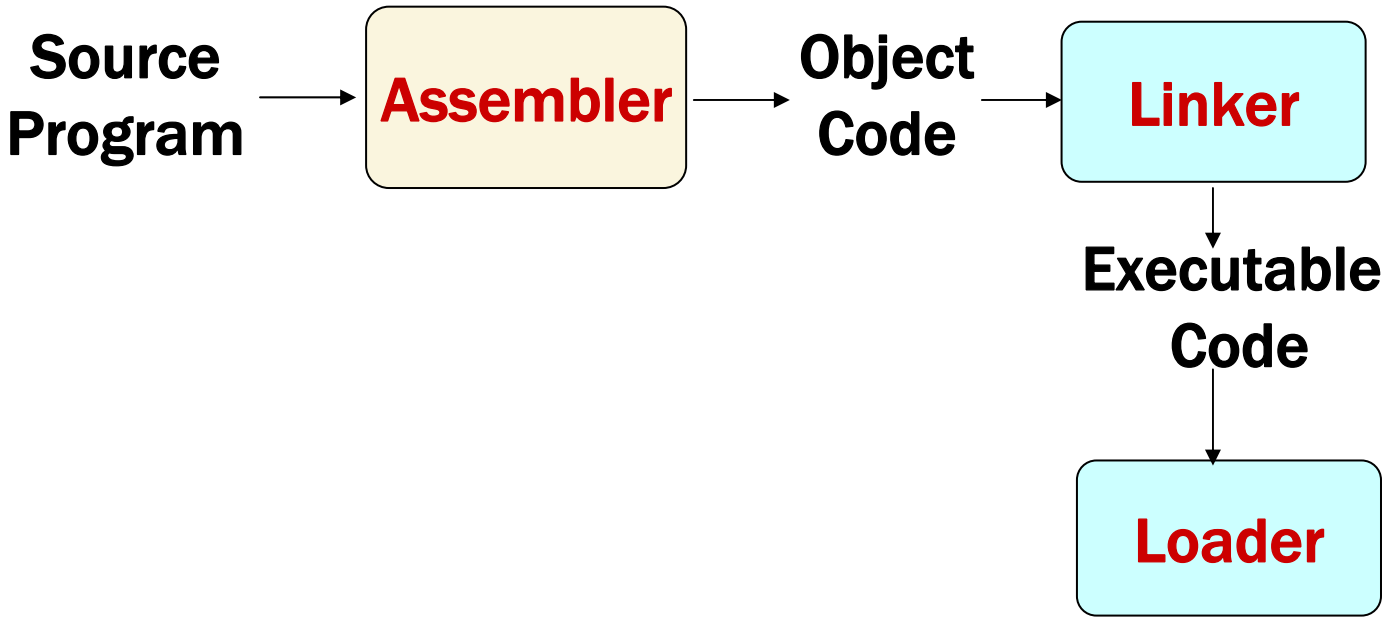
Chapter 3
Loaders and Linkers



3.1 Basic Loader Functions

- In Chapter 2, we discussions
  - *Loading:* brings the OP into memory for execution
  - *Relocating:* modifies the OP so that it can be loaded at an address different form the location originally specified.
  - *Linking:* combines two or more separate OPs (set 2.3.5)
- In Chapter 3, we will discussion
  - A *loader* brings an object program into memory and starting its execution.
  - A *linker* performs the linking operations and a separate loader to handle relocation and loading.

3.1 Basic Loader Functions
3.1.1 Design of an Absolute Loader

- Absolute loader (for SIC), in Figures 3.1 and 3.2.
  - Does not perform linking and program relocation.
  - The contents of memory locations for which there is no Text record are shown as xxxx.
  - Each byte of assembled code is given using its Hex representation in character form.

```
HCOPY  001000001007A
T0010001E14103348203900103628103030101548206 13C100300102A0C103900102D
T00101E150C10364820610810334C0000454F46000003000000
T002039 1E041030001030E0205D30203FD8205D28103030205754903 92C205E38203F
T002057 1C1010364C0000F1001000041030E02079302064509039 DC20792C1036
T00207307382064 4C000005
E001000
```

(a) Object program

3.1.1 Design of an Absolute Loader

- Absolute loader, in Figure 3.1 and 3.2.
  - STL instruction, *pair of characters* 14, when these are read by loader, they will occupy *two* bytes of memory.
  - 14 (Hex 31 34) —> 00010100 (one byte)
  - For execution, the operation code must be store in a single byte with hexadecimal value 14.
  - Each pair of bytes must be packed together into one byte.
  - Each printed character represents one half-byte.

