Chapter 3 – Loaders and Linkers

- Three fundamental processes:

  Loading – brings the object program into memory for execution.

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

  Linking – combines two or more separate object programs and supplies the information needed to allow references between them.

- A *loader* is a system program that performs the *loading* function. Many loaders also support *relocation* and *linking*. Some systems have a *linker* to perform the *linking* operations and a *separate loader* to handle *relocation* and *loading*.

3.1 Introduction

- The most fundamental functions of a loader – bringing an object program into memory and starting its execution.

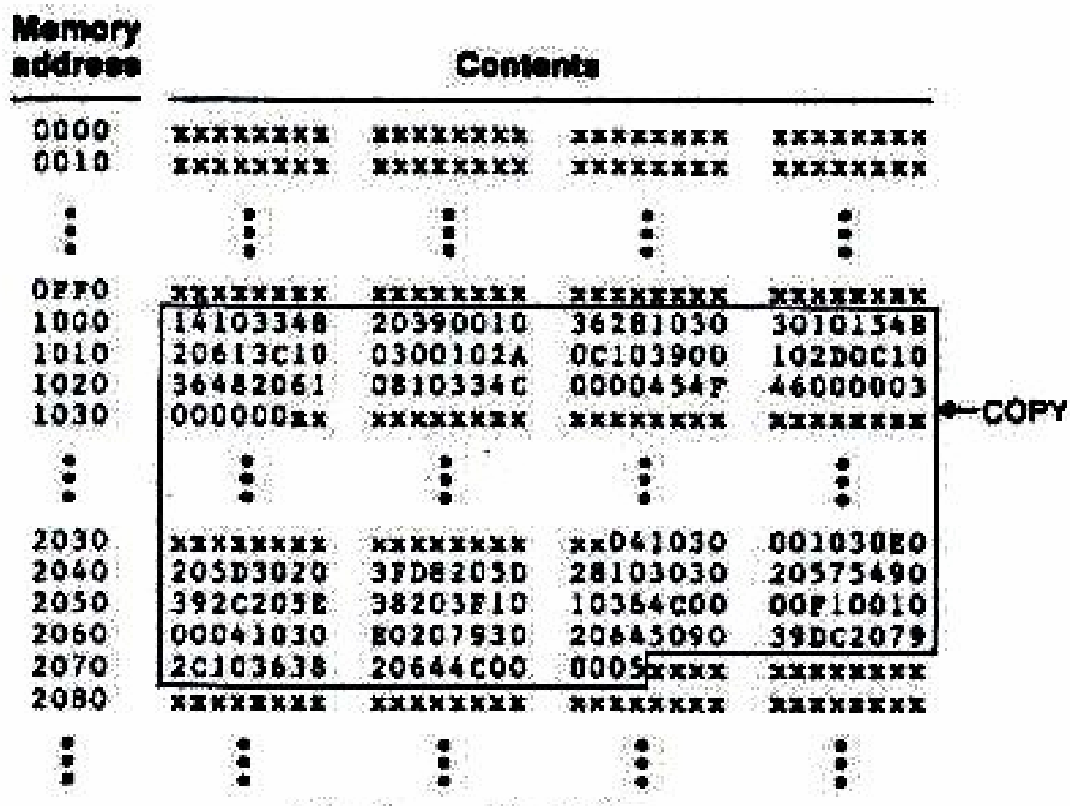
3.1.1 Design of an Absolute Loader

- An example object program is shown in Fig 3.1(a).

```
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

- For a simple absolute loader, all functions are accomplished in a single pass as follows:
  1) The <u>Header record</u> of object programs is checked to verify that the correct program has been presented for loading.
  2) As each <u>Text record</u> is read, the object code it contains is moved to the indicated address in memory.
  3) When the <u>End record</u> is encountered, the loader jumps to the specified address to begin execution of the loaded program.
- Fig 3.1(b) shows a representation of the program from Fig 3.1(a) after loading.



(b) Program loaded in memory

- Fig 3.2 shows an algorithm for the absolute loader we have discussed.

```
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 Algorithm for an absolute loader.

- 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* (a half-byte).
- Therefore, to save space and execution time of loaders, most machines store object programs in a ***binary*** form, with each byte of object code stored as a single byte in the object program.

