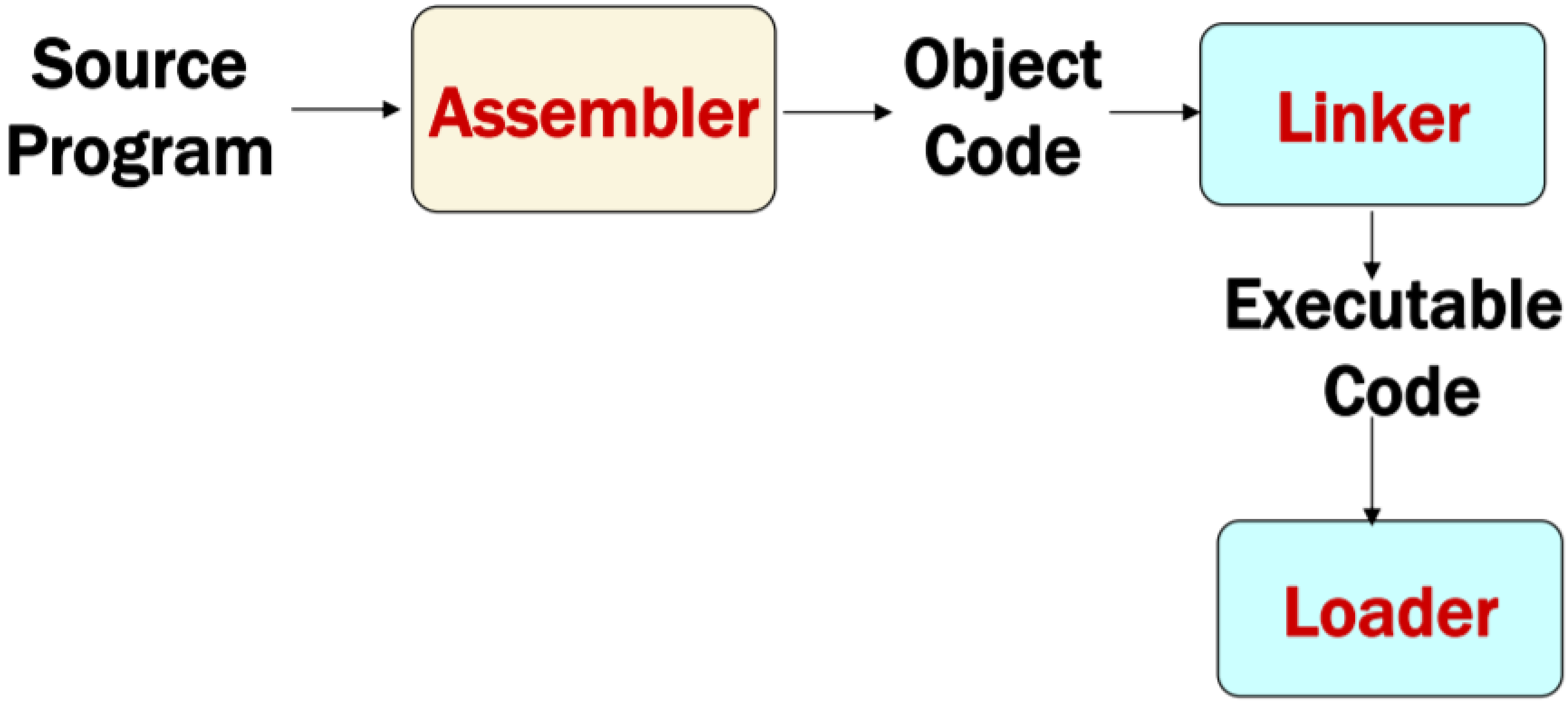
Loaders and Linkers

Introduction

- To execute an assembly language program, we need
  - **Assembler:** translates the program to object program
  - **Linking**: which combines two or more separate object programs and supplies the information needed to allow references between them
  - **Relocation:** which modifies the object program so that it can be loaded at an address different from the location originally specified
  - **Loading:** which brings the object program into memory for execution

Loaders and Linkers



Absolute Loader

- Absolute Program
  - Does not perform linking and program relocation.
  - All functions accomplished in a single pass
- Advantage
  - Simple and efficient
- Disadvantage
  - The need for programmer to specify the actual address
  - Difficult to use subroutine libraries

Absolute Loader

- For a simple absolute loader, all functions are accomplished in a single pass as follows:

  1) The Header record of object program is checked to verify that the correct program has been presented for loading (and that it will fit into the available memory).

  2) As each Text record is read, the object code it contains is moved to the indicated address in memory.

