Unit 3: Loader and Linker Design [8 Hrs]

Introduction

- An object program contains translated instructions and data values from the source program, and specifies addresses in memory where these items are to be loaded.
- Our discussions in Chapter 2 introduced the following three processes:

  *1. **Loading**, which brings the object program into memory for execution.*

  *2. **Relocation**, which modifies the object program so that it can be loaded* at an address different from the location originally specified.

  *3. **Linking**, which combines two or more separate object programs and* supplies the information needed to allow references between them.
- A ***loader*** *is a system program that performs the loading function.*
- *Many* loaders also support **relocation** and **linking**.
- Some systems have a ***linker*** *(or* ***linkage editor****) to perform the linking operations and a separate loader to handle* relocation and loading.
- In most cases all the program translators (i.e., assemblers and compilers) on a particular system produce object programs in the same format.
- Thus one system loader or linker can be used regardless of the original source programming language.

3.1: BASIC LOADER FUNCTIONS
Design of an Absolute Loader

- We consider the design of an absolute loader that might be used with the sort of assembler with object code described in Chapter 2.
- An example of such an object program is shown in Fig. 3.1(a).
- Because our loader does not need to perform such functions as linking and program relocation, its operation is very simple.
- All functions are accomplished in a single pass.
- The **Header** record is checked to verify that the correct program has been presented for loading (and that it will fit into the available memory).
- 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.
- Figure 3.1(b) shows a representation of the program from Fig. 3.1(a) after loading.
- The contents of memory locations for which there is no Text record are shown as ***xxxx****.*
- *This indicates that the previous contents of these locations* remain unchanged.

Loading of absolute program (3.1)

```
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	← 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

Algorithm for an absolute loader (3.2)

```
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
```

- Figure 3.2 shows an algorithm for the absolute loader we have discussed.
- In our object program, each byte of assembled code is given using its hexadecimal representation in character form.
- For example, the machine operation code for an **STL** instruction would be represented by the *pair of characters* "1" and "4".
- When these are read by the loader (as part of the object program), they will occupy *two bytes of memory. In the instruction as loaded for* execution, however, this operation code must be stored in a *single byte with hexadecimal value 14.*
- *Thus each pair of bytes from the object program record* must be packed together into one byte during loading.
- It is very important to realize that in Fig. 3.1(a), each printed character represents **one *byte*** *of the object* program record.
- In Fig. 3.1(b), on the other hand, each printed character represents one *hexadecimal digit in memory (i.e., a half-byte).*
- This method of representing an object program is inefficient in terms of both space and execution time.
- Therefore, most machines store object programs in a *binary form, with each byte of object code stored as a single byte in* the object program.
- In this type of representation, of course, a byte may contain any binary value.
- We must be sure that our file and device conventions do not cause some of the object program bytes to be interpreted as control characters.

A Simple Bootstrap Loader

- When a computer is first turned on or restarted, a special type of absolute loader, called a ***bootstrap loader***, *is executed.*
- *This* ***bootstrap*** *loads the first program* to be run by the computer-usually an operating system.
- In this section, we examine a very simple bootstrap loader for SIC/XE.
- In spite of its simplicity, this program illustrates almost all of the logic and coding techniques that are used in an absolute loader.
- Figure 3.3 shows the source code for our bootstrap loader.

Bootstrap loader for SIC/XE (3.3)

```
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
```

(3.3 continued)