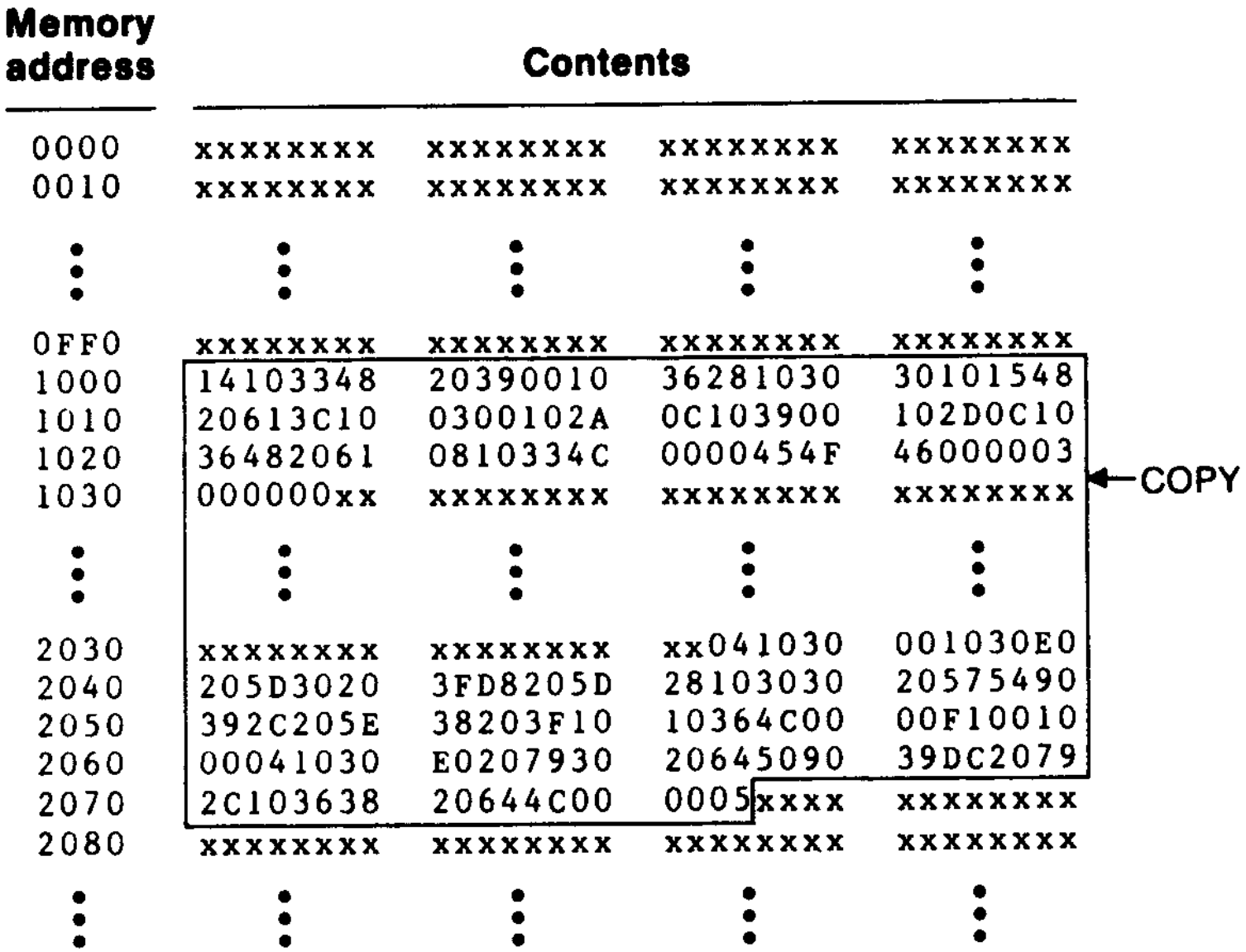


Figure 3.1 Loading of an absolute program.

```
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.

3.1.2 A Simple Bootstrap Loader

- A bootstrap loader, Figure 3.3.
  - Loads the first program to be run by the computer—usually an operating system.
  - The bootstrap itself begins at address 0 in the memory.
  - It loads the OS or some other program starting at address 80.

3.1.2 A Simple Bootstrap Loader

- A bootstrap loader, Figure 3.3.
  - Each byte of object code to be loaded is represented on device F1 as two Hex digits (by GETC subroutines).
  - The ASCII code for the character 0 (Hex 30) is converted to the numeric value 0.
  - The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80.

```
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.

3.2 Machine-Dependent Loader Features

- Absolute loader has several potential disadvantages.
  - The actual address at which it will be loaded into memory.
  - Cannot run several independent programs together, sharing memory between them.
  - It difficult to use subroutine libraries efficiently.
- More complex loader.
  - Relocation
  - Linking
  - Linking loader

3.2.1 Relocation

- Relocating loaders, two methods:
  - Modification record (for SIC/XE)
  - Relocation bit (for SIC)

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).

3.2.1 Relocation

- Modification record, Figure 3.4 and 3.5.
  - To described each part of the object code that must be changed when the program is relocated.
  - The extended format instructions on lines 15, 35, and 65 are affected by relocation. (absolute addressing)
  - In this example, all modifications add the value of the symbol COPY, which represents the starting address.
  - Not well suited for *standard version of SIC*, all the instructions except RSUB must be modified when the program is relocated. (absolute addressing)

```
HCOPY  000000001077
T0000001D17202D69202D4B10103603202629000033200 74B10105D3F2FEC032010
T00001D130F20160100030F200D4B10105D3E2003454F46
T001036 1DB410B400B44075101000E32019332FFADB2013A00433200857C003B850
T001053 1D3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T00107007 3B2FEF4F000005
M00000705+COPY
M00001405+COPY
M00002705+COPY
E000000
```

Figure 3.5 Object program with relocation by Modification records.

3.2.1 Relocation

- Figure 3.6 needs 31 Modification records.
- Relocation bit, Figure 3.6 and 3.7.
  - A *relocation bit* associated with each word of object code.
  - The relocation bits are gathered together into a *bit mask* following the length indicator in each Text record.
  - If bit=1, the corresponding word of object code is relocated.

Line	Loc	Source statement			Object code	
5	0000	COPY	START	0		
10	0000	FIRST	STL	RETADR	140033	1
15	0003	CLOOP	JSUB	RDREC	481039	1
20	0006		LDA	LENGTH	000036	1
25	0009		COMP	ZERO	280030	1
30	000C		JEQ	ENDFIL	300015	1
35	000F		JSUB	WRREC	481061	1
40	0012		J	CLOOP	3C0003	1
45	0015	ENDFIL	LDA	EOF	00002A	1
50	0018		STA	BUFFER	0C0039	1
55	001B		LDA	THREE	00002D	1
60	001E		STA	LENGTH	0C0036	1
65	0021		JSUB	WRREC	481061	1
70	0024		LDL	RETADR	080033	1
75	0027		RSUB		4C0000	0
80	002A	EOF	BYTE	C'EOF'	454F46	0
85	002D	THREE	WORD	3	000003	0
90	0030	ZERO	WORD	0	000000	0
95	0033	RETADR	RESW	1		
100	0036	LENGTH	RESW	1		
105	0039	BUFFER	RESB	4096		

```
110                 .
115                 .       SUBROUTINE TO READ RECORD INTO BUFFER
120                 .
125     1039        RDREC   LDX     ZERO            041030  1
130     103C                LDA     ZERO            001030  1
135     103F        RLOOP   TD      INPUT           E0105D  1
140     1042                JEQ     RLOOP           30103F  1
145     1045                RD      INPUT           D8105D  1
150     1048                COMP    ZERO            281030  1
155     104B                JEQ     EXIT            301057  1
160     104E                STCH    BUFFER,X        548039  1
165     1051                TIX     MAXLEN          2C105E  1
170     1054                JLT     RLOOP           38103F  1
175     1057        EXIT    STX     LENGTH          101036  1
180     105A                RSUB                    4C0000  0
185     105D        INPUT   BYTE    X'F1'           F1      0
190     105E        MAXLEN  WORD    4096            001000  0
```

```
195                 .
200                 .       SUBROUTINE TO WRITE RECORD FROM BUFFER
205                 .
210     1061        WRREC   LDX     ZERO            041030  1
215     1064        WLOOP   TD      OUTPUT          E01079  1
220     1067                JEQ     WLOOP           301064  1
225     106A                LDCH    BUFFER,X        503039  1
230     106D                WD      OUTPUT          DC1079  1
235     1070                TIX     LENGTH          2C1036  1
240     1073                JLT     LOOP            381064  1
245     1076                RSUB                    4C1000  0
250     1079        OUTPUT  BYTE    X'05'           05      0
255                         END     FIRST
```

Figure 3.6 Relocatable program for a standard SIC machine.

3.2.1 Relocation

- Relocation bit, Figure 3.6 and 3.7.
  - In Figure 3.7, T000000^1E^FFC^ (111111111100) specifics that all 10 words of object code are to be modified.
  - On line 210 begins a new Text record even though there is room for it in the preceding record.
  - Any value that is to be modified during relocation must coincide with one of these 3-byte segments so that it corresponding to a relocation bit.
  - Because of the 1-byte data value generated form line 185, this instruction must begin a new Text record in object program.

1111 1111 1100

```
HCOPY  000000001007A
T00000001EFFC140033481039000036280030300015481061 3C000300002A0C003900002D
T00001E15E000C003648106108003 34C0000454F46000003000000
T001039 1EFFC040030000030E0105D30103FD8105D280030301057548039 2C105E38103F
T001057 0A8001000364C0000F1001000
T001061 19FE0040030E01079301064508039DC10792C00363810644C000005
E000000
```

Figure 3.7 Object program with relocation by bit mask.

