KTU NOTES

The learning companion.

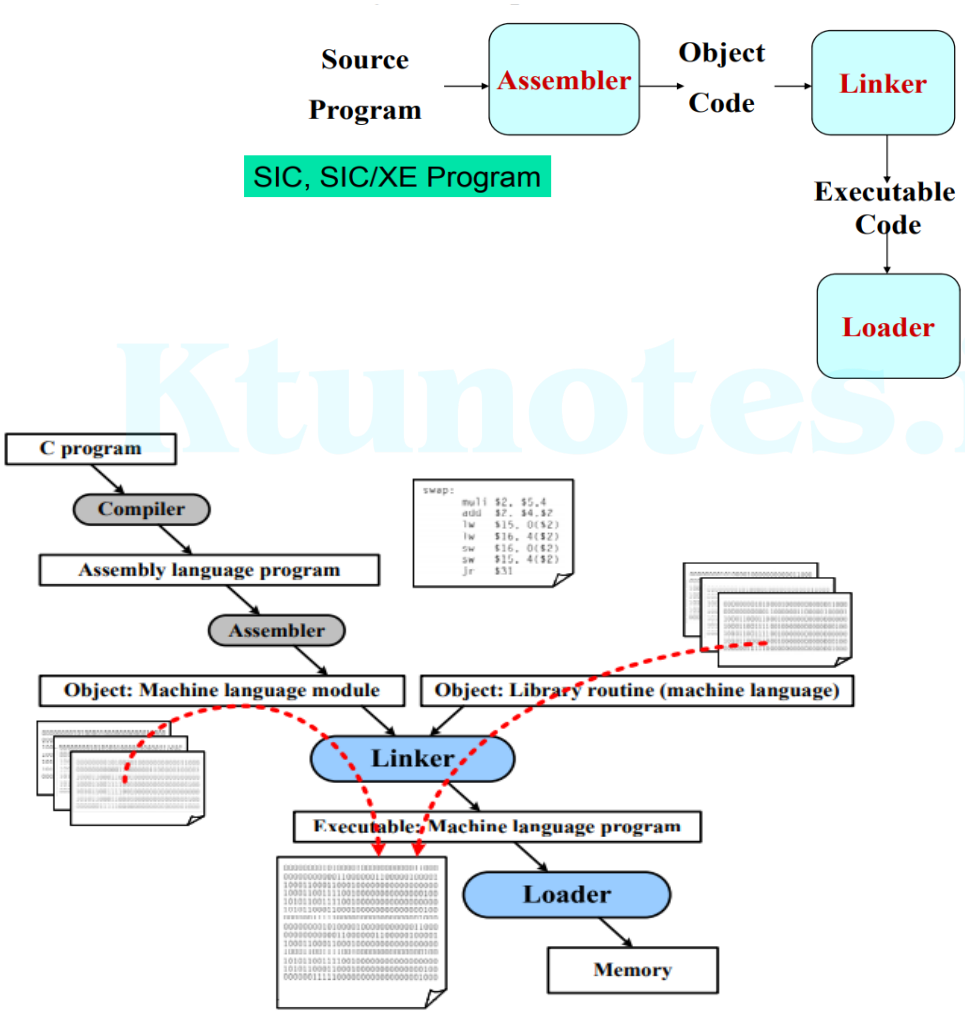
KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

Module 4

Linker and Loader

Basic Loader functions. Design of absolute loader, Simple bootstrap Loader, Machine dependent loader features- Relocation, Program Linking, Algorithm and data structures of two pass Linking Loader, Machine dependent loader features, Loader Design Options.

Overview



Loaders and Linkers

An object program contains translated instructions and data values from the source program and specifies addresses in memory where these items are to be loaded.

1. Loading : brings the object program into memory for execution
2. Relocation : modifies the object program so that it can be loaded at an address different from the location originally specified
3. 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 program linking and relocation. Some systems have a linker(or a linkage editor) to perform the linking operations and a separate loader to handle relocation and loading. In most cases all program translators (ie., assemblers and compilers) on a particular system produces object programs in the same format. Thus one system loader or linker can be used regardless of the original source programming languages. There exist a close connection between program translation and loading.

Basic Loader Functions

The major functions performed by loader are:

– Allocation

– Linking

– Relocation

- Loading

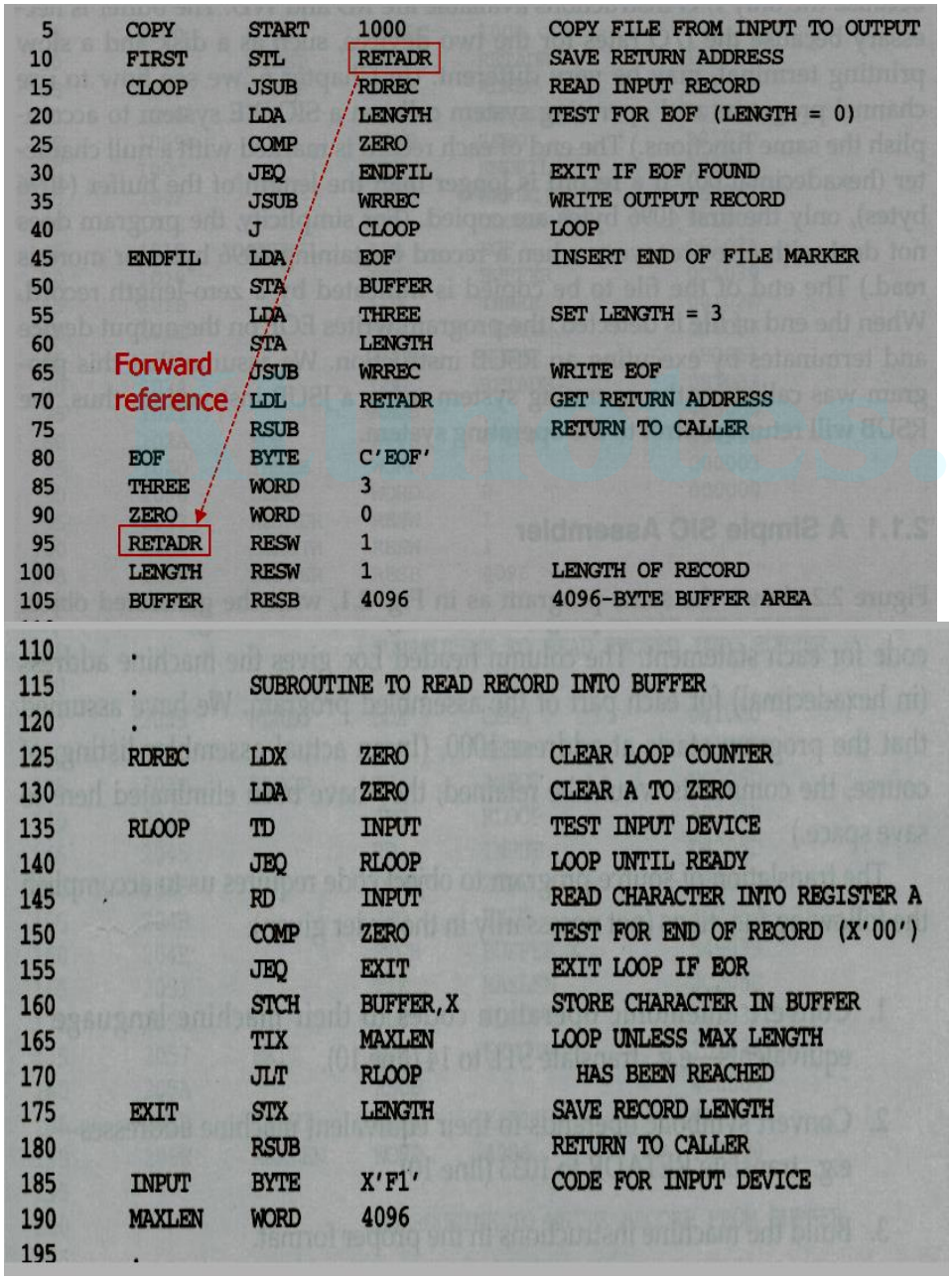
Allocation: The function performed by loader to determine and allocate the required memory space for the program to execute faithfully

Linking: It is a function performed by the loader to analyze and resolve the symbolic references made in the object modules. It is also possible to combine two or more separate object programs and supply the information needed to allow references between them

Relocation: Compilers and assemblers generally create each file of the object code with the program address, starting at zero. But only few computers let to load the program at the location zero. If a program is created from multiple subprograms, all subprograms have to be loaded at

non-overlapping addresses. Relocation is the process of assigning load addresses to the various parts of the program, adjusting the code and data in the program to reflect the assigned addresses. In many systems relocation happens more than once.

