the user to use special versions of the standard routines.
  - Suppose the main function of a program is used to gather and sort data. The program also performs an analysis of data using the routines STDDEV, PLOT and CORREL from a statistical library. Suppose this statistical analysis is not performed in a particular execution, use the following command.
    NOCALL STDDEV, PLOT, CORREL
    This instructs the loader that these external references are to remain unresolved. This avoids the overhead of loading and linking the unneeded routines, and saves the memory space that would otherwise be required.
- **Loader option 4**: It is possible to specify that no eternal references be resolved by library search. This means that an error will result if the program attempts to

make such an external references during execution. This option is useful when programs are to be linked but not executed immediately.

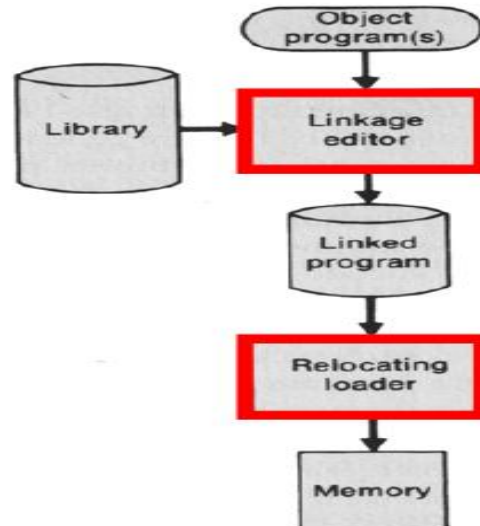
- **Loader option 5**: Can control output from loader
  - A load map may be generated during loading process
  - Through control statements the user can specify whether or not such a map is to be printed at all.
  - If a map is desired, the level of detail can be selected.
    - Ex: the map may include control section name and address only
- **Loader option 6**: Ability to specify the location at which execution is to begin (Overriding any information given in the object program).
- **Loader option 7**: To control whether or not the loader should attempt to execute the program if errors are detected during the load operation.
  - Ex: unresolved eternal references.

- **<u>LOADER DESIGN OPTIONS</u>**
  - **Linking loader**:
    - The source program is assembled or compiled, producing an object program which may contain several control sections.
    - A linking loader performs all linking and relocation operations, including automatic library search if specified, and loads the linked program directly into memory for execution.



- Disadvantage: The linking loader searches the libraries and resolves external references every time the program is executed.
- Advantage:
  - When the program is used so infrequently, it is not worthwhile to store the assembled version in a library. In such cases it is more efficient to use linking loader.
  - It is used in a program development and testing environment
- There are two alternatives design options:
  - Linkage Editor
  - Dynamic Linking
- **Linkage Editor**
  - Which perform linking prior to load time

- It produces a linked version of the program (load module or executable image), which is written to a file or library for later execution.
- A simple relocating loader can be used to load the linked version of program into memory for execution.



- The Linkage Editor performs relocation of all control sections relative to the start of the linked program.
- The only object code modification necessary is the addition of an actual load address to relative values within the program.
- Relocation is indicated by some mechanisms such as Modification Records or Bit Masks.
- Advantage:
  - The loading can be accomplished in one pass with no external symbol table required. This involves much less overhead than using a linking loader
  - Reduce the time: If Resolution of external references and library searching are only performed once. It will reduces the load time.
- One variant: If the actual address at which the program will be loaded is known in advance, the linkage editor can perform all of the needed relocation. The result is a linked program that is an exact image of the way the program will appear in memory during execution. In this case only an absolute loader is needed to load object code into memory.
- Linkage editors can perform many useful functions
  - The linkage editor can be used to replace the subroutines in the linked version
    - Eg: Suppose PLANNER is a program that uses a large number of subroutines. One subroutine among them is PROJECT. We can write a new version of PROJECT to improve its efficiency. After the new version of PROJECT is assembled, the linkage editor can be used to replace this subroutine in the linked version of all the other subroutines. Use the following code for this purpose.

```
INCLUDE     PLANNER(PROGLIB)
DELETE      PROJECT
INCLUDE     PROJECT(NEWLIB)
REPLACE     PLANNER(PROGLIB)
```

