Module 2

- **LOADERS AND LINKERS:** Basic loader functions - Design of an Absolute Loader – A Simple Bootstrap Loader -Machine dependent loader features - Relocation – Program Linking – Algorithm and Data Structures for Linking Loader - Machine-independent loader features – Automatic Library Search – Loader Options - Loader design options - Linkage Editors – Dynamic Linking –

Chapter 1

Loaders

Loaders

- It is a utility program which takes object code as input, prepares it for execution and loads the executable code into the memory.
- Thus loader is actually responsible for initiating the execution process.

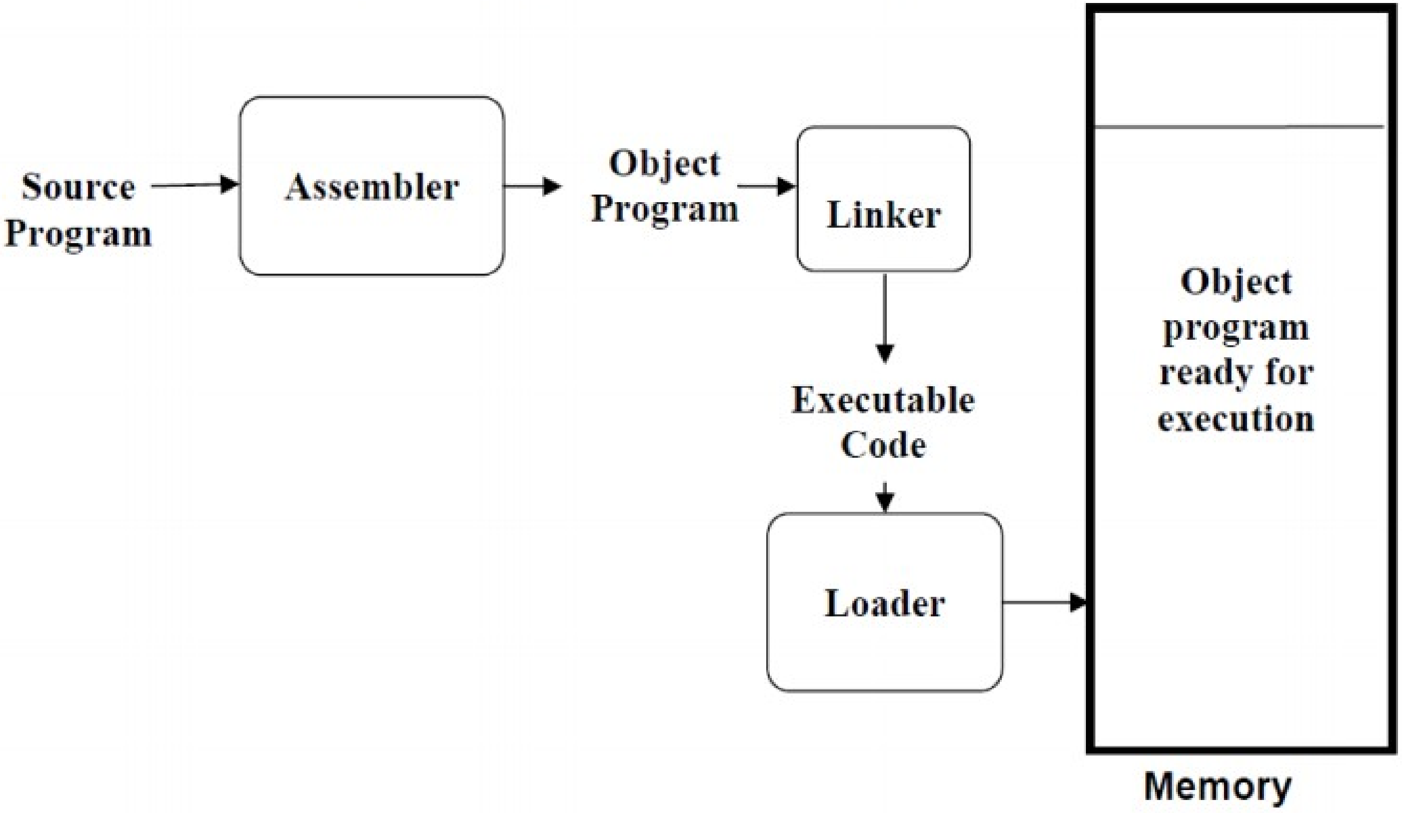


Fig: Role Of Loader and Linker

Basic Functions of Loader

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

1.ALLOCATION: it allocates the space for program in the memory, by calculating the size of the program.

2.LINKING: it resolves the symbolic references (code/data) between the object modules by assigning all the user subroutine and library subroutine addresses.

3.RELOCATION: There are some address dependent locations in the program such as address constants must be adjusted according to allocated space such activity done by loader.

4.LOADING:finally it places the machine instructions and data of corresponding programs and subroutines into the memory. Thus program now becomes ready for execution.

Absolute Loader

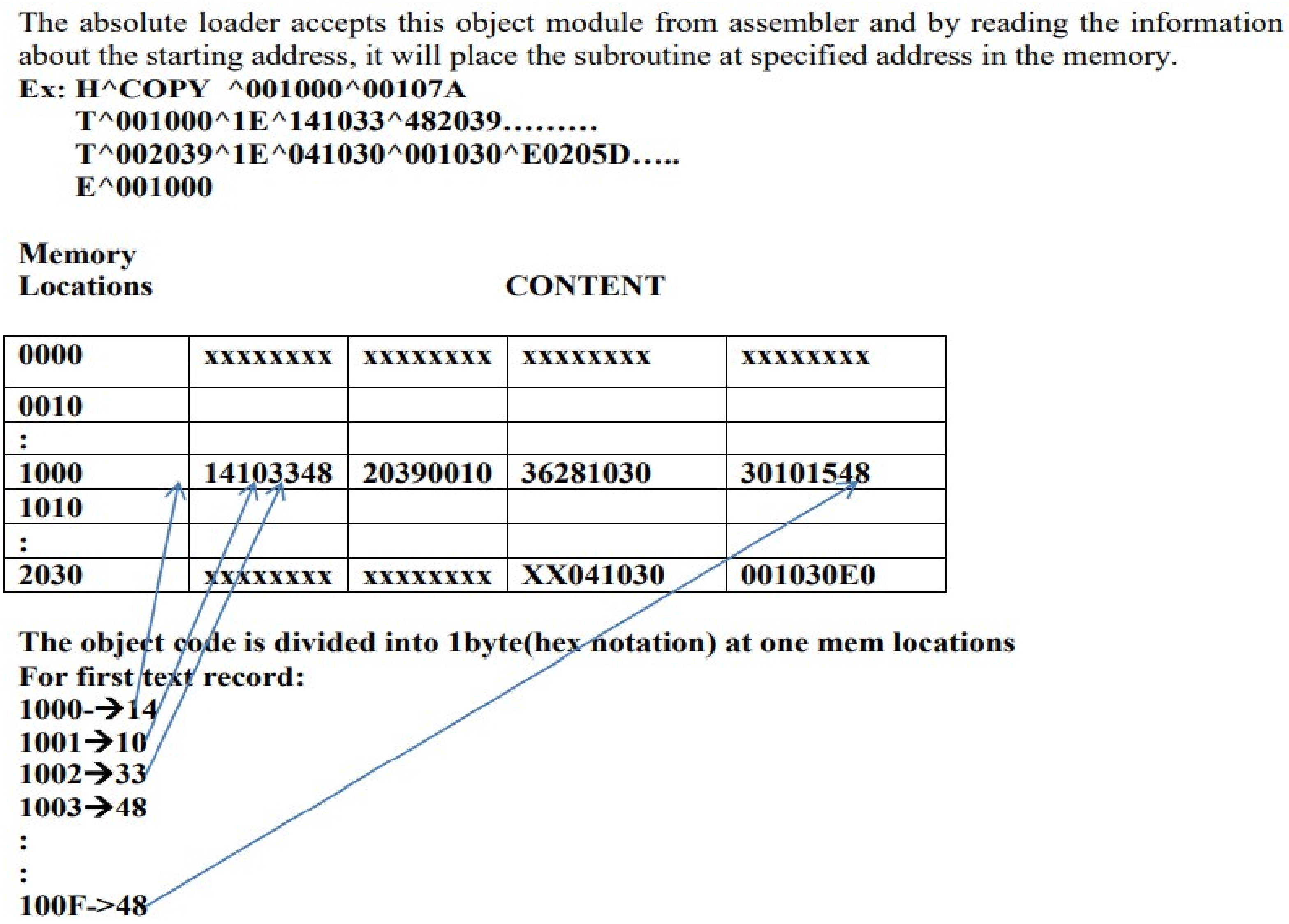
- Absolute Loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at specified locations in the memory. This type of loader is called absolute because no relocation information is needed rather it is obtained from the programmer or assembler.
- The starting address of every module is known to the programmer this corresponding address is stored in the object file, then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file.

Absolute Loader

The programmer should take care of two things :

- First, Specification of starting address of each module to be used. If some modification is done in some module then the length of that module may vary. This causes a change in the starting address of immediate next modules, its then programmer duty to make necessary changes in the starting addresses of respective modules.
- Second, while branching from one segment to another the absolute starting address of respective module is to be known by the programmer so that such address can be specified at respective JMP instruction.

Absolute Loader



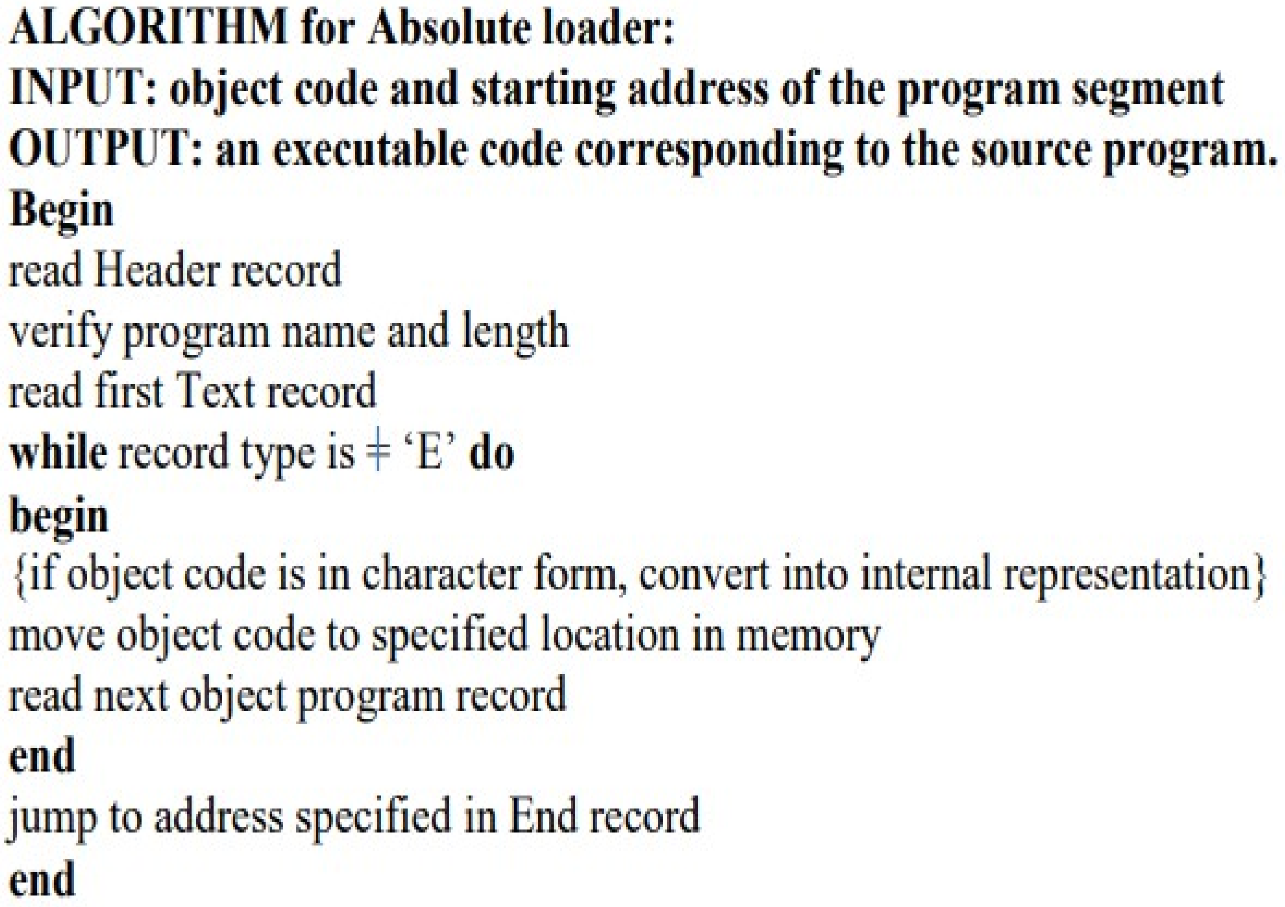
The absolute loader accepts this object module from assembler and by reading the information about the starting address, it will place the subroutine at specified address in the memory.

Ex: H^COPY ^001000^00107A
T^001000^1E^141033^482039………
T^002039^1E^041030^001030^E0205D…..
E^001000

Memory Locations	CONTENT			
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0010				
:				
1000	14103348	20390010	36281030	30101548
1010				
:				
2030	xxxxxxxx	xxxxxxxx	XX041030	001030E0

The object code is divided into 1byte(hex notation) at one mem locations
For first text record:
1000-→14
1001→10
1002→33
1003→48
:
:
100F->48

Absolute Loader



ALGORITHM for Absolute loader:

INPUT: object code and starting address of the program segment

OUTPUT: an executable code corresponding to the source program.

```
Begin
read Header record
verify program name and length
read first Text record
while record type is ≠ '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
```

ADVANTAGES:

- It is simple to implement
- The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code into main memory.
- The process of execution is efficient

DISADVANTAGES:

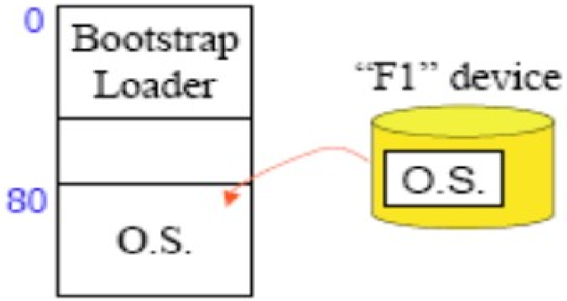
- It is the programmer duty to adjust all the inter segment addresses and manually do the linking activity .For that programmer has to know the memory management.

- If at all any modification is done the some segments, the starting addresses of immediate next segment may get changed, the programmer has to take care of this issue and he needs to update the corresponding addresses on any modification in the source.

Bootstrap Loader

- When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed.
- This bootstrap loads the first program to be run by the computer -- usually an operating system.
- The bootstrap itself begins at **address 0.**
- It loads the OS starting **address 0x80**.
- No header record or control information.
- The object code is consecutive bytes of memory.