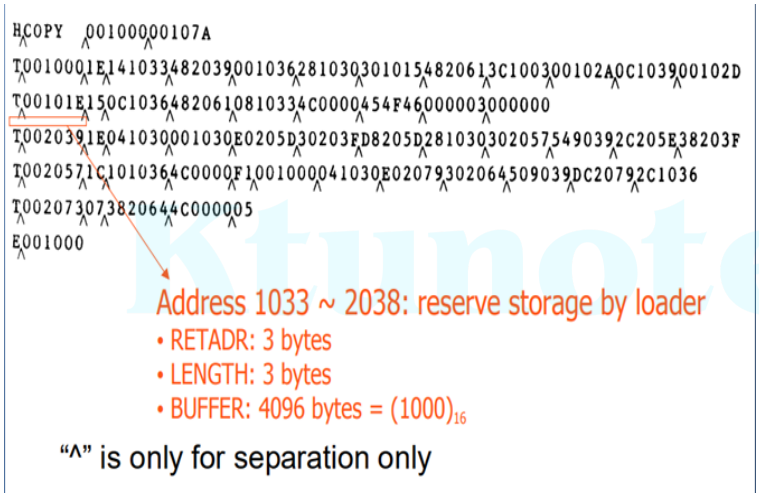
Loading: It is also defined as the process of copying a program from a secondary storage into main memory so it is ready to run. In some cases loading just involves copying the data from the disk to memory, in others it involves allocating storage, setting protection bits or arranging virtual memory to map virtual addresses to disk pages




```
 5     COPY      START   1000          COPY FILE FROM INPUT TO OUTPUT
10     FIRST     STL     RETADR        SAVE RETURN ADDRESS
15     CLOOP     JSUB    RDREC         READ INPUT RECORD
20               LDA     LENGTH        TEST FOR EOF (LENGTH = 0)
25               COMP    ZERO
30               JEQ     ENDFIL        EXIT IF EOF FOUND
35               JSUB    WRREC         WRITE OUTPUT RECORD
40               J       CLOOP         LOOP
45     ENDFIL    LDA     EOF           INSERT END OF FILE MARKER
50               STA     BUFFER
55               LDA     THREE         SET LENGTH = 3
60               STA     LENGTH
65               JSUB    WRREC         WRITE EOF
70               LDL     RETADR        GET RETURN ADDRESS
75               RSUB                  RETURN TO CALLER
80     EOF       BYTE    C'EOF'
85     THREE     WORD    3
90     ZERO      WORD    0
95     RETADR    RESW    1
100    LENGTH    RESW    1             LENGTH OF RECORD
105    BUFFER    RESB    4096          4096-BYTE BUFFER AREA
110    .
115    .         SUBROUTINE TO READ RECORD INTO BUFFER
120    .
125    RDREC     LDX     ZERO          CLEAR LOOP COUNTER
130              LDA     ZERO          CLEAR A TO ZERO
135    RLOOP     TD      INPUT         TEST INPUT DEVICE
140              JEQ     RLOOP         LOOP UNTIL READY
145              RD      INPUT         READ CHARACTER INTO REGISTER A
150              COMP    ZERO          TEST FOR END OF RECORD (X'00')
155              JEQ     EXIT          EXIT LOOP IF EOR
160              STCH    BUFFER,X      STORE CHARACTER IN BUFFER
165              TIX     MAXLEN        LOOP UNLESS MAX LENGTH
170              JLT     RLOOP           HAS BEEN REACHED
175    EXIT      STX     LENGTH        SAVE RECORD LENGTH
180              RSUB                  RETURN TO CALLER
185    INPUT     BYTE    X'F1'         CODE FOR INPUT DEVICE
190    MAXLEN    WORD    4096
195    .
```

```
195      .
200      .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205      .
210      WRREC    LDX     ZERO        CLEAR LOOP COUNTER
215      WLOOP    TD      OUTPUT      TEST OUTPUT DEVICE
220               JEQ     WLOOP       LOOP UNTIL READY
225               LDCH    BUFFER,X    GET CHARACTER FROM BUFFER
230               WD      OUTPUT      WRITE CHARACTER
235               TIX     LENGTH      LOOP UNTIL ALL CHARACTERS
240               JLT     WLOOP         HAVE BEEN WRITTEN
245               RSUB                RETURN TO CALLER
250      OUTPUT   BYTE    X'05'       CODE FOR OUTPUT DEVICE
255               END     FIRST
```

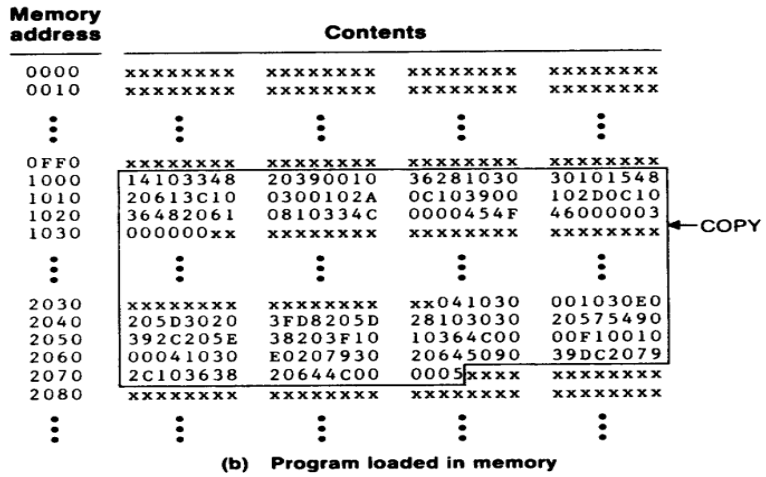
Design of an Absolute Loader



Absolute Loader is a primitive type of loader, which does only the loading function. It does not perform linking and program relocation. All functions are accomplished in a single pass:

- The loader is provided with:
  - Text of the object program, which has already been linked
  - The address of the location, where the first word of the object program is to be loaded
  - Length of the object program

The absolute loader reads the header record and check to verify that the correct program has been presented for loading(and that it will fit into the available memory). When the end record is encountered the loader jumps to the specified address to begin execution of the loader program



(b) Program loaded in memory

The contents of the memory locations for which there is no Text Record are shown as *xxxx*. This indicates that the previous contents of these locations remained unchanged.

The algorithm for an absolute loader

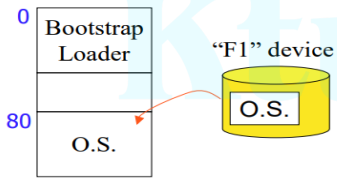
```
begin
   read Header record
   verify program name and length
   read first Text record
   while record type ≠ 'E' do
     begin
        {if object code is in character form, convert into
          internal representation}
        move object code to specified location in memory
        read next object program record
     end
   jump to address specified in End record
end
```

In the object program, each byte of assembled code is given using the hexadecimal representation in character form. For example, the machine operation code for STL instruction would be represented by the pair of the characters “1” and “4”. When these are read by loader as part of object program, they will occupy two bytes of memory. In the instruction as loaded for execution, however this operation code must be stored in a single byte with the hexadecimal value 14. Thus each pair of bytes from the object program record must be packed together into

one byte during loading. Note: In fig(a) each printed character represents one byte of the object program record. In fig (b) each printed character represents one hex-decimal digit in memory(ie,half-byte). This method of representing an object program is inefficient in terms of both space and execution time. Therefore, most machines store object program in a binary form, with each byte of object code stored as a single byte in the object program. In this type of representations, a byte may contain any binary value. We must ensure that our file and device conventions do not cause some of the object program bytes to be interpreted as control characters. For example- the convention indicating the end of the record with a byte containing hexadecimal 00 – would clearly be unsuitable for use with a binary object program.

A Simple Bootstrap Loader

When a computer is first turned on or restarted a special type of absolute loader called *bootstrap loader* is executed. This bootstrap loader loads the first program to be run by the computer-usually an operating system. A bootstrap loader for SIC/XE




```
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
```

X ← 0x80 (the address of the next memory location to be loaded)
Loop until end of input
A ← GETC (and convert from ASCII character code to the hexadecimal digit)
save the value in the high-order 4 bits of S
A ← GETC
combine the value to form one byte A ← (A+S)
(X) ← (A) (store one char.)
X ← X + 1
End of loop

GETC A ← read one character from device F1
if (A = 0x04) then jump to 0x80
if A<48 then goto GETC
A ← A-48 (0x30)
if A<10 then return
A ← A-7
return

ASCII value of
0~9 : 0x30~39
A~F : 0x41~46

The figure shows the source code of our bootstrap loader. The bootstrap itself begins at address 0 in the memory. It loads the OS or some other program starting at address 80. Each byte of object code to be loaded is represented on device F1 as two Hex digits (by GETC subroutines). No header record, end record or control information (such as addresses or lengths). The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80. After all of the object code from device F1 has been loaded the bootstrap jumps to address 80, which begins the execution of program that was loaded. Much of the work of bootstrap loader is performed by the subroutine GETC.