3.1.2 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.
- Fig 3.3 shows the source code for our bootstrap loader. The bootstrap itself begins at address 0 in the memory.

```
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.

Note: 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.

1) The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80. The

main loop of the bootstrap keeps the address of the next memory location to be loaded in register X.

2) 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.

  GETC is used to read and convert a pair of characters from device F1 representing 1 byte of object code to be loaded. For example, two bytes =C"D8"→'4438'H converting to one byte 'D8'H.

  The resulting byte is stored at the address currently in register X, using STCH instruction that refers to location 0 using indexed addressing.

  The TIXR instruction is then used to add 1 to the value in X.

3.2 Machine-Dependent Loader Features

- The absolute loader has several potential disadvantages.

  One of the most obvious is <u>the need for the programmer to specify the actual address at which it will be loaded into memory</u>.

  Writing absolute programs also makes it difficult to use subroutine libraries efficiently. This could not be done effectively if all of the subroutines had pre-assigned absolute addresses.

- 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.

3.2.1 Relocation

- Two methods for specifying relocation as part of the object program.
- *The first method*: A Modification record (The format is given in Section 2.3.5.) is used to describe each part of the object code that must be changed when the program is relocated.
- Fig 3.4 shows a SIC/XE program we use to illustrate this first method of specifying relocation.

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		.			
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 3.4 Example of a SIC/XE program (from 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.

- Fig 3.5 displays the object program corresponding to the source in Fig 3.4.

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

Figure 3.5 Object program with relocation by Modification records.

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.

- The Modification record is not well suited for use with all machine architectures. Consider, for example, the program in Fig 3.6. This is a *relocatable* program written for standard version for SIC.

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	
	.			
	.	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	

Figure 3.6 Relocatable program for a standard SIC machine.

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 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.

- *The second method*: Fig 3.7 shows this method applied to our SIC program example.

```
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
```

Figure 3.7 Object program with relocation by bit mask.

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 3.7 this mask is represented (in character form) as three hexadecimal digits.
- 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.

For example, the bit mask FFC (representing the bit string 111111111100) in the first Text record specifies that all <u>10 words</u> of object code are to be modified during relocation.

- Example: note that the LDX instruction on line 210 (Fig 3.6) begins a new Text record. <u>If it were placed in the preceding Text record, it would not be properly aligned to correspond to a relocation bit</u> because of the 1-byte data value generated from line 185.

3.2.2 Program Linking

- Consider the three (separately assembled) programs in Fig 3.8, each of which consists of a single control section.

```
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
```

```
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
```

Figure 3.8 Sample programs illustrating linking and relocation.

- Consider first the reference marked REF1.

  For the first program (PROGA), (1) REF1 is <u>simply a reference to a label</u> within the program. (2) It is assembled in the usual way as a PC relative instruction. (3) No modification for relocation or linking is necessary.

  In PROGB, <u>the same operand refers to an external symbol</u>. (1) The assembler uses an extended-format instruction with address field set to 00000. (2) The object program for PROGB (Fig 3.9) contains a Modification record instructing the loader <u>to add *the value of the symbol LISTA* to *this address field*</u> when the program is linked.

```
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
```

```
HPROGB ^000000^00007F
DLISTB ^000060^ENDB  ^000070
RLISTA ^ENDA  ^LISTC ^ENDC
:
T^000036^0B^03100000^772027^05100000
:
T^000070^0F^000000^FFFFF6^FFFFFF^FFFFF0^000060
M^000037^05^+LISTA
M^00003E^05^+ENDA
M^00003E^05^-LISTA
M^000070^06^+ENDA
M^000070^06^-LISTA
M^000070^06^+LISTC
M^000073^06^+ENDC
M^000073^06^-LISTC
M^000076^06^+ENDC
M^000076^06^-LISTC
M^000076^06^+LISTA
M^000079^06^+ENDA
M^000079^06^-LISTA
M^00007C^06^+PROGB
M^00007C^06^-LISTA
E