```
.
. 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
```

- The bootstrap itself begins at address 0 in the memory of the machine.
- It loads the operating system (or some other program) starting at address 80.
- Because this loader is used in a unique situation (the initial program load for the system), the program to be loaded can be represented in a very simple format.
- Each byte of object code to be loaded is represented on device F1 as two hexadecimal digits (just as it is in a Text record of a SIC object program).
- However, there is no Header record, End record, or control information (such as addresses or lengths).
- The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80.
- After all of the object code from device F1 has been loaded, the bootstrap jumps to address 80, which begins the execution of the program that was loaded.

- Much of the work of the bootstrap loader is performed by the subroutine **GETC**.
- This subroutine reads one character from device F1 and converts it from the ASCII character code to the value of the hexadecimal digit that is represented by that character.
- For example, the ASCII code for the character *"0“* (hexadecimal 30) is converted to the numeric value 0.
- Likewise, the ASCII codes for “1 " through *"9" (hexadecimal 31 through 39) are converted to the* numeric values 1 through 9, and the codes for *“A" through “F" (hexadecimal* 41 through 46) are converted to the values 10 through 15.
- This is accomplished by subtracting 48 (hexadecimal 30) from the character codes for "0" through *"9", and subtracting 55 (hexadecimal 37) from the codes for “ A" through "F".*
- The subroutine **GETC** jumps to address 80 when an end-of-file (hexadecimal 04) is read from device F1.
- It skips all other input characters that have ASCII codes less than hexadecimal 30.
- This causes the bootstrap to ignore any control bytes (such as end-of-line) that are read.

- The main loop of the bootstrap 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 (representing 1 byte of object code to be loaded).
- These two 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, using a STCH instruction that refers to location 0 using indexed addressing.
- The **TIXR** instruction is then used to add 1 to the value in register X. (Because we are not interested in the result of the comparison performed by **TIXR**, register X is also used as the second operand for this instruction.)

3.2: MACHINE-DEPENDENT LOADER FEATURES

- The absolute loader described is certainly simple and efficient; however, this scheme has several potential disadvantages.
- One of the most obvious is the need for the programmer to specify (when the program is assembled) the actual address at which it will be loaded into memory. If we are considering a very simple computer with a small memory (such as the standard version of SIC), this does not create much difficulty. There is only room to run one program at a time, and the starting address for this single user program is known in advance.
- On a larger and more advanced machine (such as SIC/XE), the situation is not quite as easy. We would often like to run several independent programs together, sharing memory (and other system resources) between them. This means that we do not know in advance where a program will be loaded.
- Efficient sharing of the machine requires that we write relocatable programs instead of absolute ones.
- Writing absolute programs also makes it difficult to use subroutine libraries efficiently.
- Most such libraries (for example, scientific or mathematical packages) contain many more subroutines than will be used by anyone program.
- To make efficient use of memory, it is important to be able to select and load exactly those routines that are needed.
- This could not be done effectively if all of the subroutines had preassigned absolute addresses.

- In this section we consider the design and implementation of a more complex loader.
- The loader we present is one that is suitable for use on a SIC/XE system and is typical of those that are found on most modem computers.
- This loader provides for program relocation and linking, as well as for the simple loading functions described in the preceding section.
- As part of our discussion, we examine the effect of machine architecture on the design of the loader.
- The need for program relocation is an indirect consequence of the change to larger and more powerful computers.
- The way relocation is implemented in a loader is also dependent upon machine characteristics.
- Next we discuss these dependencies by examining different implementation techniques and the circumstances in which they might be used.

Relocation

- Loaders that allow for program relocation are called ***relocating loaders*** *or* ***relative loaders****.*
- *The concept of program relocation was introduced in* **Chapter 2**.
- In this section we discuss two methods for specifying relocation as part of the object program.
- The first method we discuss is essentially the same as that introduced in **Chapter 2**.
- A Modification record is used to describe each part of the object code that must be changed when the program is relocated.
- Figure 3.4 shows a SIC/XE program we use to illustrate this first method of specifying relocation.
- The program is the same as the one in **Fig**. **2.6**.
- Most of the instructions in this program use relative or immediate addressing.
- The only portions of the assembled program that contain actual addresses are the extended format instructions on lines 15, 35, and 65.
- Thus these are the only items whose values are affected by relocation.

Example of a SIC/XE Program (3.4//2.6)

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	
110		.			

(3.4 Continued)

```
110                  .
115                  .          SUBROUTINE TO READ RECORD INTO BUFFER
120                  .
125     1036         RDREC      CLEAR     X                 B410
130     1038                    CLEAR     A                 B400
132     103A                    CLEAR     S                 B440
133     103C                   +LDT      #4096              75101000
135     1040         RLOOP      TD        INPUT             E32019
140     1043                    JEQ       RLOOP             332FFA
145     1046                    RD        INPUT             DB2013
150     1049                    COMPR     A,S               A004
155     104B                    JEQ       EXIT              332008
160     104E                    STCH      BUFFER,X          57C003
165     1051                    TIXR      T                 B850
170     1053                    JLT       RLOOP             3B2FEA
175     1056         EXIT       STX       LENGTH            134000
180     1059                    RSUB                        4F0000
185     105C         INPUT      BYTE      X'F1'             F1
195                  .
200                  .          SUBROUTINE TO WRITE RECORD FROM BUFFER
205                  .
210     105D         WRREC      CLEAR     X                 B410
212     105F                    LDT       LENGTH            774000
215     1062         WLOOP      TD        OUTPUT            E32011
220     1065                    JEQ       WLOOP             332FFA
225     1068                    LDCH      BUFFER,X          53C003
230     106B                    WD        OUTPUT            DF2008
235     106E                    TIXR      T                 B850
240     1070                    JLT       WLOOP             3B2FEF
245     1073                    RSUB                        4F0000
250     1076         OUTPUT     BYTE      X'05'             05
255                             END       FIRST
```

- Figure 3.5 displays the object program corresponding to the source in Fig.3.4.
- Notice that there is one Modification record for each value that must be changed during relocation (in this case, the three instructions previously mentioned).
- Each Modification record specifies the starting address and length of the field whose value is to be altered.
- It then describes the modification to be performed.
- In this example, all modifications add the value of the symbol COPY, which represents the starting address of the program (Fig. 3.6).

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

Figure 3.5 Object program with relocation by Modification records.

SIC/XE Relocation Loader Algorithm (3.6)