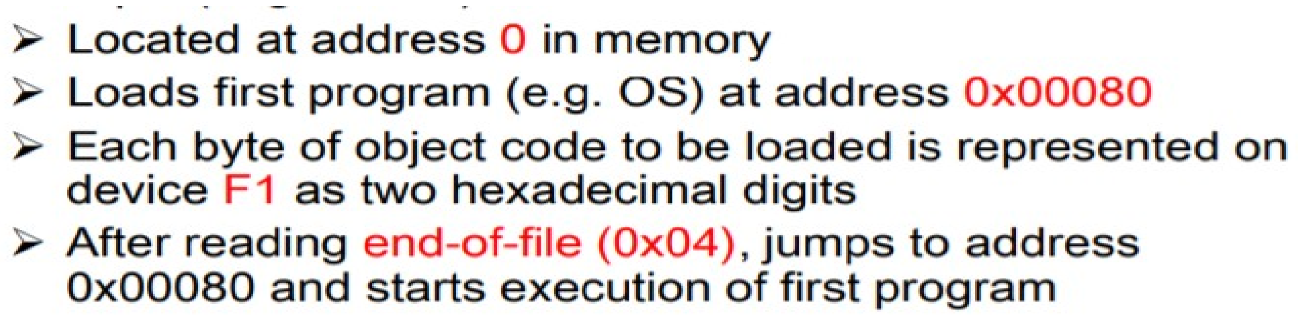
Bootstrap Loader



Various characteristics of bootstrap loader

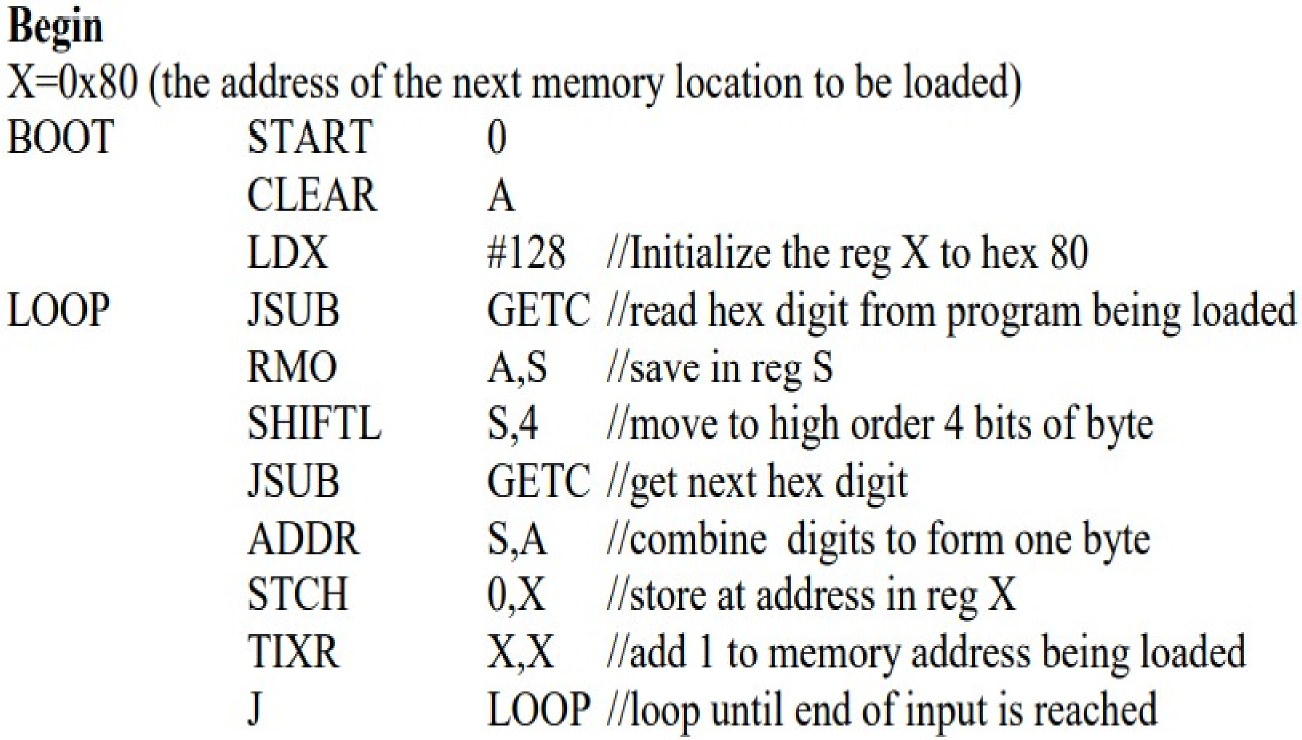
1. The bootstrap loader is a small program and it should be fitted in the ROM.
2. The bootstrap loader must load the necessary portions of OS in the main memory

3. The initial address at which the bootstrap loader is to be loaded is generally the lowest for example at location 0000 or at the highest location and not a intermediate location.



- ➢ Located at address 0 in memory
- ➢ Loads first program (e.g. OS) at address 0x00080
- ➢ Each byte of object code to be loaded is represented on device F1 as two hexadecimal digits
- ➢ After reading end-of-file (0x04), jumps to address 0x00080 and starts execution of first program

Bootstrap Loader for SIC/XE

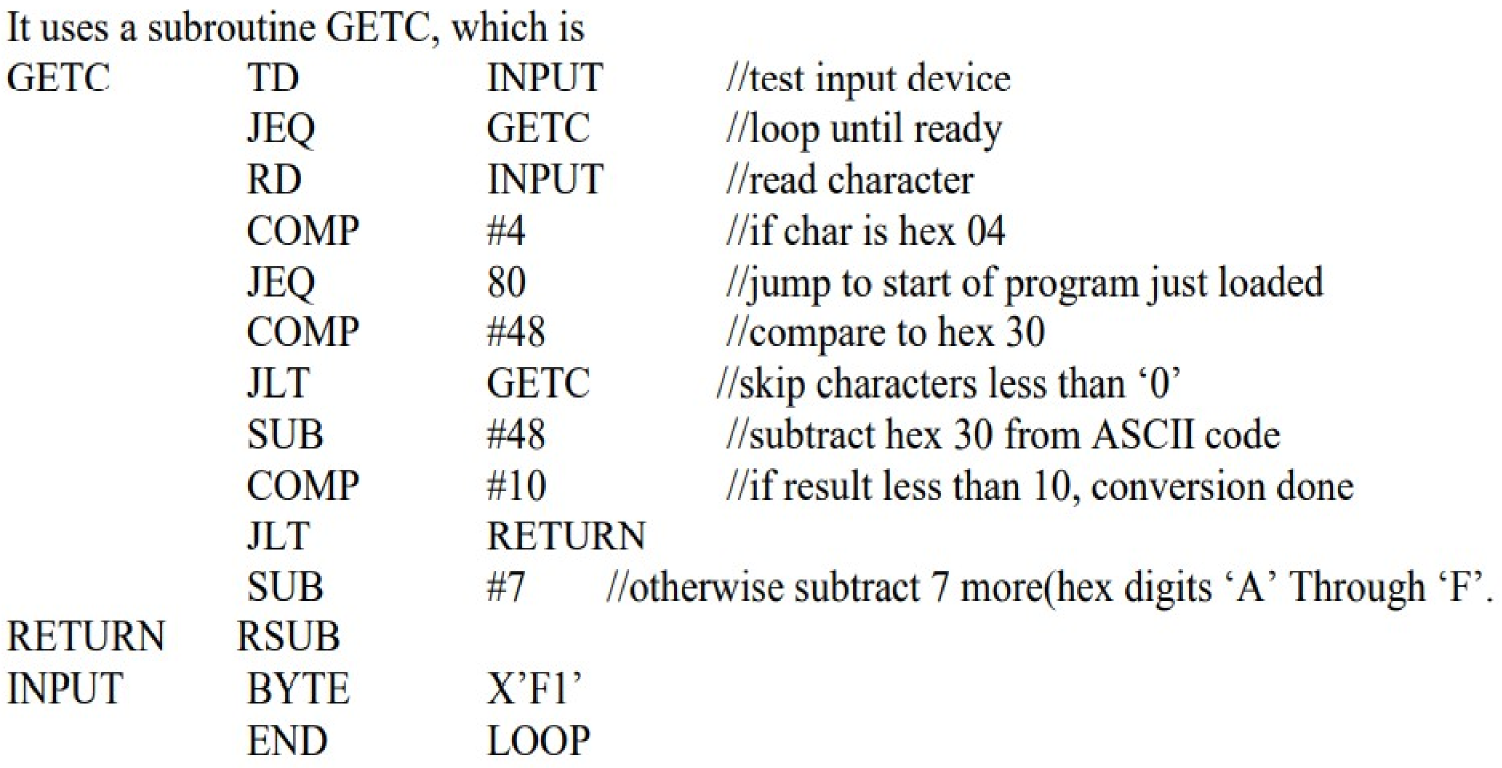


Begin

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

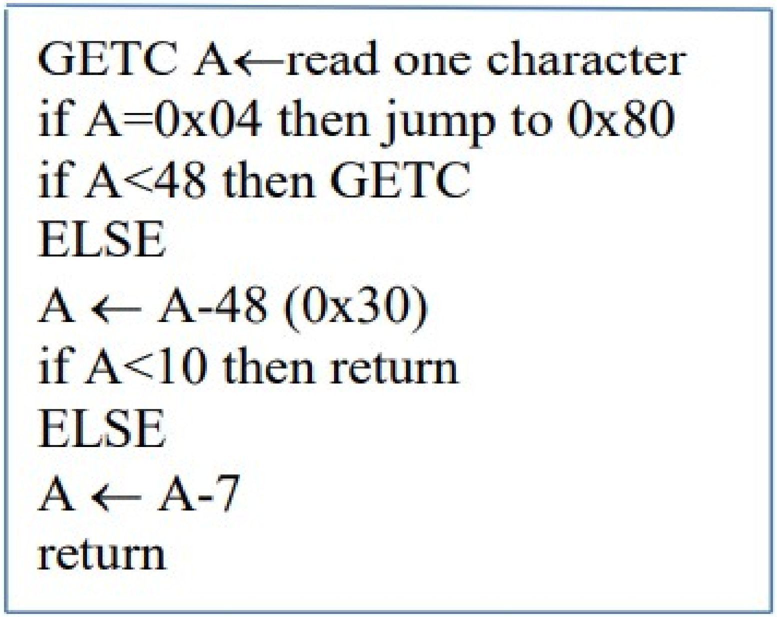
```
BOOT    START   0
        CLEAR   A
        LDX     #128    //Initialize the reg X to hex 80
LOOP    JSUB    GETC    //read hex digit from program being loaded
        RMO     A,S     //save in reg S
        SHIFTL  S,4     //move to high order 4 bits of byte
        JSUB    GETC    //get next hex digit
        ADDR    S,A     //combine  digits to form one byte
        STCH    0,X     //store at address in reg X
        TIXR    X,X     //add 1 to memory address being loaded
        J       LOOP    //loop until end of input is reached
```

Bootstrap Loader for SIC/XE



It uses a subroutine GETC, which is

```
GETC        TD          INPUT       //test input device
            JEQ         GETC        //loop until ready
            RD          INPUT       //read character
            COMP        #4          //if char is hex 04
            JEQ         80          //jump to start of program just loaded
            COMP        #48         //compare to hex 30
            JLT         GETC       //skip characters less than ‘0’
            SUB         #48         //subtract hex 30 from ASCII code
            COMP        #10         //if result less than 10, conversion done
            JLT         RETURN
            SUB         #7      //otherwise subtract 7 more(hex digits ‘A’ Through ‘F’.
RETURN     RSUB
INPUT       BYTE        X’F1’
            END         LOOP
```



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

MACHINE DEPENDENT LOADER FEATURES:

1.RELOCATION

2. PROGRAM LINKING

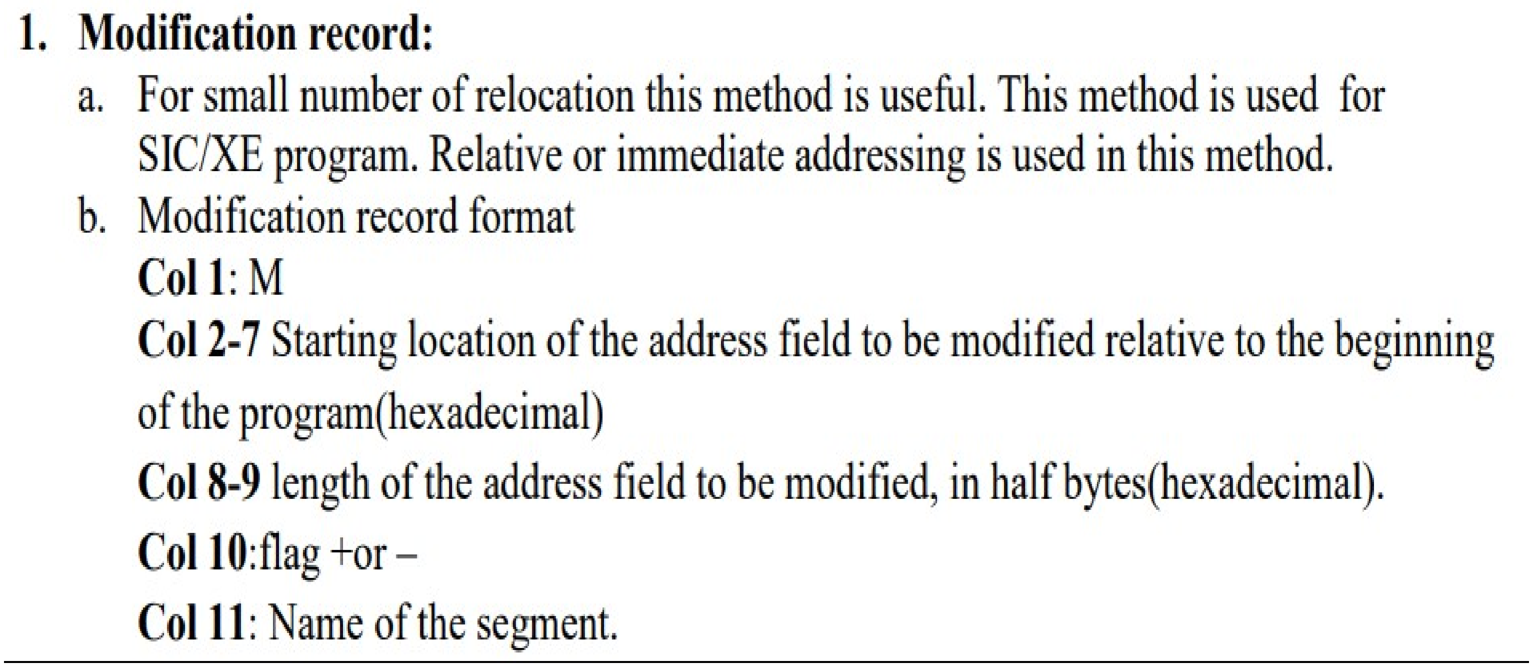
Relocation

- 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.

Relocation

- There are two methods for specifying relocation as a part of object program and those are
- 1. Modification record (SIC/XE program)
- 2. Relocation bits(SIC program)

Relocation-Modification Record



1. **Modification record:**
   a. For small number of relocation this method is useful. This method is used for SIC/XE program. Relative or immediate addressing is used in this method.
   b. Modification record format

      Col 1: M

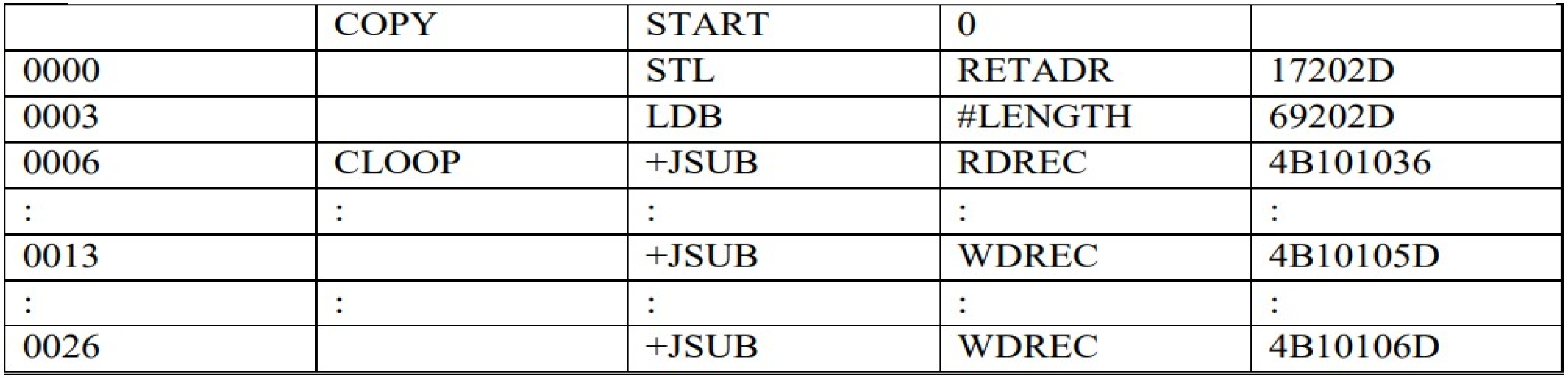
      Col 2-7 Starting location of the address field to be modified relative to the beginning of the program(hexadecimal)

      Col 8-9 length of the address field to be modified, in half bytes(hexadecimal).

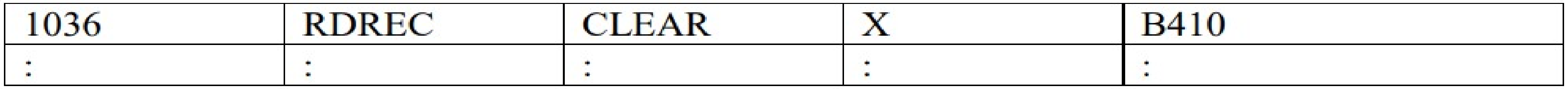
      Col 10:flag +or –

      Col 11: Name of the segment.

