

Chapter 3
Loaders and Linkers



Outline

- Basic Loader Functions
- Machine-Dependent Loader Features
- Machine-Independent Loader Features
- Loader Design Options
- Implementation Examples



Basic Loader Functions

Linker and Loader

Object program

- Contain ***translated instructions*** and ***data values*** from the source program.
- Specify ***addresses*** in memory where these items are to be loaded.

- Three important processes to load an object program:
  - Loading: Bring the object program into memory for execution.
  - Relocation: Modify the object program so that it can be loaded at an address different from the location originally specified.
  - Linking: Combine two or more separate object programs and supply information needed to allow references between them.

Linker and Loader (Cont.)

- ***Loader***
  - A system program performs the loading function.
  - Some also supports relocation and linking.
- Some systems have a ***linker*** (or ***linkage editor***) to perform the linking operations and a separate ***loader*** to handle relocation and loading.
- All the program translators (i.e., assemblers and compilers) produce the same object program format. Thus one system loader or linker can be used regardless the original source programming language.

Design of An Absolute Loader

```
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
```

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
⋮	⋮	⋮	⋮	⋮

← COPY

One byte character

Half-byte

The **Header record** is first checked.
Then, each **Text record** is read to memory.
When the **End record** is encountered, the loader jumps to the specified address.

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
```



Object Program Format

- In our object program, each byte of assembled code is given using its hexadecimal representation in ***character form***.
  - E.g., The opcode for **STL** instruction would be represented by the pair of characters "1" and "4".
    - When they are read by the loader, they occupy two bytes of memory and must be stored in a singly byte with hexadecimal value 14.
    - Each pair of bytes from the object program record must be packed together into one byte during loading.
- This method of representing an object program is inefficient in terms of ***space*** and ***execution time***.
- Therefore, most machines store object program in a ***binary form***: Each byte of object code is stored as a single byte in the object program.
  - The file and device conventions should not cause some of the object program bytes to be interpreted as control character.
  - E.g., Indicating the end of a record with a byte containing hexadecimal 00 would clearly be unsuitable for use with a binary object program.
- Obviously, object program stored in binary form do not lend themselves well to printing or to reading by human beings. Therefore, we continue to use character representations of object programs in this course.



Simple Bootstrap Loader

- ***Bootstrap loader*** is a special type of ***absolute loader*** that is executed when a computer is first turned on or restarted.
- This bootstrap loads the first program to be run by the computer – usually an operating systems.
- The bootstrap loader for SIE/XE:
  - The bootstrap begins at address 0 in the memory of the machine.
  - It loads the operating system starting at address 80.
  - Because this loader is used in a unique situation (the initial program load or the system), the program to be loaded can be represented in a very simple format.
    - Each byte of object code to be loaded is represented on device F1 as two hexadecimal digits. (No Header record, End record, or control info.)
    - The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80.
    - After loading, the bootstrap jumps to address 80 to execute loaded program.



Bootstrap Loader for SIC/XE

"A" through "F": (hex 41 to 46)

"0" through "9" (hex 30 to 39)

End-of-file: hex 04