  3) When the End record is encountered, the loader jumps to the specified address to begin execution of the loaded program.

Absolute Loader

```
begin
   read Header record
   verify program name and length
   read first Text record
   while record type is not '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
```

```
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
⋮	⋮	⋮	⋮	⋮
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

Absolute Loader

- In general
  - each byte of assembled code is given using its hexadecimal representation in character form
    - Eg: STL instruction: 14
  - When read by the loader they will occupy two bytes of memory
  - Each pair of bytes from the object program record must be packed into one byte during loading
  - We must be sure that our file and device conventions do not cause some of the program bytes to be interpreted as control characters

Absolute loader

- Disadvantages of Absolute loader
  - The actual address at which it will be loaded into memory should be known.
  - Cannot run several independent programs together, sharing memory between them.
  - It is difficult to use subroutine libraries efficiently.
  - Efficient sharing of memory requires writing relocatable programs instead of absolute ones
  - Inefficient representation in terms of space and execution time
    - So most machines store object programs in binary form

Bootstrap Loader

- When a computer is first turned on or restarted, a special type of absolute loader, called *bootstrap loader* is executed
- This bootstrap loads the first program to be run by the computer - usually an operating system
- The bootstrap itself begins at address 0 in the memory of the machine
- It loads the operating system starting at address 0x80
- No header record or control information
- The object code is loaded into consecutive bytes of memory starting at address 80

GETC Subroutine

- Reads one character from device F1
- Converts it from the ASCII character code to the hexadecimal digit that is represented by the character
- The ASCII code for character 0 (Hex 30) is converted to the numeric value 0.
- The ASCII codes for 1 through 9 (Hex 31 to 39) are converted to the numeric values 1 to 9.
- The codes for A to F (Hex 41 to 46) are converted to values 10 through 15
- Accomplished by
  - Subtract 48 (hex 30) from character codes ‘0’ to ‘9’
  - Subtract 55 (hex 37) from codes ‘A’ through ‘F’
- Jumps to address 80 when EOF is read from F1
- Skips all characters having hexadecimal codes less than 30

Bootstrap Loader

- STL: machine equivalent is 14.
- **Packing of bytes**
- Invoke GETC subroutine
  - Read the first character '1' to Accumulator
  - '1' will be read as ASCII '31' (hex)
  - Subtract hex 30 (decimal 48) from 31 (hex) to get 1
- Store in a register S and shift left by 4 bits.
  - Now the high order 4 bits will contain 1.
- GETC subroutine will be invoked twice
  - Read the next character '4'. It will be read as ASCII '34'.
  - Subtract 30 to get 4
- Add this to register S
- Register will contain '14'

Register S

0000 0001

0001 0000

Shift left by 4 bits

Register A

0000 0100

0001 0100

```
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
```

GETC subroutine

- **Characters 0 to 9**
  - Character 5 will be read as decimal 53 (hex 35)
  - Subtract decimal 48 (hex 30) to get 5
  - Check whether the number is less than 10
  - If less than 10, conversion is complete
- **Characters A to F**
  - Loader reads a character A (decimal 65)
  - Initially subtract 48 (65-48=17)
  - Check whether the result is less than 10.
  - If less than 10, conversion is complete
  - Else subtract 7 from the result (17-7=10)

Machine Dependent Loader Features

- Relocation
- Program Linking

- **Relocation**
- Loaders that allow program relocation are called ***relocating loaders or relative loaders***
- Motivation
  - Efficient sharing of the machine with larger memory and when several independent programs are to be run together
  - Support the use of subroutine libraries efficiently

Relocating Loaders

- Two methods for specifying relocation
  - Modification record
    - To describe each part of the object code that must be changed when the program is relocated.
    - The extended format instructions are affected by relocation (absolute addressing)
  - Relocation bit
    - Each instruction is associated with one relocation bit
    - The relocation bits in a Text record is gathered into bit masks

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	

Relocating Loaders

- Modification record
  - 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 since all the instructions except RSUB must be modified when the program is relocated (absolute addressing)
- **Modification record**

  col 1 : M

  col 2-7: Starting address of the field to be modified (hexadecimal)

  col 8-9: length of the field to be modified (half byte)

  col 10 : modification flag (+/-)

  col 11-17: segment name

Object Program

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

Modification record
- col 1: M
- col 2-7: relocation address
- col 8-9: length (halfbyte)
- col 10: flag (+/-)
- col 11-17: segment name

Relocation

- Relocation bit
  - 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

- Twelve-bit mask is used in each Text record
  - Since each text record contains less than 12 words
  - Unused words are set to 0
  - Any value that is to be modified during relocation must coincide with one of these 3-byte segments

Relocation

- Text record
  - col 1: T
  - col 2-7: starting address of the object code in this text record
  - col 8-9: length of the object code in this text record
  - col 10-12: relocation bits
  - col 13-72: object code

Line	Loc	Source statement			Object cod	
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		

Relocation bit

- T000000^1E^FFC^ (111111111100) specifics that all 10 words of object code are to be modified.
- 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 relocation bits are E00 (1110 0000 0000) for the second text record. It indicates that the first three words of the object code needs to be modified

Program Linking

- Programs can be made up of control sections.
- Assembled together or assembled independently.
- In either case, they would appear as separate segments of code after assembly

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		