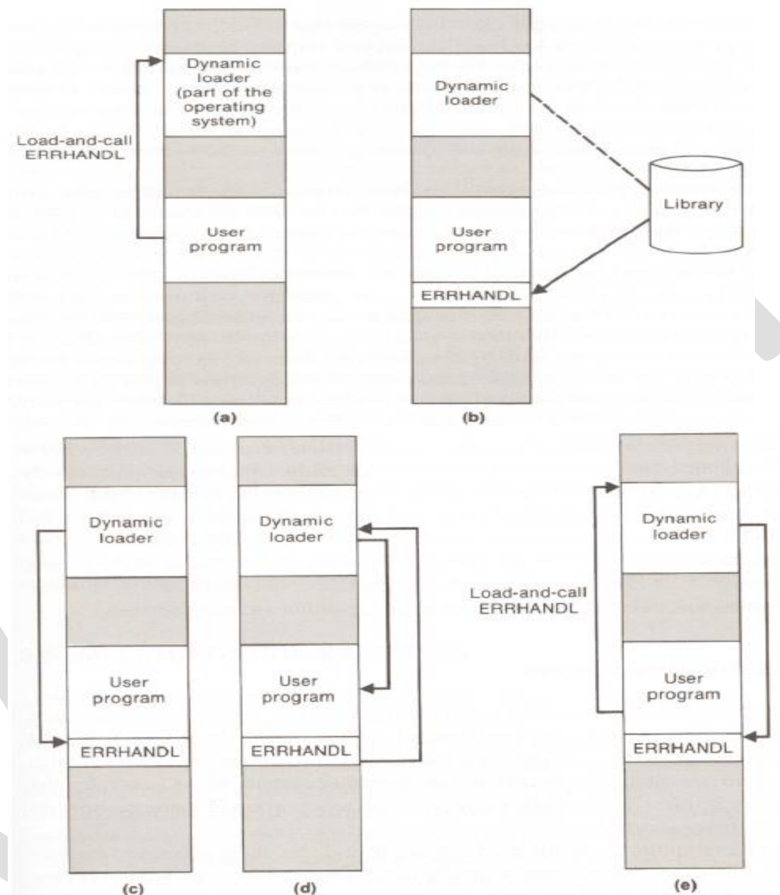
- Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together.

```
INCLUDE     READR(FTNLIB)
INCLUDE     WRITER(FTNLIB)
INCLUDE     BLOCK(FTNLIB)
INCLUDE     DEBLOCK(FTNLIB)
INCLUDE     ENCODE(FTNLIB)
INCLUDE     DECODE(FTNLIB)
.
.
.
SAVE        FTNIO(SUBLIB)
```

  - Suppose there are few closely related subroutines in the directory FTNLIB. The above command sequence combines these subroutines from the directory FTNLIB and produces a linked module called FTNIO. It is then insert in to the directory SUBLIB. Thus the search of SUBLIB before FTNLIB would retrieve FTNIO instead of the separate routines.

- **Dynamic Linking (Dynamic Loading or Load on call)**
  - The linking function is performed at execution time.
  - A subroutine is loaded and linked to the rest of the program when it is first called
  - Dynamic linking provides the ability to load the routines only when they are needed
  - Advantage:
    - Allow several executing programs to share one copy of a subroutine or library
    - Allow to share one object by several programs:
      - In object-oriented system, it allows the implementation of the object and its methods to be determined at the time the program is run.
    - Load the routines only when they are needed:
      - Suppose a program contains subroutines that correct errors in the input data during execution. If such errors are rare, this subroutine may not be used for all executions. Dynamic linking provides the ability to load the routines only when they are needed. This will save time and memory space.
    - Avoid to load the entire library for each execution
      - Suppose a program uses a library that contains large number of subroutines. The exact routine can not be predicted until the program examines its input. Dynamic linking loads the required subroutines instead of entire library for each execution.