1110 0000 0000

3.2.2 Program Linking

- In Section 2.3.5 showed a program made up of three controls sections.
  - Assembled together or assembled independently.

3.2.2 Program Linking

- Consider the three programs in Fig. 3.8 and 3.9.
  - Each of which consists of a single control section.
  - A list of items, LISTA—ENDA, LISTB—ENDB, LISTC—ENDC.
  - Note that each program contains exactly the same set of references to these external symbols.
  - Instruction operands (REF1, REF2, REF3).
  - The values of data words (REF4 through REF8).
  - Not involved in the relocation and linking are omitted.

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	+LISTA-1	00003F
005D	REF7	WORD	ENDA-LISTA-( )	000014
0060	REF8	WORD	-LISTA	FFFFC0
		END	REF1	

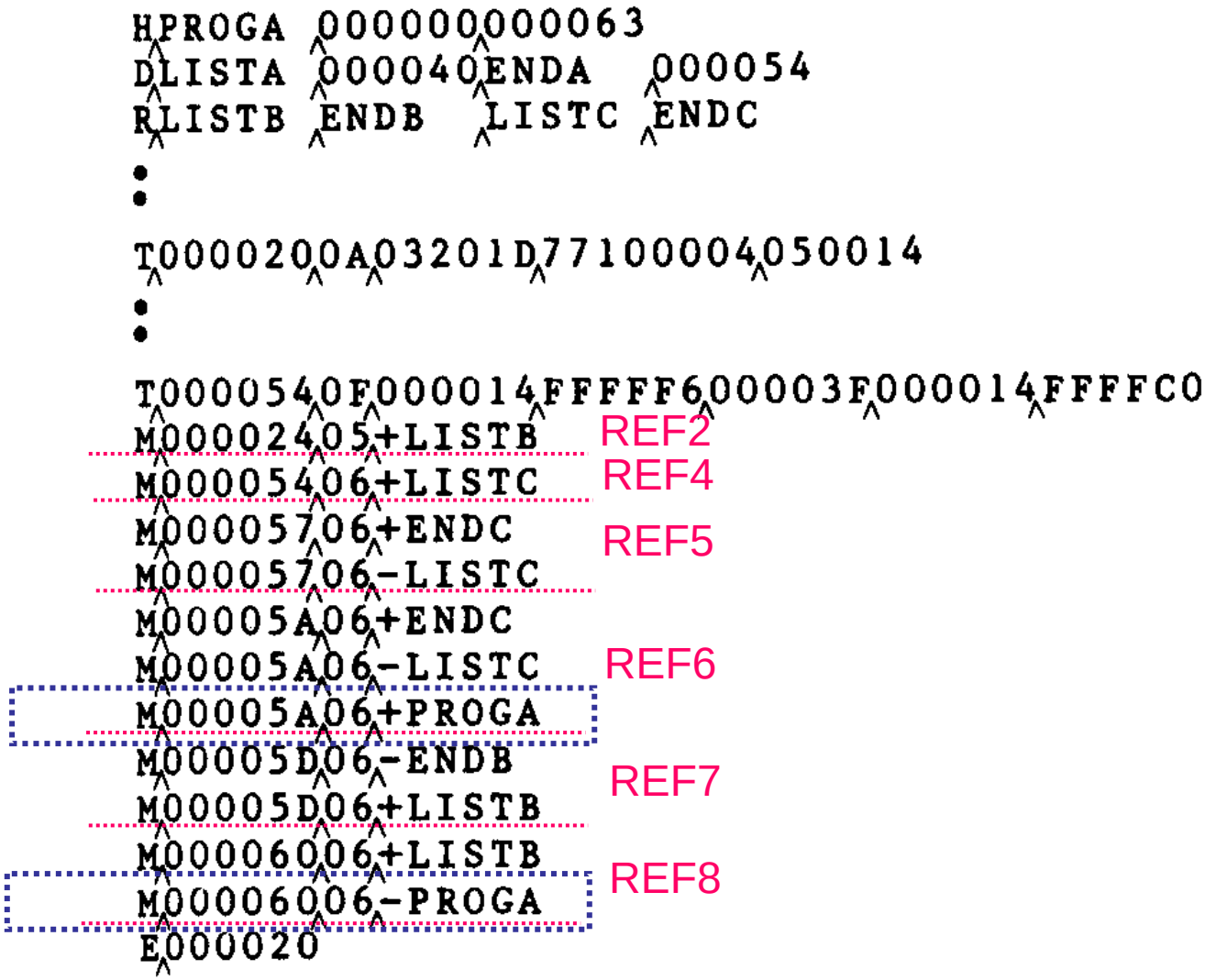
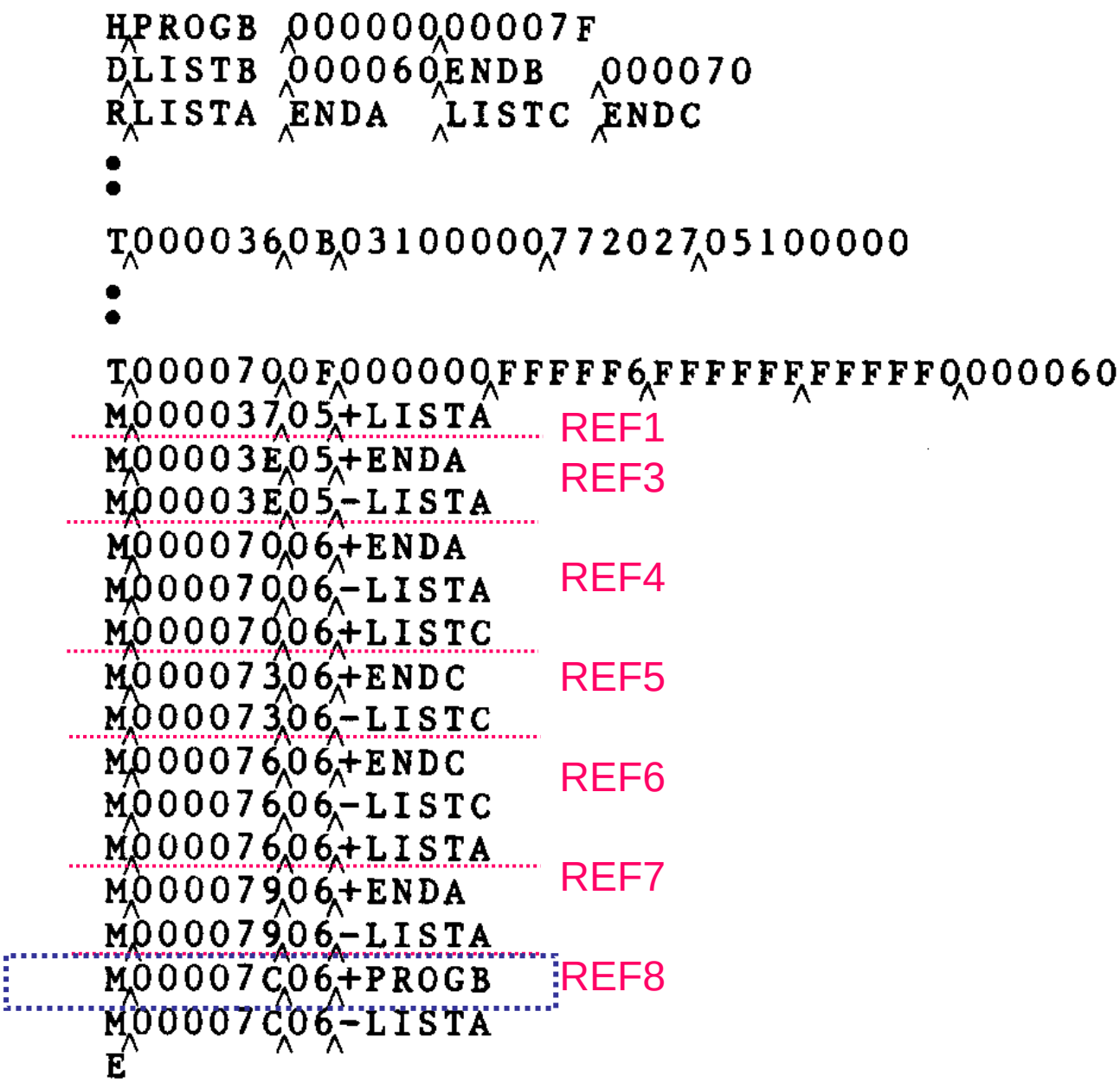