```
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
```

- The Modification record scheme is a convenient means for specifying program relocation; however, it is not well suited for use with all machine architectures.
- Consider, for example, the program in Fig. 3.7.
- This is a relocatable program written for the standard version of SIC.
- The important difference between this example and the one in Fig. 3.4 is that the standard SIC machine does not use relative addressing.
- In this program the addresses in all the instructions except RSUB must be modified when the program is relocated.
- This would require 31 Modification records, which results in an object program more than twice as large as the one in Fig. 3.5.

Relocatable Program for a Standard SIC Machine (3.7)

Loc	Source statement			Object code
0000	COPY	START	0	
0000	FIRST	STL	RETADR	140033
0003	CLOOP	JSUB	RDREC	481039
0006		LDA	LENGTH	000036
0009		COMP	ZERO	280030
000C		JEQ	ENDFIL	300015
000F		JSUB	WRREC	481061
0012		J	CLOOP	3C0003
0015	ENDFIL	LDA	EOF	00002A
0018		STA	BUFFER	0C0039
001B		LDA	THREE	00002D
001E		STA	LENGTH	0C0036
0021		JSUB	WRREC	481061
0024		LDL	RETADR	080033
0027		RSUB		4C0000
002A	EOF	BYTE	C'EOF'	454F46
002D	THREE	WORD	3	000003
0030	ZERO	WORD	0	000000
0033	RETADR	RESW	1	
0036	LENGTH	RESW	1	
0039	BUFFER	RESB	4096	
	.			

(3.7 Continued)

```
                .       SUBROUTINE TO READ RECORD INTO BUFFER
                .
        1039    RDREC   LDX     ZERO                040030
        103C            LDA     ZERO                000030
        103F    RLOOP   TD      INPUT               E0105D
        1042            JEQ     RLOOP               30103F
        1045            RD      INPUT               D8105D
        1048            COMP    ZERO                280030
        104B            JEQ     EXIT                301057
        104E            STCH    BUFFER,X            548039
        1051            TIX     MAXLEN              2C105E
        1054            JLT     RLOOP               38103F
        1057    EXIT    STX     LENGTH              100036
        105A            RSUB                        4C0000
        105D    INPUT   BYTE    X'F1'               F1
        105E    MAXLEN  WORD    4096                001000
                .
                .       SUBROUTINE TO WRITE RECORD FROM BUFFER
                .
        1061    WRREC   LDX     ZERO                040030
        1064    WLOOP   TD      OUTPUT              E01079
        1067            JEQ     WLOOP               301064
        106A            LDCH    BUFFER,X            508039
        106D            WD      OUTPUT              DC1079
        1070            TIX     LENGTH              2C0036
        1073            JLT     LOOP                381064
        1076            RSUB                        4C0000
        1079    OUTPUT  BYTE    X'05'               05
                        END     FIRST
```

- On a machine that primarily uses direct addressing and has a fixed instruction format, it is often more efficient to specify relocation using a different technique.
- Figure 3.8 shows this method applied to our SIC program example.
- There are no Modification records.
- The Text records are the same as before except that there is a *relocation bit associated with each word of **object code**.*
- *Since* all SIC instructions occupy one word, this means that there is one relocation bit for each possible instruction.
- The relocation bits are gathered together into a *bit mask following the length indicator in each Text record.*
- *In Fig. 3.8 this* mask is represented (in character form) as three hexadecimal digits.
- These characters are underlined for easier identification in the figure.

Object Program with Relocation by Bit Mask (3.8)

```
H^COPY  ^000000^00107A
T^000000^1E^FFC^140033^481039^000036^280030^300015^481061^3C0003^00002A^0C0039^00002D
T^00001E^15^E00^0C0036^481061^080033^4C0000^454F46^000003^000000
T^001039^1E^FFC^040030^000030^E0105D^30103F^D8105D^280030^301057^548039^2C105E^38103F
T^001057^0A^800^100036^4C0000^F1^001000
T^001061^19^FE0^040030^E01079^301064^508039^DC1079^2C0036^381064^4C0000^05
E^000000
```

- If the relocation bit corresponding to a word of object code is set to 1, the program's starting address is to be added to this word when the program is relocated.
- A bit value of 0 indicates that no modification is necessary.
- If a Text record contains fewer than 12 words of object code, the bits corresponding to unused words are set to 0.
- Thus the bit mask FFC (representing the bit string 111111111100) in the first Text record specifies that all 10 words of object code are to be modified during relocation.
- These words contain the instructions corresponding to lines 10 through 55 in Fig. 3.7.
- The mask E00 in the second Text record specifies that the first three words are to be modified.
- The remainder of the object code in this record represents data constants (and the RSUB instruction) and thus does not require modification.

- The other Text records follow the same pattern.
- Note that the object code generated from the LDX instruction on line 210 begins a new Text record even though there is room for it in the preceding record.
- This occurs because each relocation bit is associated with a 3-byte segment of object code in the Text record.
- Any value that is to be modified during relocation must coincide with one of these 3-byte segments so that it corresponds to a relocation bit.
- The assembled LDX instruction does require modification because of the direct address.
- However, if it were placed in the preceding Text record, it would not be properly aligned to correspond to a relocation bit because of the 1-byte data value generated from line 185.
- Therefore, this instruction must begin a new Text record in the object program.

SIC Relocation Loader Algorithm (3.9)

```
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
```

Figure 3.9 SIC relocation loader algorithm.

Program Linking

- In this section we consider more complex examples of external references between programs and examine the relationship between relocation and linking.
- Figure 2.15 in Chapter 2 showed a program made up of three control sections. These control sections could be assembled together (that is, in the same invocation of the assembler), or they could be assembled independently of one another.
- In either case, however, they would appear as separate segments of object code after assembly (see Fig. 2.17).
- The programmer has a natural inclination to think of a program as a logical entity that combines all of the related control sections.
- From the loader's point of view, however, there is no such thing as a program in this sense-there are only control sections that are to be linked, relocated, and loaded.
- The loader has no way of knowing (and no need to know) which control sections were assembled at the same time.

Sample Programs Illustrating Linking and Relocation (3.10)