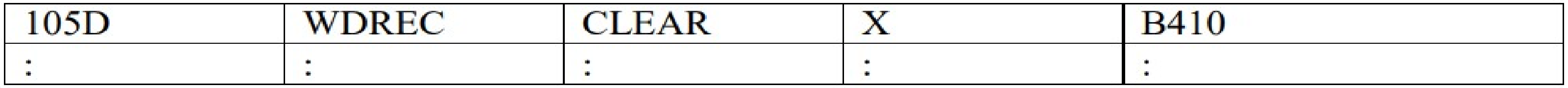
Relocation-Modification Record



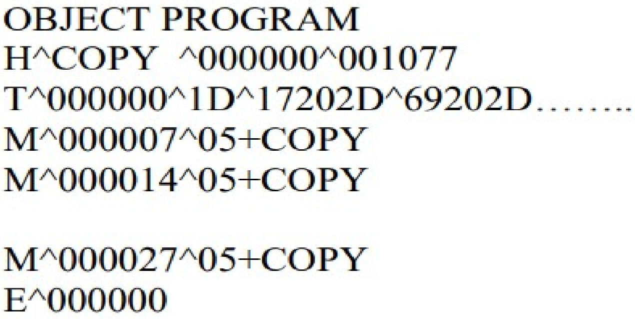
	COPY	START	0	
0000		STL	RETADR	17202D
0003		LDB	#LENGTH	69202D
0006	CLOOP	+JSUB	RDREC	4B101036
:	:	:	:	:
0013		+JSUB	WDREC	4B10105D
:	:	:	:	:
0026		+JSUB	WDREC	4B10106D



1036	RDREC	CLEAR	X	B410
:	:	:	:	:



105D	WDREC	CLEAR	X	B410
:	:	:	:	:



```
OBJECT PROGRAM
H^COPY  ^000000^001077
T^000000^1D^17202D^69202D……..
M^000007^05+COPY
M^000014^05+COPY

M^000027^05+COPY
E^000000
```

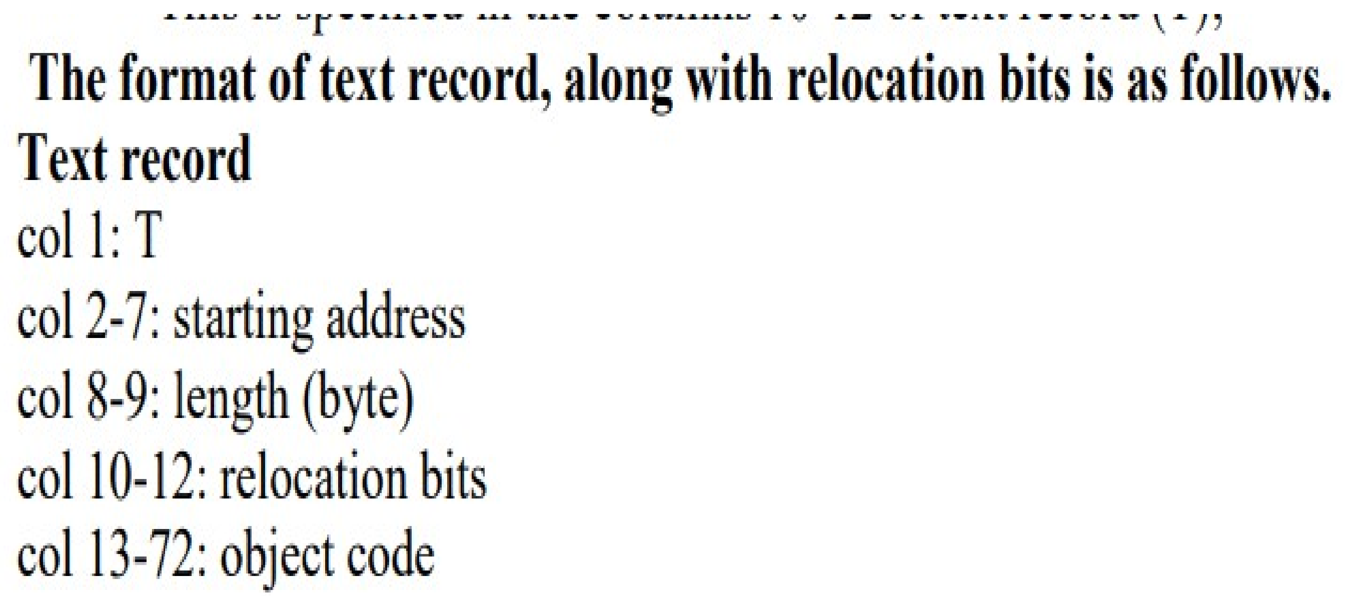
Disadvantage: This method is not well suited for SIC program . because these programs will require lot of modified records and then the size of object program will get increased

Relocation Bits

- The relocation bit method is used for simple machines. **Relocation bit is 0: no modification is necessary,**

  1: modification is needed.

This is specified in the columns 10-12 of text record (T).



This is specified in the columns 10-12 of text record (T),

The format of text record, along with relocation bits is as follows.

Text record

col 1: T

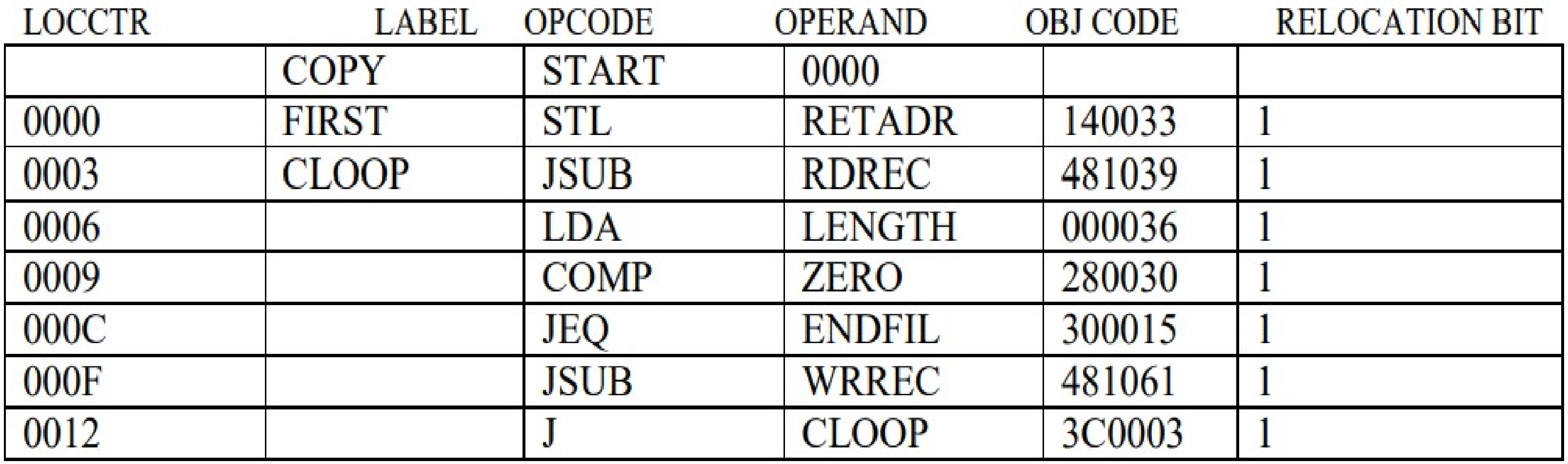
col 2-7: starting address

col 8-9: length (byte)

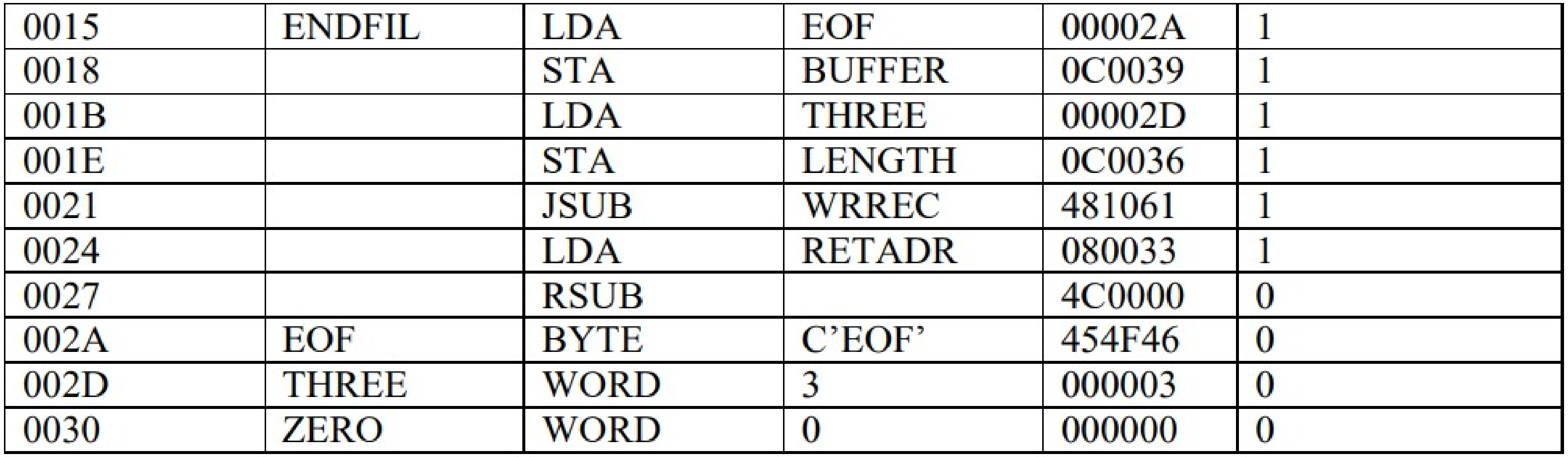
col 10-12: relocation bits

col 13-72: object code

Relocation Bits

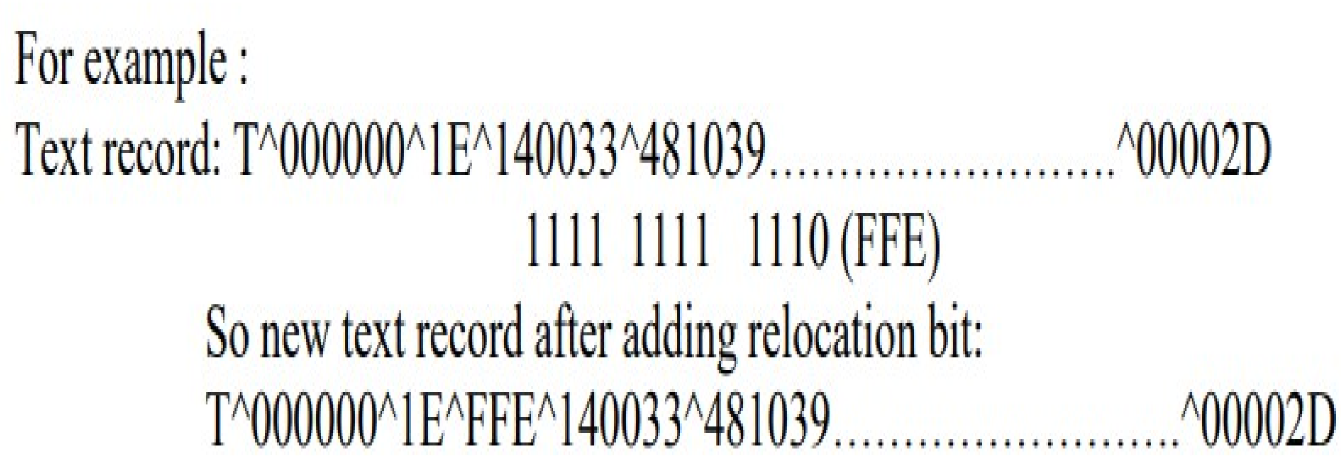


LOCCTR	LABEL	OPCODE	OPERAND	OBJ CODE	RELOCATION BIT
	COPY	START	0000		
0000	FIRST	STL	RETADR	140033	1
0003	CLOOP	JSUB	RDREC	481039	1
0006		LDA	LENGTH	000036	1
0009		COMP	ZERO	280030	1
000C		JEQ	ENDFIL	300015	1
000F		JSUB	WRREC	481061	1
0012		J	CLOOP	3C0003	1



0015	ENDFIL	LDA	EOF	00002A	1
0018		STA	BUFFER	0C0039	1
001B		LDA	THREE	00002D	1
001E		STA	LENGTH	0C0036	1
0021		JSUB	WRREC	481061	1
0024		LDA	RETADR	080033	1
0027		RSUB		4C0000	0
002A	EOF	BYTE	C’EOF’	454F46	0
002D	THREE	WORD	3	000003	0
0030	ZERO	WORD	0	000000	0

Relocation Bits



For example :

Text record: T^000000^1E^140033^481039……………………..^00002D

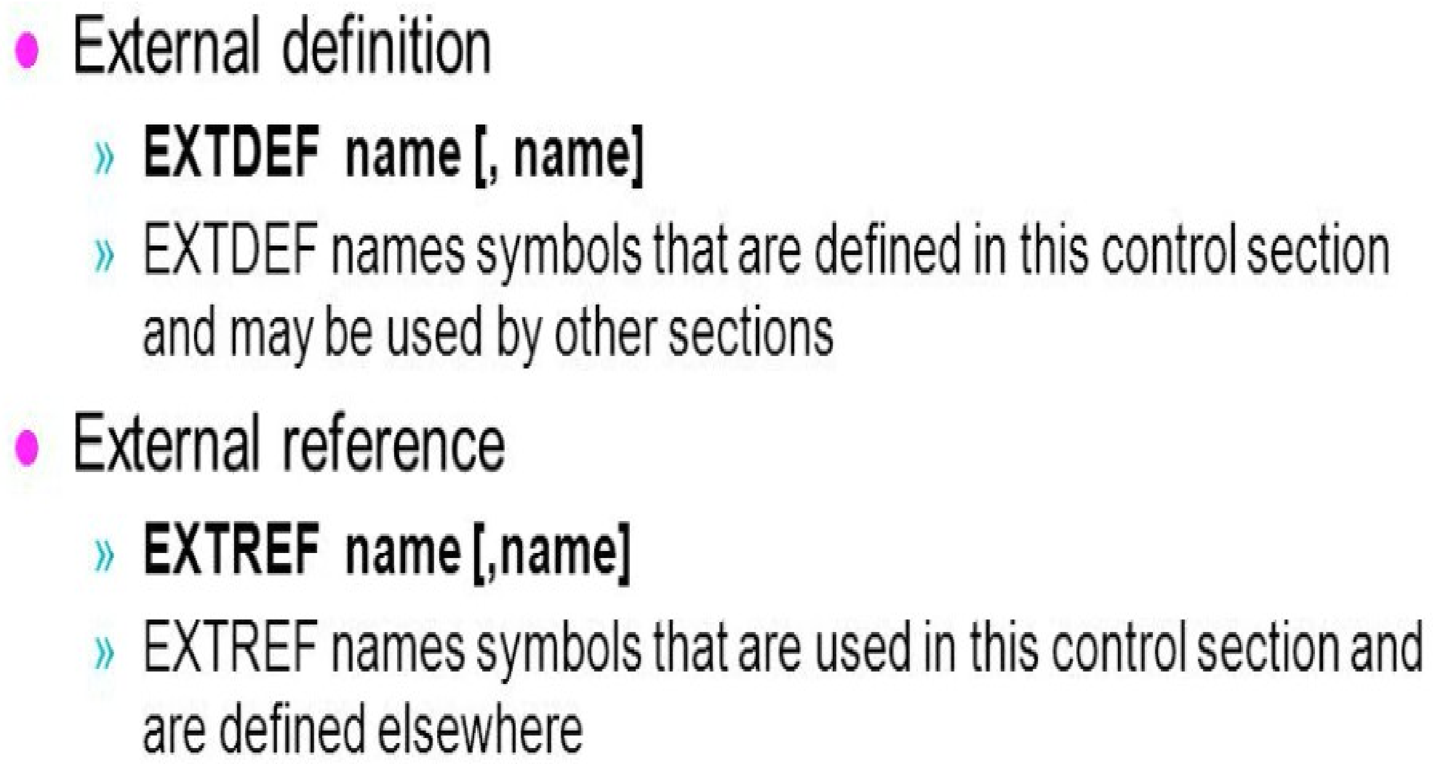
1111 1111 1110 (FFE)

So new text record after adding relocation bit:

T^000000^1E^FFE^140033^481039……………………..^00002D

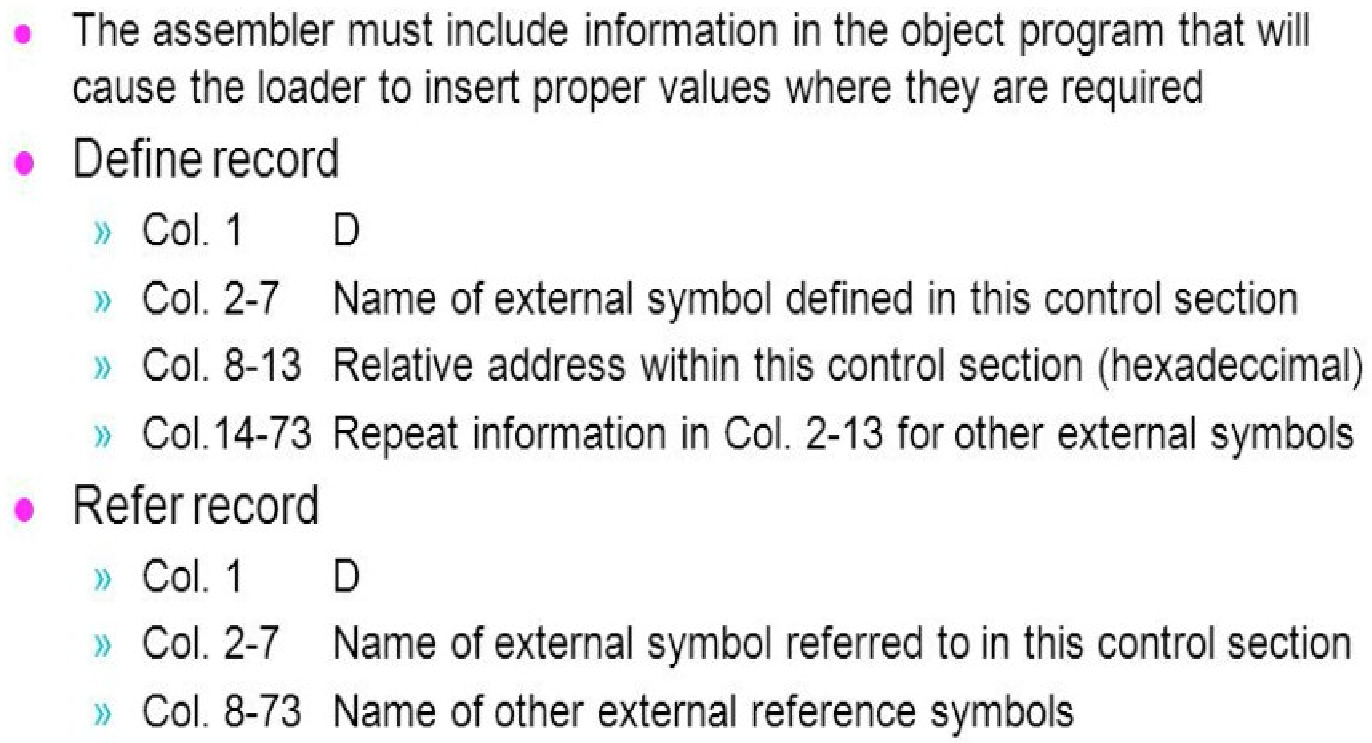
- Control sections
  - can be loaded and relocated independently of the others
  - are most often used for subroutines or other logical subdivisions of a program
  - **secname CSECT**
  - the programmer can assemble, load, and manipulate each of these control sections separately
  - because of this, there should be some means for linking control sections together
  - example: instruction in one control section may need to refer to instructions or data located in another section