Figure 3.9 Object programs corresponding to Fig. 3.8.

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.


HPROGB 000000000007F
DLISTB 000060ENDB 000070
RLISTA ENDA LISTC ENDC
T0000360B03100000772027051000000
T00007000F000000FFFFF6FFFFFFFFFFF0000060
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
REF1
REF3
REF4
REF5
REF6
REF7
REF8

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		

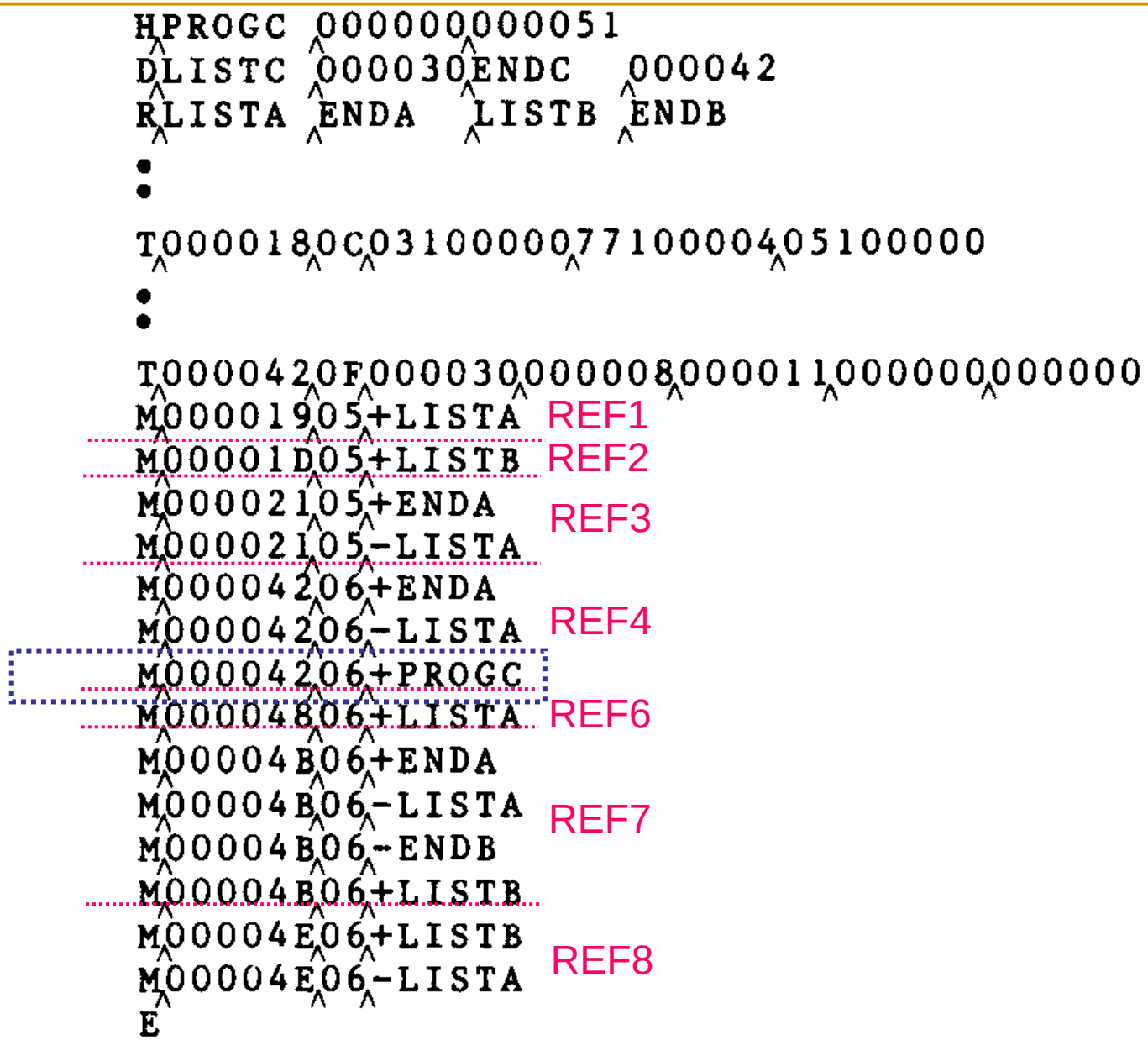


Figure 3.9 (*cont'd*)

3.2.2 Program Linking

- REF1,

```
LDA     LISTA     03201D      03100000
```

  - In the PROGA, REF1 is simply a reference to a label.
  - In the PROGB and PROGC, REF1 is a reference to an external symbols.
  - Need use extended format, Modification record.

- REF2 and REF3.

```
LDT     LISTB+4       772027          77100004
LDX     #ENDA-LISTA   050014          05100000
```

3.2.2 Program Linking

- REF4 through REF8,
  - `WORD  ENDA-LISTA+LISTC  000014+000000`
- Figure 3.10(a) and 3.10(b)
  - Shows these three programs as they might appear in memory after loading and linking.
  - PROGA 004000, PROGB 004063, PROGC 0040E2.
  - REF4 through REF8 in the same value.
  - For the references that are instruction operands, the calculated values after loading do not always appear to be equal.
  - Target address, REF1 4040.