```
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
```

(3.10 Continued)

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		

(3.10 Continued)

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		

- Consider the three (separately assembled) programs in Fig. 3.10, each of which consists of a single control section.
- Each program contains a list of items (**LISTA**, **LISTB**, **LlSTC**); the ends of these lists are marked by the labels **ENDA**, **ENDB**, **ENDC**.
- The labels on the beginnings and ends of the lists are external symbols (that is, they are available for use in linking).
- Note that each program contains exactly the same set of references to these external symbols.
- Three of these are instruction operands (**REF1** through **REF3**), and the others are the values of data words (**REF4** through **REF8**).
- In considering this example, we examine the differences in the way these identical expressions are handled within the three programs.
- This emphasizes the relationship between the relocation and linking processes.
- To focus on these issues, we have not attempted to make these programs appear realistic.
- All portions of the programs not involved in the relocation and linking process are omitted.
- The same applies to the generated object programs shown in Fig. 3.11

- Consider first the reference marked **REF1**.
- For the first program (**PROGA**), **REF1** is simply a reference to a label within the program.
- It is assembled in the usual way as a program-counter relative instruction.
- No modification for relocation or linking is necessary.
- In **PROGB**, on the other hand, the same operand refers to an external symbol.
- The assembler uses an extended-format instruction with address field set to 00000.
- The object program for **PROGB** (see Fig. 3.11) 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 reference is handled in exactly the same way for **PROGC**.

- The reference marked **REF2** is processed in a similar manner.
- For **PROGA**, the operand expression consists of an external reference plus a constant.
- The assembler stores the value of the constant in the address field of the instruction and a Modification record directs the loader to add to this field the value of **LISTB**.
- In **PROGB**, the same expression is simply a local reference and is assembled using a program-counter relative instruction with no relocation or linking required.

- **REF3** is an immediate operand whose value is to be the difference between **ENDA** and **LISTA** (that is, the length of the list in bytes).
- In **PROGA**, the assembler has all of the information necessary to compute this value.
- During the assembly of **PROGB** (and **PROGC**), however, the values of the labels are unknown.
- In these programs, the expression must be assembled as an external reference (with two Modification records) even though the final result will be an absolute value independent of the locations at which the programs are loaded.

Object programs corresponding to Fig. 3.10 (3.11)

```
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
```

(3.11 Continued)