HPROGC ^000000^000051
DLISTC ^000030^ENDC  ^000042
RLISTA ^ENDA  ^LISTB ^ENDB
:
T^000018^0C^03100000^77100004^05100000
:
T^000042^0F^000030^000008^000011^000000^000000
M^000019^05^+LISTA
M^00001D^05^+LISTB
M^000021^05^+ENDA
M^000021^05^-LISTA
M^000042^06^+ENDA
M^000042^06^-LISTA
M^000042^06^+PROGC
M^000048^06^+LISTA
M^00004B^06^+ENDA
M^00004B^06^-LISTA
M^00004B^06^-ENDB
M^00004B^06^+LISTB
M^00004E^06^+LISTB
M^00004E^06^-LISTA
E
```

Figure 3.9 Object programs corresponding to Fig. 3.8.

For PROGC, REF1 is handled in exactly the same way.

- The reference marked REF2 is processed in a similar manner.
- 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), 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.
- 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 '000014'H and one Modification record.

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 ('000030'H). 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.

- Fig 3.10(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.

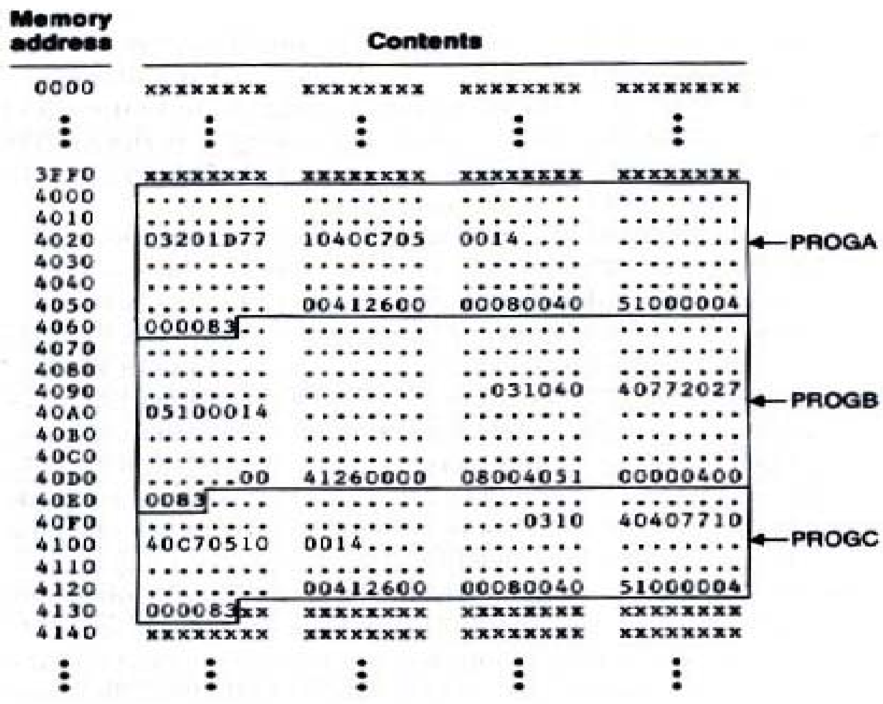


Figure 3.10(a) Programs from Fig. 3.8 after linking and loading.

- For example, the value for reference REF4 in PROGA is located at address 4054 (the beginning address of PROGA plus 0054). Fig 3.10(b) shows the details of how this value is computed.

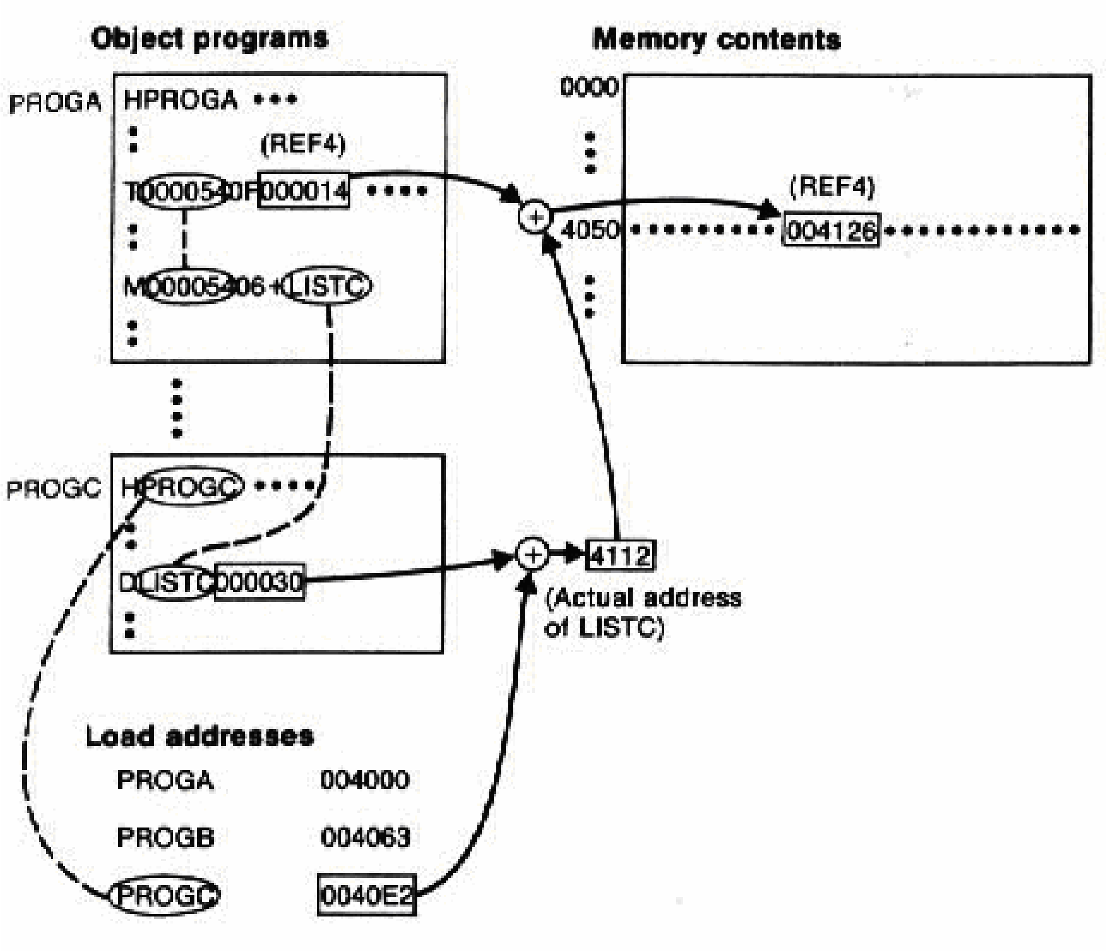


Figure 3.10(b) Relocation and linking operations performed on REF4 from PROGA.

The initial value (from the Text record) is 000014. To this is added the address assigned to LISTC, which 4112 (<u>the beginning address of PROGC</u> plus <u>30</u>).

3.2.3 Algorithm and Data Structures for a Linking Loader

- The algorithm for a *linking loader* is considerably more complicated than the *absolute loader* algorithm discussed in Section 3.1.

- A linking loader usually makes *two passes* over its input, just as an assembler does. In terms of general function, the two passes of a linking loader are quite similar to the two passes of an assembler:

Pass 1 assigns addresses to all external symbols.

Pass 2 performs the actual loading, relocation, and linking.