```
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         REF2
M^000054^06^+LISTC         REF4
M^000057^06^+ENDC
M^000057^06^-LISTC         REF5
M^00005A^06^+ENDC
M^00005A^06^-LISTC         REF6
M^00005A^06^+PROGA
M^00005D^06^-ENDB
M^00005D^06^+LISTB         REF7
M^000060^06^+LISTB
M^000060^06^-PROGA         REF8
E^000020
```

```
H^PROGB ^000000^00007F
D^LISTB ^000060^ENDB  ^000070
R^LISTA ^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
```

REF1

REF3

REF4

REF5

REF6

REF7

REF8

```
HPROGC 000000000051
DLISTC 000030ENDC  000042
RLISTA ENDA  LISTB ENDB
•
•
T0000180C03100000771000004051000000
•
•
T0000420F000030000008000011000000000000
M00001905+LISTA      REF1
M00001D05+LISTB      REF2
M00002105+ENDA       REF3
M00002105-LISTA
M00004206+ENDA       REF4
M00004206-LISTA
M00004206+PROGC
M00004806+LISTA      REF6
M00004B06+ENDA
M00004B06-LISTA      REF7
M00004B06-ENDB
M00004B06+LISTB
M00004E06+LISTB      REF8
M00004E06-LISTA
E
```

Program Linking