This subroutine reads one character from the device F1 and converts it from the ASCII character code to the value of the hexadecimal digit that is represented by that characterThe ASCII code for the character 0 (Hex 30) is converted to the numeric value 0. Likewise, the ASCII code for

"1" through "9" (hex 31 through 39) are converted to the numeric values 1 through 9 and the codes for "A" through "F" (hex 41 through 46) are converted to the values 10 through 15. This is accomplished by subtracting 48(hex30) from the character codes for "0" through "9" and subtracting 55 (hex 37) from the codes for "A" through "F". The subroutine GETC jumps to address 80 when an end of the file (hex 04) is read from device F1. It skips all other input characters that have ASCII code less than hexadecimal 30. This causes the bootstrap to ignore any control bytes (such as end-of-line) that are used. The main loop of the bootstrap keeps the address of the next memory location to be loaded in register X.

GETC is used to read and convert a pair of characters from device F1 representing 1 byte of object code to be loaded. These two hexadecimal digit values are combined into a single byte by shifting the first one left 4 bit positions and adding the second to it. The resulting byte is stored at the address currently in the 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 register X. The register X is also used as the second operand for this instruction

Drawback of absolute loaders

1. Programmer needs to specify the actual address at which it will be loaded into memory.
2. Difficult to run several programs concurrently, sharing memory between them.
3. Difficult to use subroutine libraries.

Machine Dependent Loader Features

- Solution:
- A more complex loader that provides
  - Program relocation
  - Program linking

Program Relocation

The need for program relocation is an indirect consequence of the change to larger and more powerful computers. The way relocation is implemented in loader is also dependent upon machine characteristics. Loader that allow program relocation are called relocating loaders or relative loaders. A modification record is used to describe each part of the object code that must be changed when the program is relocated. Format of the modification record we have studied in mod.2. Here we are taking the same example studied in mod.2

```
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
```

```
HCOPY  000000001077
T0000001D17202D69202D4B10103603202629000033200074B10105D3F2FEC032010
T00001D130F20160100030F200D4B10105D3E2003454F46
T001036 1DB410B400B44075101000E32019332FFADB2013A004332008 57C003B850
T001053 1D3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T001070073B2FEF4F000005
M00000705+COPY
M00001405+COPY
M00002705+COPY
E000000
```

Figure 3.5 Object program with relocation by Modification records.

Most of the instructions in this program use relative addressing or immediate addressing. The only portion of the assembled program that contains actual addresses are the extended format instructions(line 15,35 and 65). So these are get affected with relocation. Note that one modification record for each value that must be changed during relocation. Each modification record specifies the starting address and the length of the field whose value is to be altered. It 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. *Later in this module we will study the algorithm which loader uses to perform these modifications*. Modification record scheme is convenient for machines with small architecture. It is not well suited to use this for complex architectures. So we are considering the relocatable program written for SIC

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	140033
15	0003	CLOOP	JSUB	RDREC	481039
20	0006		LDA	LENGTH	000036
25	0009		COMP	ZERO	280030
30	000C		JEQ	ENDFIL	300015
35	000F		JSUB	WRREC	481061
40	0012		J	CLOOP	3C0003
45	0015	ENDFIL	LDA	EOF	00002A
50	0018		STA	BUFFER	0C0039
55	001B		LDA	THREE	00002D
60	001E		STA	LENGTH	0C0036
65	0021		JSUB	WRREC	481061
70	0024		LDL	RETADR	080033
75	0027		RSUB		4C0000
80	002A	EOF	BYTE	C'EOF'	454F46
85	002D	THREE	WORD	3	000003
90	0030	ZERO	WORD	0	000000
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	040030
130	103C		LDA	ZERO	000030
135	103F	RLOOP	TD	INPUT	E0105D
140	1042		JEQ	RLOOP	30103F
145	1045		RD	INPUT	D8105D
150	1048		COMP	ZERO	280030
155	104B		JEQ	EXIT	301057
160	104E		STCH	BUFFER,X	548039
165	1051		TIX	MAXLEN	2C105E
170	1054		JLT	RLOOP	38103F
175	1057	EXIT	STX	LENGTH	100036
180	105A		RSUB		4C0000
185	105D	INPUT	BYTE	X'F1'	F1
190	105E	MAXLEN	WORD	4096	001000
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		

```
205
210     1061    WRREC   LDX     ZERO            040030
215     1064    WLOOP   TD      OUTPUT          E01079
220     1067            JEQ     WLOOP           301064
225     106A            LDCH    BUFFER,X        508039
230     106D            WD      OUTPUT          DC1079
235     1070            TIX     LENGTH          2C0036
240     1073            JLT     LOOP            381064
245     1076            RSUB                    4C0000
250     1079    OUTPUT  BYTE    X'05'           05
255                     END     FIRST
```

Figure 3.6 Relocatable program for a standard SIC machine.

This SIC program does not use relative addressing.
The addresses in all the instructions except RSUB must be modified.
This would require 31 Modification records.

If a machine primarily uses direct addressing and has a fixed instruction format. There are many addresses needed to be modified. It is often more efficient to specify relocation using relocation bit. There are no modification records, the text records are same as before except that there is a relocation bit associated with each word of the object code. All SIC instructions occupy one word, this means that there is one relocation bit for each possible instruction. For second Text record, mask E00, specifies that the first three words are to be modified. The remainder of the object code in this record is represented data constants, RSUB instruction and thus do not require modification. Other text record follows the same pattern. Note that: the object code generated for the LDX instruction on line 210 begins a new Text record even though there is room for it in the preceding record. This occurs because each relocation bit is associated with a 3-byte segment of object code in the Text Record.

- **Any value that is to be modified during relocation must coincide with one of these 3-byte segments so that it 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.**

- Relocation bit
  - 0: no modification is needed
  - 1: modification is needed

Text record
col 1: T
col 2-7: starting address
col 8-9: length (byte)
col 10-12: relocation bits
col 13-72: object code

```
HCOPY  000000001078
T00000001EFFC140033481039000036280030300015481061C000300002A0C003900002D
T00001E15E00DC003648106108003340000454F46000003000000
T0010391EFFC040030000030E0105D30103FD8105D280030301057548039 2C105E38103F
T0010570A800100036 4C0000F1001000
T00106119FE0040030E01079301064508039DC10792C00363810644C000005
E000000
```

F1 is one-byte

Figure 3.7 Object program with relocation by bit mask.

Two methods for specifying relocation as part of the object program.

- **Modification Record** (for SIC/XE)
- **Relocation Bit** (for SIC)

Some computers provide hardware relocation capability that eliminates some of the need for the loader to perform program relocation. For example, some such machines consider all memory references to be relative to the beginning of the user's assigned area of memory. The conversion of these relative addresses to actual addresses is performed as the program is executed.

Program Linking

The control sections could be assembled together (ie., in the same invocation of the assembler) or they could be assembled independently of one another. In either case they would appear as separate segments of object code after assembly. The loader has to link, relocate and load the control section. The loader does not know which control sections were assembled at the same time. Consider the three separately assembled programs in Fig. Each consists of a single control section. Each program contains a list of items, LISTA, LISTB, LIST C; and it ends with ENDA,ENDB,ENDC. LISTA---ENDA, LISTB---ENDB, LISTC---ENDC. The labels on the beginnings and ends of the lists are external symbols(they are available for linking). **Note:** Each program contains exactly the same set of references to these external symbols.REF1, REF2,

REF3 for Instruction operands. REF4, REF5, REF6, REF7, REF8 are the values of data words. Omitted the instructions which do not involve linking and relocation.