- The main data structure needed for our linking loader is an *external symbol table* ESTAB.

  This table, which is analogous to SYMTAB in our assembler algorithm, is used to store the *name* and *address* of each external symbol in the set of control sections being loaded.

  A *hashed organization* is typically used for this table.

  Two other important variables are PROGADDR (program load address) and CSADDR (control section address).

  PROGADDR is *the beginning address in memory* where the linked program is to 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. This value is added to all relative addresses within the control section to convert them to actual addresses.

- The algorithm is presented in Fig 3.11.
- During Pass 1 (Fig 3.11(a)), the loader is concerned only with Header and Define record types in the control sections.

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

Pass 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.

1) The beginning load address for the linked program (PROGADDR) is obtained from the OS. This becomes the starting address (CSADDR) for the first control section in the input sequence.

2) The control section name from Header record is entered into ESTAB, with value given by CSADDR. All *external symbols* appearing in the Define record for the control section are also entered into ESTAB. Their addresses are obtained by adding the value specified in the Define record to CSADDR.

3) When the End record is read, the control section length

CSLTH (which was saved from the End record) is added to CSADDR. This calculation gives the starting address for the next control section in sequence.

- At the end of Pass 1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each.
- Many loaders include as an option the ability to print a *load map* that shows these symbols and their addresses. For the example of Figs 3.9 and 3.10, such a load map might look like as shown on the top of page 143.
- Pass 2 (Fig 3.11(b)) of our loader performs the actual *loading*, *relocation*, and *linking* of the program.