```
0020   REF1    LDA  LISTA (PROGA)    03201D
0036   REF1   +LDA  LISTA (PROGB)    03100000
0018   REF1   +LDA  LISTA (PROGC)    03100000
```

- PROGA
  - REF1 is simply a reference to a label within the program.
  - It is assembled in the usual way as a PC relative instruction.
  - No relocation or linking is necessary.
- PROGB and PROGC
  - The same operand (LISTA) refers to an external symbol.
  - The assembler uses an extended format instruction with address field set to 00000.
  - The object program for PROGB and PROGC contains a Modification record instructing the loader to add the value of the symbol LISTA to this address field when the program is linked.
- The assembler evaluates as much of the expression it can
- The remaining terms are passed on to the loader via modification records

Program Linking

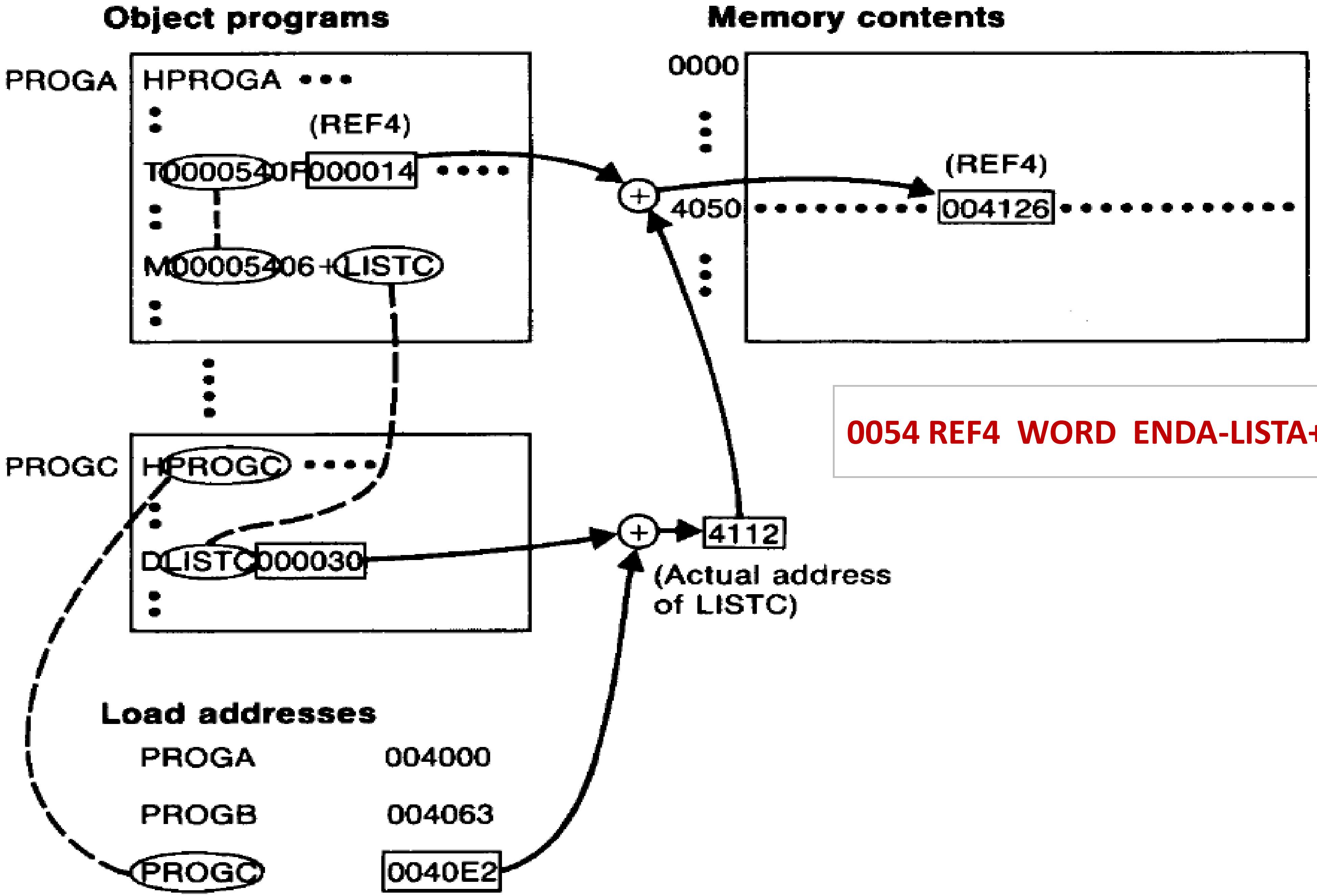
0023	REF2	+LDT	LISTB+4	77100004	(PROGA)
001C	REF2	LDT	LISTB+4	772027	(PROGB)
003A	REF2	+LDT	LISTB+4	77100004	(PROGC)

- **PROGA & PROGC**
  - 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
  - A modification record directs the loader to add to this field the value of LISTB
- **PROGB**
  - The same expression is simply a local reference
  - Assembled using a program counter relative instruction with no relocation or linking required
- The assembler evaluates as much of the expression it can
- The remaining terms are passed on to the loader via modification records

Program Linking

0054	REF4	WORD	ENDA-LISTA+LISTC	000014	(PROGA)
0070	REF4	WORD	ENDA-LISTA+LISTC	000000	(PROGB)
0042	REF4	WORD	ENDA-LISTA+LISTC	000030	(PROGC)

- **PROGA**
  - The assembler evaluates all of the expression in REF4 except the value of LISTC
  - The result is an initial value of 000014 (hex) and one Modification record for LISTC
  - M^000054^06^+^LISTC
- **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 ENDA, LISTA and LISTC
- **PROGC**
  - Evaluates LISTC
  - The result is an initial value of 30 and three Modification records for ENDA, LISTA and a modification record to add the starting address of PROGC (to obtain the actual address of LISTC)