```
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
```

S ← (A)

A ← (S)+(A)

X ← (X)+1; (X):(X)



Machine-Dependent Loader Features

Issues of Absolute Loaders

- On a larger and more advanced machine, we do not know in advance where a program will be loaded.
- Efficient sharing of the machine requires that we write relocatable programs instead of absolute one.
- Write absolute programs makes it difficult to use subroutine libraries efficiently.
  - Most such libraries contain many more subroutines than will be used by any one program.
  - To make efficient use of memory, it is important to be able to select and load exactly those routines that are needed.



Machine-Dependent Loader Features

- Program relocation is an indirect consequence of the change to larger and more powerful computers.
  - The way relocation is implemented in a loader is also dependent upon machine characteristics.
- Linking is not a machine-dependent function, but it has the same implementation techniques for loaders.
  - The process of linking usually involves relocation of some of the routines being linked together.



Relocation

- Loaders that allow for program relocation are called ***relocating loaders*** or ***relative loaders***.
- Two methods for specifying relocation in object programs:
  - Use ***Modification records***.
    - A Modification record is used to describe each part of the object code that must be changed when the program is relocated.
    - It is not well suited for use with all machine architectures. E.g., SIC machine doesn’t not use relative addressing. (*Need too many Modification records*.)
  - Use ***direct addressing*** with ***relocation bits***.
    - It is suitable for machines that do not use relative addressing.

Example

Relocatable program

Use Modification records

Line number	Machine address	Label	Instruction	Operand	Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
12	0003		LDB	#LENGTH	69202D
13			BASE	LENGTH	
15	0006	CLOOP	+JSUB	RDREC	4B101036
20	000A		LDA	LENGTH	032026
25	000D		COMP	#0	290000
30	0010		JEQ	ENDFIL	332007
35	0013		+JSUB	WRREC	4B10105D
40	0017		J	CLOOP	3F2FEC
45	001A	ENDFIL	LDA	EOF	032010
50	001D		STA	BUFFER	0F2016
55	0020		LDA	#3	010003
60	0023		STA	LENGTH	0F200D
65	0026		+JSUB	WRREC	4B10105D
70	002A		J	@RETADR	3E2003
80	002D	EOF	BYTE	C'EOF'	454F46
95	0030	RETADR	RESW	1	
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
125	1036	RDREC	CLEAR	X	B410
130	1038		CLEAR	A	B400
132	103A		CLEAR	S	B440
133	103C		+LDT	#4096	75101000
135	1040	RLOOP	TD	INPUT	E32019
140	1043		JEQ	RLOOP	332FFA
145	1046		RD	INPUT	DB2013
150	1049		COMPR	A,S	A004
155	104B		JEQ	EXIT	332008
160	104E		STCH	BUFFER,X	57C003
165	1051		TIXR	T	B850
170	1053		JLT	RLOOP	3B2FEA
175	1056	EXIT	STX	LENGTH	134000
180	1059		RSUB		4F0000
185	105C	INPUT	BYTE	X'F1'	F1
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
210	105D	WRREC	CLEAR	X	B410
212	105F		LDT	LENGTH	774000
215	1062	WLOOP	TD	OUTPUT	E32011
220	1065		JEQ	WLOOP	332FFA
225	1068		LDCH	BUFFER,X	53C003
230	106B		WD	OUTPUT	DF2008
235	106E		TIXR	T	B850
240	1070		JLT	WLOOP	3B2FEF
245	1073		RSUB		4F0000
250	1076	OUTPUT	BYTE	X'05'	05
255			END	FIRST	



Example (Cont.)

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

Modification records for +JSUB

Example for relocation bits (SIC Machine)

Machine address	Label	Instruction	Operand	Object code
1000	COPY	START	1000	
1000	FIRST	STL	RETADR	141033
1003	CLOOP	JSUB	RDREC	482039
1006		LDA	LENGTH	001036
1009		COMP	ZERO	281030
100C		JEQ	ENDFIL	301015
100F		JSUB	WRREC	482061
1012		J	CLOOP	3C1003
1015	ENDFIL	LDA	EOF	00102A
1018		STA	BUFFER	0C1039
101B		LDA	THREE	00102D
101E		STA	LENGTH	0C1036
1021		JSUB	WRREC	482061
1024		LDL	RETADR	081033
1027		RSUB		4C0000
102A	EOF	BYTE	C'EOF'	454F46
102D	THREE	WORD	3	000003
1030	ZERO	WORD	0	000000
1033	RETADR	RESW	1	
1036	LENGTH	RESW	1	
1039	BUFFER	RESB	4096	
	.			
	.	SUBROUTINE TO READ RECORD INTO BUFFER		
	.			
2039	RDREC	LDX	ZERO	041030
203C		LDA	ZERO	001030
203F	RLOOP	TD	INPUT	E0205D
2042		JEQ	RLOOP	30203F
2045		RD	INPUT	D8205D
2048		COMP	ZERO	281030
204B		JEQ	EXIT	302057
204E		STCH	BUFFER,X	549039
2051		TIX	MAXLEN	2C205E
2054		JLT	RLOOP	38203F
2057	EXIT	STX	LENGTH	101036
205A		RSUB		4C0000
205D	INPUT	BYTE	X'F1'	F1
205E	MAXLEN	WORD	4096	001000
	.			
	.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
	.			
2061	WRREC	LDX	ZERO	041030
2064	WLOOP	TD	OUTPUT	E02079
2067		JEQ	WLOOP	302064
206A		LDCH	BUFFER,X	509039
206D		WD	OUTPUT	DC2079
2070		TIX	LENGTH	2C1036
2073		JLT	WLOOP	382064
2076		RSUB		4C0000
2079	OUTPUT	BYTE	X'05'	05
		END	FIRST	

Starting address

PC ← (L)

Begin a new Text record

Object Program with Relocation Bits

Example for relocation bits (SIC Machine)

Bit mask: Each **relocation bit** is associated with each word of object code.

1: program's starting address needs to be added to this word.
0: No need to be addedd.

No modification is needed for RSUB.