- **What are the different ways these identical expressions are handled within three programs?**

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		

REF1 (LISTA)

- Control section A
  - LISTA is defined within the control section.
  - Its address is available using *PC-relative addressing.*
  - No modification for relocation or linking is necessary.
- Control sections B and C
  - LISTA is an *external reference.*
  - Its address is not available
    - An *extended-format instruction* with *address field set to 00000* is used.
  - A modification record is inserted into the object code
    - Instruct the loader to *add the value of LISTA to this address field.*

REF2 (LISTB+4)

- Control sections A and C
  - REF2 is an *external reference* (LISTB) plus a constant (4).
  - The address of LISTB is not available
    - An ***extended-format instruction*** with ***address field set to 00004*** is used.
  - A modification record is inserted into the object code
    - Instruct the loader to add the value of LISTB to this address field.
- Control section B
  - LISTB is defined within the control section.
  - Its address is available using *PC-relative addressing.*
  - No modification for relocation or linking is necessary.

REF3 (#ENDA-LISTA)

- Control section A
  - ENDA and LISTA are defined within the control section.
  - The difference between ENDA and LISTA is immediately available.
  - No modification for relocation or linking is necessary.
- Control sections B and C
  - ENDA and LISTA are *external references*.
  - The difference between them is not available
    - An *extended-format instruction* with address field set to 00000 is used.
  - ***Two*** modification records are inserted into the object code
    - +ENDA
    - -LISTA

REF4 (ENDA-LISTA+LISTC)

- Control section A
  - The values of ENDA and LISTA are internal. Only the value of LISTC is unknown.
  - The address field is initialized as 000014 (ENDA-LISTA).
  - **One** Modification record is needed for LISTC:
    - +LISTC
- Control section B
  - ENDA, LISTA, and LISTC are all unknown.
  - The address field is initialized as 000000.
  - **Three** Modification records are needed:
    - +ENDA
    - -LISTA
    - +LISTC
- Control section C
  - LISTC is defined in this control section but ENDA and LISTA are unknown.
  - The address field is initialized as the ***relative address of LISTC (000030)***
  - **Three** Modification records are needed:
    - +ENDA
    - -LISTA
    - **+PROGC (***for relocation***) // Thus, relocation also use modification record**

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	

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
RLISTB ENDB  LISTC ENDC
:
:
T0000200A03201D771000004050014
:
:
T0000540F000014FFFFF600003F000014FFFFC0
M00002405+LISTB     REF2
M00005406+LISTC     REF4
M00005706+ENDC      REF5
M00005706-LISTC
M00005A06+ENDC
M00005A06-LISTC     REF6
M00005A06+PROGA
M00005D06-ENDB      REF7
M00005D06+LISTB
M00006006+LISTB     REF8
M00006006-PROGA
E000020
```

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		

```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
RLISTA ENDA  LISTC ENDC
.
T000036 0B 03100000 772027 05100000
.
T000070 0F 000000 FFFFF6 FFFFFF FFFFF0 000060
M000037 05+LISTA        REF1
M00003E 05+ENDA         REF3
M00003E 05-LISTA
M000070 06+ENDA         REF4
M000070 06-LISTA
M000070 06+LISTC
M000073 06+ENDC         REF5
M000073 06-LISTC
M000076 06+ENDC         REF6
M000076 06-LISTC
M000076 06+LISTA        REF7
M000079 06+ENDA
M000079 06-LISTA
M00007C 06+PROGB        REF8
M00007C 06-LISTA
E
```

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		