0054 REF4 WORD ENDA-LISTA+LISTC 000014

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

Algorithm and Data Structures for a Linking Loader

- A linking loader usually makes two passes
- Pass 1 assigns addresses to all external symbols by creating External Symbol Table .
- Pass 2 performs the actual loading, relocation, and linking by using ESTAB.
- The main data structure is **External Symbol Table (ESTAB)**
    - Analogous to SYMTAB
    - Stores the name and address of each external symbol (in EXTDEF statement) in the set of control sections being loaded
    - Also indicates in which control section the symbol is defined
    - Organized as hash table

Algorithm and Data Structures for a Linking Loader

- Two variables **PROGADDR and CSADDR**.
- **PROGADDR (Program load address)**
  - PROGADDR is the beginning address in memory where the linked program is to be loaded.
  - Value is supplied to the loader by the operating system

- **CSADDR (Control section address)**
  - 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

Algorithm for a Linking Loader

- **Pass 1**
  - Loader is concerned only with the Header and Define records
  - The beginning load address for the program, PROGADDR is obtained from the OS.
  - This becomes the starting address (CSADDR) for the first control section being loaded
  - The control section name from Header record is entered into ESTAB, with value given by CSADDR
  - All the external symbols appearing in the Define record are entered into ESTAB with address equal to value in Define record + CSADDR
  - When the END record is encountered, the control section length CSLTH (which was saved from the Header record) is added to CSADDR to get the address of the next control section in sequence
  - At the end of Pass 1, the external symbol table ESTAB contains all the external symbols defined in the control sections and their addresses

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}
```