```
HCOPY  00000000107A
T0000001EFFC1400334810390000362800303000154810613C000300002A0C003900002D
T00001E15E000C00364810610800334C0000454F46000003000000
T0010391EFFC040030000030E0105D30103FD8105D2800303010575480392C105E38103F
T0010570A8001000364C0000F1001000
T00106119FE0040030E01079301064508039DC10792C00363810644C000005
E000000
```

Data content. No modification is needed

If it were placed in the preceding Text record, it would not be properly aligned to correspond to a relocation bit because of the 1-byte data value from Line 185.

Each relocation bit is associated with a 3-byte segment of object code in the Text record.



Program Linking

- Control sections could be assembled together, or they can be assembled independently of one another.
  - The programmer has a natural inclination to think of a program as a logical entity that combines all of the related control sections.
  - The loader has no such thing in this sense:
    - There are only control sections that are to be linked, relocated, and added.
    - The loader has no way of knowing which control sections were assembled at the same time.

Program Linking Example (PROGA)

```
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
```

```
H^PROGA ^000000^000063
D^LISTA ^000040^ENDA  ^000054
R^LISTB ^ENDB  ^LISTC ^ENDC
•
•
T^000020^0A^03201D^77100004^050014
•
•
T^000054^0F^000014^FFFFF6^00003F^000014^FFFFC0
M^000024^05^+LISTB
M^000054^06^+LISTC
M^000057^06^+ENDC
M^000057^06^-LISTC
M^00005A^06^+ENDC
M^00005A^06^-LISTC
M^00005A^06^+PROGA
M^00005D^06^-ENDB
M^00005D^06^+LISTB
M^000060^06^+LISTB
M^000060^06^-PROGA
E^000020
```

Program Linking Example (PROGB)

LISTC = PROGC + 0030
= 40E2 + 0030 = 4112

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

ENDA=4054, LISTA = 4040, LISTC=4112

Program Linking Example (PROGC)

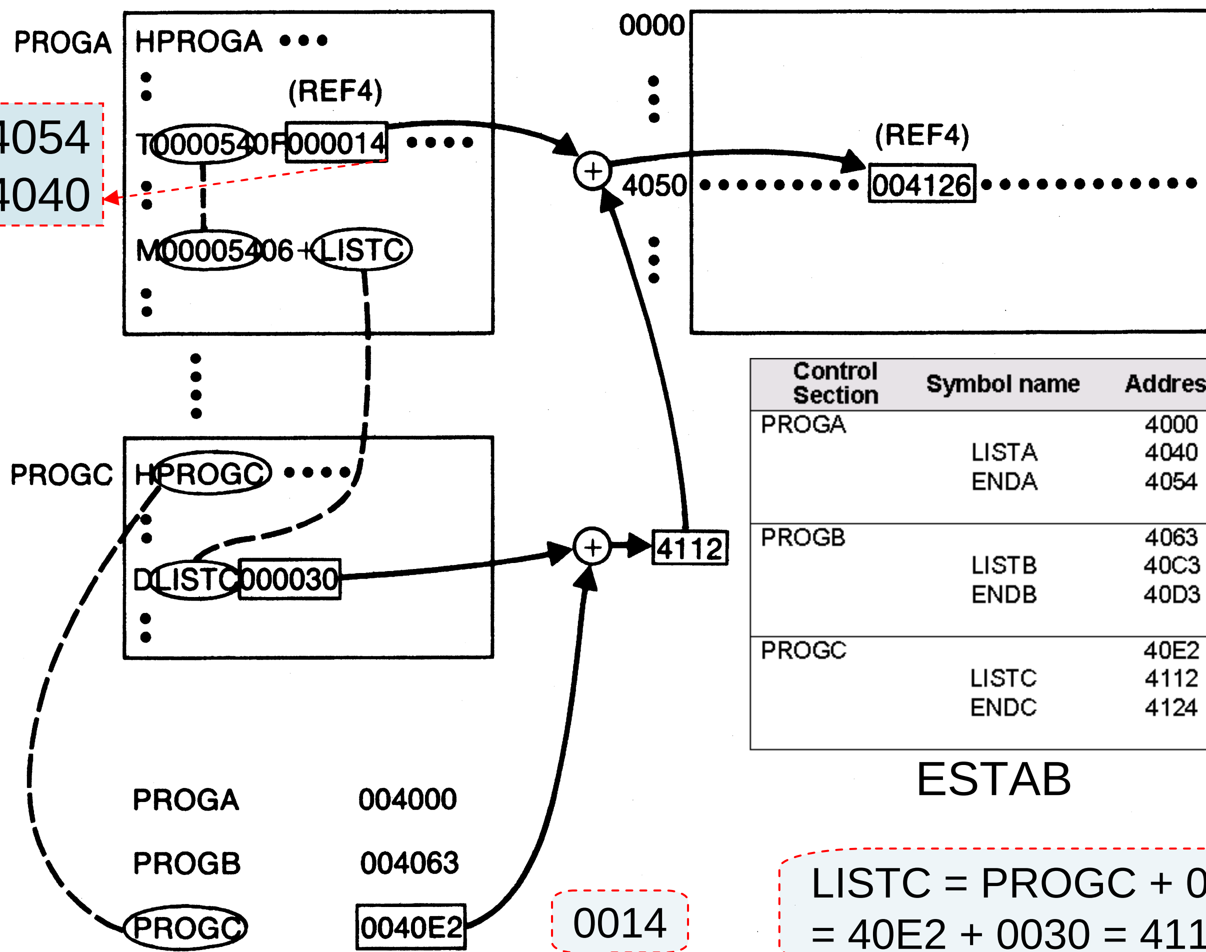
```
0000    PROGC   START   0
                EXTDEF  LISTC,ENDC
                EXTREF  LISTA,ENDA,LISTB,ENDB
                .
                .
                .
0018    REF1    +LDA    LISTA                   03100000
001C    REF2    +LDT    LISTB+4                 77100004
0020    REF3    +LDX    #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
•
T0000180C031000007710000405100000
•
T0000420F000030000008000011000000000000
M00001905+LISTA
M00001D05+LISTB
M00002105+ENDA
M00002105-LISTA
M00004206+ENDA
M00004206-LISTA
M00004206+PROGC
M00004806+LISTA
M00004B06+ENDA
M00004B06-LISTA
M00004B06-ENDB
M00004B06+LISTB
M00004E06+LISTB
M00004E06-LISTA
