MODULE- 4

LOADERS AND LINKERS

Introduction

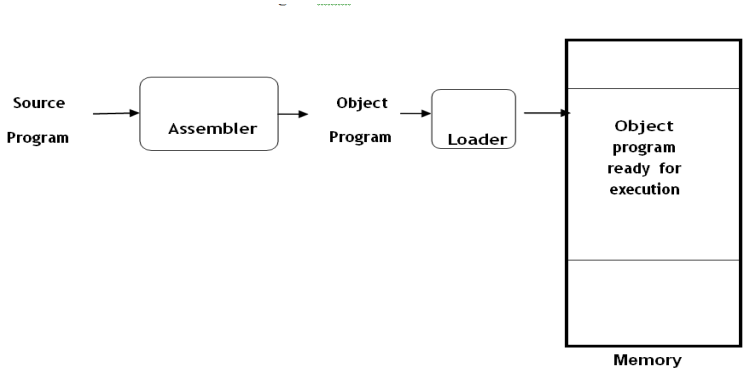
The Source Program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution. This conversion either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies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

This contains the following three processes, and they are,

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

4. 1 Basic Loader Functions:

- A loader is a system software 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.



Type of Loaders

The different types of loaders are, absolute loader, bootstrap loader, relocating loader (relative loader), and, linking loader. The following sections discuss the functions and design of all these types of loaders.

4.1.1 Design of 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. Linking and relocation is not done.
- The algorithm for this type of loader is given here.

```
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
```

Algorithm for Absolute loader

- In this all functions are done in a single pass. The header is checked to veriy 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.

```
HCOPY  001000001007A
T001000 1E 141033482039001036281030301015482061 3C 100300102A 0C 1039001 02D
T00101E 15 0C 1036482061081033 4C 0000454F46 0C 0003000000
T002039 1E 041030001030E0205D30203FD8205D28103030205754903920C205E38203F
T002057 1C 1010364C0000F1001000041030E02079302064509039DC2079 2C1036
T002073 07 3820644C000005
E001000
```

(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 ← COPY
⋮	⋮	⋮	⋮	⋮
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
⋮	⋮	⋮	⋮	⋮

(b) Program loaded in memory

- The figure (b) shows the representation of program from figure (a) after loading.
- In the object program each byte of assembled code is given using its hexadecimal representation in character form.

- In the object program , each byte of assembled code is given using its hexadecimal representation in character form. For example, the machine opcode for an STL instruction would be represented by the pair of characters “1” and “4”. When these are read by the loader , they will occupy two bytes of memory. This opcode must be stored in a single byte with hexa decimal value 14. Thus each pair of bytes from the object program must be packed together into one byte during loading.

4.1.2 A simple 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 80.

- Working: Consider the bootstrap loader for SIC/XE. The bootstrap loader begins at address 0 in the memory. It loads the OS starting at address 80. Each byte of object code to be loaded is represented on device F1 as two hexa decimal digits(Text record) . Object code is loaded to consecutive memory locations starting at address 80. After all the object code from device F1 has been loaded the bootstrap jumps to the address 80.

- GETC subroutine – This subroutine reads one character from device F1 and converts from ASCII to hex. This is done by subtracing 48 if the character is from 0 to 9. For characters A to F subtract 55. Subroutine jumps to address 80 when end of line is reached.

- Main loop of the bootstrap loader- This keeps the address of the next memory location to be loaded in register X. GETC is used to read and convert a pair of characters from device F1(represents one byte of object code). These two hexadecimal values are combined to a single byte by shifting the first one left by 4 bit positions and adding the second to it. The resulting byte is stored at address currently in register X

The algorithm for the bootstrap loader is as follows

```
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
         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
```

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

Figure 3.3 Bootstrap loader for SIC/XE.

4.2Machine-Dependent Loader Features

- Absolute loader is simple and efficient, but the scheme has potential disadvantages. One of the most disadvantage is the programmer has to specify the actual starting address, from where the program to be loaded. This does not create difficulty, if one program to run, but not for several programs. Further it is difficult to use subroutine libraries efficiently.
- This needs the design and implementation of a more complex loader. The loader must provide program relocation and linking, as well as simple loading functions. This depends on machine architecture.