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	

```
H^PROGA ^000000^000063
D^LISTA ^000040^ENDA   ^000054
R^LISTB ^ENDB   ^LISTC ^ENDC
.
```

```
H^PROGB ^000000^00007F
D^LISTB ^000060^ENDB   ^000070
R^LISTA ^ENDA   ^LISTC ^ENDC
.
```

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

Algorithm for a Linking Loader

- **Pass 2**
- Performs actual loading, relocation and linking of a program
- 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 modified is looked up in ESTAB.
- This value is then added to or subtracted from the indicated location in memory.
- The control is transferred to the loaded program to begin execution
  - The loader uses the address given in the End record as the transfer point
  - If more than one control section specifies a transfer address, the loader uses the last one encountered
  - If no transfer point is given, the loader uses the PROGADDR as the transfer point

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}
```

Linking Loader

- Modification record using reference number
  - The algorithm can be made more efficient.
  - Each external symbol is assigned a reference number
  - This number is used in the Modification records instead of external symbol.
  - The number 01 to the control section name.
  - 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

```
H^PROGB ^000000^00007F
D^LISTB ^000060^ENDB  ^000070
R^02LISTA ^03ENDA  ^04LISTC ^05ENDC
•
•
T^000036^0B^03100000^772027^05100000
•
•
T^000070^0F^000000^FFFFF6^FFFFFF^FFFFF0^000060
M^000037^05^+02
M^00003E^05^+03
M^00003E^05^-02
M^000070^06^+03
M^000070^06^-02
M^000070^06^+04
M^000073^06^+05
M^000073^06^-04
M^000076^06^+05
M^000076^06^-04
M^000076^06^+02
M^000079^06^+03
M^000079^06^-02
M^00007C^06^+01
M^00007C^06^-02
E
```

Loader Design Options

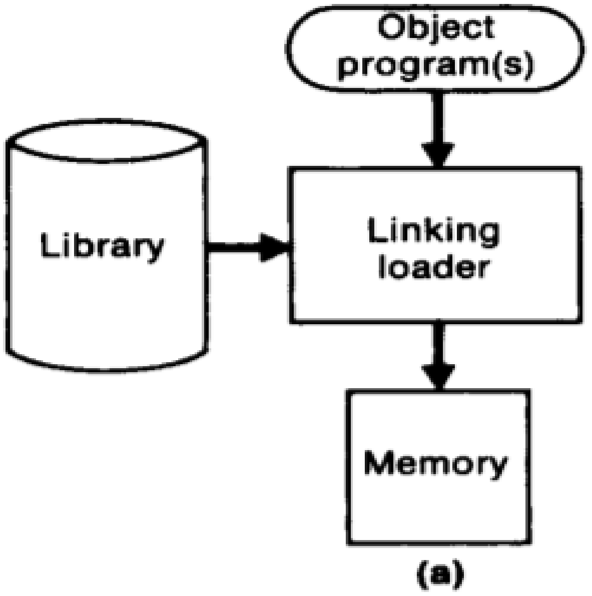
- Linking loaders
  - Performs all linking and relocation operations, including library search if specified, and loads the linked program directly into memory for execution
- Linkage editors
  - Produces a linked version of the program (often called a load module or an executable image), which is written to a file or library for later execution
- Dynamic linking
  - Perform linking at execution time

Linkage Editors

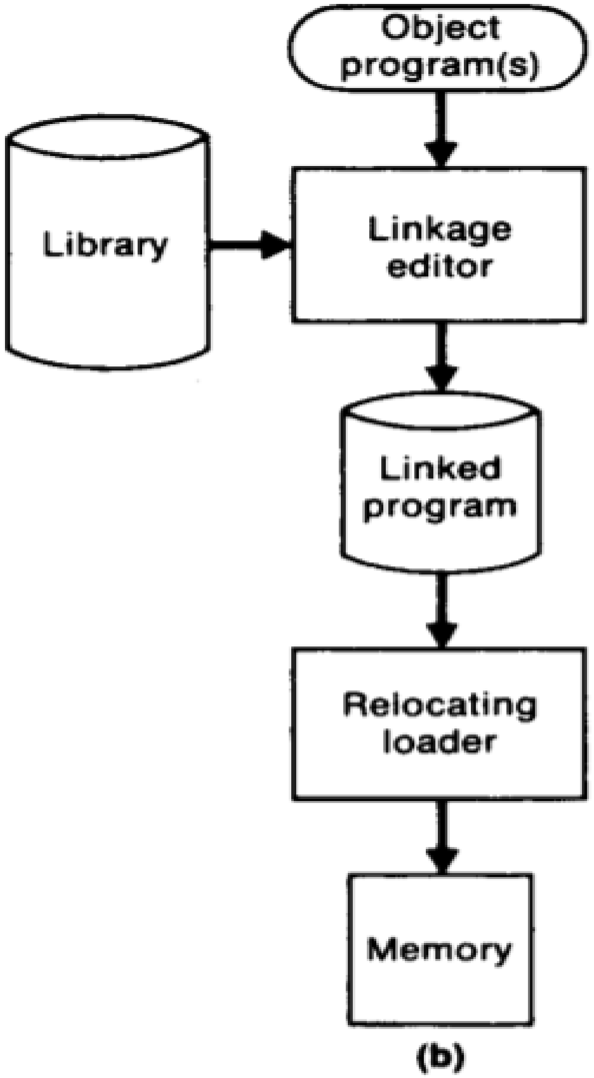
- Performs relocation of all control sections relative to the start of the linked program
- Resolve all external reference
- Output a relocatable module for later execution
- When the user is ready to run the linked program, a simple relocating loader can be used to load the program into memory.
- The loading can be accomplished in one pass with no external symbol table required
- The only object code modification necessary is the addition of an actual load address to relative values within the program.

Linkage Editor

- 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
  - In the linked version of programs, all external references are resolved, and relocation is indicated by some mechanism such as modification records or a bit mask
- External references is often retained in the linked program
  - To allow subsequent relinking of the program to replace control sections, modify external references, etc.



Linking Loader



Linkage Editor

Linking Loader

1. Performs linking and relocation at load time
2. Loads the linked program directly into memory. Therefore, steps for writing and reading the linked program are avoided
3. Linking loader searches libraries and resolves external references every time the program is executed
4. Used in situations where the program has to be reassembled for every execution.
   - program development and testing
   - when a program is used so infrequently that it is not worthwhile to store the assembled and linked version
5. Has less flexibility and control
6. Less complexity and overhead

Linkage Editor

1. Linking is done prior to load time
2. Writes a linked version of program into a file, which is later executed by the relocating loader
3. Resolution of external references and library searching are performed only once
4. Linkage editors can be efficiently used when a program is to be executed many times without being reassembled every time
5. Has more flexibility and control
6. Increase in complexity and overhead

Additional Functions of Linkage Editors

- Linkage editor can perform many useful functions besides simply preparing an object program for execution
  - Replacement of subroutines in the linked program
  - Construction of a package for subroutines generally used together
  - Specification of external references not to be resolved by automatic library search

Additional Functions of Linkage Editors

- Replacement of subroutines in the linked program
- For example: improve a subroutine (PROJECT) of a program (PLANNER) without going back to the original versions of all of the other subroutines