E
```

Relocation and Linking on REF4 from PROGA

ENDA: 4054
LISTA: 4040

Program is loaded starting at address 4000.



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	

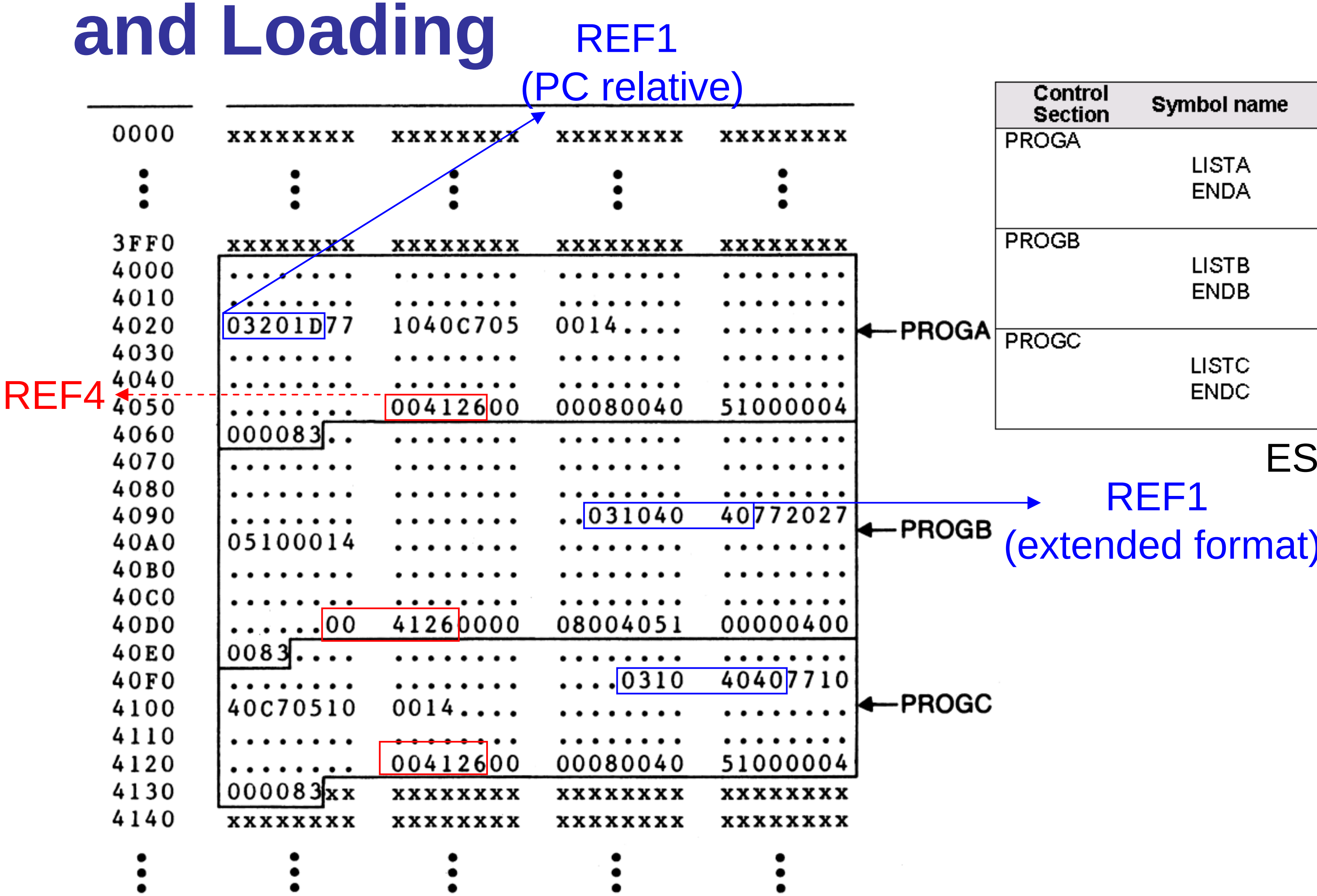
ESTAB

0014

LISTC = PROGC + 0030
= 40E2 + 0030 = 4112

0054 REF4 WORD ENDA-LISTA+LISTC 000014

Program Linking Example after Linking and Loading



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	

ESTAB



Instruction Operand Calculation

- For the references that are instruction operands, the calculated values after loading do not always appear to be equal.
  - This is because there is an additional address calculation step involved for *program-counter relative* (or *base relative*) instruction.
  - In these cases, it is the ***target addresses*** that are the same.
  - ***For example:***
    - In PROGA, the reference ***REF1*** is a PC relative instruction with displacement ***01D***. When this instruction is executed, the program counter contains the value ***4023***. (TA = 4023 + 01D = 4040)
    - No relocation is needed for this this instruction because the PC will always contains the actual address of the next instruction.
      - This is considered as relocation at execution time automatically through target address calculation.
    - In PROGB, reference ***REF1*** is *an extended format instruction* that contains a direct address. This address (after linking) is 4040.



External Reference Issue in Linking Loaders

- The input to a loader consists of a set of object programs (i.e., control sections) that are to be linked together.
  - In a control section, ***external reference*** to a symbol whose definition does not appear until later in this input stream (.i.e., other control sections).
  - In such a case, the required linking operation cannot be performed until an address is assigned to the external symbol (i.e., until the later control section is read).
- In order to resolve the address of external references, a linking loader usually makes two passes over its input.
  - ***Pass 1***: Assign address to all external symbols.
  - ***Pass 2***: Perform the actual loading, relocation, and linking.



Data Structure for Linking Loader

- ***External symbol table*** (***ESTAB***) is the main data structure needed for the linking loader.
  - It is to store the ***name*** and ***address*** of each external symbol.
  - It is similar to SYMTAB in the assembler.
  - It indicates in which control section the symbol is defined.