4.2.1Relocation(Relocating loader)

- Loaders that allow program relocation are called relocating loaders.
- There are two methods for providing relocation as part of the object program.
    - Modification record
    - Bit masking

Modification Record

- A modification record is used to describe each part of the object code that must be changed when the program is relocated.
- Consider SIC/XE programs, Most of the instructions in this program uses relative or immediate addressing. So modification not required. Only format 4 instructions require modification
- Each modification record specifies the starting address and length of the field to be modified and what modification to be performed.(adding the start address).

```
HCOPY  000000001077
T0000001D17202D69202D4B10103603202629000033200074B10105D3F2FEC032010
T00001D130F20160100030F200D4B10105D3E2003454F46
T001036 1DB410B400B44075101000E32019332FFADB2013A00433200857C003B850
T001053 1D3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T00107007 3B2FEF4F000005
M00000705+COPY
M00001405+COPY
M00002705+COPY
E000000
```

Figure 3.5 Object program with relocation by Modification records.

Algorithm for SIC/XE relocation loader

```
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
```

Bitmasking

- In SIC program relative addressing is not used. So every instruction needs modification. We can not write modification records for all instructions.
- So relocation bits are used. Each instruction object code is associated with relocation bit.
- Relocation bits for each text record is written together into bitmask after the length using 3 hexadecimal digits.(12 bits)
- Example:

```
HCOPY  000000001007A
T00000001EFFC140033481039000036280030300015481061 3C0003 00002A0C0039 00002D
T00001E15E000C0036481061080033 4C0000 454F46 000003 000000
T001039 1EFFC 040030 000030 E0105D 30103F D8105D 280030 301057 548039 2C105E 38103F
T001057 0A8001 000364 C0000F 1001000
T001061 19FE0 040030 E01079 301064 508039 DC1079 2C0036 381064 4C0000 05
E000000
```

Figure 3.7 Object program with relocation by bit mask.

- If the relocation bit is 1 program starting address is to be added to this word.

  FFC= 111111111100

SIC relocation loader algorithm

```
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 M_i = 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
```

4.2.2Program Linking

- Consider the program of control sections. The program is made up of 3 control sections.

    1. Main program
    2. Read subroutine
    3. Write subroutine

- These control sections could be assembled together or they could be assembled independently as separate segments of object code after assembly.
- The programmer thinks the three control sections together as a single program. But loader considers this as separate control sections which are to be linked , relocated and loaded.
- Consider the three separate programs PROGA,PROGB,PROGC. In this example, there are differences in handling the identical expressions within the 3 programs.
- Consider the references and the corresponding modification records.
- The general approach is assembler evaluate as much as of the expression it can. The remaining terms are passed on to the loader through modification records.

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	

```
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
```

Figure 3.8 Sample programs illustrating linking and relocation.

```
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
```

- Each program contains a list of items(LISTA, LISTB, LISTC). The ends of these lists are marked by ENDA, ENDB, ENDC. Each program contains the same set of references to these external symbols. Three of these are instruction operands(REF1,REF2,REF3). and the others are the values of data words.(REF4 through REF8).
- Consider first reference marked REF1.For PROGA REF1 is simply a reference to a label within the program. It is assembled in the usual way as PC relative instruction.In PROGB the same operand refers to an external symbol. The assembler uses an extended format instruction with addess field set to 00000. Object program for PROGB contains a modification record instructing the loader to add the value of the symbol LISTA to this address field when the program is linked.This reerence is handled exactly in the same way for PROGC.

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
RLISTB ENDB  LISTC ENDC
.
.
T0000200A03201D771000040500 14
.
.
T000054 0F000014FFFFF600003F000014FFFFC0
M00002405+LISTB
M00005406+LISTC
M00005706+ENDC
M00005706-LISTC
M00005A06+ENDC
M00005A06-LISTC
M00005A06+PROGA
M00005D06-ENDB
M00005D06+LISTB
M00006006+LISTB
M00006006-PROGA
E000020
```

Figure 3.9 Object programs corresponding to Fig. 3.8.

```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
RLISTA ENDA  LISTC ENDC
.
.
T0000360B0310000077202705100000
.
.
T0000700F000000FFFFF6FFFFFFFFFFF0000060
M00003705+LISTA
M00003E05+ENDA
M00003E05-LISTA
M00007006+ENDA
M00007006-LISTA
M00007006+LISTC
M00007306+ENDC
M00007306-LISTC
M00007606+ENDC
M00007606-LISTC
M00007606+LISTA
M00007906+ENDA
M00007906-LISTA
M00007C06+PROGB
M00007C06-LISTA
E
```