```
INCLUDE PLANNER(PROGLIB)
DELETE PROJECT {delete from existing PLANNER}
INCLUDE PROJECT(NEWLIB) {include new version}
PLANNER(PROGLIB)        [creates new version of PLANNER]
```

Additional Functions of Linkage Editors

- Build packages of subroutines or other control sections that are generally used together
- For example: build a new linked module FTNIO instead of search all subroutines in FINLIB

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

Additional Functions of Linkage Editors

- Specify that external references are not to be resolved by automatic library search
  - Can avoid multiple storage of common libraries in programs.
  - If 100 programs using the routines on the same library a total copy of 100 libraries would be stored, waste space
  - User could specify that no library search be performed during linkage editing, only external references between user written routines would be resolved
  - Need a linking loader to combine the common libraries at execution time.

Dynamic Linking

- Postpone the linking function until execution time
- A subroutine is loaded and linked to the rest of the program when it is first called
- **Called Dynamic linking, dynamic loading, or load on call**
- Allow several executing programs to share one copy of a subroutine or library (Dynamic Link Library, DLL)
- Dynamic linking provides the ability to load the routines only when they are needed
- **Advantages**
- Load the routines when they are needed
    - Time and memory space will be saved
    - Avoids the necessity of loading the entire library for each execution

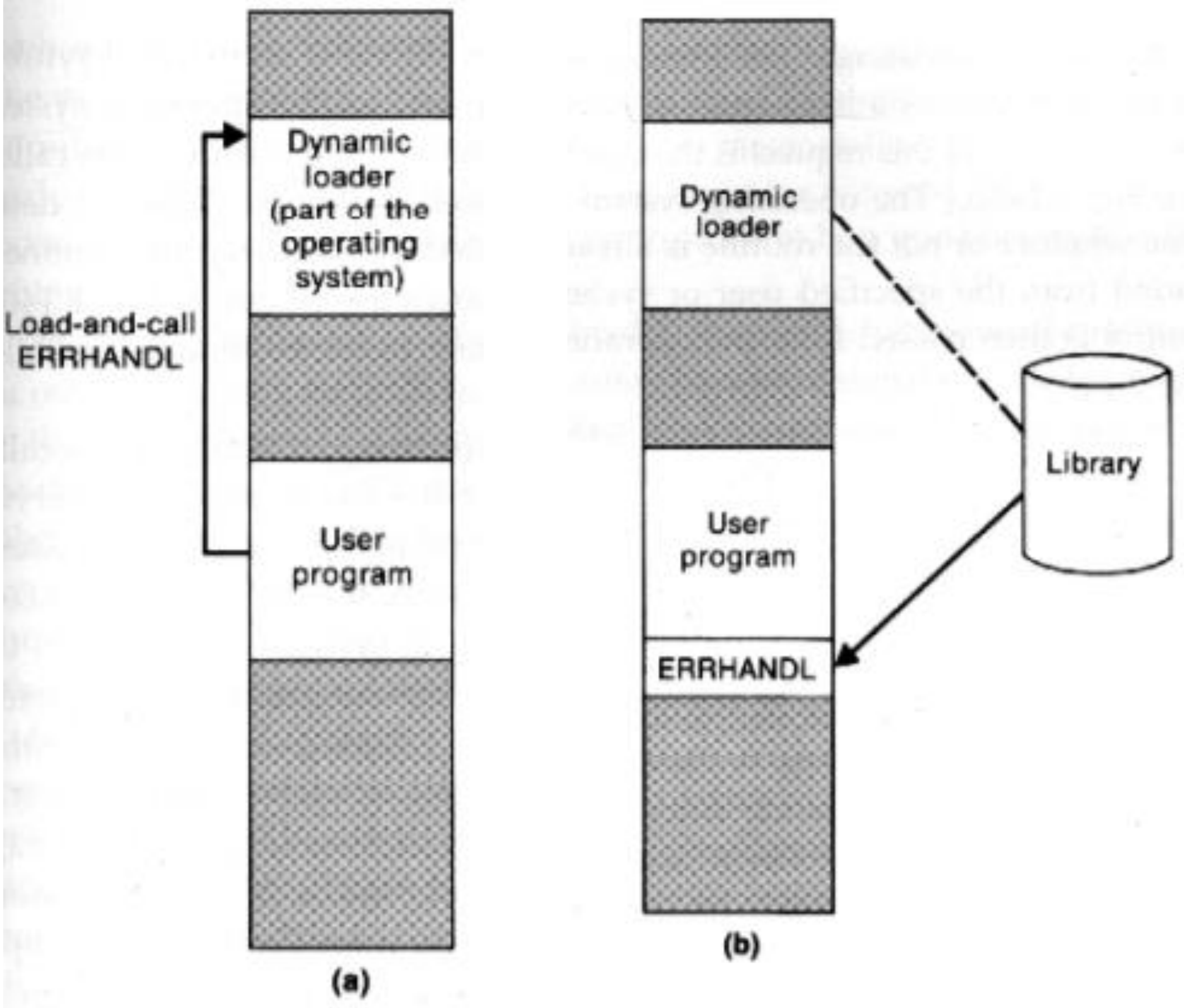
Dynamic Linking

- Dynamic linking provides the ability to load the routines only when (and if) they are needed.
- Example:
  - A program contains subroutines that correct or clearly diagnose error in the input data during execution.
  - If such errors 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.

Dynamic Linking

- Need the help of OS
  - Dynamic loader is one part of the OS
  - OS should provide load-and-call system call
- Instead of executing a JSUB instruction, the program makes a load-and-call service request to the OS
  - The parameter of this request is the symbolic name of the routine to be called
- Processing procedures of load-and-call request
  - Pass control to operating system’s dynamic loader
  - OS checks whether the routine is in memory or not.
    - If in memory, pass control to the routine.
    - If not, load the routine and pass control to the routine.

Loading and Calling a Subroutine Using Dynamic Linking



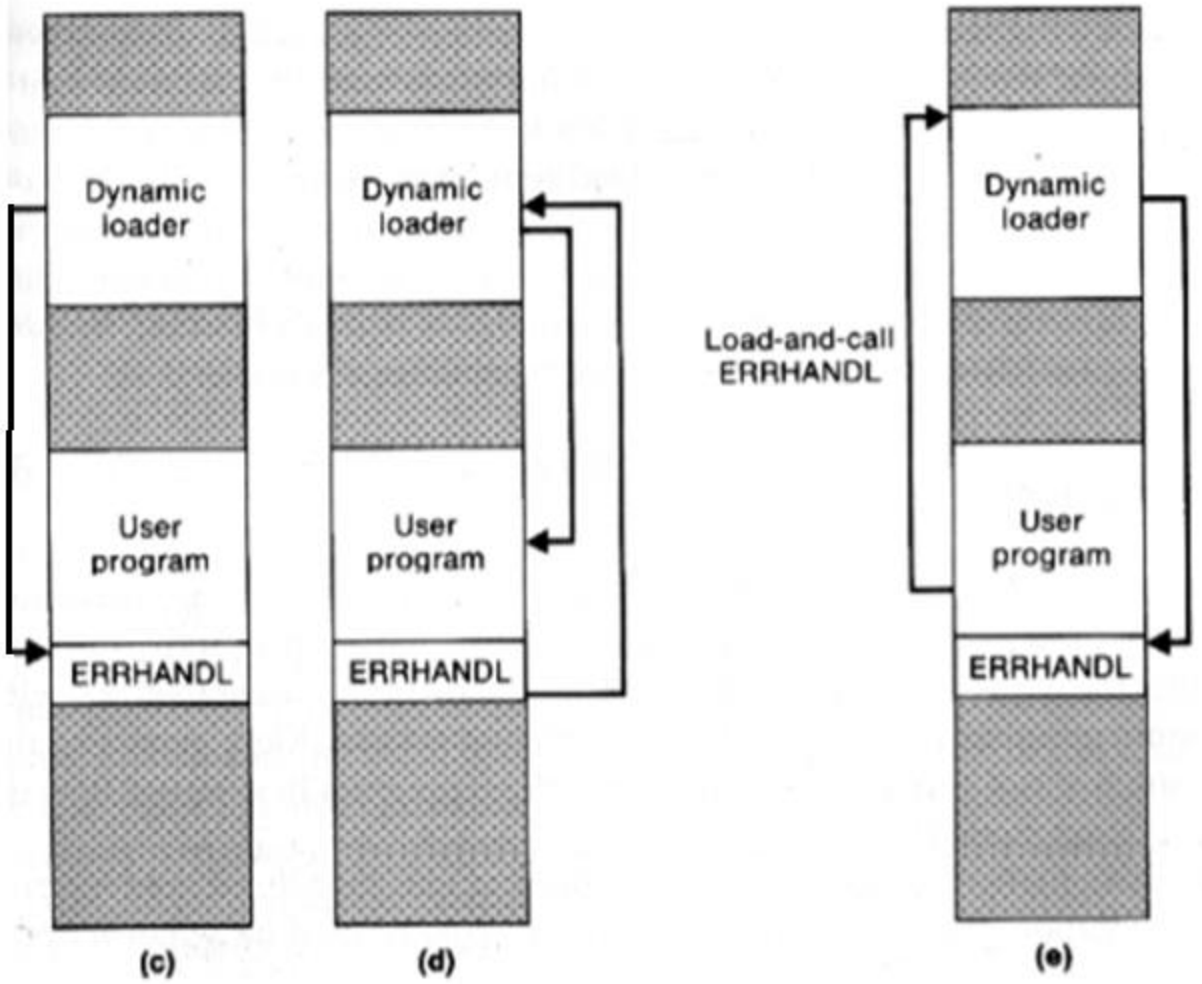
The program makes a load-and-call service request to the OS

Load-and-call service

a) OS examines its internal tables to determine whether or not the routine is already loaded

b) Routine is loaded from the specified user or system libraries

Loading and Calling a Subroutine Using Dynamic Linking



(c) Control is passed from OS to the called subroutine

d) Subroutine is finished, control is passed to the caller, i.e. the OS routine that processed the load and call request. OS returns control to the user program.

e) Calling to a subroutine which is already in memory. Control passed from the dynamic loader to the called routine

Dynamic Linking

- The association of an actual address with the symbolic name of the called routine is not made until the call statement is executed.
- Binding of the name to an actual address is delayed from load time until execution time.
- Delayed binding
  - More flexibility
  - Requires more overhead since OS must intervene in the calling process