EXTDEF and EXTREF directives



- External definition
  - **EXTDEF name [, name]**
  - EXTDEF names symbols that are defined in this control section and may be used by other sections
- External reference
  - **EXTREF name [,name]**
  - EXTREF names symbols that are used in this control section and are defined elsewhere

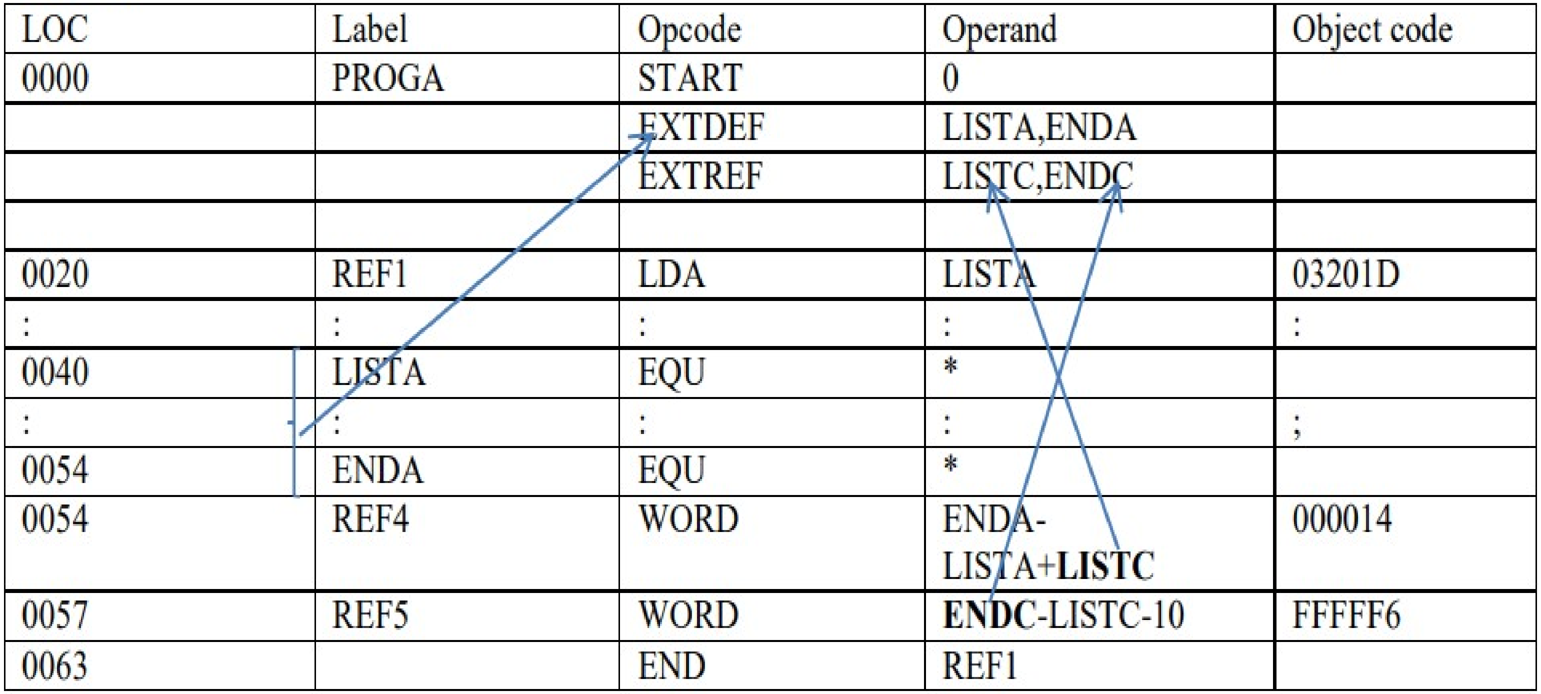
Define and Refer Record



- The assembler must include information in the object program that will cause the loader to insert proper values where they are required
- Define record
  - » Col. 1 D
  - » Col. 2-7 Name of external symbol defined in this control section
  - » Col. 8-13 Relative address within this control section (hexadeccimal)
  - » Col.14-73 Repeat information in Col. 2-13 for other external symbols
- Refer record
  - » Col. 1 D
  - » Col. 2-7 Name of external symbol referred to in this control section
  - » Col. 8-73 Name of other external reference symbols

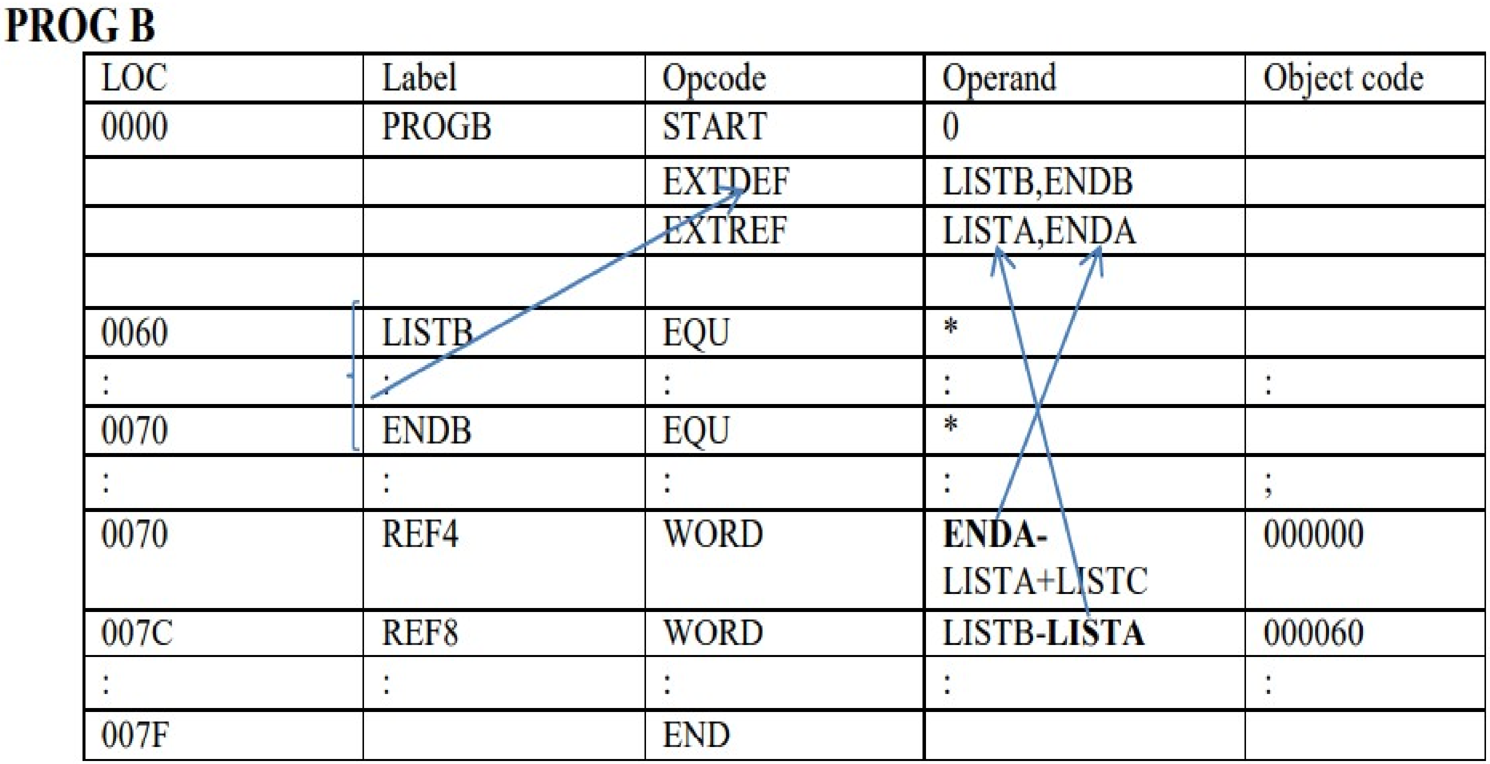
Relocation and Linking

PROG A



LOC	Label	Opcode	Operand	Object code
0000	PROGA	START	0	
		EXTDEF	LISTA,ENDA	
		EXTREF	LISTC,ENDC	
0020	REF1	LDA	LISTA	03201D
:	:	:	:	:
0040	LISTA	EQU	*	
:	:	:	:	;
0054	ENDA	EQU	*	
0054	REF4	WORD	ENDA- LISTA+**LISTC**	000014
0057	REF5	WORD	**ENDC**-LISTC-10	FFFFF6
0063		END	REF1	

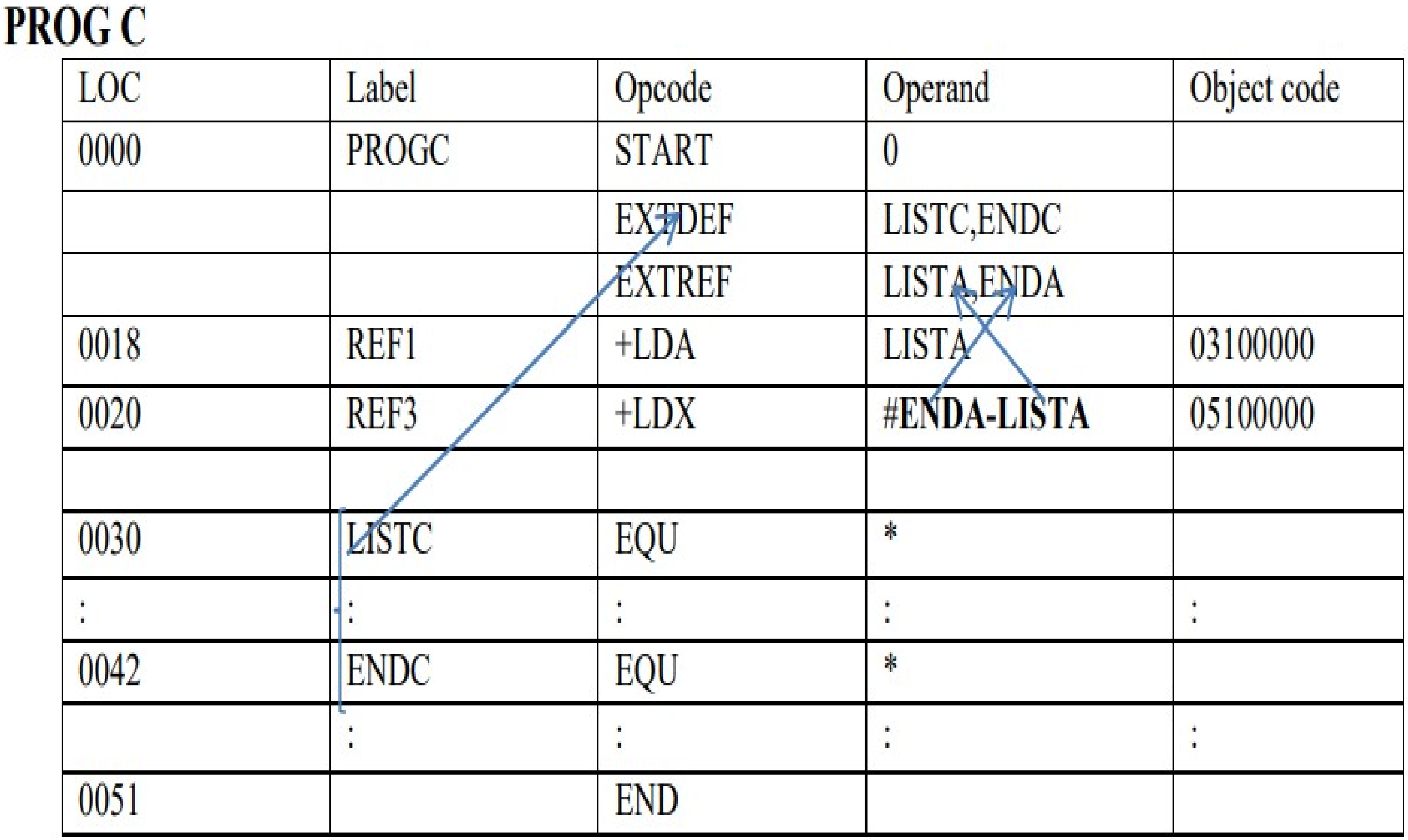
Relocation and Linking



PROG B

LOC	Label	Opcode	Operand	Object code
0000	PROGB	START	0	
		EXTDEF	LISTB,ENDB	
		EXTREF	LISTA,ENDA	
0060	LISTB	EQU	*	
:	:	:	:	:
0070	ENDB	EQU	*	
:	:	:	:	;
0070	REF4	WORD	**ENDA-** LISTA+LISTC	000000
007C	REF8	WORD	LISTB-**LISTA**	000060
:	:	:	:	:
007F		END		

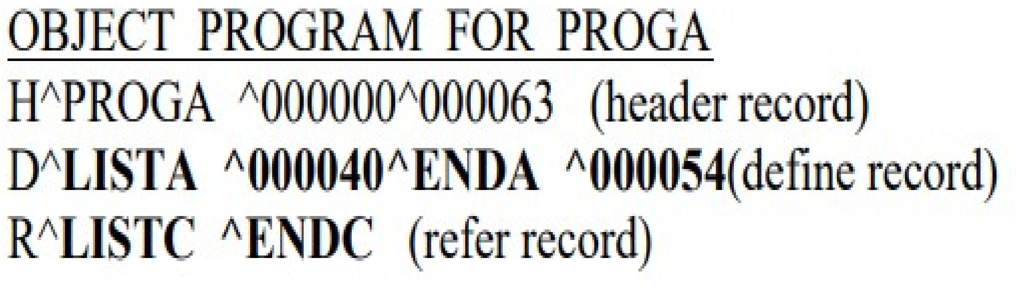
Relocation and Linking



PROG C

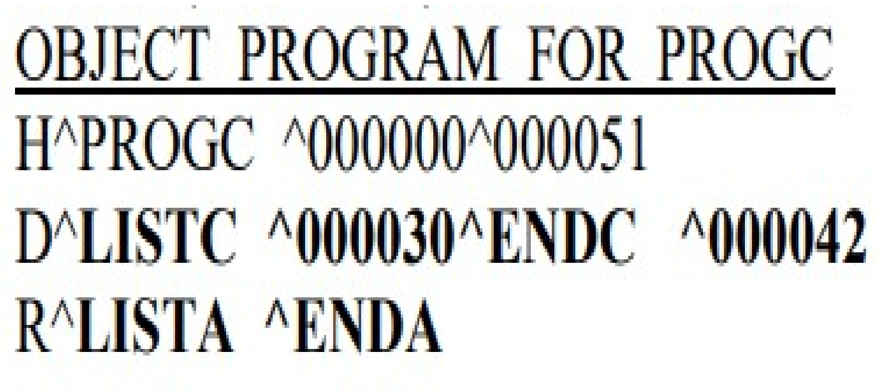
LOC	Label	Opcode	Operand	Object code
0000	PROGC	START	0	
		EXTDEF	LISTC,ENDC	
		EXTREF	LISTA,ENDA	
0018	REF1	+LDA	LISTA	03100000
0020	REF3	+LDX	**#ENDA-LISTA**	05100000
0030	LISTC	EQU	*	
:	:	:	:	:
0042	ENDC	EQU	*	
	:	:	:	:
0051		END		

Define and Refer record- Example



OBJECT PROGRAM FOR PROGA

```
H^PROGA  ^000000^000063   (header record)
D^LISTA  ^000040^ENDA  ^000054(define record)
R^LISTC  ^ENDC   (refer record)
```