```
HPROGC ^000000000051
DLISTC ^000030ENDC  ^000042
RLISTA ^ENDA  ^LISTB ^ENDB
:
T0000180C03100000771000040510000 0
:
T0000420F000030000008000011000000000000
M00001905+LISTA
M00001D05+LISTB
M00002105+ENDA
M00002105-LISTA
M00004206+ENDA
M00004206-LISTA
M00004206+PROGC
M00004806+LISTA
M00004B06+ENDA
M00004B06-LISTA
M00004B06-ENDB
M00004B06+LISTB
M00004E06+LISTB
M00004E06-LISTA
E
```

Figure 3.9 (*cont'd*)

- The figure below shows how the three programs are loaded into memory.

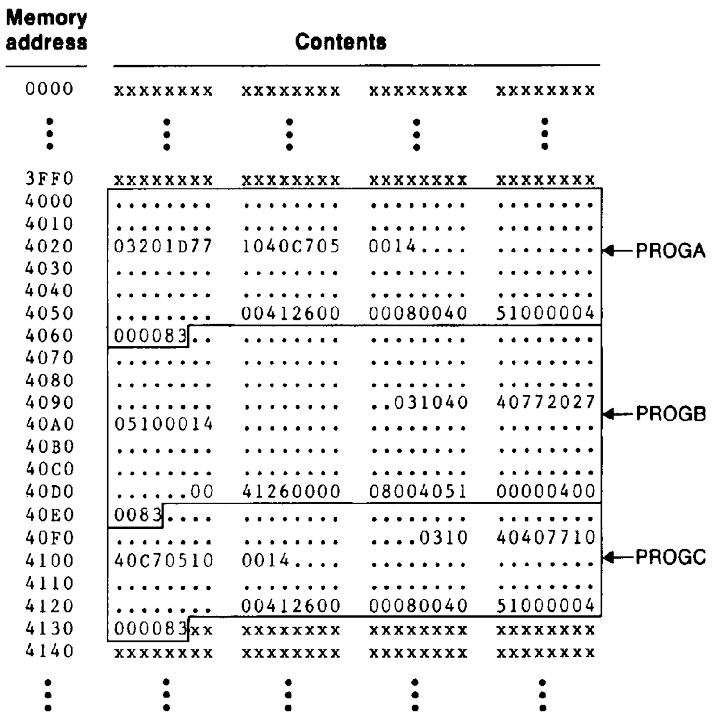


Figure 3.10(a) Programs from Fig. 3.8 after linking and loading.

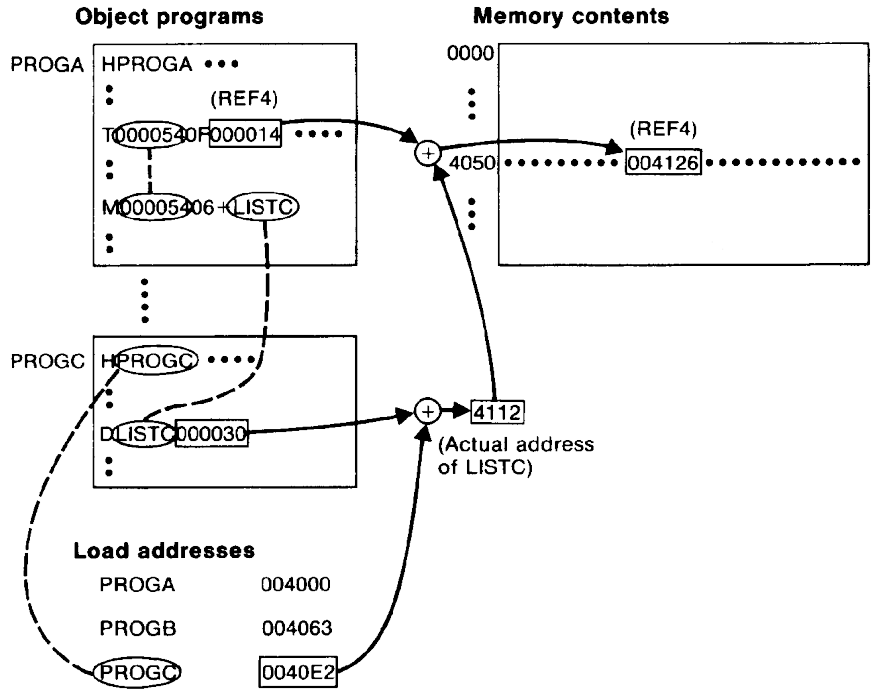


Figure 3.10(b) Relocation and linking operations performed on REF4 from PROGA.

- The values of REF4 through REF8 are same in all the three programs because the same source expression appeared in each program.

4.2.3 Algorithm and data structures for a linking loader

- Consider the algorithm for a linking and relocating loader.
- We use modification records for both relocating and linking
- This type of loader is found on SIC/XE machines whose relative addressing makes relocation unnecessary.
- Input- consists of a set of object programs (control sections) that are to be linked together.
- Control sections or programs contain external references whose definition does not appear in the same program or control section. So linking can not be done until an address is assigned to the external symbol. So it requires two passes.
    - Pass1- Assigns addresses to all external symbols.
    - Pass2- performs the actual loading relocation and linking.
- The **main data structure** for the linking loader is an external symbol table **ESTAB**. It is analogous to SYMTAB. It stores the name and address of each external symbol in the control section. The table also indicates in which control section the symbol is defined.
- Two variables: PROGADDR- Program starting address in memory where the linked program should be loaded. Its value is supplied to the loader by the OS.CSADDR-contains the starting address assigned to the control section currently being scanned by the loader.
- Example: Consider the object programs of PROGA, PROGB, PROGC in fig 3.9 as input to the loader.

Pass1

- 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 OS. This becomes the starting address for the first control section(CSADDR).
- The control section name 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 gives the starting address for the next control section.

- At the end of pass1 , ESTAB contains all external symbols defined in the control sections together with addresses assigned to each.

- Many loaders include the ability to print a **load map** that shows these symbols and their addresses.

Output of pass1

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	

Algorithm for pass1 of a linking loader

Pass 1:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR {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 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'}
      add CSLTH to CSADDR {starting address for next control section}
   end {while not EOF}
end {Pass 1}
```