```
HPROGC 000000000051
DLISTC 000030ENDC  000042
RLISTA ENDA  LISTB ENDB
.
.
T000018 0C 03100000 771000040 5100000
.
.
T000042 0F 000030 000008 000011 000000 000000
M000019 05+LISTA    REF1
M000001D 05+LISTB   REF2
M000021 05+ENDA     REF3
M000021 05-LISTA
M000042 06+ENDA
M000042 06-LISTA    REF4
M000042 06+PROGC
M000048 06+LISTA    REF6
M00004B 06+ENDA
M00004B 06-LISTA    REF7
M00004B 06-ENDB
M00004B 06+LISTB
M00004E 06+LISTB    REF8
M00004E 06-LISTA
E
```

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

The fig. (a) shows that three programs might look like this after being linking and loading. PROGA has been loaded at starting address 4000, PROGB and PROGC are following. Note:

each of REF4 through REF8 has resulted (after relocation and linking is performed) in the same value in each of the three programs because the same source expression appeared in each program.

Calculation of REF4 (ENDA-LISTA+LISTC)

- Control section A
  - The address of REF4 is 4054 (4000 + 54)
  - The address of LISTC is:

    0040E2 + 000030 = 004112

    (starting address of PROGC) (relative address of LISTC in PROGC)
  - The value of REF4 is:

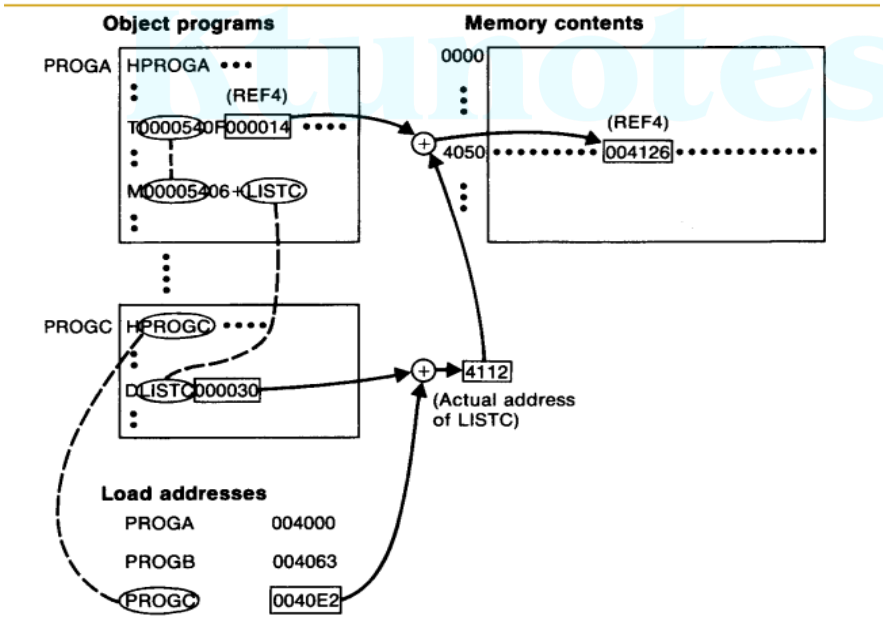
    000014 + 004112 = 004126

    (initial value) (address of LISTC)
- Control section B
  - The address of REF4 is 40D3 (4063 + 70)
  - The value of REF4 is:

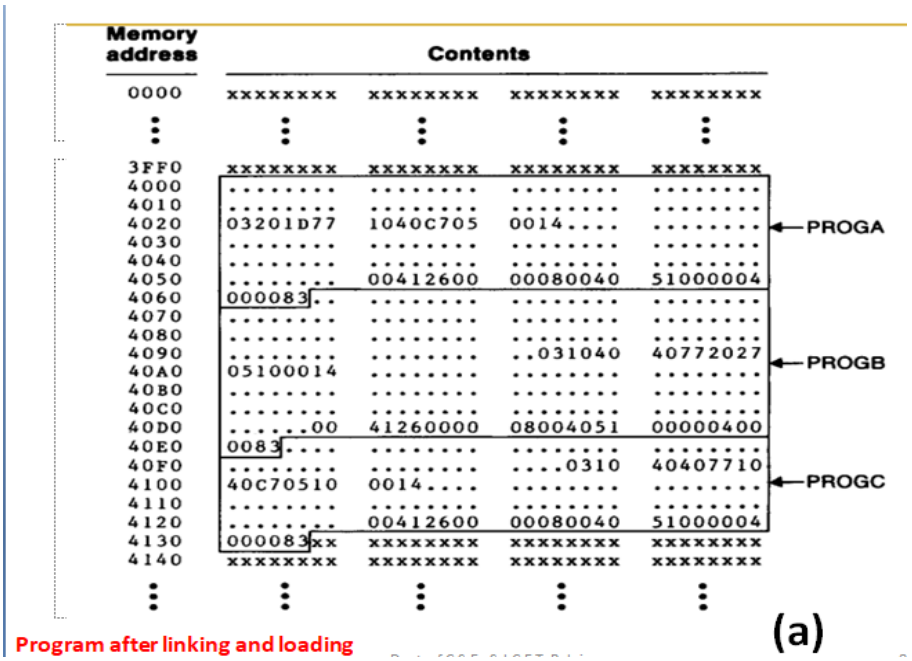
    000000 + 004054 - 004040 + 004112 = 004126

    (initial value) (address of ENDA) (address of LISTA) (address of LISTC)

Target Address are the same

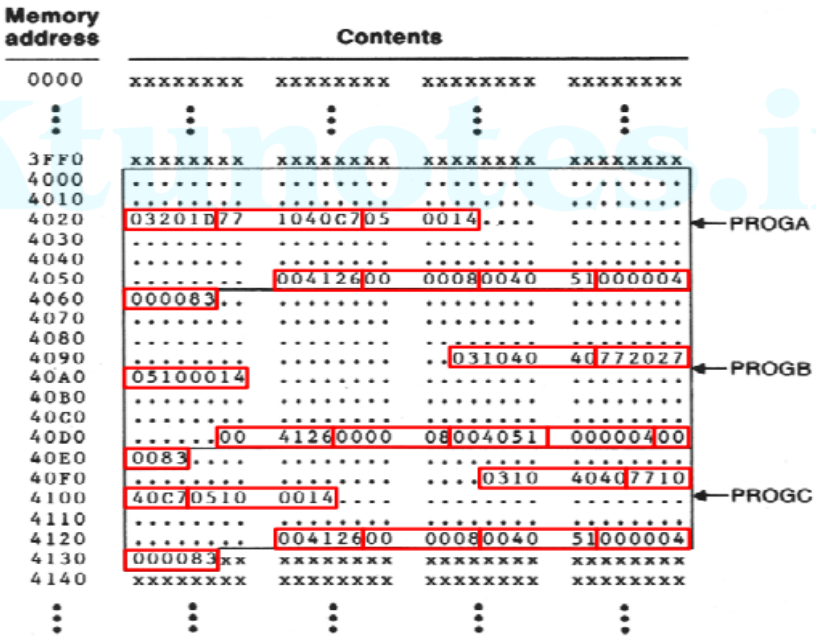


Relocation and Linking operations performed on REF4 from PROG A



Program after linking and loading

(a)



Programs From Fig 3.8 After Linking and Loading (Fig. 3.10a)

Values of REF4, REF5, …, REF8 in three places are all the same.

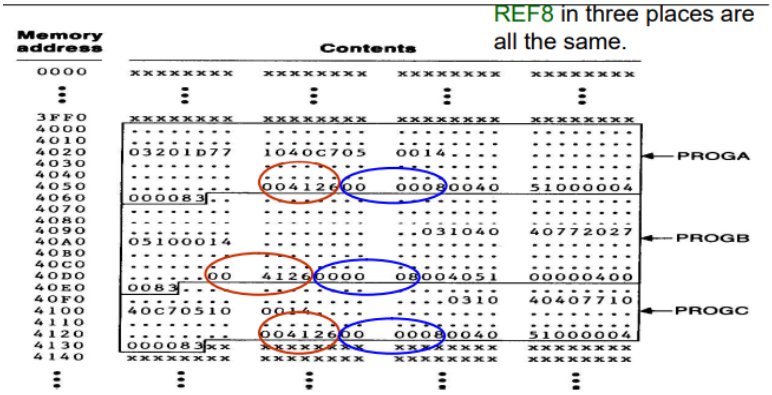


Figure 3.10(a) Programs from Fig. 3.8 after linking and loading.

- After these control sections are linked, relocated, and loaded
  - Each of REF4 through REF8 should have the same value in each of the three control sections.
    - They are data labels and have the same expressions
  - But not for REF1 through REF3 (instruction operation)
    - Depends on PC-relative, Base-relative, or direct addressing used in each control section
      - In PROGA, REF1 is a PC-relative
      - In PROGB, REF1 is a direct (actual) address
    - However, the ***target address*** of REF1~REF3 in each control section are the same
      - Target address of REF1 in PROGA, PROGB, PROGC are all 4040

In PROGA, the reference REF1 is a program counter relative with displacement 01D . When this instruction is executed, the program counter contains the value 4023(the actual address of the next instruction). The resulting target address is 4040.

Algorithm and Data Structures for a two-pass linking loader

The input to a two-pass linking loader consists of a set of object programs (ie., control sections) that are to be linked together. It is possible (and common) for a control section to make an external reference to a symbol whose definition does not appear until later in this input stream. In such a case, the required linking operation cannot be performed until an address is assigned to the external symbol involved (ie., until the later control section is read). Thus a linking loader usually makes two passes over its input, just as an assembler does.

- In Pass1: assign addresses to all external symbols
- In Pass 2: perform the actual linking, relocating and loading

Data Structures

- **ESTAB**
- **PROGADDR**
- **CSADDR**
- **CSLTH :Control section length**

ESTAB

The main data structure needed for our linking loader is an external symbol table ESTAB. Analogous to SYMTAB. It is used to store the name and address of each external symbol in the set of control section being loaded. The table indicates in which control section the symbol is defined A table has hashed organisation.

PROGADDR

Is a variable. Program load Address. Is the beginning address in memory where the linked program is to be loaded. Its value is supplied to the loader by the operating system.

CSADDR

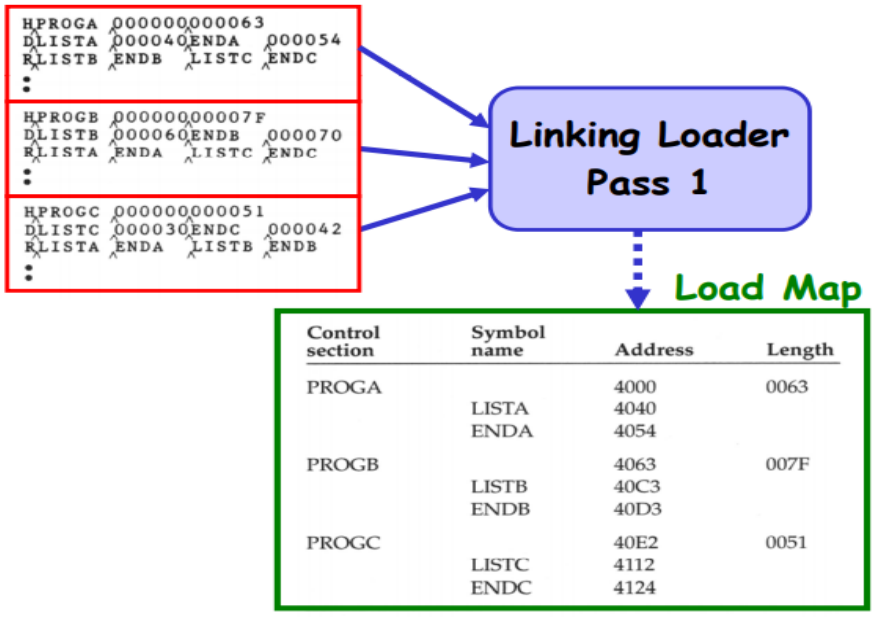
Is a variable. Control Section address. It contains the starting address assigned to the control section currently being scanned by the loader. This value is added to all relative addresses within the control section to convert them to actual addresses. During Pass 1, the loader is concerned only with the header and define record types in the control sections. The beginning load address for linked program (PROGADDR) is obtained from the operating system. This becomes the starting address (CSADDR) for the first control section in the input sequence. The control section name from the header record is entered into ESTAB, with the value given by CSADDR. All external symbols appearing in the Define Record for the control section are also entered into ESTAB. Their addresses are obtained by adding the value specified in the define record to

CSADDR. When the end record is read, the control section length CSLTH (which is saved in header record) is added to CSADDR. This calculation gives the starting address for the next control section in sequence.

ESTAB (External Symbol Table) may also look like Load MAP

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	

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 the ability to print a load map that shows these symbols and their addresses as an option. This information is useful in program debugging.



The pass 2 of the loader performs actual loading, relocation and linking of the program. In pass 2 also CSADDR is used, as it always contains the actual starting address of the control section currently being loaded. As each text record is read, the object code is moved to the specified address (plus the current value of CSADDR). When 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 memory location in memory. The last step performed by loader is usually the transferring of control to the loaded program to begin execution. *On some systems, the address where execution begins is simply passed back to the operating system. The user must enter a separate execute command*

The end record for each control section may contain the address of the first instruction in that control section to be executed. Here the 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. Most commonly, if no control section contains a transfer address, the loader uses the beginning of the linked program (ie., the PROGADDR) as the transfer point. Normally, the transfer address would be placed in the End record for a main program, but not for a subroutine. Thus the correct execution address would be specified regardless of the order in which the control sections were presented for loading.

Pass 1: (only ***Header*** and ***Define*** records are concerned)