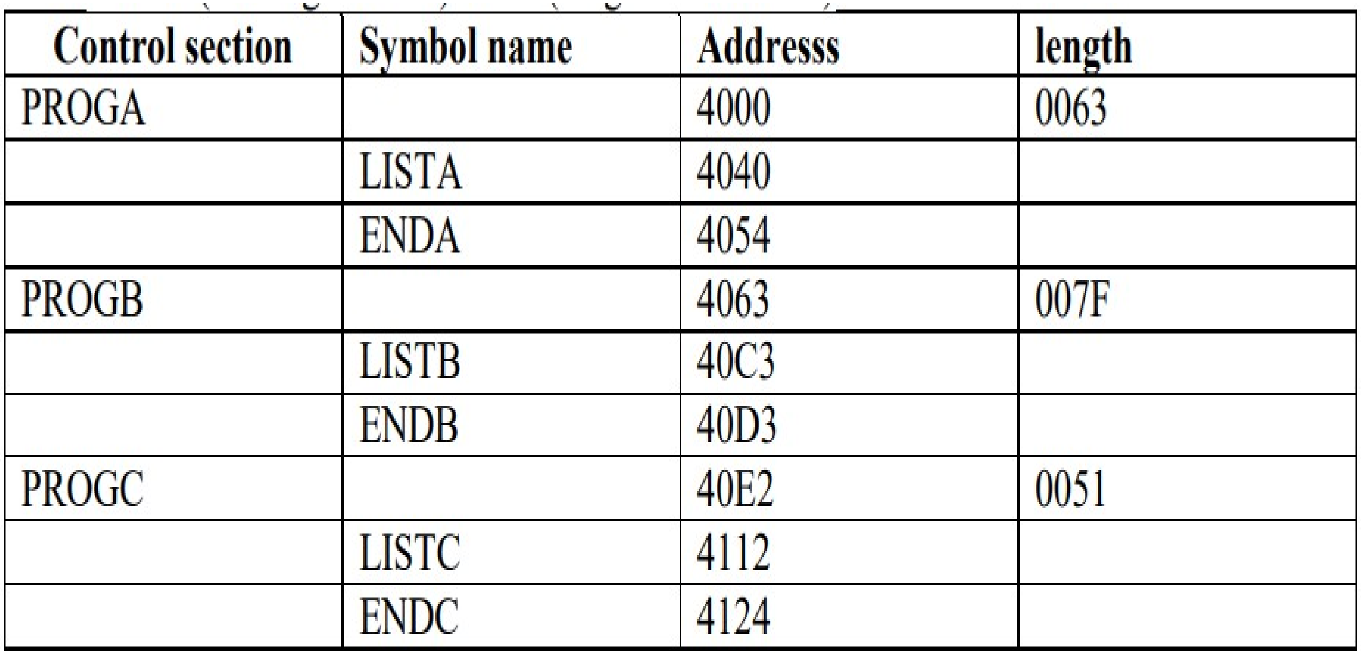


OBJECT PROGRAM FOR PROGC

```
H^PROGC  ^000000^000051
D^LISTC  ^000030^ENDC   ^000042
R^LISTA  ^ENDA
```

Relocation and Linking

- PROGA**4000**(Starting address)+**0063**(length of PROGA)->**4063**
- PROGB**4063**(staring address)+**007F**(length of PROGB)**40E2**
- PROGC**40E2**(starting address)+**0051**(length of PROGC)**4133**



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

Data Structures for Linking Loader

- **Data Structures:**

1.ESTAB:External Symbol Table

- This table is used to store the name and address of each external symbol in the set of control sections being loaded.
- The table also often indicates in which control sections the symbol is defined.
- A hashed organization is typically used for this table.

Data Structures for Linking Loader

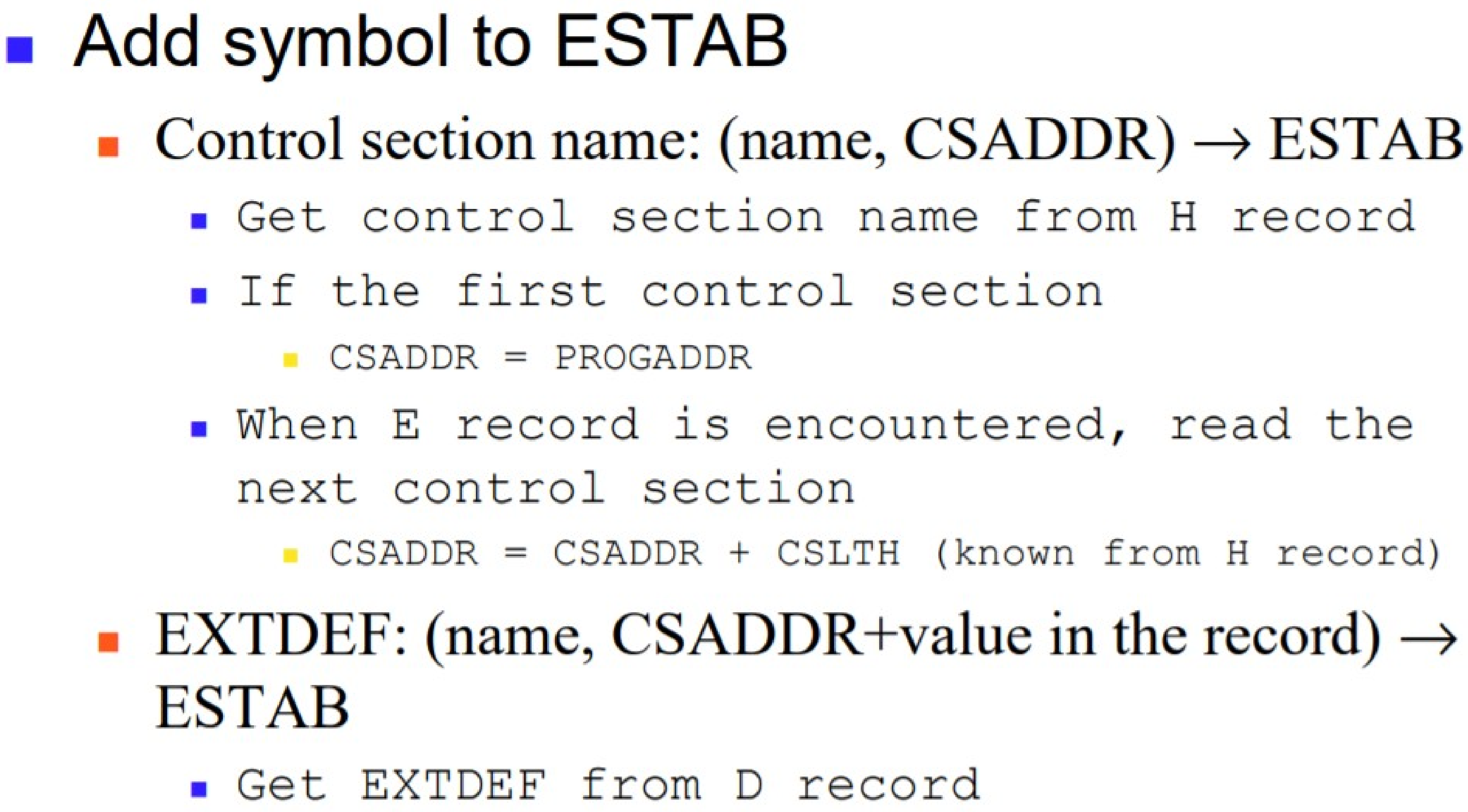
2. PROGADDR: Program load address

- It is the beginning address in memory where the linked program is to be loaded.
- Its value is supplied by OS to the loader.

3. CSADDR : Control Section address

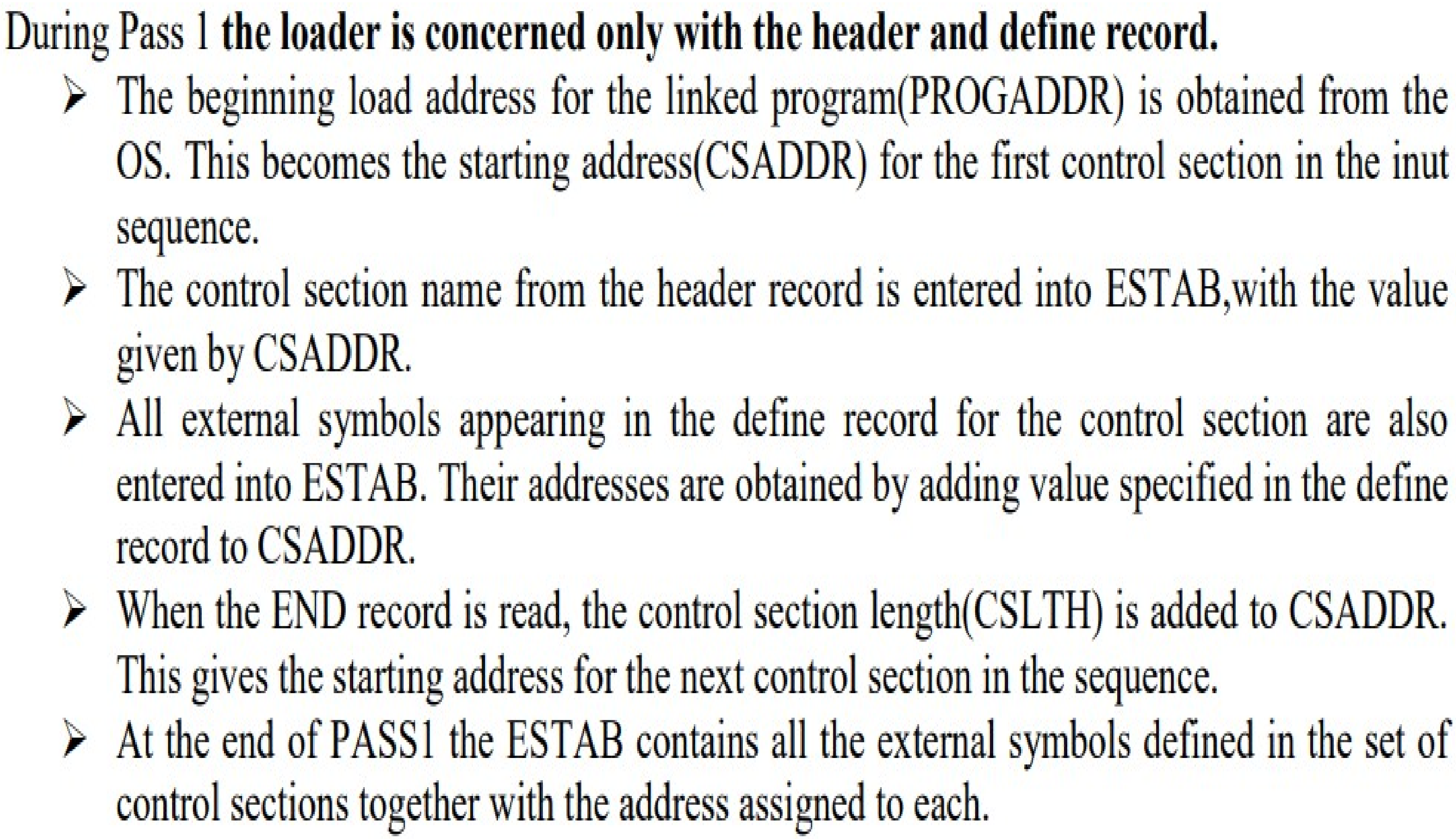
- Contains the starting address assigned to control sections currently being scanned by the loader.
- This value is added to all relative addresses within the control section to convert them to actual addresses.

Pass 1 of Linking loader



- Add symbol to ESTAB
  - Control section name: (name, CSADDR) → ESTAB
    - Get control section name from H record
    - If the first control section
      - CSADDR = PROGADDR
    - When E record is encountered, read the next control section
      - CSADDR = CSADDR + CSLTH (known from H record)
  - EXTDEF: (name, CSADDR+value in the record) → ESTAB
    - Get EXTDEF from D record

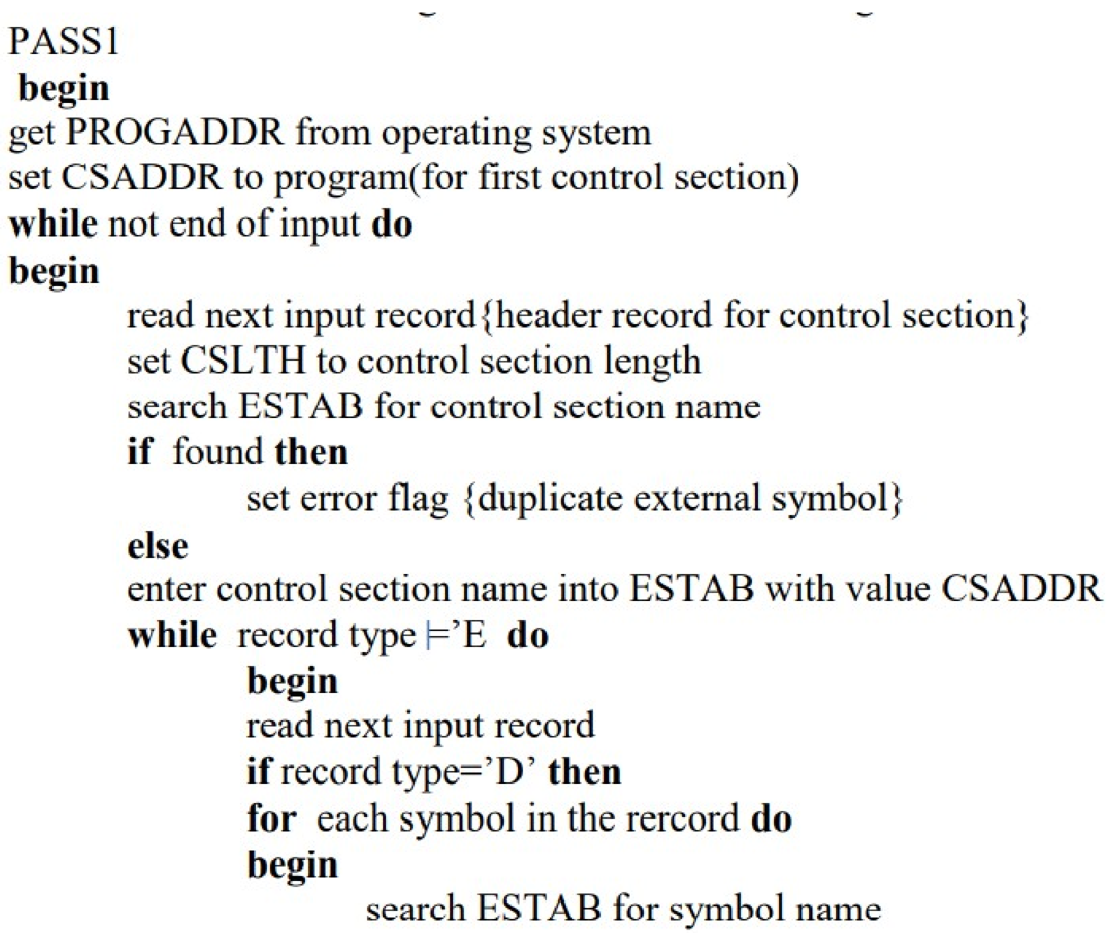
Pass 1 of a Linking Loader



During Pass 1 **the loader is concerned only with the header and define record.**

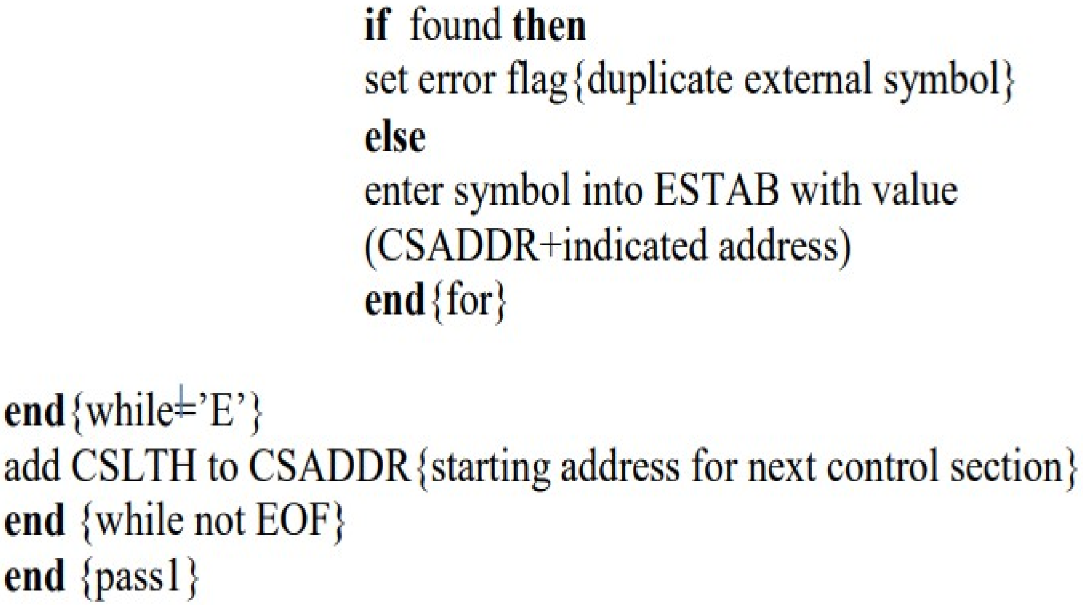
- The beginning load address for the linked program(PROGADDR) is obtained from the OS. This becomes the starting address(CSADDR) for the first control section in the inut sequence.
- The control section name from the header record is entered into ESTAB,with the value given by CSADDR.
- All external symbols appearing in the define record for the control section are also entered into ESTAB. Their addresses are obtained by adding value specified in the define record to CSADDR.
- When the END record is read, the control section length(CSLTH) is added to CSADDR. This gives the starting address for the next control section in the sequence.
- At the end of PASS1 the ESTAB contains all the external symbols defined in the set of control sections together with the address assigned to each.