Figure 3.11(a) Algorithm for Pass 1 of a linking loader.

Pass2

- Performs the actual loading, relocation and linking of the program.
- CSADDR holds the 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 the 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 transferring of control to the loaded program to begin execution.

Pass2 Algorithm

Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
   begin
       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
                  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)
                      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)
       add CSLTH to CSADDR
   end  {while not EOF}
jump to location given by EXECADDR {to start execution of loaded program}
end {Pass 2}
```

Figure 3.11(b) Algorithm for Pass 2 of a linking loader.

- The algorithm can be made more efficient if a slight change is made in the object program format. that is assigning a reference number to each external symbol referred to in a control section. This reference number is used in modification records.

4.3 Machine Independent loader features

4.3.1 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 library as they are needed during linking.

- Loader can automatically include routines from a library into the program being loaded.

- The programmer has to only give the subroutine name in the external reference. The routine will be automatically fetched from the library and linked with the main program.

- Working: Enter symbols from Refer record into the symbol table(ESTAB) . When the definition is encountered the address is assigned to the symbol. At the end of pass the symbols in ESTAB remain undefined represent unresolved external references . The loader searches the library for the routines and process the subroutines as if they are part of the input stream.

- The libraries to be searched by the loader contain assembled or compiled versions of the object program(sub program). A special file structure is used for libraries. This is known as directory. This contains the name of the subroutine and a pointer to its address within the file.

4.3.2 Loader Options

- Many loader allow the user to specify options that modify standard processing.
- Loaders have special command language that is used to specify options. Sometimes there is a separate input file to the loader that contains such control statements. The programmer can even include loader control statements in the source program.
  Some of the loader options are:
  1. Selection of alternative sources of input:
     INCLUDE programname(libraryname)
     This command direct the loader to read the designated object program from a library and treat it as if it were primary loader input.
  2. Command to delete external symbols or entire control section
     DELETE csectname
     This instruct the loader to delete the control section from the set of programs being loaded.