```
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}
```

```
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 // the next control section
        end  {while not EOF}
    jump to location given by EXECADDR {to start execution of loaded program}
    end {Pass 2}
```

The efficiency of the algorithm can be improved by making a slight change in the object program format. This modification involves assigning a reference number to each symbol referred to in a control section. This reference number is used (instead of the symbol name) in modification record. Suppose we always assign the reference number 01 to the control section name. The other external symbols may be assigned numbers as part of the Refer record for the control section.

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
R02LISTB 03ENDB  04LISTC 05ENDC
:
T0000200A03201D77100004050014
:
T0000540F0000014FFFFF600003F000014FFFFC0
M000024 05+02
M000054 06+04
M000057 06+05
M000057 06-04
M00005A 06+05
M00005A 06-04
M00005A 06+01
M00005D 06-03
M00005D 06+02
M000060 06+02
M000060 06-01
E000020
```

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

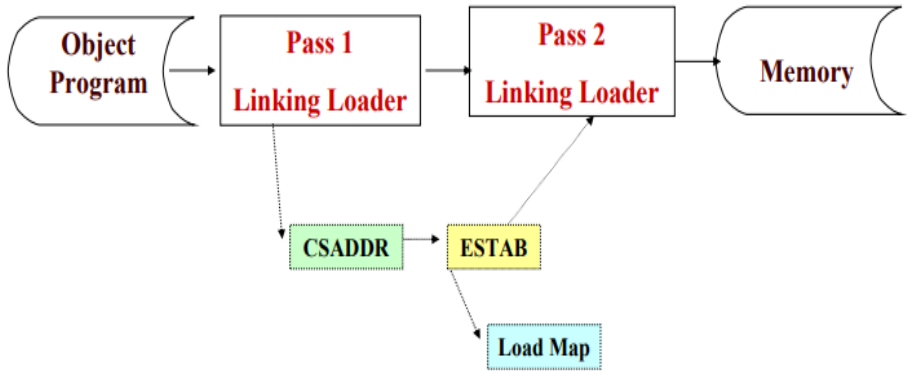
```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
R02LISTA 03ENDA  04LISTC 05ENDC
:
T000036 0B 03 10 00 00 77 20 27 05 10 00 00
:
T000070 0F 000000 FFFFF6 FFFFFF FFFFF0 000060
M000037 05 +02
M00003E 05 +03
M00003E 05 -02
M000070 06 +03
M000070 06 -02
M000070 06 +04
M000073 06 +05
M000073 06 -04
M000076 06 +05
M000076 06 -04
M000076 06 +02
M000079 06 +03
M000079 06 -02
M00007C 06 +01
M00007C 06 -02
E
```

```
HPROGC 000000000051
DLISTC 000030ENDC  000042
R02LISTA 03ENDA  04LISTB 05ENDB
:
T000018 0C 03 10 00 00 77 10 00 04 05 10 00 00
:
T000042 0F 000030 000008 000011 000000 000000
M000019 05 +02
M00001D 05 +04
M000021 05 +03
M000021 05 -02
M000042 06 +03
M000042 06 -02
M000042 06 +01
M000048 06 +02
M00004B 06 +03
M00004B 06 -02
M00004B 06 -05
M00004B 06 +04
M00004E 06 +04
M00004E 06 -02
E
```


The common use of a technique such as reference numbers is one of the reason for including refer records in the object programs. In algorithm we do not mention about the refer record. Main advantage of reference numbers:

Avoiding *multiple searches* of ESTAB for the same symbol during the loading of a control section.

- Search of ESTAB for each external symbol can be performed **once** and the result is **stored in a new table** indexed by the *reference number*.
- The values for code modification can then be obtained by simply indexing into the table.



Machine Independent Loader Features

Loader features that are not directly related to machine architecture and design

Loading and linking are often thought of as operating system service functions.

1. Automatic library search
2. Loader options

Automatic Library Search

Automatic Library Search for handling external references:

1. Allows programmers to use standard subroutines without explicitly including them in the program to be loaded.
2. The routines are automatically retrieved from a library as they are needed during linking.

Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded. There is a standard library for this purpose. Other libraries may be specified by control statements or by parameters to the loader. This feature allows the programmer to use the subroutines from one or more libraries. The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program, and loaded. The programmer does not need to take any action beyond mentioning the subroutines name as external references in the source program. One some systems this feature is called automatic library call. Linking loaders that support automatic library search must keep track of external symbols that are referred to, but not defined, in the primary input to the loader.

One easy way to do this is to enter the symbols from each Refer record into ESTAB unless these symbols are already present. These entries are marked to indicate that the symbol has not yet been defined. When the definition is encountered, the address assigned to the symbol is filled in to complete the entry. At the end of pass1, the symbols in ESTAB that remain undefined represent unresolved external references. The loader searches the library or library specified for routines that contain the definitions of these symbols, and processes the subroutine found by this search exactly as if they had been part of primary input stream. Note: The subroutines fetched from library in this way also 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, this must be treated as errors.

The process also allows the programmer to override the standard subroutines in the library by supplying his or her own routines. For ex: The main program refers to a standard subroutine named SQRT. Ordinarily, the subroutine with this name would automatically be included via the library search function. A programmer who for some reason wanted to use a different version of SQRT could do so simply by including it as input to the loader. By the end of pass1 of the loader, SQRT would already be defined and it might not be included in automatic library search. The libraries to be searched by the loader ordinarily contains assembled or compiled versions of the subroutines (ie, object program). It is possible to search this libraries by scanning the define records for all of the object programs on the library – it might be inefficient. In most cases a special file structure is used for the libraries. This structure contains a directory that gives the name of each subroutine and a pointer to its address within the file. If the subroutine is to be callable by more than one name (using different entry points), both names are entered in to the directory. The object program is stored only once. Both directory entries point to the same copy of the routine. Thus the library search itself really involves a search of the directory followed by reading the object programs indicated by this search. Some operating systems can keep the directory for commonly used libraries permanently in memory. This can expedite the search process if a large number of external references are to be resolved.

Loader Options

Common options that can be selected at the time of loading and linking.

Including capabilities such as specifying alternative sources of input, changing or deleting external references and controlling the automatic processing of external references. Many loaders allow user to specify the options that modify the standard processing. Many uses a special command language to specify the options. Sometimes there is a separate input file to the loader that contains such control statements. Sometimes, the statements can be embedded in the primary input stream between object programs.On a few systems the programmer can even include loader control statements in the source program, and the assembler or compiler retains these commands as a part of the object program. On some systems options are specified as a part of the job control language that is processed by the operating system. When this approach is used, the operating system incorporates the options specified into a control block that is made available on the loader when it is invoked. One typical loader option allows the selection of alternative sources of input

- Eg: the command:

```
INCLUDE   program-name(library-name)
```

Might direct the loader to read the designated object program from a library and treat it as if it were part of the primary loader input. Next command:

```
DELETE        csect-name
```

Allow users to delete external symbols or entire control sections. It may be possible to change external references within the programs being loaded and linked. The command might instruct the loader to delete the named control section(s) from the set of programs being loaded. Next command,

```
CHANGE        name1, name2
```

Might cause the external symbol name1 to be changed to name2 wherever it appears in the object program.

```
INCLUDE   READ(UTLIB)
INCLUDE   WRITE(UTILB)
DELETE    RDREC, WRREC
CHANGE    RDREC, READ
CHANGE    WRREC, WRITE
```

The command would direct the loader to include the control sections READ and WRITE from the library UTLIB. The command to delete the control sections RDREC and WRREC from the load. The first CHANGE command would cause all external references to symbol RDREC to be changed to refer to symbol READ. Similarly, references to WRREC would be changed to WRITE.

Next option, automatic inclusion of library routines for external reference(*as we have discussed in the previous class*). Most loaders allow the user to specify alternative libraries to be searched using a statement such as,

```
LIBRARY    MYLIB
```

Such user-defined libraries are searched before the standard system libraries. This allows the user to use special versions of the standard routines. Loaders that perform automatic library search to satisfy external references often allow user to specify that some references not be resolved in this way. Suppose for example, that a certain program has as its main function the gathering and storing of data. However, the program can also perform an analysis of the data using the routines STDDEV, PLOT and CORREL from a statistical library. The user may request this analysis at the execution time. Since the program contains external references to these three routines, they would ordinarily be loaded and linked with the program.

If it is known that the statistical analysis is not to be performed in a particular execution of this program, the user could include a command such as:

```
NOCALL     name

NOCALL          STDDEV, PLOT, CORREL
```

It instruct the loader that these external references are to remain unresolved. This avoid the overhead of loading and linking the unneeded routines and saves the memory space that would otherwise be required. Another option is that it is possible to specify that no external references be resolved by library search. Of course this means an error will result if the program attempts to make such an external reference during execution. This option is more useful when programs are

to be linked but not executed immediately. In such a case, it is often desirable to postpone the resolution of external references. Linkage editor performs this task*(we can discuss).*

Another common option involves output from the loader. Loadmap might be generated during the loading process. Through the control statements the user can specify whether or not such a map is to be printed at all. If a map is desired, the level of detail can be selected. For example, the map may include control section names and addresses only. It may also include external symbol addresses or even a cross reference table that shows references to each symbol.