Pass 1 of a Linking Loader



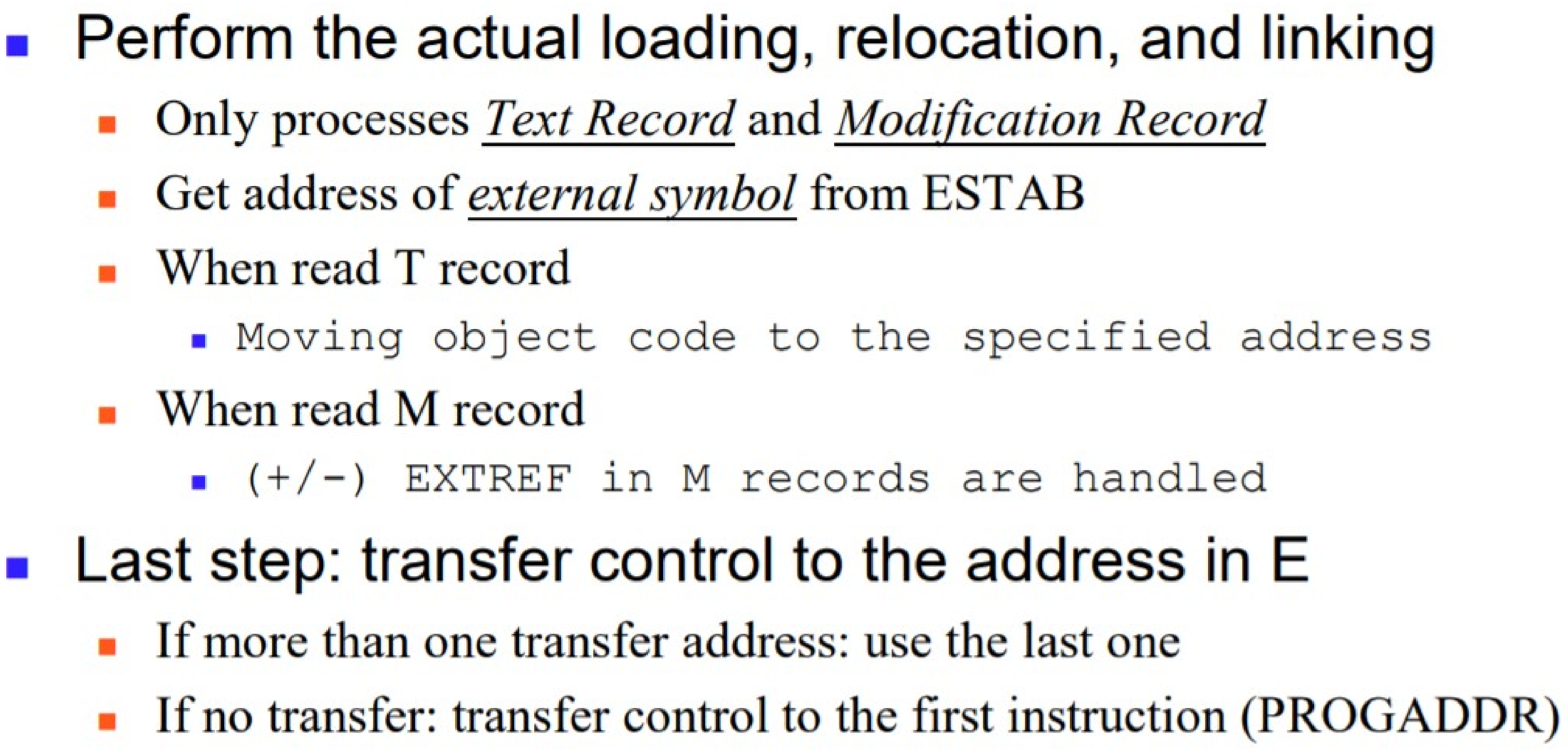
```
PASS1
 begin
get PROGADDR from operating system
set CSADDR to program(for first control section)
while not end of input do
begin
        read next input record{header record for control section}
        set CSLTH to control section length
        search ESTAB for control section name
        if  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
                for  each symbol in the rercord do
                begin
                        search ESTAB for symbol name
```

Pass 1 of a Linking Loader



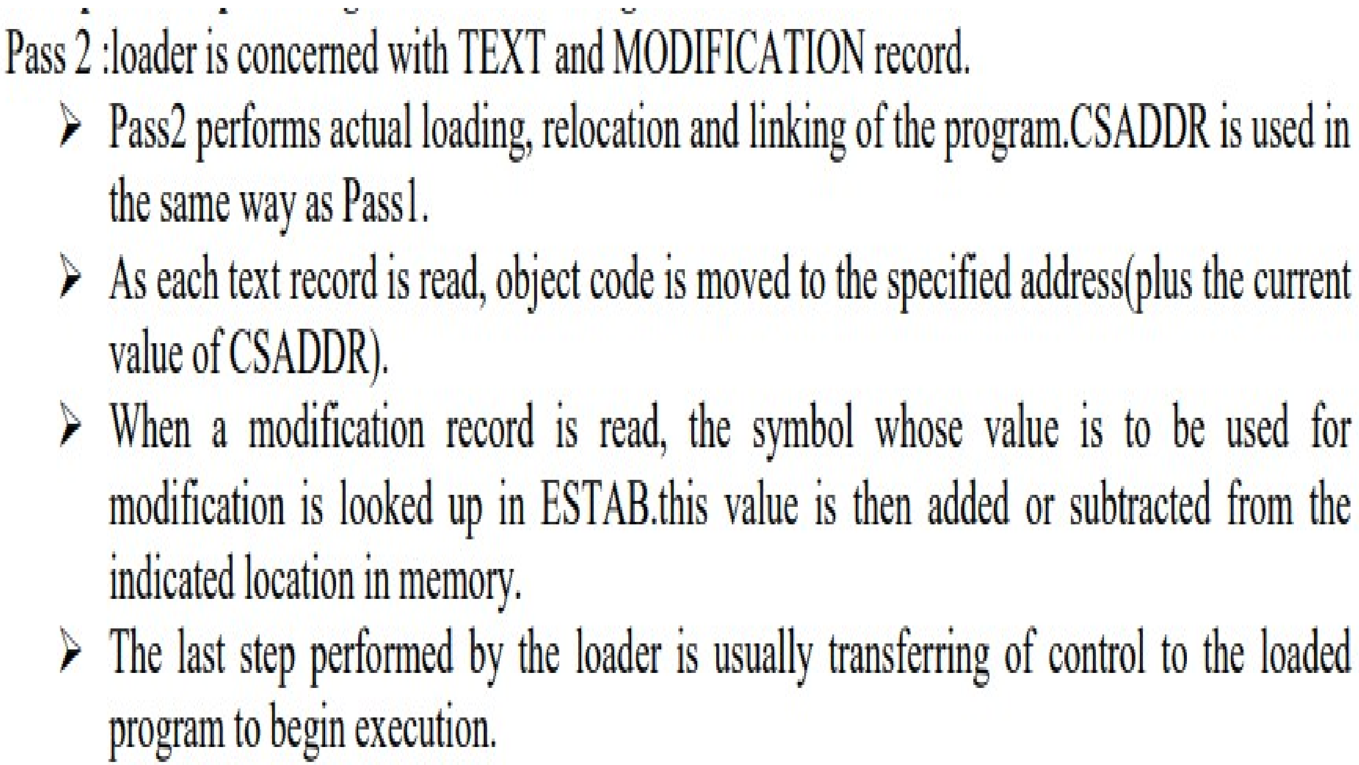
```
                    if found then
                    set error flag{duplicate external symbol}
                    else
                    enter symbol into ESTAB with value
                    (CSADDR+indicated address)
                    end{for}

end{while='E'}
add CSLTH to CSADDR{starting address for next control section}
end {while not EOF}
end {pass1}
```

Pass 2 of a Linking Loader



- Perform the actual loading, relocation, and linking
  - Only processes *Text Record* and *Modification Record*
  - Get address of *external symbol* from ESTAB
  - When read T record
    - Moving object code to the specified address
  - When read M record
    - (+/-) EXTREF in M records are handled
- Last step: transfer control to the address in E
  - If more than one transfer address: use the last one
  - If no transfer: transfer control to the first instruction (PROGADDR)

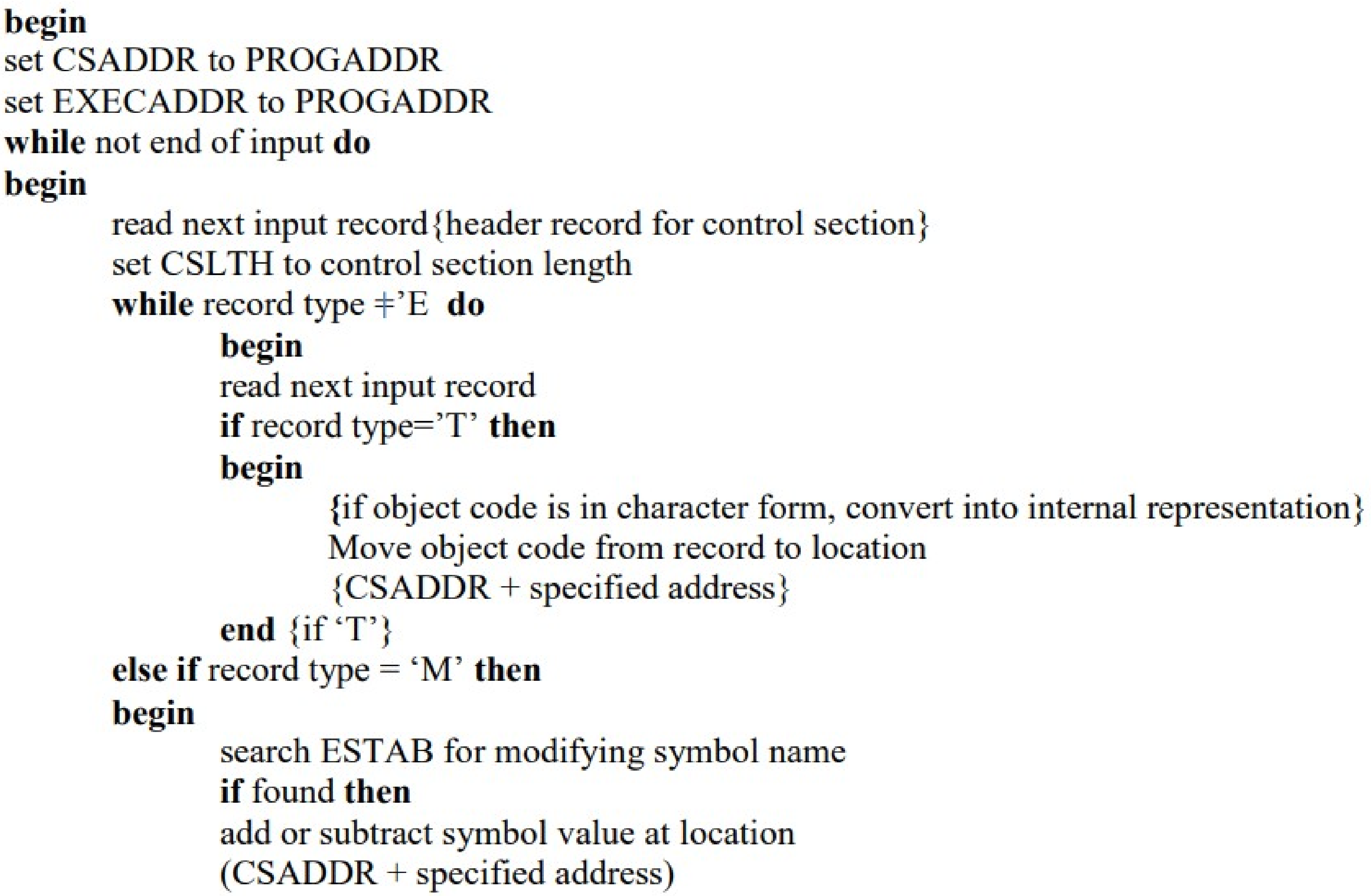
Pass 2 of a Linking Loader



Pass 2 :loader is concerned with TEXT and MODIFICATION record.

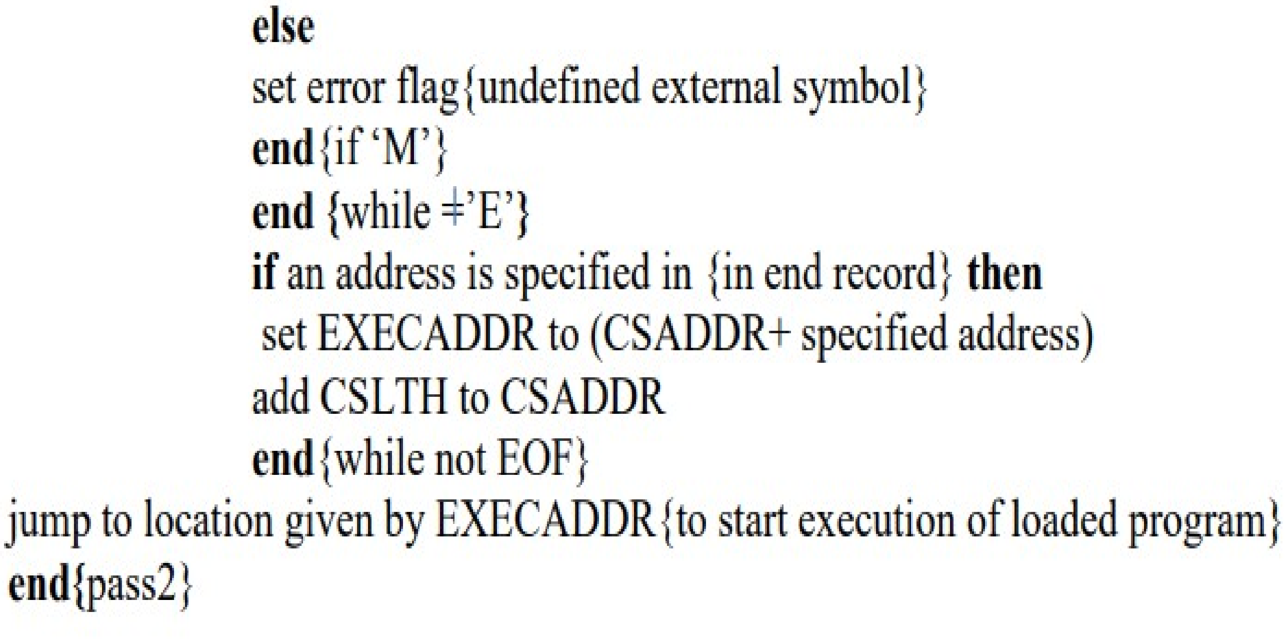
- Pass2 performs actual loading, relocation and linking of the program.CSADDR is used in the same way as Pass1.
- As each text record is read, object code is moved to the specified address(plus the current value of CSADDR).
- When a modification record is read, the symbol whose value is to be used for modification is looked up in ESTAB.this value is then added or subtracted from the indicated location in memory.
- The last step performed by the loader is usually transferring of control to the loaded program to begin execution.