```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
RLISTA ENDA  LISTC ENDC
•
•
T0000360B03100000772027051000000
•
•
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
```

(3.11 Continued)

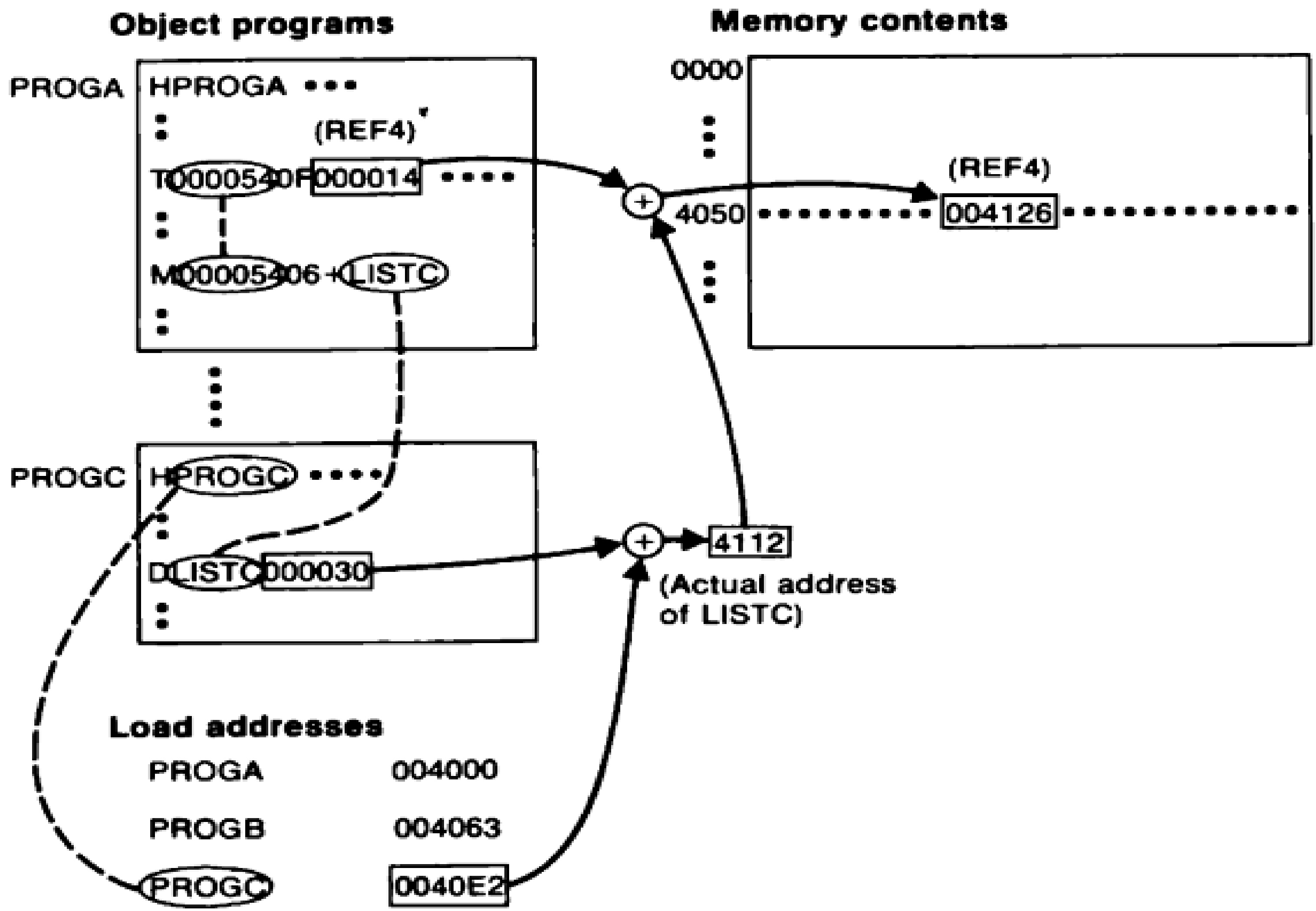
```
HPROGC 000000000051
DLISTC 000030ENDC  000042
RLISTA ENDA  LISTB ENDB
:
T0000180C03100000771000040510 0000
:
T0000420F000030000008000011000000000000
M00001905+LISTA
M00001D05+LISTB
M00002105+ENDA
M00002105-LISTA
M00004206+ENDA
M00004206-LISTA
M00004206+PROGC
M00004806+LISTA
M00004B06+ENDA
M00004B06-LISTA
M00004B06-ENDB
M00004B06+LISTB
M00004E06+LISTB
M00004E06-LISTA
E
```

- The remaining references illustrate a variety of other possibilities.
- The general approach taken is for the assembler to evaluate as much of the expression as it can.
- The remaining terms are passed on to the loader via Modification records.
- To see this, consider **REF4**. The assembler for **PROGA** can evaluate all of the expression in **REF4** except for the value of **LISTC**.
- This results in an initial value of (hexadecimal) **000014** and one Modification record.
- However, the same expression in **PROGB** contains no terms that can be evaluated by the assembler.
- The object code therefore contains an initial value of **000000** and three Modification records.
- For **PROGC**, the assembler can supply the value of **LISTC** relative to the beginning of the program (but not the actual address, which is not known until the program is loaded).
- The initial value of this data word contains the relative address of **LISTC** (hexadecimal **000030**).
- Modification records instruct the loader to add the beginning address of the program (i.e., the value of **PROGC**), to add the value of **ENDA**, and to subtract the value of **LISTA**.
- Thus the expression in **REF4** represents a simple external reference for **PROGA**, a more complicated external reference for **PROGB**, and a combination of relocation and external references for **PROGC**.

Programs from Fig. 3.10 after linking and loading (3.12 a)

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	
⋮	⋮	⋮	⋮	⋮	

Relocation and linking operations performed on REF4 from PROGA (3.12 b)



- Figure **3.12(a)** shows these three programs as they might appear in memory after loading and linking.
- **PROGA** has been loaded starting at address **4000**, with **PROGB** and **PROGC** immediately following.
- Note that each of **REF4** through **REF8** has resulted (after relocation and linking is performed) in the same value in each of the three programs.
- For example, the value for reference **REF4** in **PROGA** is located at address **4054** (the beginning address of **PROGA** plus **0054**, the relative address of **REF4** within **PROGA**).

- Figure **3.12(b)** shows the details of how this value is computed.
- The initial value (from the Text record) is **000014**.
- To this is added the address assigned to LISTC, which is **4112** (the beginning address of **PROGC** plus **30**).
- In **PROGB**, the value for **REF4** is located at relative address **70** (actual address **40D3**). To the initial value (**000000**), the loader adds the values of **ENDA** (**4054**) and **LISTC** (**4112**), and subtracts the value of **LlSTA** (**4040**).
- The result, **004126**, is the same as was obtained in **PROGA**.
- Similarly, the computation for **REF4** in **PROGC** results in the same value.
- The same is also true for each of the other references **REF5** through **REF8**.

- For the references that are instruction operands, the calculated values after loading do not always appear to be equal.
- This is because there is an additional address calculation step involved for program-counter relative (or base relative) instructions.
- In these cases it is the *target addresses that are the same.*
- For example, in **PROGA** the reference **REF1** is a program-counter relative instruction with displacement **01D**.
- When this instruction is executed, the program counter contains the value **4023** (the actual address of the next instruction).
- The resulting target address is **4040**. No relocation is necessary for this instruction since the program counter will always contain the actual (not relative) address of the next instruction.
- We could also think of this process as automatically providing the needed relocation at execution time through the target address calculation.