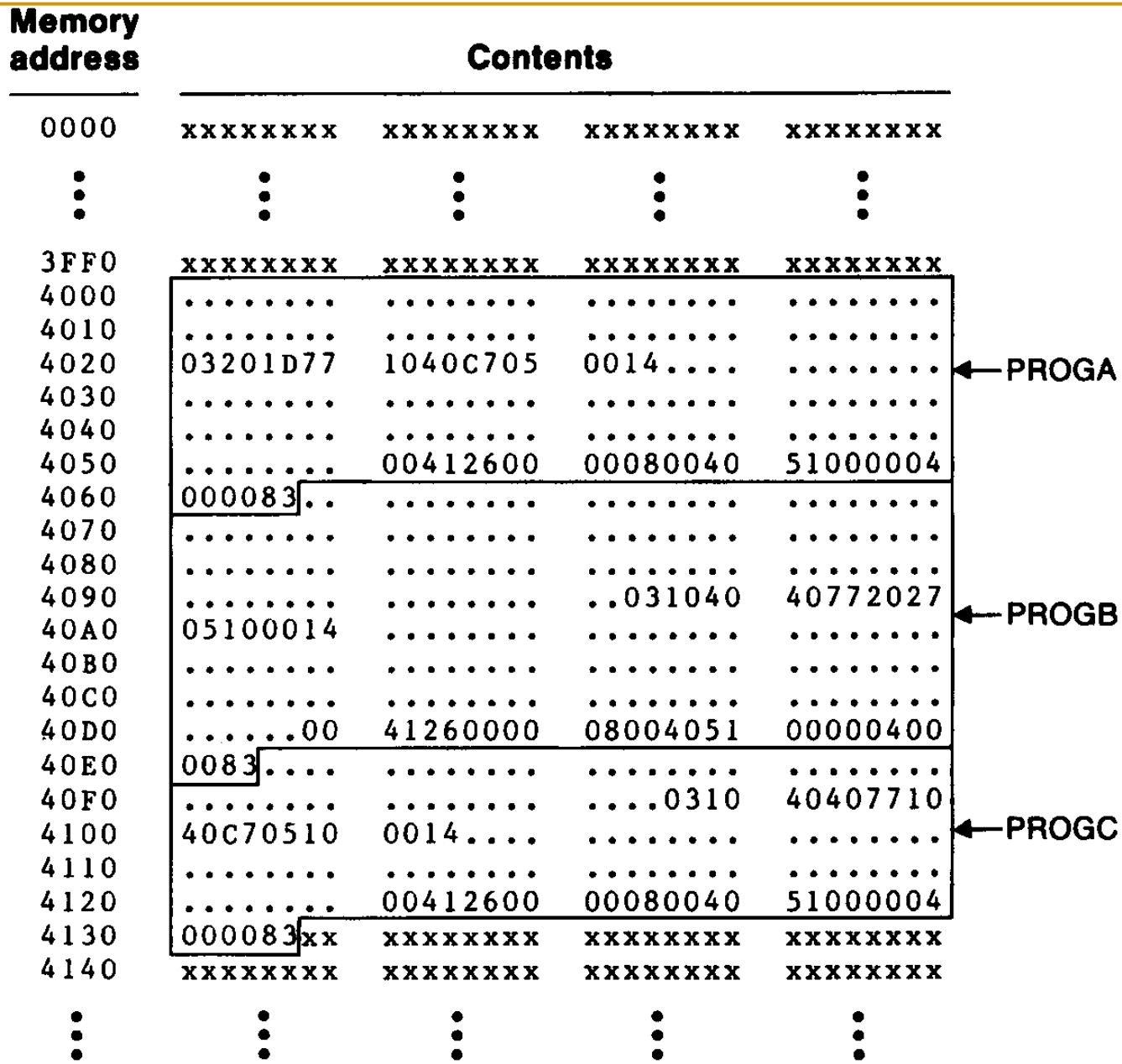


Figure 3.10(a) Programs from Fig. 3.8 after linking and loading.

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

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

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

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

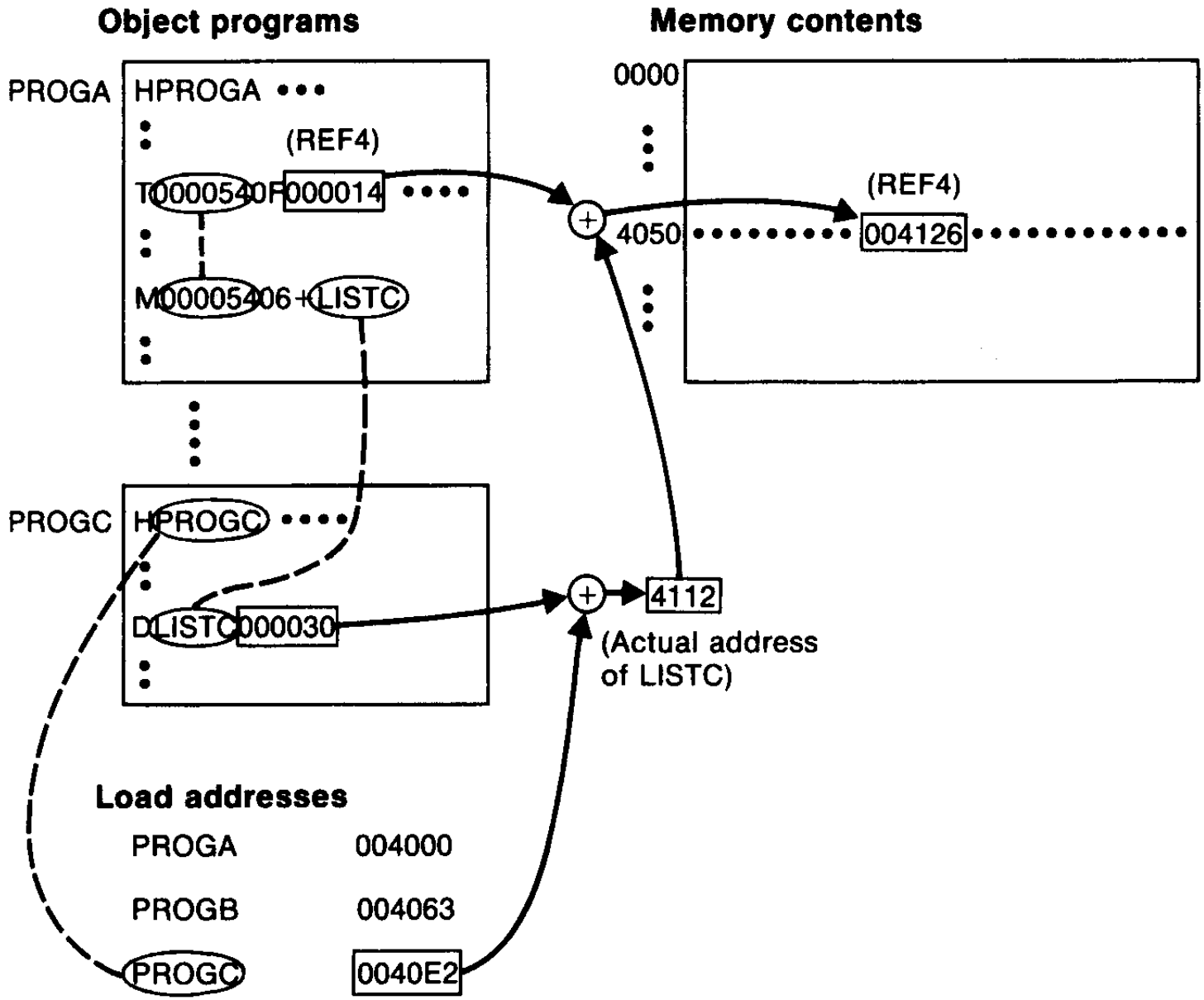


Figure 3.10(b) Relocation and linking operations performed on REF4 from PROGA.