Pass 2 of a Linking Loader



```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
begin
        read next input record{header record for control section}
        set CSLTH to control section length
        while record type ≠’E  do
                begin
                read next input record
                if record type=’T’ then
                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
        begin
                search ESTAB for modifying symbol name
                if found then
                add or subtract symbol value at location
                (CSADDR + specified address)
```

Pass 2 of a Linking Loader

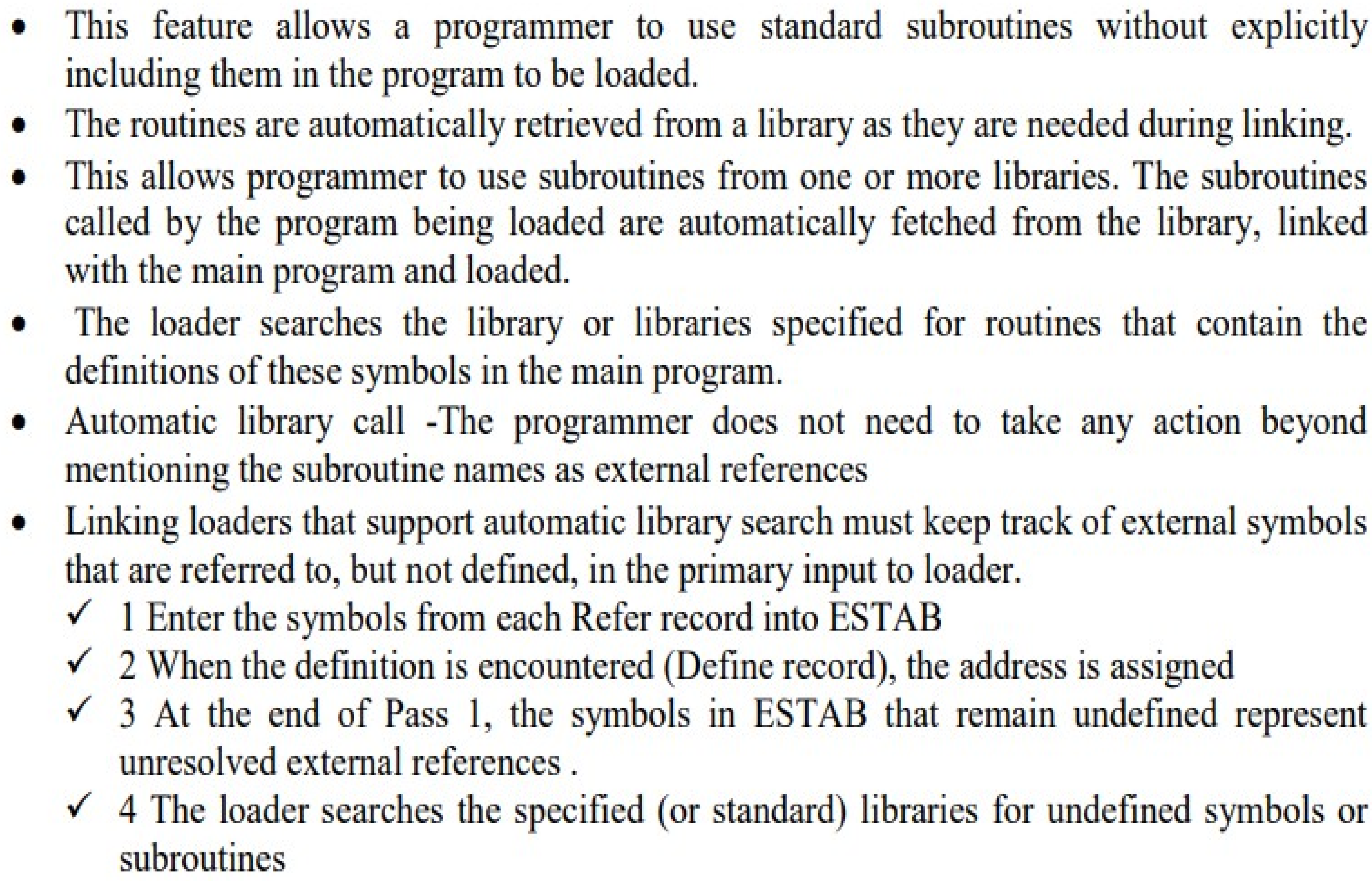


```
          else
          set error flag{undefined external symbol}
          end{if 'M'}
          end {while ≠'E'}
          if an address is specified in {in end record} then
           set EXECADDR to (CSADDR+ specified address)
          add CSLTH to CSADDR
          end{while not EOF}
jump to location given by EXECADDR{to start execution of loaded program}
end{pass2}
```

MACHINE INDEPENDENT LOADER FEATURES

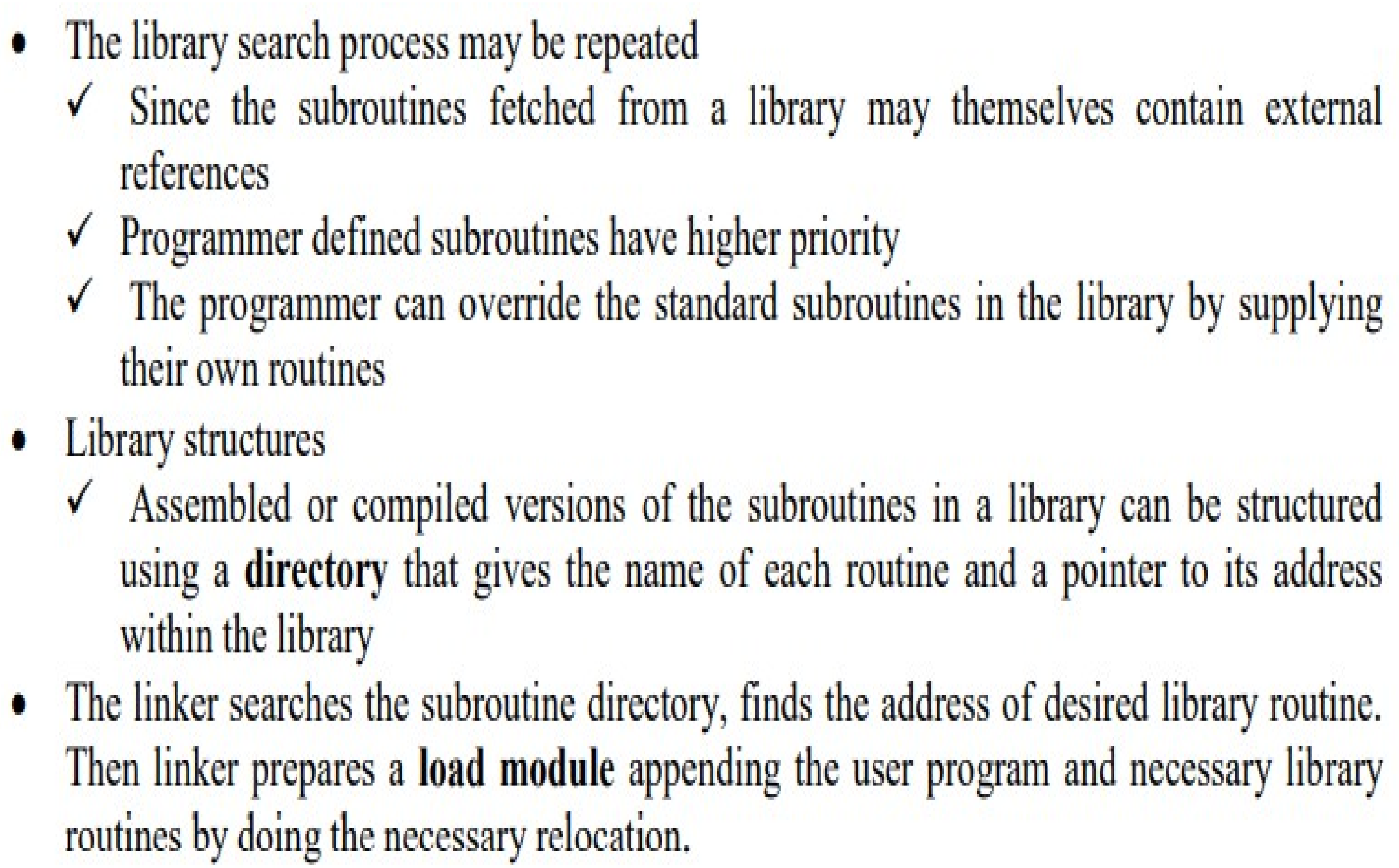
1. AUTOMATIC LIBRARY SEARCH
2. USE OF LOADER OPTIONS

Automatic Library Search



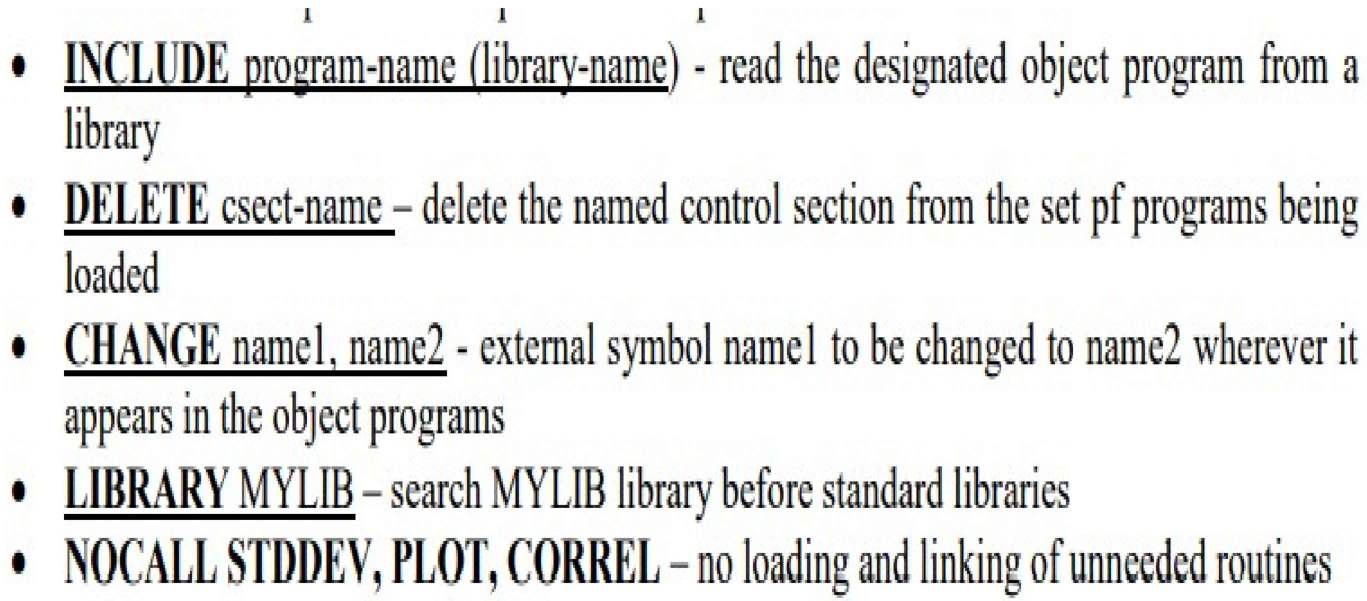
- This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded.
- The routines are automatically retrieved from a library as they are needed during linking.
- This allows programmer to use subroutines from one or more libraries. The subroutines 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 call -The programmer does not need to take any action beyond mentioning the subroutine names as external references
- Linking loaders that support automatic library search must keep track of external symbols that are referred to, but not defined, in the primary input to loader.
  - ✓ 1 Enter the symbols from each Refer record into ESTAB
  - ✓ 2 When the definition is encountered (Define record), the address is assigned
  - ✓ 3 At the end of Pass 1, the symbols in ESTAB that remain undefined represent unresolved external references .
  - ✓ 4 The loader searches the specified (or standard) libraries for undefined symbols or subroutines

Automatic Library Search

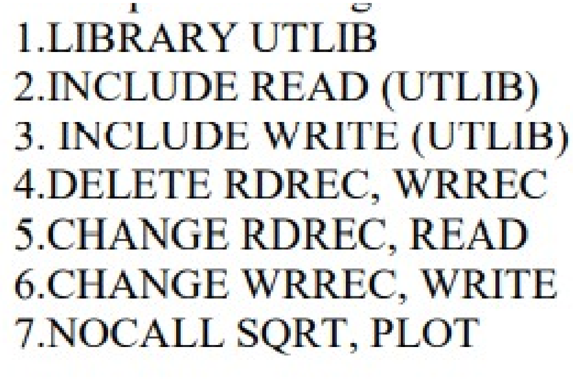


- The library search process may be repeated
  - ✓ Since the subroutines fetched from a library may themselves contain external references
  - ✓ Programmer defined subroutines have higher priority
  - ✓ The programmer can override the standard subroutines in the library by supplying their own routines
- Library structures
  - ✓ Assembled or compiled versions of the subroutines in a library can be structured using a **directory** that gives the name of each routine and a pointer to its address within the library
- The linker searches the subroutine directory, finds the address of desired library routine. Then linker prepares a **load module** appending the user program and necessary library routines by doing the necessary relocation.

Loader Option Commands

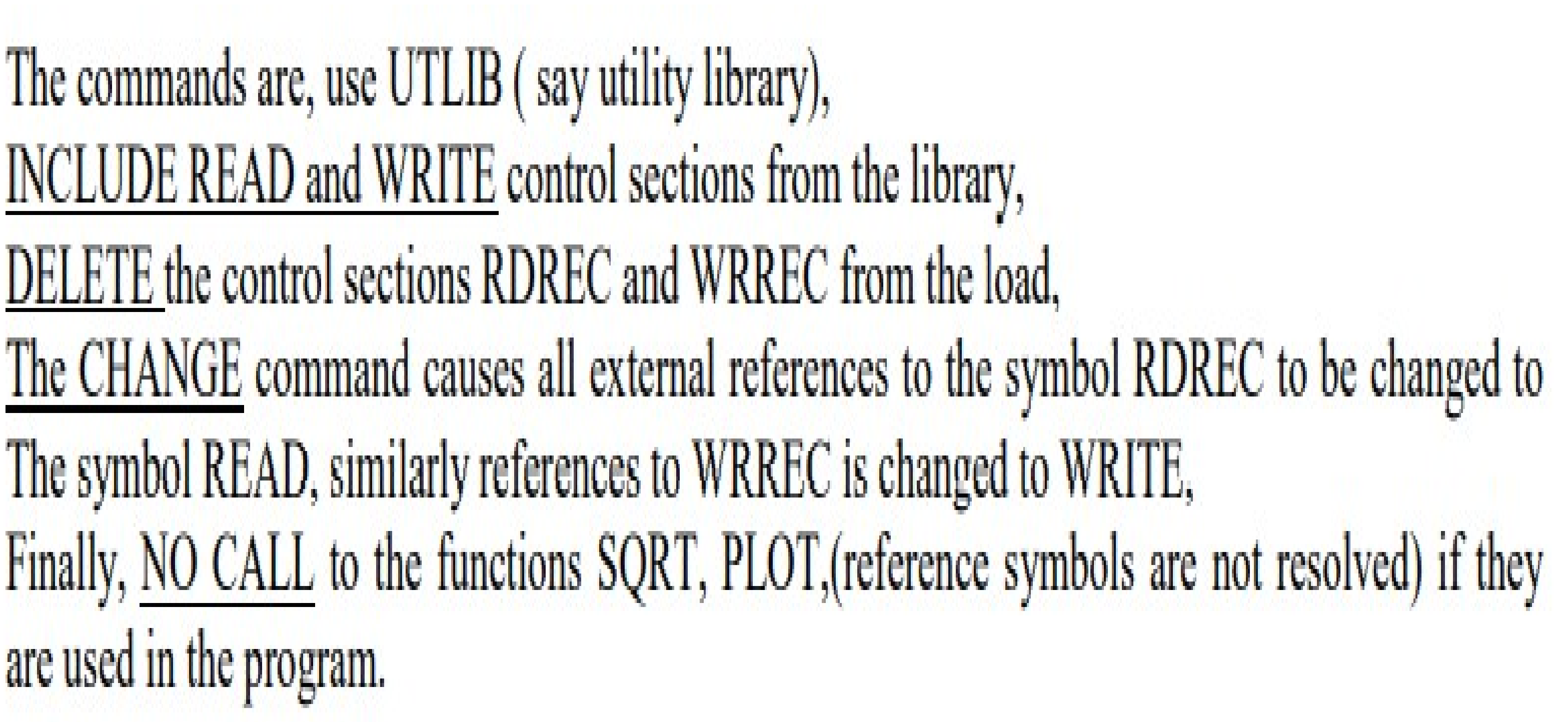


- **INCLUDE** program-name (library-name) - read the designated object program from a library
- **DELETE** csect-name – delete the named control section from the set pf programs being loaded
- **CHANGE** name1, name2 - external symbol name1 to be changed to name2 wherever it appears in the object programs
- **LIBRARY** MYLIB – search MYLIB library before standard libraries
- **NOCALL STDDEV, PLOT, CORREL** – no loading and linking of unneeded routines



1.LIBRARY UTLIB
2.INCLUDE READ (UTLIB)
3. INCLUDE WRITE (UTLIB)
4.DELETE RDREC, WRREC
5.CHANGE RDREC, READ
6.CHANGE WRREC, WRITE
7.NOCALL SQRT, PLOT

Loader Option Commands



The commands are, use UTLIB (say utility library),

INCLUDE READ and WRITE control sections from the library,

DELETE the control sections RDREC and WRREC from the load,

The CHANGE command causes all external references to the symbol RDREC to be changed to The symbol READ, similarly references to WRREC is changed to WRITE,

Finally, NO CALL to the functions SQRT, PLOT,(reference symbols are not resolved) if they are used in the program.

LOADER DESIGN OPTIONS:

- LINKAGE EDITOR
- DYNAMIC LINKAGE.

Linking Loader

- 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.