  3. CHANGE name1,name2
     This command causes the external symbol name1 to be changed to name2 wherever it appears in the object program.
     Eg: Consider the object program COPY. Here main program is COPY and the two subroutines are RDREC and WRREC. Each of these is a separate control section. Suppose that a set of utility routines are available on the computer system. Two of these READ and WRITE are are designed to perform the same functions as RDREC and WRREC. If we want to use READ and WRITE we can give the loader commands

     INCLUDE READ(UTLIB)
     INCLUDE WRITE(UTLIB)
     DELETE RDREC, WRREC
     CHANGE RDREC,READ
     CHANGE WRREC,WRITE
  4. Another common loader option involves the automatic inclusion of library routines to satisfy external references. Most loaders allow the user to specify alternative libraries to be searched using a statement such as LIBRARY MYLIB . Such user specified libraries are normally searched before the standard libraries. This allows the user to use special versions of the standard routines.

5. Loaders that perform automatic library search to satisfy external reference allows the user to avoid some references using the command NOCALL. Eg: NOCALL STDDEV, PLOT. This avoids the overhead of loading and linking the unwanted routines
6. Other options:
    - No external reference should be resolved.
    - Specify the output from the loader(load map)
    - Specify the location at which the execution is to begin

4.4 Loader Design

- Loaders do loading , relocation and linking.
- There are 4 types
    - Linkage editor- links the program stores it in a file and later loads.
    - Linking loader- linking during load time
    - Dynamic linking- linking during execurion time
    - Bootstrap loader- loads the first program or OS.

4.4.1Differences between Linkage editor and linking loader

Linking loader	linkage editor
1. Performs all linking and relocation operations and loads the linked program directly into memory for execution	1. Produces a linked version of the program called load module which is written to a file for later execution
2. A linking loader searches the library and resolves external references every time the program is executed.	2. Resolution of external references and library searching are only performed once.
3. More than one pass required.	3. The loading can be accomplished in one pass and no external symbol table required, much less overhead than a linking loader.

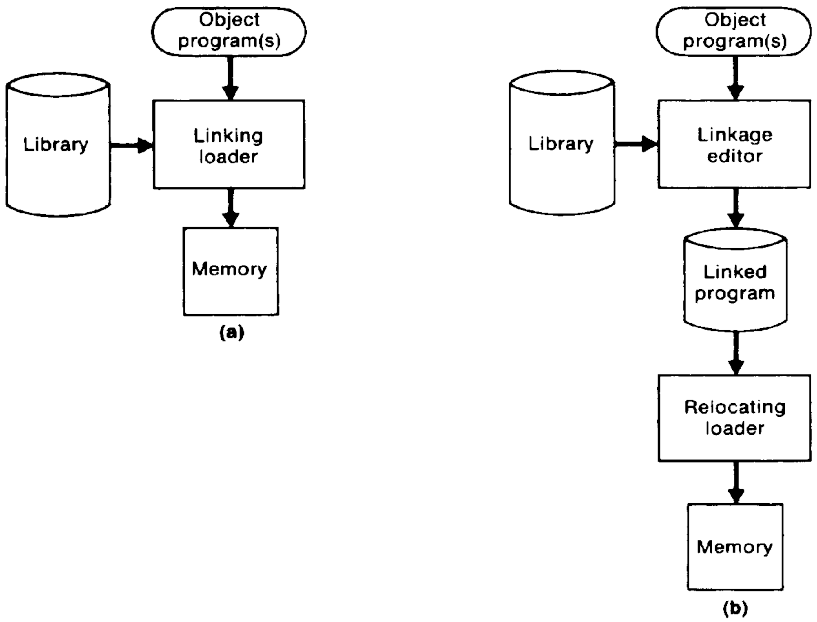


Figure 3.13 Processing of an object program using (a) linking loader and (b) linkage editor.

Advantages of Linkage editors

- Linkage editors can perform many useful functions besides simply preparing an object program for execution. Consider the example, a program PLANNER that uses a large number of subroutines. Suppose that one subroutine called PROJECT is changed. After new version of PROJECT is assembled the linkage editor can be used to replace this subroutine in the linked version of PLANNER.