  - A *hash organization* is typically used for this table.
  - Two variables are defined:
    - ***PROGADDR (program load address)***:
      - Indicate the beginning address in memory where the linked program is to be loaded. Its value is supplied to loader by the operating system.
    - ***CSADDR (control section address)***:
      - Contain the starting address assigned to the control section currently being scanned by the loader.



Pass 1 of Linking Loader

- In Pass 1, the loader is concerned only with ***Header*** and ***Define*** record types in the control sections.
  - Initialization:
    - The beginning load address for the linked program (**PROGADDR**) is obtained from the operating system.
    - This becomes the starting address (**CSADDR**) for the **first control section** in the input sequence.
  - Record scanning:
    - The ***control section name*** from the **Header record** is entered into **ESTAB**, with value given by CSADDR.
    - All ***external symbols*** appearing in the **Define record** for the control section are also entered into **ESTAB**.
    - External symbols' addresses are obtained by adding the value specified in the Define record to CSADDR. (**specified address + CSADDR**)
    - When **End record** is read, the control section length **CSLTH** (obtained from Header record) is added to CSADDR to calculate the starting address for the ***next control section***. (**CSADDR = CSAADR + CSLTH**)



Pass 1 of Linking Loader (Cont.)

- At the end of Pass 1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each.
- Many loaders include as an option the ability to print a ***load map*** that shows these symbols and their addresses.
  - This information is useful in program debugging.
- The following table is the ESTAB of the previous example at the end of Pass 1.

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	

Algorithm for Pass 1

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR {for first control section}
while not end of input do    // Each iteration processes one control section
    begin
        read next input record {Header record for control section}
        set CSLTH to control section length        // Header record
        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    // Define record
                    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'}    // reach End record
        add CSLTH to CSADDR {starting address for next control section}
    end {while not EOF}
end {Pass 1}
```