3.2.3 Algorithm and Data Structure for a Linking Loader

- A linking loader usually makes two passes
  - Pass 1 assigns addresses to all external symbols by creating ESTAB.
  - Pass 2 performs the actual loading, relocation, and linking by using ESTAB.
  - The main data structure is ESTAB (hashing table).

3.2.3 Algorithm and Data Structure for a Linking Loader

- A linking loader usually makes two passes
  - ESTAB is used to store the name and address of each external symbol in the set of control sections being loaded.
  - Two variables PROGADDR and CSADDR.
  - PROGADDR is the beginning address in memory where the linked program is to be loaded.
  - CSADDR contains the starting address assigned to the control section currently being scanned by the loader.

3.2.3 Algorithm and Data Structure for a Linking Loader

- The linking loader algorithm, Fig 3.11(a) & (b).
  - In Pass 1, concerned only Header and Defined records.
  - CSADDR+CSLTH = the next CSADDR.
  - A load map is generated.
  - In Pass 2, as each Text record is read, the object code is moved to the specified address (plus the current value of 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.

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.

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.

3.2.3 Algorithm and Data Structure for a Linking Loader

- The algorithm can be made more efficient.
  - A *reference number*, is used in Modification records.
  - The number 01 to the control section name.
  - Figure 3.12, the main advantage of this reference-number mechanism is that it avoids multiple searches of ESTAB for the same symbol during the loading of a control section.

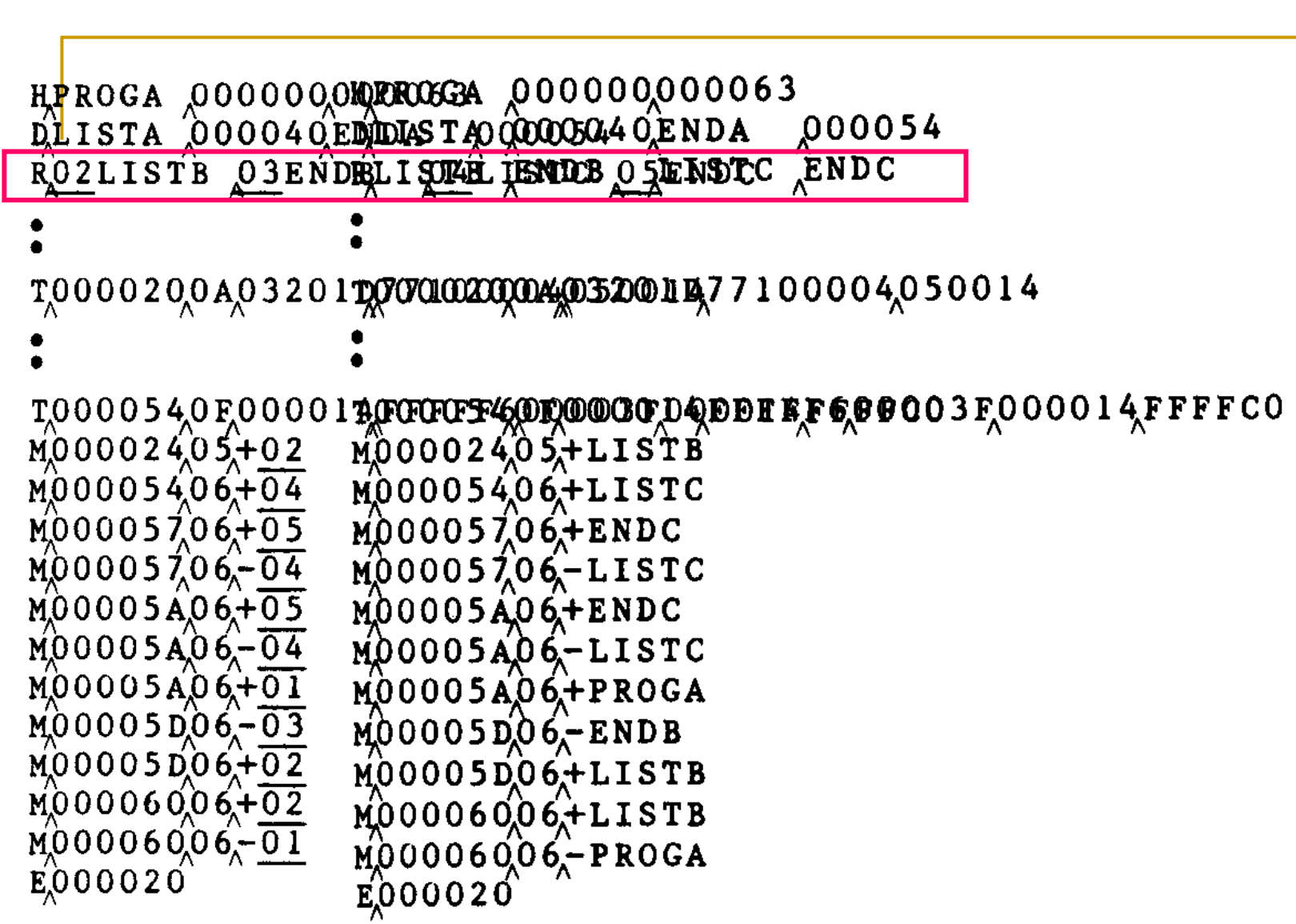


Figure 3.12 Object programs for Fig. 3.8. (Reference numbers are underlined for easier reading.)

```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
R02LISTA 03ENDA  04LISTC 05ENDC
•
•
T0000360B031000007720270510000O
•
•
T000070 0F000000FFFFF6FFFFFFFFFF0000060
M00003705+02
M00003E05+03
M00003E05-02
M00007006+03
M00007006-02
M00007006+04
M00007306+05
M00007306-04
M00007606+05
M00007606-04
M00007606+02
M00007906+03
M00007906-02
M00007C06+01
M00007C06-02
E
```

```
HPROGC 000000000051
DLISTC 000030ENDC  000042
R02LISTA 03ENDA  04LISTB 05ENDB
•
•
T0000180C0310000077100004051000000
•
•
T0000420F000030000008000011000000000000
M00001905+02
M00001D05+04
M0000210 5+03
M0000210 5-02
M0000420 6+03
M0000420 6-02
M0000420 6+01
M0000480 6+02
M00004B0 6+03
M00004B0 6-02
M00004B0 6-05
M00004B0 6+04
M00004E0 6+04
M00004E0 6-02
E
```

Figure 3.12 (*cont'd*)

3.3 Machine-Independent Loader Features
3.3.1 Automatic Library Search

- Many linking loaders
  - Can automatically incorporate routines form a subprogram library into the program being loaded.
  - A standard system library
  - The subroutines called by the program begin loaded are automatically fetched from the library, linked with the main program, and loaded.

3.3.1 Automatic Library Search

- Automatic library call
  - At the end of Pass 1, the symbols in ESTAB that remain undefined represent unresolved external references.
  - The loader searches the library

3.3.2 Loader Options

- Many loaders allow the user to specify options that modify the standard processing.
  - Special command
  - Separate file
  - INCLUDE program-name(library-name)
  - DELETE csect-name
  - CHANGE name1, name2

```
INCLUDE     READ(UTLIB)
INCLUDE     WRITE(UTLIB)
DELETE      RDREC, WRREC
CHANGE      RDREC, READ
CHANGE      WRREC, WRITE
LIBRARY     MYLIB
NOCALL      STDEV, PLOT, CORREL
```

3.4 Loader Design Options
3.4.1 Linkage Editors

- Fig 3.13 shows the difference between linking loader and linkage editor.
  - The source program is first assembled or compiled, producing an OP.
- Linking loader
  - A linking loader performs all linking and relocation operations, including automatic library search if specified, and loads the linked program directly into memory for execution.