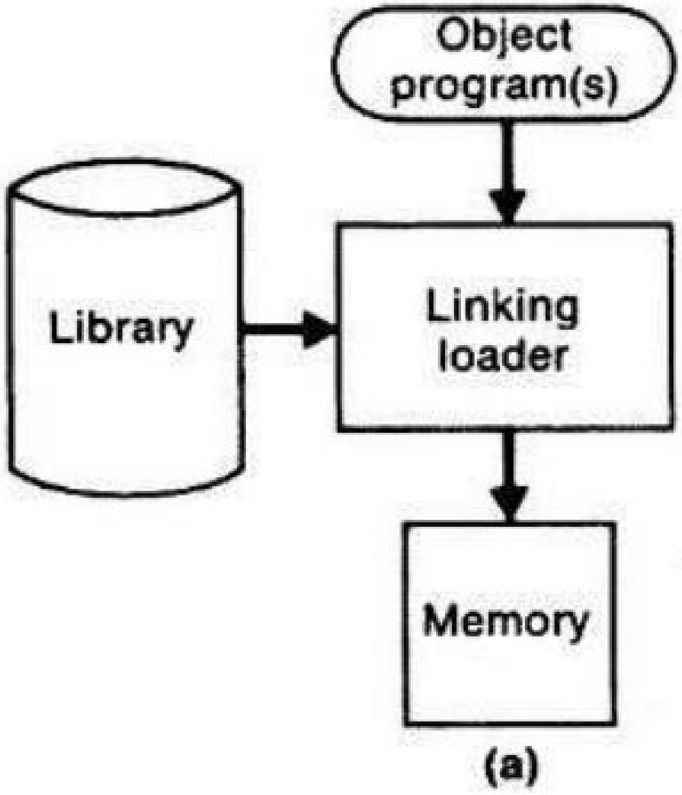
- There is no need of relocating loader.

- The linking loader searches the libraries and resolves the external references every time the program is executed.

- When program is in development stage then at that time the linking loader can be used.

- The loading requires two passes

Linking Loader



Linkage Editor

- The linkage editor produces a linked version of the program. Such a linked version is also called as load module. This load module is generally written in a file or library for later execution.

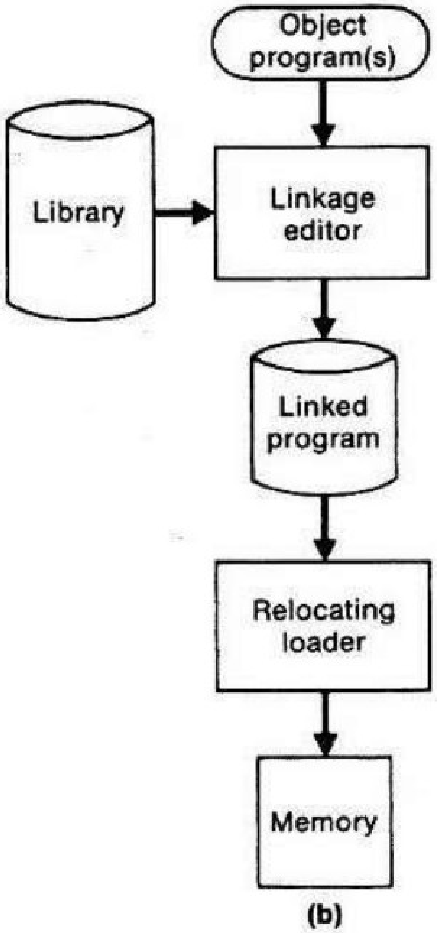
- The relocating loader loads the load module into memory.

- If the program is executed many times without being reassembled then linkage editor is the best choice.

- When program development is finished or when the library is built, then linkage editor is the best choice.

- The loading can be done in one pass.

Linkage editor



- A linkage editor produces a linked version of the program – often called a **load module or an executable image** – which is written to a file or library for later execution.
- The linked program produced is generally in a form that is suitable for processing by a relocating loader.
- Some useful functions of Linkage editor are, an absolute object program can be created, if starting address is already known.
- New versions of the library can be included without changing the source program. Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together.

- Linkage editors often allow the user to specify that external references not to be resolved by automatic library search – linking will be done later by linking loader – linkage editor + linking loader – savings in space.

Dynamic Linkage

- In overlay structure certain selective subroutines can be resident in the memory.
- That means it is not necessary to resident all the subroutines in the memory for all time.
- only necessary routines can be present in the main memory and during execution the required subroutines can be loaded in the memory.
- **The process of postponing linking and loading of external references until execution is called dynamic linking dynamic loading or load on call.**

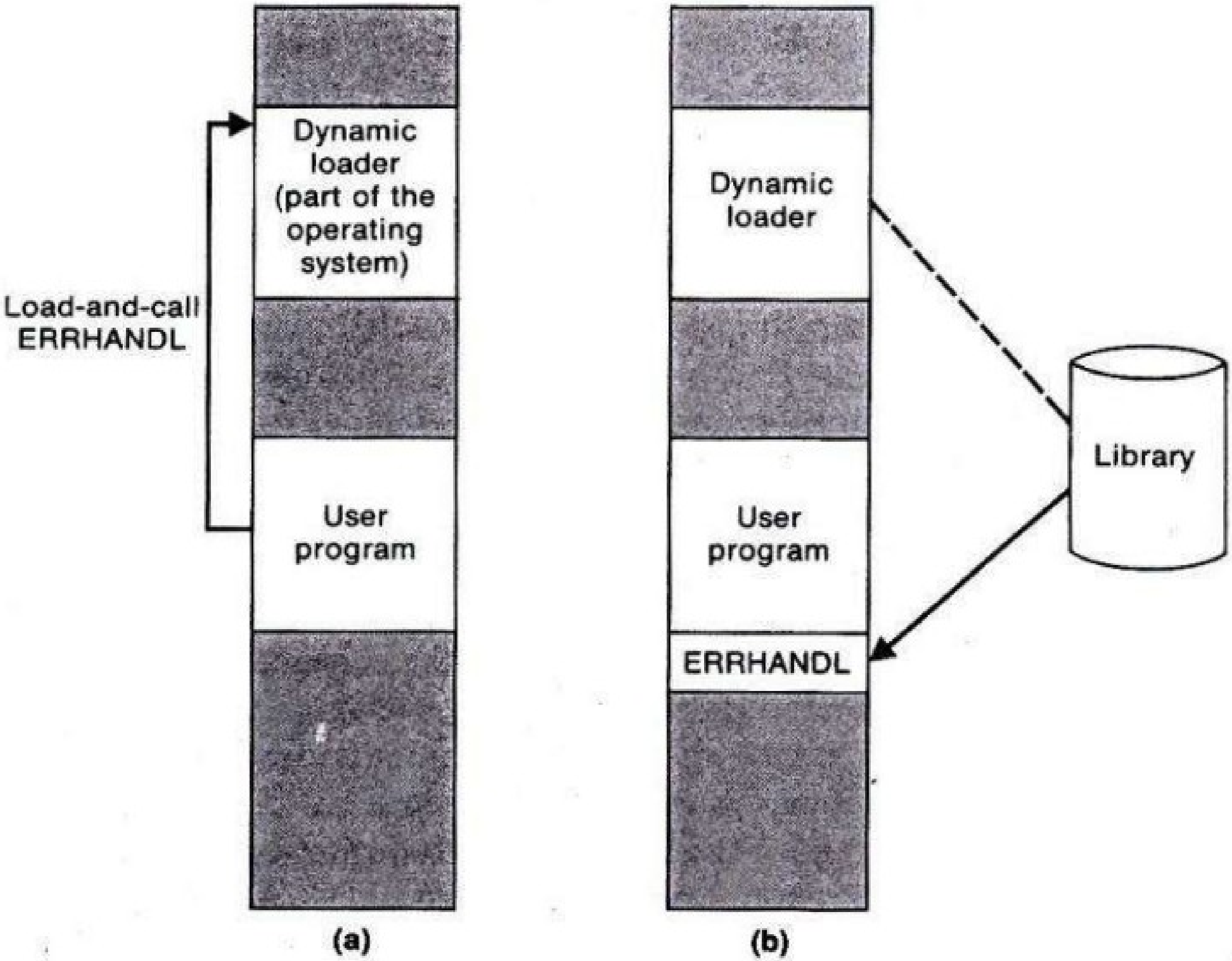
Dynamic Linking

- Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library.
- for ex run time support routines for a high level language like C could be stored in a dynamic link library.
- A single copy of the routines in this library could be loaded into the memory of the computer. All c programs currently in execution could be linked to this one copy , instead of linking a separate copy into each object program.

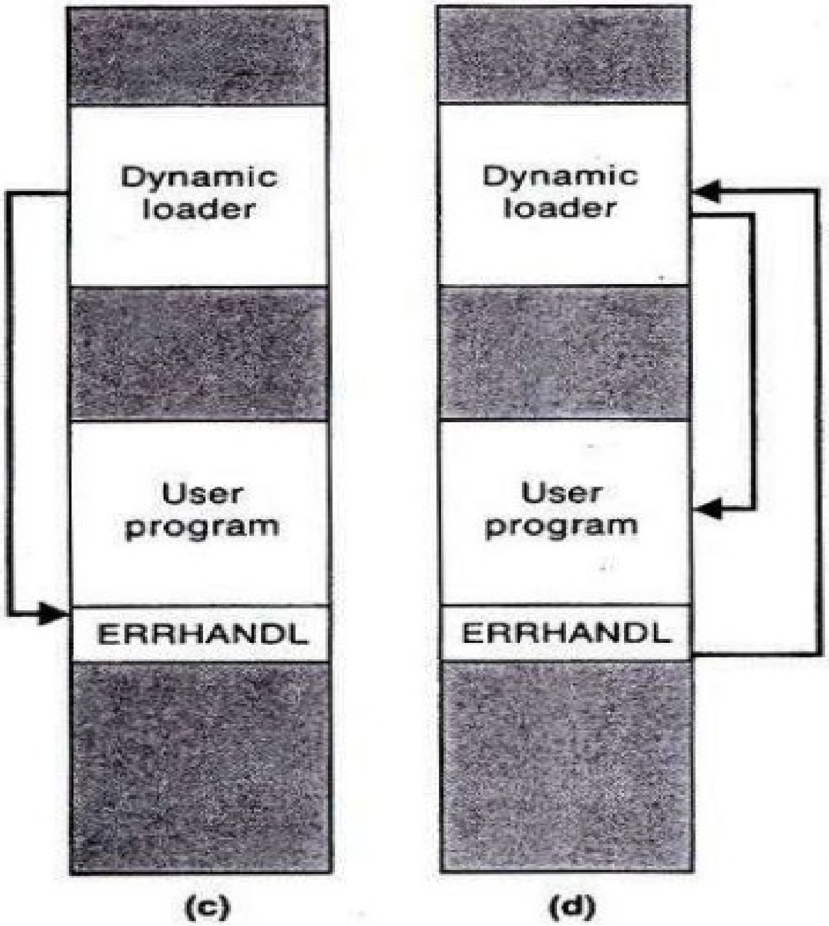
Dynamic Linking

- Suppose that in any execution, a program uses only a few out of large number of possible routines, but the exact routine needed cannot be predicted until program receives its input. dynamic linking avoids the necessity of loading the entire library for each execution.

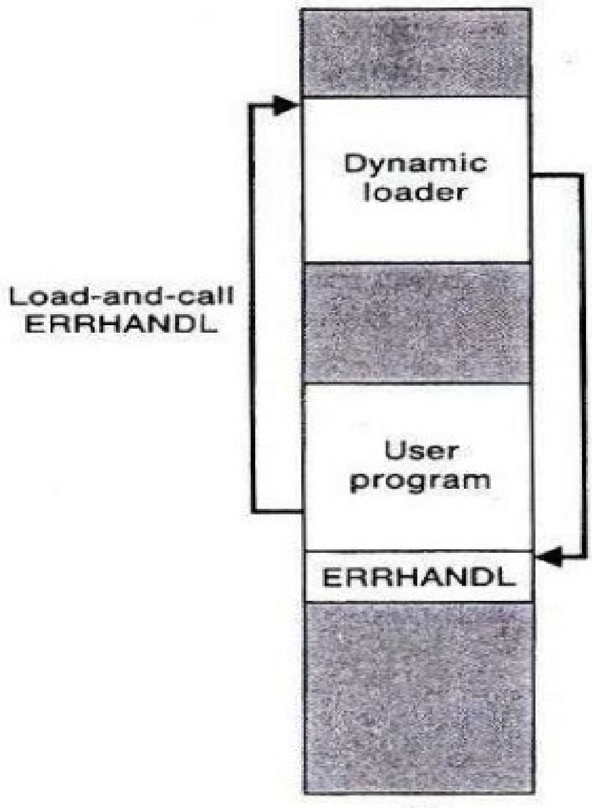
Dynamic Linking



Dynamic Linking

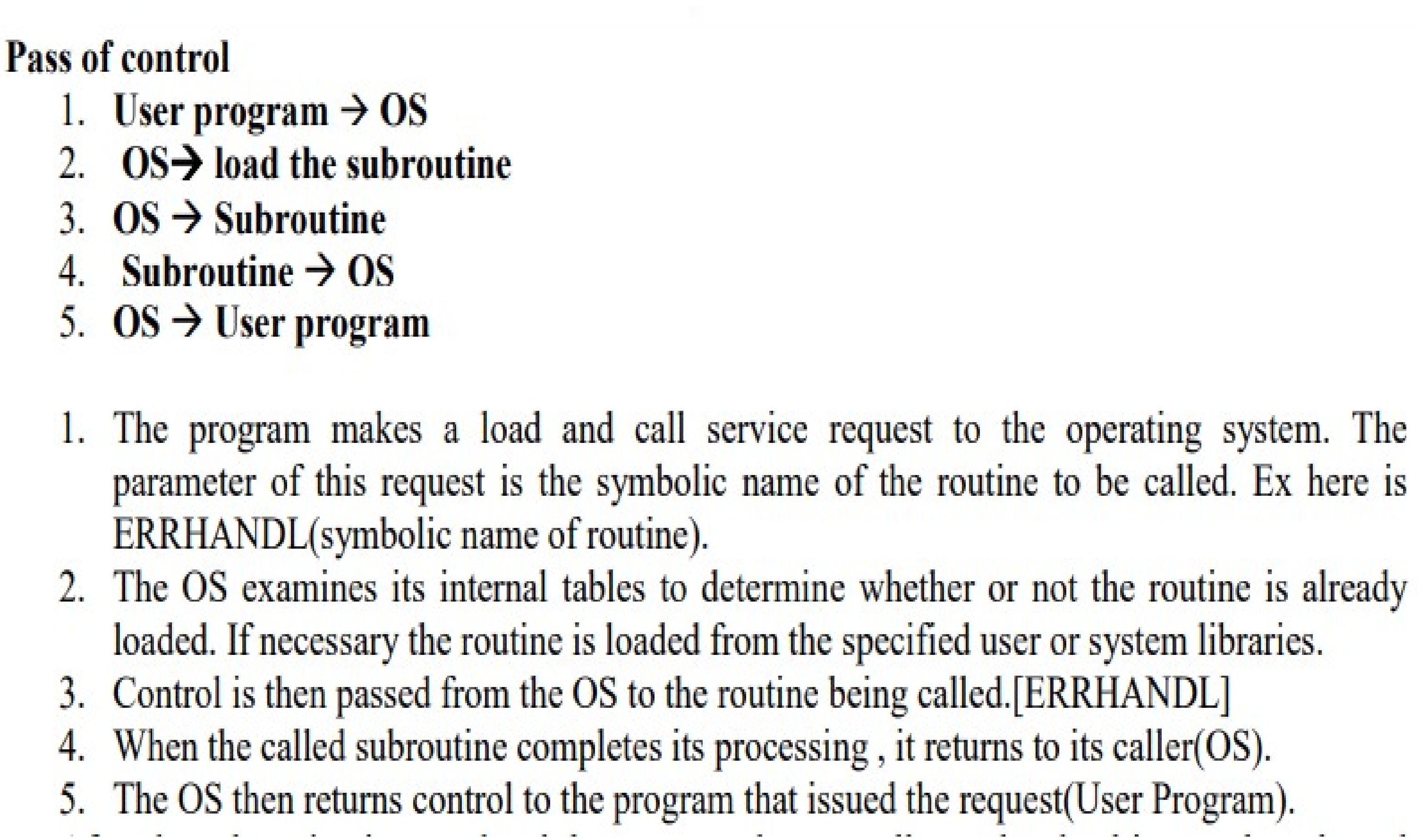


Dynamic Linking



(e)

Dynamic Linking



Pass of control

1. **User program → OS**
2. **OS→ load the subroutine**
3. **OS → Subroutine**
4. **Subroutine → OS**
5. **OS → User program**

1. The program makes a load and call service request to the operating system. The parameter of this request is the symbolic name of the routine to be called. Ex here is ERRHANDL(symbolic name of routine).
2. The 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.
3. Control is then passed from the OS to the routine being called.[ERRHANDL]
4. When the called subroutine completes its processing , it returns to its caller(OS).
5. The OS then returns control to the program that issued the request(User Program).

Dynamic Linking

- **Advantages:**

1. The overhead on the loader is reduced. The required subroutine will be loaded in the main memory only at the time of execution.

2. The system can be dynamically configured.

- **Disadvantages** The linking and loading has to be suspended until execution. During the execution if at all any subroutine is needed then the process of execution needs to be suspended until the require subroutine gets loaded in the main memory