- In **PROGB**, on the other hand, reference **REF1** is an extended format instruction that contains a direct (actual) address.
- This address, after linking, is **4040** - the same as the target address for the same reference in **PROGA**.

3.3: MACHINE-INDEPENDENT LOADER FEATURES

- In this section we discuss some loader features that are not directly related to machine architecture and design.
- Loading and linking are often thought of as operating system service functions.
- The programmer's connection with such services is not as direct as it is with, for example, the assembler during program development.
- Therefore, most loaders include fewer different features (and less varied capabilities) than are found in a typical assembler.
- Section 3.3.1 discusses the use of an automatic library search process for handling external references. 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.
- Section 3.3.2 presents some common options that can be selected at the time of loading and linking. These include such capabilities as specifying alternative sources of input, changing or deleting external references, and controlling the automatic processing of external references.

Automatic Library Search

- Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded.
- In most cases there is a standard system library that is used in this way.
- Other libraries may be specified by control statements or by parameters to the loader. This feature allows the programmer to use subroutines from one or more libraries (for example, mathematical or statistical routines) almost as if they were a part of the programming language.
- The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program, and loaded.
- The programmer does not need to take any action beyond mentioning the subroutine names as external references in the source program.
- On some systems, this feature is referred to as ***automatic library call***.
- *We use the term library search to avoid confusion with the call feature found in most program*ming languages.

- 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 the loader.
- One easy way to do this is to enter symbols from each Refer record into the symbol table (**ESTAB – External Table**) unless these symbols are already present.
- These entries are marked to indicate that the symbol has not yet been defined.
- When the definition is encountered, the address assigned to the symbol is filled in to complete the entry.
- 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 process just described allows the programmer to override the standard subroutines in the library by supplying his or her own routines.
- For example, suppose that the main program refers to a standard subroutine named **SQRT**.
- Ordinarily the subroutine with this name would automatically be included via the library search function.
- A programmer who for some reason wanted to use a different version of **SQRT** could do so simply by including it as input to the loader.
- By the end of Pass 1 of the loader, **SQRT** would already be defined, so it would not be included in any library search that might be necessary.

- The libraries to be searched by the loader ordinarily contain assembled or compiled versions of the subroutines (that is, object programs).
- It is possible to search these libraries by scanning the Define records for all of the object programs on the library, but this might be quite inefficient.
- 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.
- If a subroutine is to be callable by more than one name (using different entry points), both names are entered into the directory.
- The object program itself, of course, is only stored once. Both directory entries point to the same copy of the routine.
- Thus the library search itself really involves a search of the directory, followed by reading the object programs indicated by this search.
- Some operating systems can keep the directory for commonly used libraries permanently in memory.
- This can expedite the search process if a large number of external references are to be resolved.

Loader Options

- In this section we discuss some typical loader options and give examples of their use.
- Many loaders have a special command language that is used to specify options.
- Sometimes there is a separate input file to the loader that contains such control statements.
- Sometimes these same statements can also be embedded in the primary input stream between object programs.