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



Pass 2 of Linking Loader

- In Pass 2, loader performs the actual *loading*, *relocation*, and *linking* of the program.
  - **CSADDR** is used in the same way as it was in Pass 1.
    - 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. (**specified address + CSADDR**)
  - When a **Modification record** is encountered, the symbol whose value is to be used for modification is looked up in **ESTAB**.
    - This value is then added to or subtracted from the indicated location in memory.



Pass 2 of Linking Loader (Cont.)

- The last step of Pass 2 is to transfer control to the loaded program to begin execution.
  - The **End record** for each control section may contain the address of the first instruction in that control section to be executed.
  - Two scenarios could be encountered:
    - 1. If more than one control section specifies a ***transfer address***, the loader arbitrarily uses the last one encountered.
    - 2. If no control section contains a transfer address, the loader uses the beginning of the linked program (i.e., **PROGADDR**) as the transfer point.
  - Normally, a transfer address would be placed in the **End record** for a ***main program***, but not for a subroutine.



Algorithm for Pass 2

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
    begin                                    // Each iteration processes one control section
        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    // Text record
                    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    // Modification record
                    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)  // End record: transfer address is specified
        add CSLTH to CSADDR    // Move to next CS
    end {while not EOF}
jump to location given by EXECADDR {to start execution of loaded program}
end {Pass 2}
```



Advanced Method for External Symbols

- We can assign a ***reference number*** to each external symbol referred to in a control section.
- This reference number is used in **Modification records**.
- E.g.,
  - ***Control section name*** with reference number 01.
  - The ***other external reference symbols*** are assigned numbers as part of the **Refer record** for the control section.
- 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***.
  - An external reference symbol can be looked up in ESTAB once for each control section that uses it.
  - The value for code modification can then be obtained by simply ***indexing into an array of these values***.

Advanced Method for External Symbols (PROGA)

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
RLISTB ENDB  LISTC ENDC
•
•
T0000200A03201D77100004050014
•
•
T0000540F000014FFFF600003F000014FFFFC0
M00002405+LISTB
M00005406+LISTC
M00005706+ENDC
M00005706-LISTC
M00005A06+ENDC
M00005A06-LISTC
M00005A06+PROGA
M00005D06-ENDB
M00005D06+LISTB
M00006006+LISTB
M00006006-PROGA
E000020
```

→

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
R02LISTB 03ENDB  04LISTC 05ENDC
•
•
T0000200A03201D77100004050014
•
•
T0000540F000014FFFF600003F000014FFFFC0
M00002405+02
M00005406+04
M00005706+05
M00005706-04
M00005A06+05
M00005A06-04
M00005A06+01
M00005D06-03
M00005D06+02
M00006006+02
M00006006-01
E000020
```

Advanced Method for External Symbols (PROGB)

```
HPROGB 00000000007F
DLISTB 000060ENDB  000070
RLISTA ENDA  LISTC ENDC
•
•
T0000360B0310000077202705100000
•
•
T0000700F000000FFFFF6FFFFFFFFFFF0000060
M00003705+LISTA
M00003E05+ENDA
M00003E05-LISTA
M00007006+ENDA
M00007006-LISTA
M00007006+LISTC
M00007306+ENDC
M00007306-LISTC
M00007606+ENDC
M00007606-LISTC
M00007606+LISTA
M00007906+ENDA
M00007906-LISTA
M00007C06+PROGB
M00007C06-LISTA
E
```

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

Advanced Method for External Symbols (PROGC)

```
HPROGC 000000000051
DLISTC 000030ENDC  000042
RLISTA ENDA  LISTB ENDB
•
•
T0000180C031000007710000405100000
•
•
T0000420F000030000008000011000000000000
M00001905+LISTA
M00001D05+LISTB
M00002105+ENDA
M00002105-LISTA
M00004206+ENDA
M00004206-LISTA
M00004206+PROGC
M00004806+LISTA
M00004B06+ENDA
M00004B06-LISTA
M00004B06-ENDB
M00004B06+LISTB
M00004E06+LISTB
M00004E06-LISTA
E
```

→