- The essential difference between a linkage editor and a linking loader

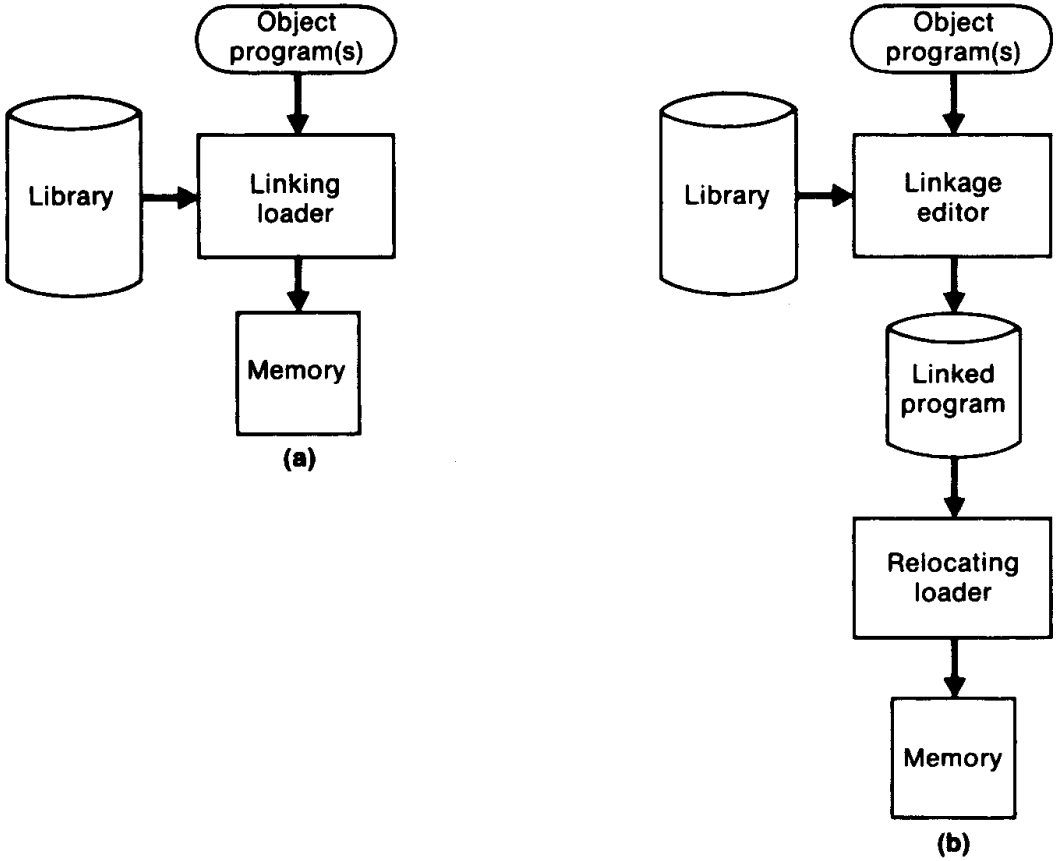


Figure 3.13 Processing of an object program using (a) linking loader and (b) linkage editor.

3.4.1 Linkage Editors

- Linkage editor
  - 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.
  - 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 LE performs relocation of all control sections relative to the start of the linked program.