- On a few systems the programmer can even include loader control statements in the source program, and the assembler or compiler retains these commands as a part of the object program.
- We discuss loader options in this section as though they were specified using a command language, but there are other possibilities.
- On some systems options are specified as a part of the job control language that is processed by the operating system.
- When this approach is used, the operating system incorporates the options specified into a control block that is made available to the loader when it is invoked.
- The implementation of such options is, of course, the same regardless of the means used to select them.

- One typical loader option allows the selection of alternative sources of input.
- For example, the command: **INCLUDE program-name (library-name)**

  might 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.
- Other commands allow the user to delete external symbols or entire control sections.
- It may also be possible to change external references within the programs being loaded and linked.
- For example, the command: **DELETE csect-name**

  might instruct the loader to delete the named control section(s) from the set of programs being loaded.
- The command: **CHANGE name1, name2**

  might cause the external symbol ***name1*** *to be changed to* ***name2*** *wherever it ap*pears in the object programs.

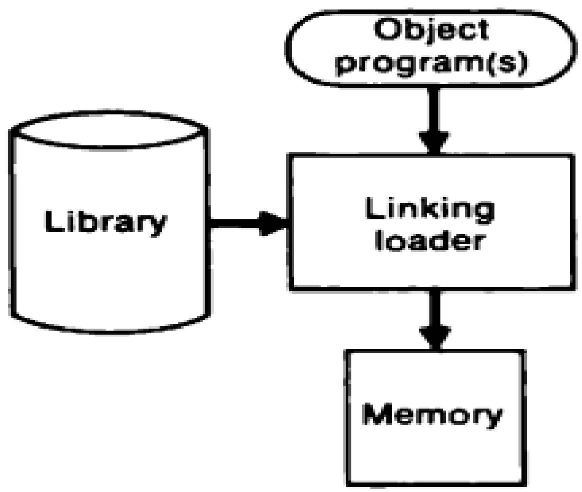
3.4: LOADER DESIGN OPTIONS

- In this section we discuss some common alternatives for organizing the loading functions, including relocation and linking.
- Linking loaders, as described earlier perform all linking and relocation at load time.
- We discuss two alternatives to this: **linkage editors**, which perform linking prior to load time, and **dynamic linking**, in which the linking function is performed at execution time.
- Section 3.4.1 discusses **linkage editors**, which are found on many computing systems instead of or in addition to the linking loader. A linkage editor performs linking and some relocation; however, the linked program is written to a file or library instead of being immediately loaded into memory. This approach reduces the overhead when the program is executed. All that is required at load time is a very simple form of relocation.
- Section 3.4.2 introduces **dynamic linking**, which uses facilities of the operating system to load and link subprograms at the time they are first called. By delaying the linking process in this way, additional flexibility can be achieved. However, this approach usually involves more overhead than does a linking loader.
- In Section 3.4.3 we discuss **bootstrap loaders**. Such loaders can be used to run stand-alone programs independent of the operating system or the system loader. They can also be used to load the operating system or the loader itself into memory.

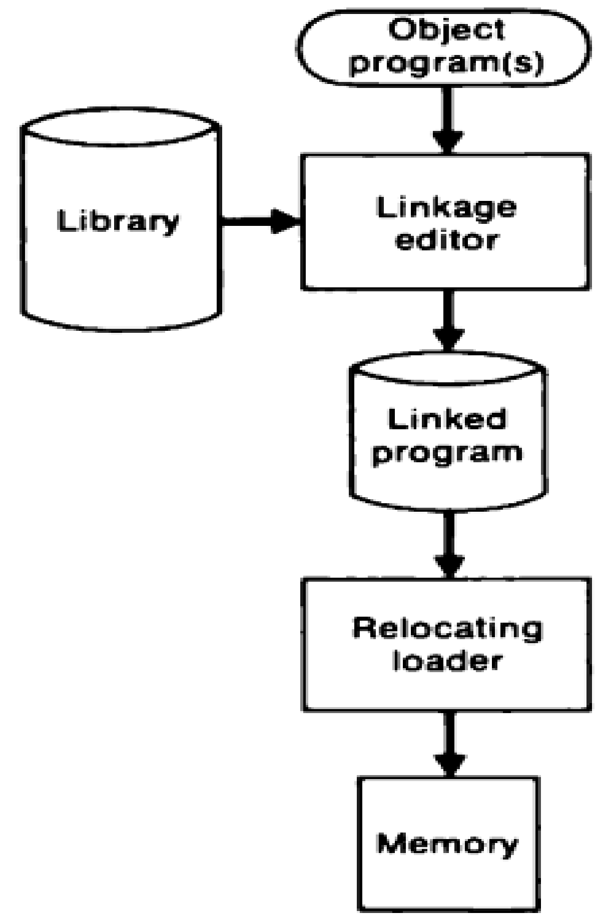
Linkage Editors

- The essential difference between a **linkage editor** and a **linking loader** is illustrated in Fig. 3.13.
- The source program is first assembled or compiled, producing an object program (which may contain several different 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.
- A ***linkage editor****, on the other hand, 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.

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



(a)



(b)

- When the user is ready to run the linked program, a simple relocating loader can be used to load the program into memory.
- The only object code modification necessary is the addition of an actual load address to relative values within the program.
- The **linkage editor** performs relocation of all control sections relative to the start of the linked program.
- Thus, all items that need to be modified at load time have values that are relative to the start of the linked program.
- This means that the loading can be accomplished in one pass with no external symbol table required.
- This involves much less overhead than using a linking loader.

- If a program is to be executed many times without being reassembled, the use of a **linkage editor** substantially reduces the overhead required.
- Resolution of external references and library searching are only performed once (when the program is link edited). In contrast, a linking loader searches libraries and resolves external references every time the program is executed.