  INCLUDE PLANNER(PROGLIB)
  DELETE PROJECT (delete from existing planner)
  INCLUDE PROJECT(NEWLIB) (include new version)
  REPLACE PLANNER(PROGLIB)

- Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. Eg: For FORTRAN programs there are a number of subroutines that are used for input and output. They are read and write datablocks, encode and decode data items etc. Linkage editor can be used to combine these subroutines into a package with the following commands.

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

INCLUDE   READR(FTNLIB)
INCLUDE   WRITER(FTNLIB)

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

- Linkage editors can also allow the user to specify that external references are not to be resolved by automatic library search.

4.4.2 Dynamic Linking

- In dynamic linking the linking function is done at execution time. That is a subroutine is loaded and linked to the rest of the program when it is first called.
- Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library. For eg: in C such fuctions are stored in dynamic linking library.. A single copy of the routines in this library could be loaded into memory and all programs share this.
- In object oriented program dynamic linking is often used for references to software objects.
- Advantage:- Dynamic linking provide the ability to load the routines only when they are required. For eg: consider the subroutine which diagnose the error in input data during execution. If such errors are rare these subroutines need not be used.
- Consider the following example of dynamic linking. Here the routines that are to be dynamically loaded must be called via an OS service request.

Loading and calling a subroutine via dynamic linking

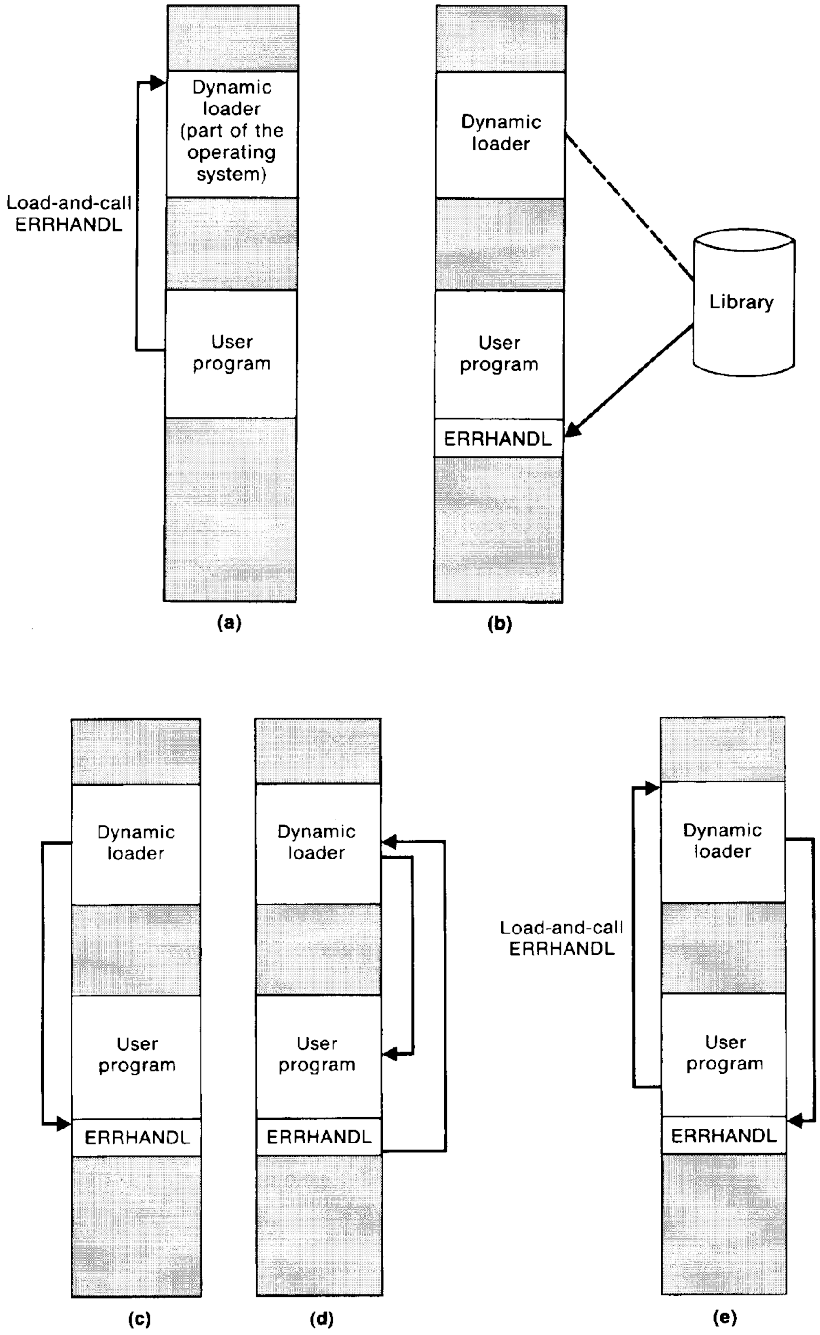


Figure 3.14 Loading and calling of a subroutine using dynamic linking.

- When the dynamic linking is used the association of an actual address with the symbolic name of the called routine is done at execution time.. This is known as **dynamic binding**.

4.3.3 Bootstrap loaders

- Consider how the loader itself is loaded into memory. OS loads the loader. How the OS gets loaded.
- In an idle system if we specify the absolute address the program can be loaded at that location. that is a mechanism of absolute loader is required.
- One solution to this is to have a built in hardware function that reads a fixed length record from some device into memory at some fixed location. This device can be selected via console switches. After the read operation is complete the 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.
- If the loading process requires more instructions than can be read in a single record this first record causes the reading of others and in turn other records . Hence the name **Bootstrap.**