Another option, the loader has the ability to specify the location at which execution is to begin(overridding any information given in the object program).

Another is the ability to control whether or not the loader should attempt to execute the program if errors are detected during the load (for example unresolved external references)

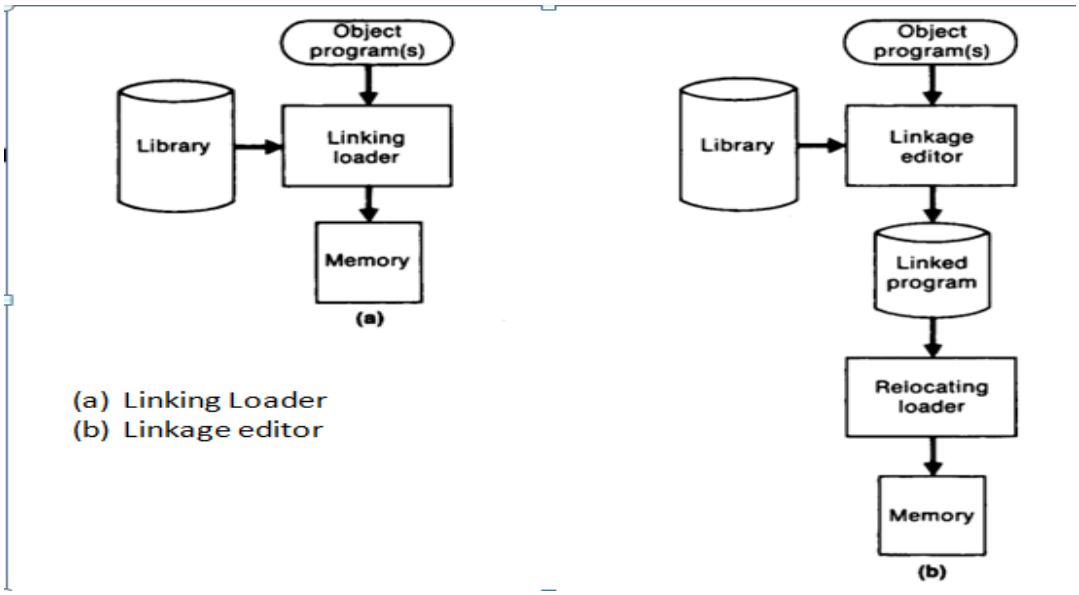
Loader design Options

Alternatives for organizing the loading functions, including relocation and linking. We have studied two pass linking loader algorithm, in that linking loaders perform all linking and relocation at load time. As an alternative to this:

1. Linkage editors: perform linking prior to load time.
2. Dynamic linking: in which linking function is performed at execution time.

Linkage Editor

Are found in many computing system instead of or in addition to the linking loader. A linkage editor performs linking and some relocation. However, the linked program is written to a file instead of being immediately loaded into memory. This approach reduces the overhead when the program is executed. All that is required at load time is a very simple form of relocation. The essential differences between linkage editor and linkage loader is given in the figure:



The source program is first assembled or compiled, producing an object program(which may contain several different control sections). A **linking loader** performs all linking and relocation operations including automatic library search if specified and loads the linked program directly into the memory for execution. In **Linkage editor**, it produces a linked version of the program called **load module** or an **executable image**, which is written to a file or library for later execution. When a user is ready to run the linked program, a simple relocating loader can be used to load program into memory. The only object code modification is necessary in the addition of actual load address to relative values within the program.

The linkage editor performs relocation of all the control sections relative to the start of the linked program. Thus all items that need to be modified at load time have values that are relative to the start of the linked program. This means that the loading can be accomplished in one pass with no external symbol table required. This involves much less overhead than using a linking loader. If a program is to be executed many times without being reassembled, the use of a linkage editor substantially reduces the overhead required.Resolution of external references and library searching are only performed once, (ie., when the program is link edited). *In contrast, linking loader searches libraries and resolves external references every time the program is executed. Sometimes, a program is reassembled for nearly every execution. Situation occurs in a program development and testing environment. Also occurs when a program is used so infrequently, it is*

not worth to store the assembled versions in a library. In such case, it is more efficient to use a linking loader , which avoids the steps of writing and reading the linked program.

The linked program produced by the linkage editor is generally in a form that is suitable for processing by a relocating loader. All external references are resolved, and relocation is indicated by some mechanism such as modification records or bit mask. Even though all linking has been performed, information concerning external references is often retained in the linked program. This allows subsequent re-linking of the program to replace control sections, modify external references, etc. If this information is not retained, the linked program cannot be reprocessed by the linkage editor, it can only loaded and executed. If the actual address at which the program will be loaded is known in advance, the linkage editor can perform all of the needed relocation. The result is a linked program that is an exact image of the way the program will appear in memory during execution.

The content and processing of such an image are the same as for an absolute object program. Normally, however the added flexibility of being able to load the program at any location is easily worth, the slight additional overhead for performing relocation at load time.

Linkage editors can perform many useful **functions** besides simply preparing object program for execution.For ex: Consider a program (PLANNER) that uses large number of subroutines

1. Suppose **that one subroutine (PROJECT) used by the program is changed to correct an error or to improve efficiency.**

    - After new version of PROJECT is assembled or compiled, the linkage editor can be used to replace this subroutine in the linked version of PLANNER

    - It is not necessary to go back to the original (separate)versions of all of the other subroutines