- Sometimes, however, a program is reassembled for nearly every execution.
- This situation might occur in a program development and testing environment (for example, student programs).
- It also occurs when a program is used so infrequently that it is not worthwhile to store the assembled version in a library.
- In such cases it is more efficient to use a linking loader, which avoids the steps of writing and reading the linked program.

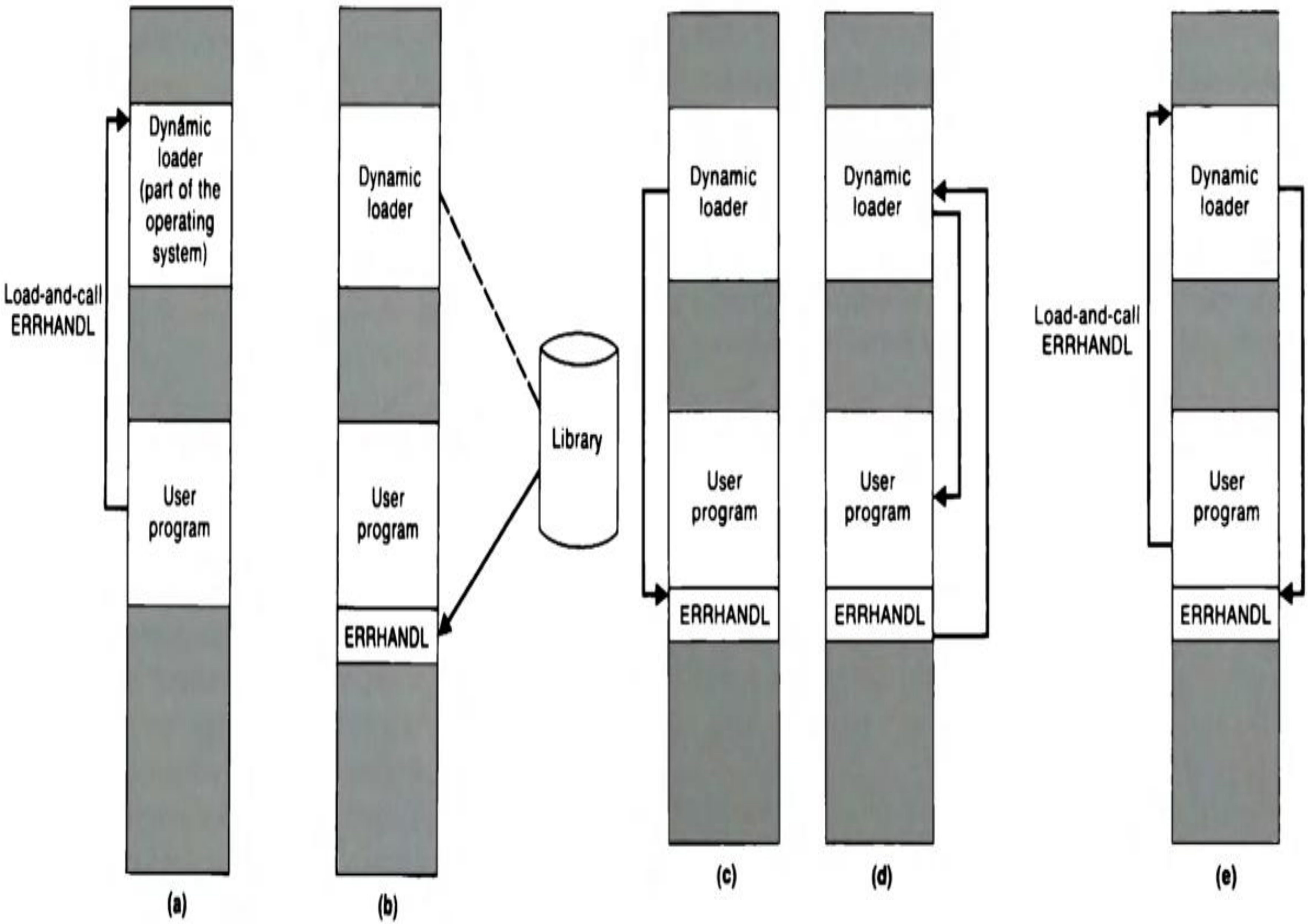
- The linked program produced by the linkage editor is generally in a form that is suitable for processing by a relocating loader.
- All external references are resolved, and relocation is indicated by some mechanism such as Modification records or a bit mask. Even though all linking has been performed, information concerning external references is often retained in the linked program.
- This allows subsequent relinking of the program to replace control sections, modify external references, etc.
- If this information is not retained, the linked program cannot be reprocessed by the linkage editor; it can only be loaded and executed.
- 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.
- The content and processing of such an image are the same as for an absolute object program.
- Normally, however, the added flexibility of being able to load the program at any location is easily worth the slight additional overhead for performing relocation at load time.

Dynamic Linking

- Linkage editors perform linking operations before the program is loaded for execution.
- Linking loaders perform these same operations at load time.
- In this section we discuss a scheme that postpones the linking function until execution time: a subroutine is loaded and linked to the rest of the program when it is first called.
- This type of function is usually called ***dynamic linking***, ***dynamic loading***, *or* ***load on call***.
- Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library.
- For example, 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.

- In an object-oriented system, dynamic linking is often used for references to software objects – allows the implementation of the object and its methods to be determined at the time the program is run.
- The implementation can be changed at any time, without affecting the program that makes use of the object.
- Dynamic linking also makes it possible for one object to be shared by several programs, as discussed previously.
- Dynamic linking also offers some other advantages over the other types of linking we have discussed.
- Suppose, for example, that a program contains subroutines that correct or clearly diagnose errors in the input data during execution. If such errors are rare, the correction and diagnostic routines may not be used at all during most executions of the program.
- However, if the program were completely linked before execution, these subroutines would need to be loaded and linked every time the program is run.
- Dynamic linking provides the ability to load the routines only when (and if) they are needed.
- If the subroutines involved are large, or have many external references, this can result in substantial savings of time and memory space.

Loading and calling of a subroutine using dynamic linking (3.14)



- Instead of executing a **JSUB** instruction that refers to an external symbol, 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 [3.13 (a)].
- The operating system 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 [3.14(b)].
- Control is then passed from the operating system to the routine being called [3.14(c)].
- When the called subroutine completes its processing, it returns to its caller (that is, to the operating system routine that handles the load-and-call service request).
- The operating system then returns control to the program that issued the request.

- This process is illustrated in Fig. 3.14(d). It is important that control be returned in this way so that the operating system knows when the called routine has completed its execution.
- After the subroutine is completed, the memory that was allocated to load it may be released and used for other purposes.
- However, this is not always done immediately. Sometimes it is desirable to retain the routine in memory for later use as long as the storage space is not needed for other processing.
- 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 [3.14(e)].