```
HPROGC 000000000051
DLISTC 000030ENDC  000042
R02LISTA 03ENDA  04LISTB 05ENDB
•
•
T0000180C031000007710000405100000
•
•
T0000420F000030000008000011000000000000
M00001905+02
M00001D05+04
M00002105+03
M00002105-02
M00004206+03
M00004206-02
M00004206+01
M00004806+02
M00004B06+03
M00004B06-02
M00004B06-05
M00004B06+04
M00004E06+04
M00004E06-02
E
```



Machine-Independent Loader Features



Automatic Library Search

- Many linking loaders can automatically incorporate routines from a **subprogram library** into the program being loaded.
  - In most cases, there is a ***standard system library*** that is used in this way.
  - ***Other libraries*** may be specified by *control statements* or by *parameters* to the loader.
- Automatic library search allows programmer to use subroutines from one or more libraries.
- The programmer does not need to take any action beyond mentioning the subroutine names as external references in the source program.



Automatic Library Search (Cont.)

- Linking loaders must keep track of *external symbols* that are *referred to (but not defined)* in the primary input to the loader.
  - 1. Symbols from each **Refer record** are entered **ESTAB**.
  - 2. When the definition for a symbol is encountered, the address assigned to the symbol is filled I to complete the symbol entry.
  - 3. At the end of Pass 1, the symbols in ESTAB that remain undefined represent *unresolved external references*.
  - 4. The loader searches the libraries specified for routines that contain the definitions of these symbols.
  - 5. The loader then processes the subroutines found by this search exactly as if they had been part of the primary input stream.
- The subroutines fetched from a library may themselves contain external references. It is necessary to repeat the library search process until all references are resolved.



Automatic Library Search (Cont.)

- Automatic library search allows programmers to override the standard subroutines in the library by supplying his or her own routines.
- ***For example:***
  - Suppose that the main program refers to a standard subroutine name **SQRT**.
  - A programmer who wanted to use a different version of **SQRT** could simply include the new SQRT as input to the loader.
  - By the end of Pass 1 of the loader, SQRT would already be defined, so the original SQRT would not be included in any library search.



Automatic Library Search (Cont.)

- The libraries to be searched by the loader ordinarily contain ***assembled*** or ***compiled*** versions of the subroutines (i.e., object code).
  - It is possible to search these libraries by *scanning the **Define records*** for all of the object programs in the library, but *it is lack of efficiency*.
  - In most cases, a *special file structure* is used for the libraries:
    - This structure contains a ***directory*** that gives
      - The ***name*** of each routine and
      - A ***pointer*** to its address within the file.
  - If a subroutine is callable by ***more than one name*** (using different entry points), both names are entered into the directory.
  - The library search itself involves
    - 1) a search of the directory, followed by
    - 2) reading the object programs indicated by this search.
  - If the directory could be stored in memory, the search process could be accelerated.



Loader Options - Special Command Language

- Many loaders have a ***special command language*** that is used to specify options.
  - Sometimes there is a ***separate input file*** to the loader that contains such control statements.
  - Sometimes these same statements can also be ***embedded in the primary input stream between object programs***.
  - On a few systems, the programmer can even include ***loader control statements in the source program***, and ***assembler*** or ***compiler*** retains these commands as a part of the object program.
- Note: Some systems use ***job control language*** that is processed by the operating system.
  - When this approach is used, the OS incorporates the options specified in a ***control block*** that is made available to the loader when it is invoked.

Loader Options – Alternative Sources

- One typical loader option allows the selection of ***alternative sources of input***.
- ***For example:***

INCLUDE program-name (library-name)

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.

DELETE csect-name

Allow users to delete the named *control sections*.

CHANGE name 1, name2

Change the external symbol from *name1* to *name2*. whenever it appears in the object program.

LIBRARY MYLIB

Allow the user to specify alternative libraries to be searched. Such user-specified libraries are normally *searched before the standard system libraries*.

Loader Options - Example

- Suppose that a set of utility subroutines is made available on the computer system.
  - Two of these (READ and WRITE) are designed to perform the same functions as RDREC and WRREC.