3.4.1 Linkage Editors

- All items that need to be modified at load time have values that are relative to the start of the linked program.
- If a program is to be executed many times without being reassembled, the use of a LE substantially reduces the overhead required.
- LE can perform many useful functions besides simply preparing an OP for execution.

```
INCLUDE   PLANNER(PROGLIB)
DELETE    PROJECT              {DELETE from existing PLANNER}
INCLUDE   PROJECT(NEWLIB)      {INCLUDE new version}
REPLACE   PLANNER(PROGLIB)

INCLUDE   READR(FTNLIB)
INCLUDE   WRITER(FTNLIB)

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

3.4.2 Dynamic Linking

- Linking loaders perform these same (linking) operations at load time.
- Linkage editors perform linking operations before the program is load for execution.
- Dynamic Linking postpones the linking function until execution time.

3.4.2 Dynamic Linking

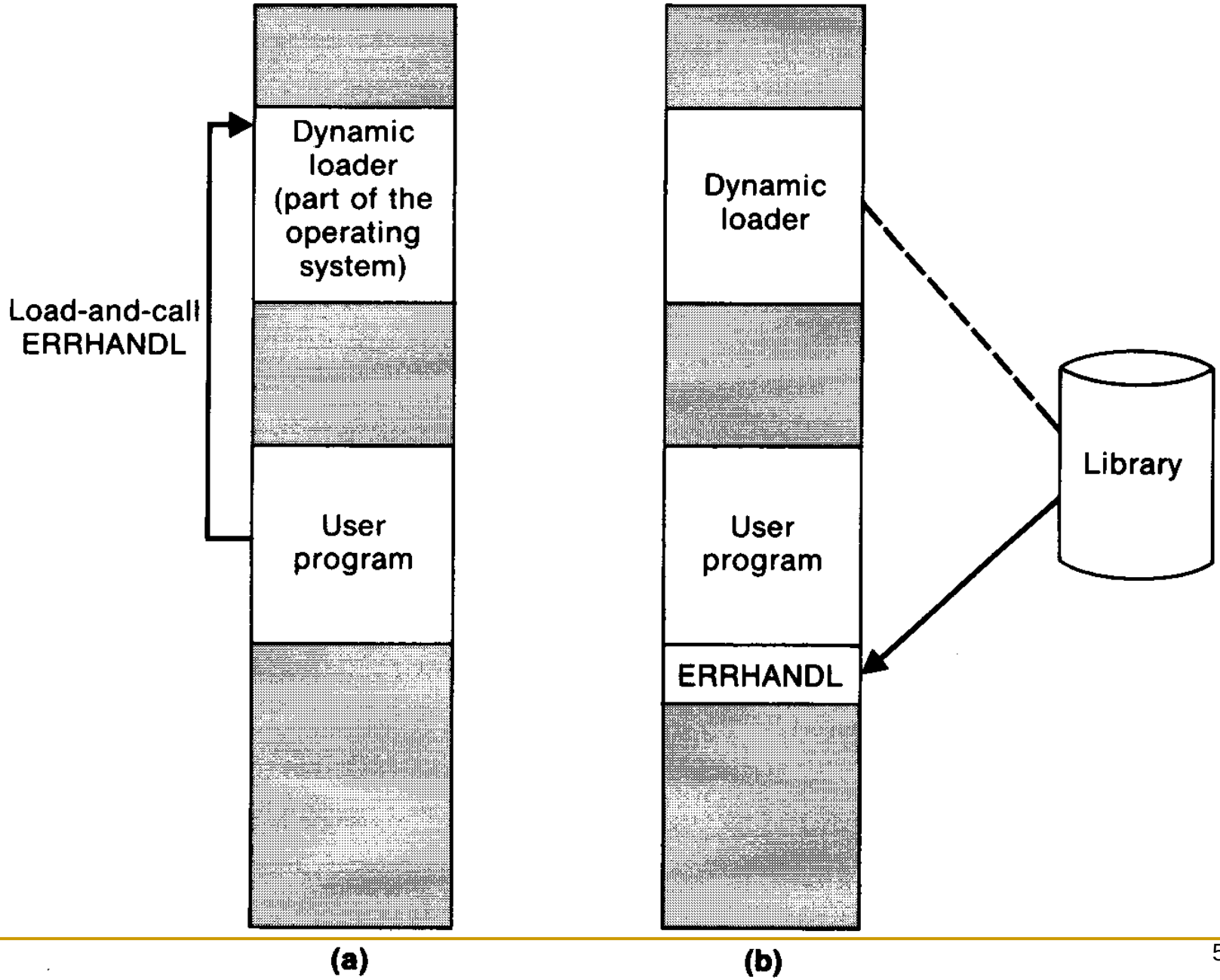
- Dynamic linking (dynamic loading, load on call)
  - Postpones the linking function until execution time.
  - A subroutine is loaded and linked to the rest the program when is first loaded.
  - Dynamic linking is often used to allow several executing program to share one copy of a subroutine or library.
    - Run-time library (C language), dynamic link library
    - A single copy of the routines in this library could be loaded into the memory of the computer.

3.4.2 Dynamic Linking

- Dynamic linking provides the ability to load the routines only when (and if) they are needed.
  - For example, that a program contains subroutines that correct or clearly diagnose error in the input data during execution.
  - If such error are rare, the correction and diagnostic routines may not be used at all during most execution of the program.
  - However, if the program were completely linked before execution, these subroutines need to be loaded and linked every time.

3.4.2 Dynamic Linking

- Dynamic linking avoids the necessity of loading the entire library for each execution.
- Fig. 3.14 illustrates a method in which routines that are to be dynamically loaded must be called via an operating system (OS) service request.


Dynamic loader (part of the operating system)
Load-and-call ERRHANDL
User program
(a)
Dynamic loader
User program
ERRHANDL
Library
(b)

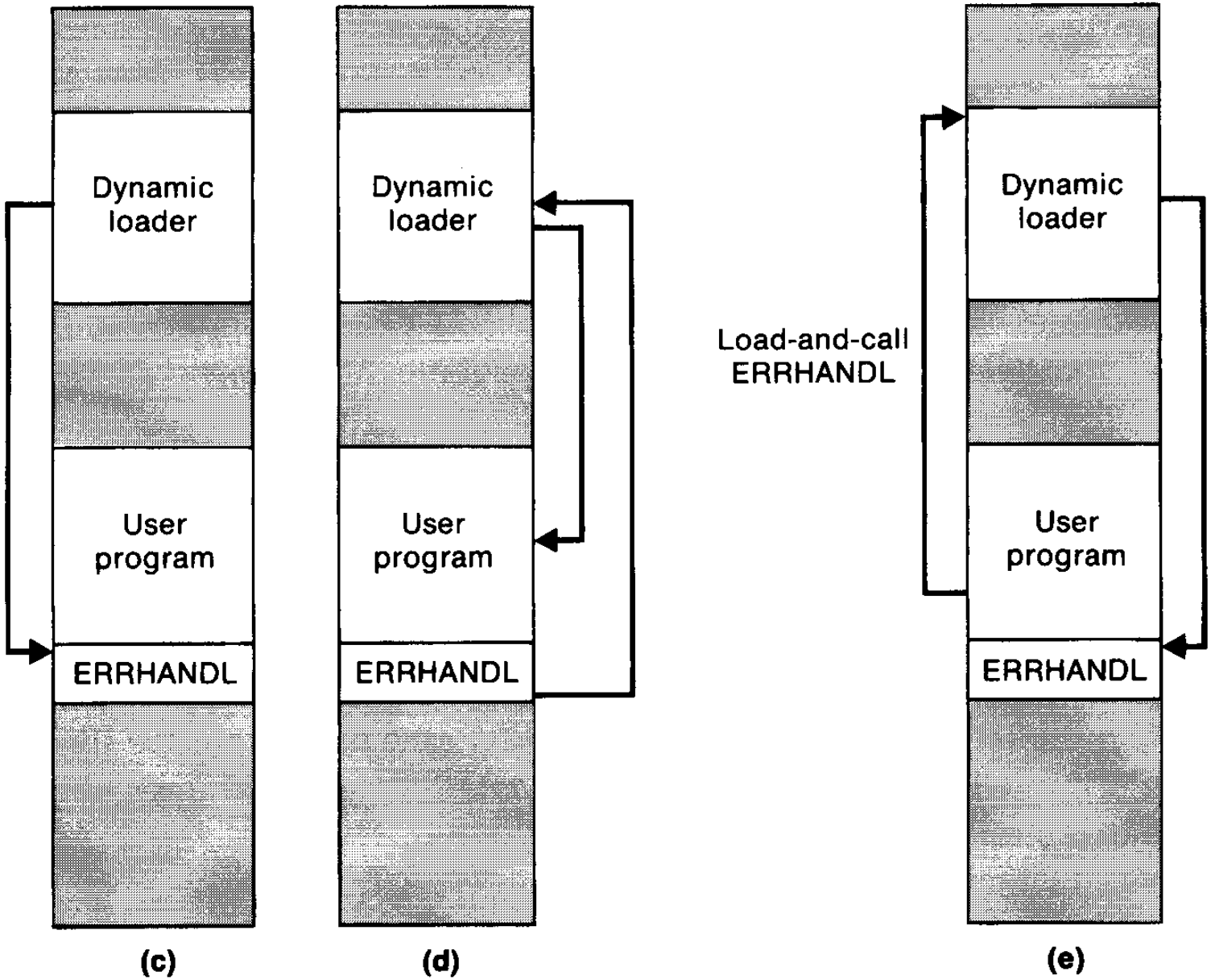


Figure 3.14 Loading and calling of a subroutine using dynamic linking.

3.4.2 Dynamic Linking

- The program makes a load-and-call service request to OS. The parameter argument (ERRHANDL) of this request is the symbolic name of the routine to be loaded.
- OS examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded form the specified user or system libraries.
- Control id then passed form OS to the routine being called.
- When the called subroutine completes its processing, OS then returns control to the program that issued the request.
- If a subroutine is still in memory, a second call to it may not require another load operation.

3.4.3 Bootstrap Loaders

- An absolute loader program is permanently resident in a read-only memory (ROM)
  - Hardware signal occurs
- The program is executed directly in the ROM
- The program is copied from ROM to main memory and executed there.

3.4.3 Bootstrap Loaders

- Bootstrap and bootstrap loader
  - Reads a fixed-length record form some device into memory at a fixed location.
  - After the read operation is complete, control is automatically transferred to the address in memory.
  - If the loading process requires more instructions than can be read in a single record, this first record causes the reading of others, and these in turn can cause the reading of more records.