- Dynamic loading must be called via an operating system service requests (Load-and-call services)
  - OS examines its internal tables to determine whether or not the routine is already loaded
  - If not present
    - Routine is loaded from library
    - Control is passed from OS to the called subroutine
    - Subroutine is finished
  - Else
    - Control is passed to a subroutine which is already in memory



  - Fig (a): The user program contains 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.
  - Fig (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.
  - Fig (c): Control is then passed from OS to the routine being called
  - Fig (d): 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.

        - Fig (e): 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.
        - This subroutine should retain in memory for later use as long as there is enough space. If the program requires more space, it may removed from memory.
    - *Binding* of the name to an actual address is delayed from load time until execution time

- **Bootstrap Loaders**
    - Given an idle computer with no program in memory, how do we get things started?
    - With the machine empty and idle there is no need for program relocation.
    - We can specify the absolute address for which the OS is first loaded.
    - An absolute loader is used to accomplish this task.
    - Different options:
        - Required an operator to enter the object code for an absolute loader into memory using switches on the computer console.
            - Disadvantage: Inconvenient and error prone
    - The absolute loader program may permanently resident in a ROM.
        - When some hardware signal occurs (the operator pressing a system start switch), the machine begins to execute this ROM program.
        - There are 2 options
            - The program is executed directly in the ROM
            - The program is copied from ROM to main memory and executed there.
        - Disadvantage:
            - Some systems do not have such read only storage
            - It is inconvenient to change the ROM program, if the modification in the absolute loader is required.
    - Have a built in hardware function (or a short ROM program) that reads a fixed length record from some device into memory at a fixed location.
        - This device may be selected via console switches.
        - After the read operation is complete, control is automatically transferred to the address in memory where the record was stored.
        - This record contains machine instructions that load the absolute program that follows.
        - The first record is generally referred to as bootstrap loader
        - This first record causes the reading of others, and these in turn can cause the reading of still more records – hence the term boots trap.
        - Such a loader is added to the beginning of all object programs.
        - This includes the OS itself and all stand-alone programs that are to be run without an OS.

Previous Year University Questions

1. Give the algorithm for an absolute loader
2. Given an idle computer with no programs in memory, how do we get things started?
3. What is the use of bitmask in program relocation? Illustrate with example
4. Describe the data structures used for the linking loader algorithm. Give the algorithm for pass 1 of the linking loader.
5. Which are the data structures used during the operation of a linking loader? Write the algorithm for Pass 2 of a Linking Loader
6. Give the algorithm for pass 2 of a linking loader.
7. With the data structures used, state and explain two pass algorithm for a linking loader.
8. Develop the records (excluding header, text and end records) for the following control section named COPY

Loc	Source Statement		
0000	COPY	START	0
		EXTDEF	BUFFER, BUFFEND, LENGTH
		EXTREF	RDREC,WRREC
0000	FIRST	STL	RETADR
0003	CLOOP	+JSUB	RDREC
0007		LDA	LENGTH
000A		COMP	#0
000D		JEQ	ENDFIL
0010		+JSUB	WRREC
0014		J	CLOOP
0017	ENDFIL	LDA	=C 'EOF'
001A		STA	BUFFER
001D		LDA	#3
0020		STA	LENGTH
0023		+JSUB	WRREC
0027		J	@RETADR
002A	RETADR	RESW	1
002D	LENGTH	RESW	1
		LTORG	
0030	*	=C 'EOF'	
0033	BUFFER	RESB	4096
1033	BUFEND	EQU	*
1000	MAXLEN	EQU	BUFEND-BUFFER

9. Write notes on machine independent loader features.
10. Explain any one machine independent loader feature.

11. What is Automatic Library Search?
12. Write notes on the different loader design options.
13. With a help of neat diagram explain what is a linkage editor?
14. Differentiate between linking loaders and linkage editors
15. What is Dynamic Linking? Explain with example.

VJCET