```
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.

1) As each Text record is read, the object code is moved to the specified address (plus the current value of CSADDR).

2) When a Modification record is encountered, the symbol whose value is to be used for modification is looked up in ESTAB.

3) This value is then added to or subtracted from the indicated location in memory.

4) The last step performed by the loader is usually the transferring of control to the loaded program to begin execution.

- The End record for each control section may contain the address of the first instruction in that control section to be executed. Our loader takes this as the transfer point to begin execution.

  If more than one control section specifies a transfer address, the loader arbitrarily uses the last one encountered.

  If no control section contains a transfer address, the loader uses the beginning of the linked program (i.e., PROGADDR) as the transfer point.

  Normally, a transfer address would be placed in the End record for a main program, but not for a subroutine.

- This algorithm can be made more efficient. Assign a reference number, which is used (instead of the symbol name) in Modification records, to each external symbol referred to in a control section.

  Suppose we always assign the reference number 01 to the control section name.

  Fig 3.12 shows the object programs from 3.9 with this change.

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

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

HPROGC 000000000051
DLISTC 000030ENDC  000042
R02LISTA 03ENDA  04LISTB 05ENDB
:
T0000180C03100000771000040510000
:
T0000420F000030000008000011000000000000
M00001905+02
M00001D05+04
M00002105+03
M00002105-02
M00004206+03
M00004206-02
M00004206+01
M00004806+02
M00004B06+03
M00004B06-02
M00004B06-05
M00004B06+04
M00004E06+04
M00004E06-02
E
```

Figure 3.12 Object programs corresponding to Fig. 3.8 using reference numbers for code modification. (Reference numbers are underlined for easier reading.)

3.3 Machine-Independent Loader Features

- Loading and linking are often thought of as OS service functions. Therefore, most loaders include fewer different features than are found in a typical assembler. They include the use of an automatic library search process for handling external reference and some common options that can be selected at the time of loading and linking.

3.3.1 Automatic Library Search

- Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded.
- 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.
- 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.