    - The following is 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)
```

2. **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 the subroutine libraries that support high-level programming languages
   - Ex: In FORTRAN, there are a large number of subroutines that are used to handle formatted input and output
   - These include routines to read and write data blocks, to block and de-block records, to encode and decode data items according to format specifications
   - There are a large number of cross reference between these subprograms because of their closely related functions
   - However it is desirable that they remain as separate control sections for reasons of program modularity and maintainability
   - If a program using formatted I/O were linked in the usual way, all of the cross-reference between these library subroutines would have to be processed individually
   - Exactly, the same set of cross-references would need to be processed for almost every FORTRAN program linked
   - This represents a substantial amount of overhead
   - The linkage editor could be used to combine the appropriate subroutines into a package with a command sequence :

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

- The linked module named FTNIO could be indexed in the directory SUBLIB under the same names as the original subroutines.
- Thus a search of SUBLIB before FTNLIB would retrieve FTNIO instead of the separate routines

- Since FTNIO already has all of the cross-reference between subroutines resolved, these linkages would not be reprocessed when each user's program is linked
- The result would be much more efficient linkage editing operation for each program and a considerable overall savings for the system

3. **Linkage editors often allow the user to specify that external references are not to be resolved by automatic library search**

- Using commands, the user could specify that no library search be performed during linkage editing
- Thus only user-written routines would be resolved
- A linking-loader could then be used to combine the linked user routines with linked module at execution time
- Because this process involves two separate linking operations, it would require slightly more overhead; however it would result in a large savings in library space
- Compared to linking loaders, linkage editors in general tend to offer more flexibility and control with the corresponding increase in complexity and overhead

Dynamic Linking

Linkage editors perform linking operations before the program ids loaded for execution. Linking loaders perform these same operations at load time. A scheme that postpones the linking function until execution time; a subroutine is loaded and linked to the rest of the program when it is first called. This type of function is usually called **dynamic linking, dynamic loading or load on call. Dynamic linking** is often used to allow several executing programs to share one copy of a subroutine or library. For ex: run-time support routines for a high level language like C could be stored in a dynamic link library. A single copy of the routines in this library could be loaded into the memory of the computer. All C programs currently in execution could be linked to the one copy, instead of linking a separate copy into each object program. **In object oriented system, dynamic linking is often used for reference to software objects.** This allows the implementation of the object and its method to be determined at the time the program is run. The

implementation can be changed at any time, without affecting the program that makes use of the object. Dynamic linking also makes it possible for one object to be shared by several programs. Dynamic linking –**Advantages:**

- For ex: a program contains subroutines that correct or clearly diagnose errors in the input data during execution
- If such errors are rare, the correction and diagnostic routines may not be used at all during most executions of the program
- However, if the program were completely linked before execution, these subroutines would need to be loaded and linked every time the program is run
- Dynamic linking provides the ability to load the routines only when (and if) they are needed
- If the subroutines involved are large, or have many external references, this can result in substantial savings of time and memory space
- Similarly, suppose that in one execution a program uses only a few out of a large number of possible subroutines, but the exact routines needed cannot be predicted until the program examines its input
- This situation could occur with a program that allows its user to interactively, call any of the subroutines of a large mathematical and statistical library
- The input data could be supplied by the user, and the result could be displayed at the terminal
- In this case, all of the library subroutines could potentially be needed , but only few will be actually be used in any one execution
- Dynamic linking avoids the necessity of loading the entire library for each execution
- Mechanism in which routines that are to be dynamically loaded must be called via, an operating system service request

- This method could also be thought of as a request to be a part of the loader that is kept in memory during execution of the program

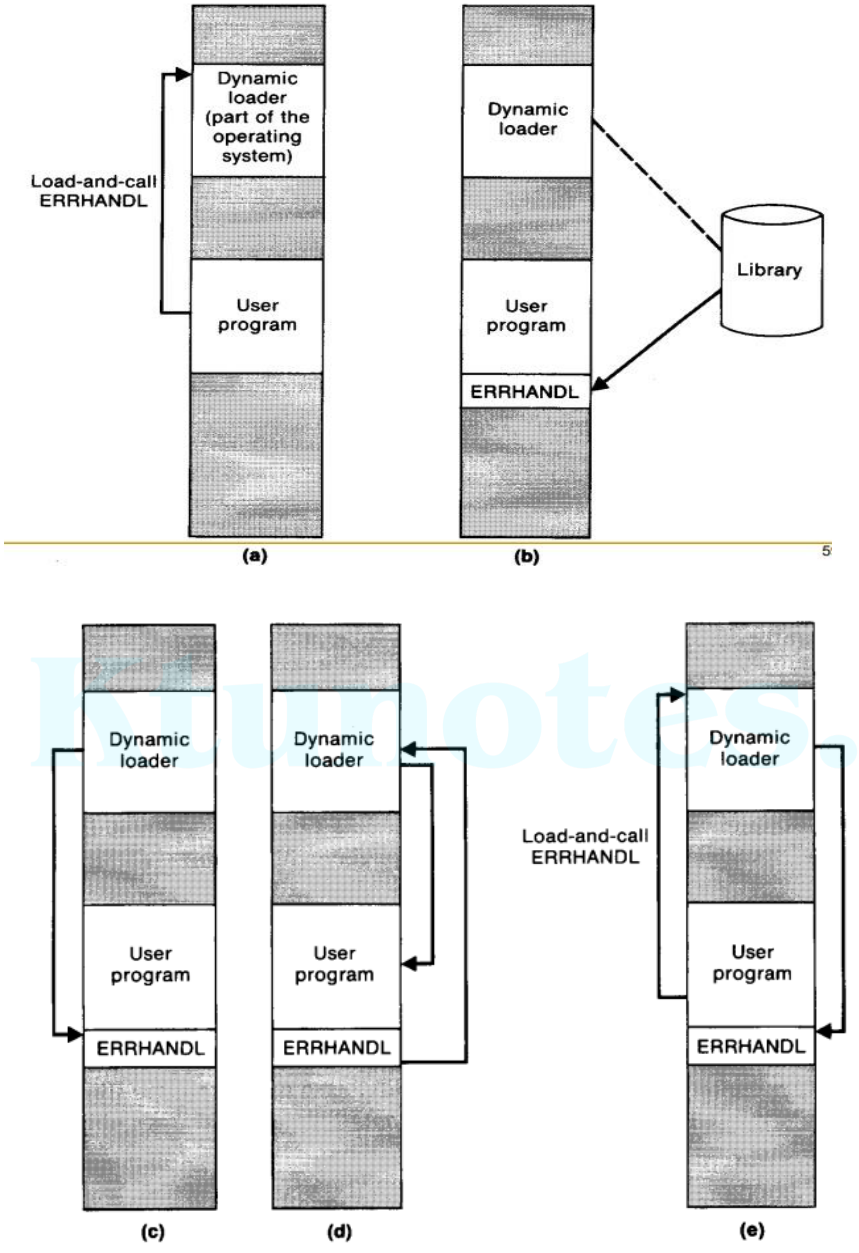
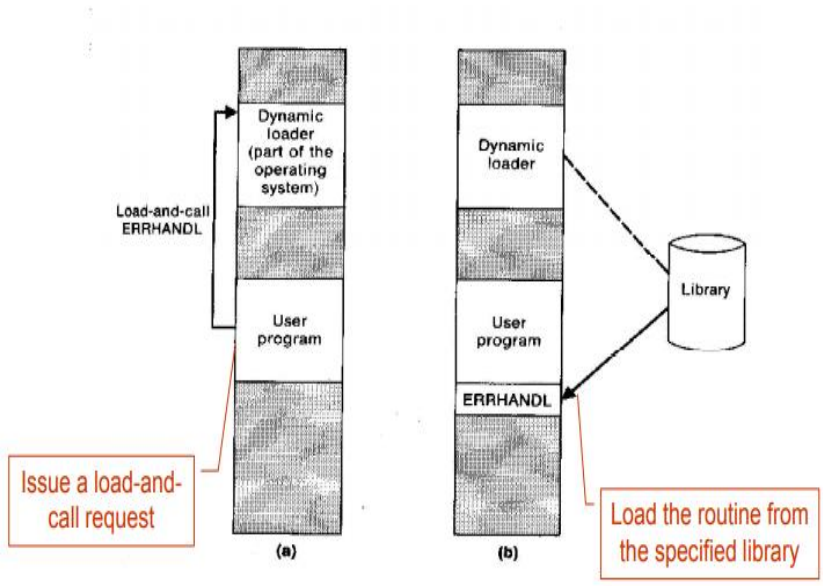


Figure 3.14 Loading and calling of a subroutine using dynamic linking.



Instead of executing a JSUB instruction that refers to an external symbol, program makes a load-and-call service request to operating system. The parameter of this request is the symbolic name of the routine to be called (fig.a). The operating system examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded from the specified user or system libraries (fig.b). Control is then passed from the operating system to the routine being called (fig.c). When the called subroutine completes its processing, it returns to caller (ie., to the OS routine that handles the load-and-call service request).

The OS then returns control to the program that issued the request (fig.d). It is important that control be returned in this way so that the operating system knows when the called routine has completed its execution. After the subroutine is completed, the memory that was allocated to load it may be released and used for other purposes. However, it is not done immediately. It is desirable; to retain the routine in memory for later use as long as the storage space is not needed for other processing. If the subroutine is still in the 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.

When dynamic linking is used, the association of an actual address with the symbolic name of the called routine is not made until the call statement is executed. Another way of describing this is to say that the binding of the name to an actual address is delayed from load time until

execution time. This delayed binding results in greater flexibility. It also requires more overhead since the operating system must intervene in the calling process.

Advantages

- Load the routines when they are needed, the time and memory space will be saved.
- Avoid the necessity of loading the entire library for each execution
  - i.e. load the routines only when they are needed
- Allow several executing programs to share one copy of a subroutine or library (**Dynamic Link Library**, **DLL**)

Bootstrap Loader

- How is loader itself loaded into memory?
  - By OS
- How is OS itself loaded into memory?
  - ***Bootstrap loader***

Given an idle computer with no program in memory, how do things get started? In this situation, with the machine empty and idle, there is no need for program relocation. We can simply specify the absolute address for whatever program is first loaded. Most often the program will be operating system, which occupies a predefined location in memory. This means that we need some means of accomplishing the functions of an absolute loader. Some early computers required the operator to enter into memory the object code for an absolute loader, using switches on the computer console. However, this process is much too in convenient and error-prone to be a good solution to the problem. On some computers, an absolute loader program is permanently resident in ROM (Read-only Memory). When some hardware signal occurs (for ex: operator pressing a "system start " switch), the machine begins to execute this ROM program. On some computers the program is executed directly in the ROM. On others; the program is copied from ROM to main memory and executed there. However, some systems do not have such read only storage. In addition, it is inconvenient to change a ROM program if modifications in the absolute loader are required. An intermediate solution is to have a built –in hardware function that reads a

fixed-length record from some device into memory at fixed location. The particular device to be used can often be selected via console switches. After the read operation is complete, the control automatically gets transferred to the address in memory where the record was stored. This record contains machine instructions that load the absolute program that follows. If the loading process requires more instructions that can be read in a single record, this record causes reading of others, and these in turn can cause the reading of still more records –hence the term Bootstrap. The first record is generally referred to as a bootstrap loader. Such a loader is added to the beginning of all object programs that are to be loaded into an empty and idle.