  Note that the subroutines fetched from a library in this way may themselves contain external references. It is therefore necessary to repeat the library search process until all references are resolved.

  If unresolved external references remain after the library search is completed, these must be treated as errors.

3.3.2 Loader Options

- Many loaders allow the user to specify options that modify the standard processing described in Section 3.2.

- Typical loader option 1: allows the selection of alternative sources of input. Ex.,

  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.

- Loader option 2: allows the user to delete external symbols or entire control sections. Ex.,

  DELETE csect-name

  might instruct the loader to delete the named control section(s) from the set of programs being loaded.

  CHANGE name1, name2

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

- Loader option 3: involves the automatic inclusion of library routines to satisfy external references. Ex.,

  LIBRARY MYLIB

  Such user-specified libraries are normally searched before the standard system libraries. This allows the user to use special versions of the standard routines.

  NOCALL STDDEV, PLOT, CORREL

  To instruct the loader that these external references are to remain unresolved. This avoids the overhead of loading and linking the unneeded routines, and saves the memory space that would otherwise be required.

3.4 Loader Design Options

- Linking loaders perform all linking and relocation at load time. There are two alternatives: ***Linkage editors***, which *perform linking prior to load time*, and ***dynamic linking***, in which *the linking function is performed at execution time*.

- Precondition: The source program is first assembled or compiled, producing an object program.

  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* produces a linked version of the program (*load module* or *executable image*), which is written to a file or library for later execution.

- The essential difference between a *linkage editor* and a *linking loader* is illustrated in Fig 3.13.

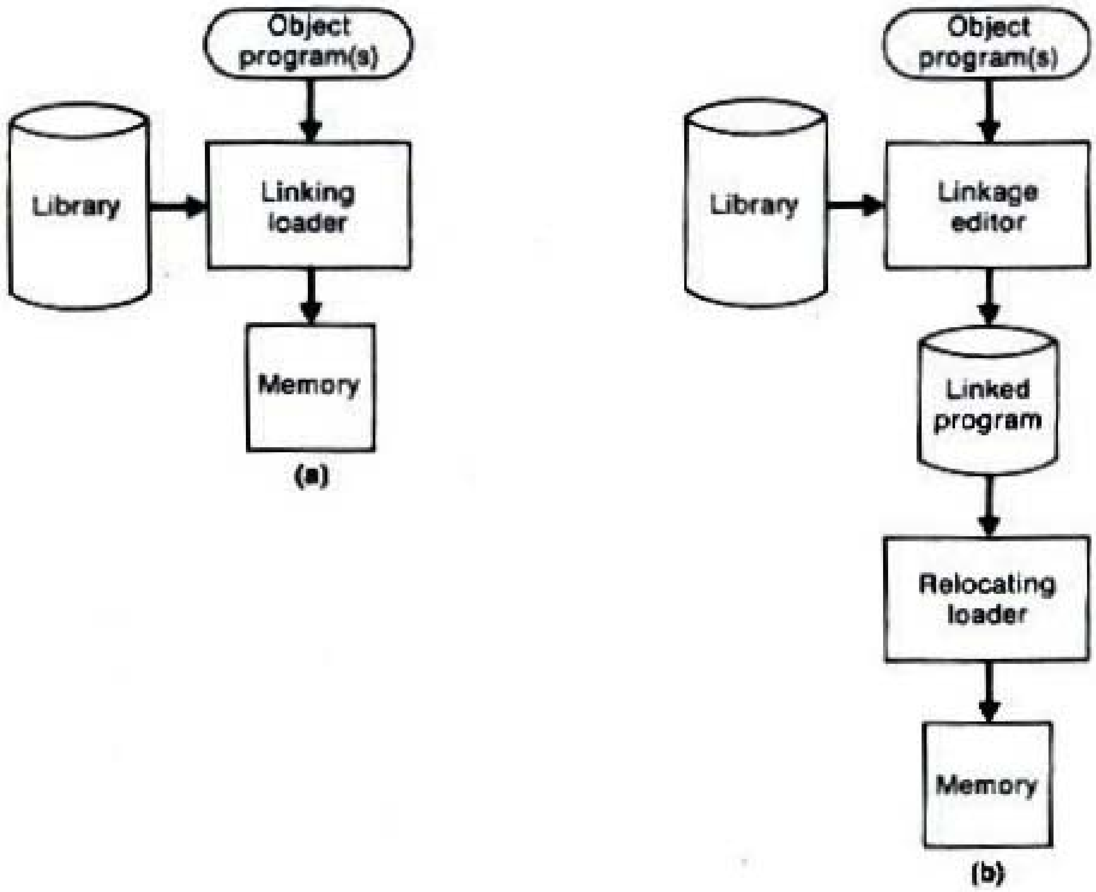


Figure 3.13 Processing of an object program using (a) linking loader and (b) linkage editor.

3.4.1 Linkage Editors

- 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.

  If a program is to be executed many times without being reassembled, the use of a linkage editor substantially reduces the overhead required.

- Linkage editors can perform many useful functions besides simply preparing an object program for execution. Ex., a typical sequence of linkage editor commands used:

```
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. This can be useful when dealing with subroutine libraries that support high-level programming languages.

- Linkage editors often include a variety of other options and commands like those discussed for linking loaders. Compared to linking loaders, linkage editors in general tend to offer more flexibility and control.

3.4.2 Dynamic Linking

- *Linkage editors* perform linking operations before the program is loaded for execution.

  Linking loaders perform these same operations at load time.

Dynamic linking, *dynamic loading*, or *load on call* postpones the linking function until execution time: a subroutine is loaded and linked to the rest of the program when it is first called.

- Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library, ex. run-time support routines for a high-level language like C.
- With a program that allows its user to interactively call any of the subroutines of a large mathematical and statistical library, all of the library subroutines could potentially be needed, but only a few will actually be used in any one execution.

  Dynamic linking can avoid the necessity of loading the entire library for each execution except those necessary subroutines.
- Fig 3.14 illustrates a method in which routines that are to be dynamically loaded must be called via an *OS service request*.

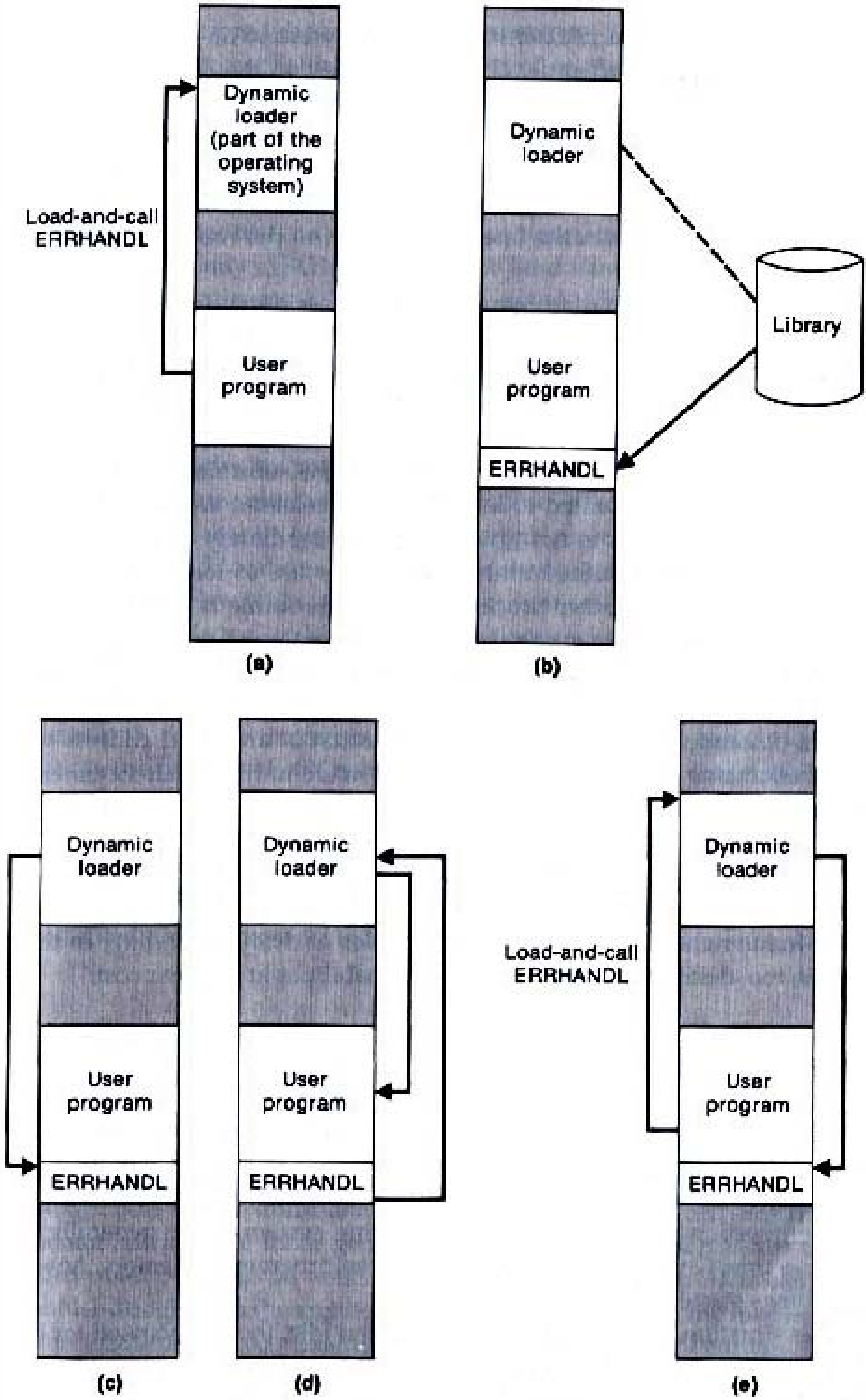


Figure 3.14 Loading and calling of a subroutine using dynamic linking.

Fig 3.14(a): Instead of executing a JSUB instruction referring to an external symbol, the program makes a load-and-call service request to OS. The parameter of this request is the symbolic name of the routine to be called.

Fig 3.14(b): OS examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded from the specified user or system libraries.

Fig 3.14(c): Control is then passed from OS to the routine being called

Fig 3.14(d): When the called subroutine completes it processing, it returns to its caller (i.e., OS). OS then returns control to the program that issued the request.

Fig 3.14(e): If a subroutine is still in memory, a second call to it may not require another load operation. Control may simply be passed from the dynamic loader to the called routine.

3.4.3 Bootstrap Loaders

- Given an idle computer with no program in memory, how do we get things started?
- On some computers, an absolute loader program is permanently resident in a read-only memory (ROM). When some hardware signal occurs, the machine begins to execute this ROM program. This is referred to as a *bootstrap loader*.

3.5 Implementation Examples

(Skip)