  - The following sequence of loader commands could be used to make this change without reassembling the program:

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

Change external references to RDREC to refer to symbol READ.

Loader Options – Non-Resolved External References

- Loaders might allow uses to specify that some references not be resolved.
- ***For example***:
  - A certain program can perform an analysis of the data using the routines STDDEV, PLOT, CORREL from a statistical library.
  - Since the program contains external references to these three routines, they are ordinarily 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 STDDEV, PLOT, CORREL
    ```

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



Loader Options – Other Options

- It is also possible to specify that ***no external references to be resolved by library search.***
  - This option is useful when programs are to be linked but not executed immediately.
  - It is desirable to postpone the resolution of external references in some cases. (e.g., **dynamic linking**)
- Other options:
  - The ability to output *load map* that shows the detailed information (e.g., *control section names*, *addresses*, *external symbol addresses*, *cross-reference table*) during loading.
  - The ability to specify the location at which execution is to begin overriding any information given in the object programs.
  - The ability to control whether or not the loader should attempt to execute the program if errors detected during the load (e.g., unresolved external references)

Loader Design Options



Loader Design Options

- Linkage editors
  - Perform linking prior to load time.
- Dynamic linking
  - Link functions at execution time.
- Bootstrap loaders
  - Loaders that can be used to run stand-alone programs independent of the operating system or the system loader.

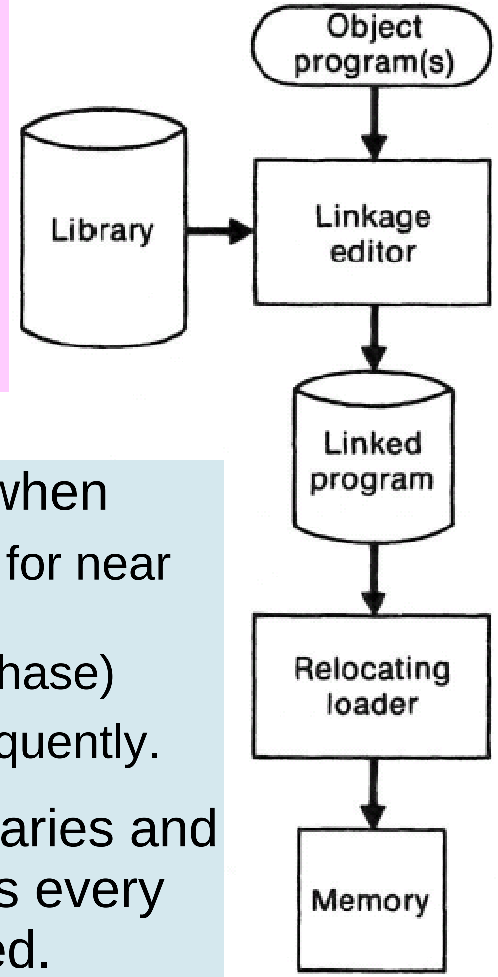


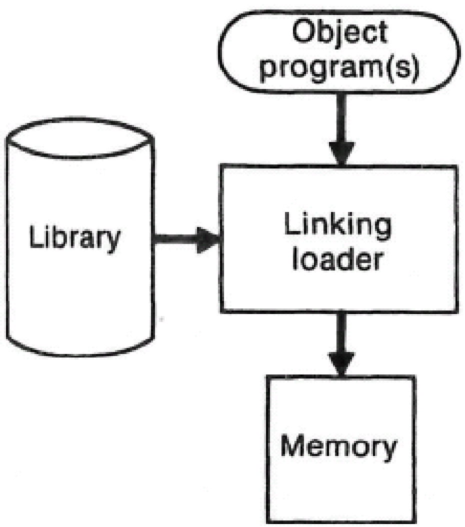
Linkage Editors

- The linkage editor produces a linked version of the program (called a *load module* or an *executable image*) that is written to a file or library for later execution.
- When the user is ready to run the linked program, a simple ***relocating loader*** can be used to load the program into memory.
  - The only object code modification is the addition of an actual load address to relative values within the program.
- The linkage editor performs relocation of all control sections ***relative to the start of the linked program***.
  - Thus, all items that need to be modified at load time have values that are relative to the start of the linked program.
  - This means that the loading can be accomplished in one pass with no external symbol table required.
- All external references are resolved, and relocation is indicated by *Modification records* or a *bit mask*.
- Information concerning external references is often retained in the linked program to allow ***subsequent relinking*** of the program to ***replace control sections***, ***modify external references***.

Linking Loader vs. Linkage Editors

- Linkage editor is preferred when
  - A program is to be executed many times without being reassembled.
- Linkage editor resolves external references and searches library once.





- Linking loader is preferred when
  - A program is reassembled for near every execution. (during development and testing phase)
  - A program is used so infrequently.
- Linking loader searches libraries and resolves external references every time the program is executed.

Linking Loader

Linkage Editors



Linkage Editors – Subroutine Replacement

- 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.
- Linkage editors can replace existing subroutines without going back to the original versions of all other subroutines.
  - Consider a program (**PLANNER**) that uses a large number of subroutines.
  - One subroutine (**PROJECT**) used by the program is changed to correct an error or to improve efficiency.

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



Linkage Editors – Building Packages

- Linkage editors can be used to build packages of subroutines or other control sections that are used together.
- ***For example***:
  - In FORTRAN, there are a large number of subroutines that are used to handle formatted input and output:
    - Read and write data blocks (READR, WRITER)
    - Block and deblock records (BLOCK, DEBLOCK)
    - Encode and decode data items (ENCODE, DECODE)
  - There are a large number of cross-references between these subprograms because of their closely related functions.
  - If a program using formatted I/O were linked in the usual way all of the cross-references between these library subroutines would need to processed individually. (every time a FORTRAN program is linked)
  - The linkage editor could combine the appropriate subroutines into a ***package*** to reduce linking overheads.

A search of SUBLIB before FTNLIB would retrieve FTNIO instead of the separate routines. FINTO already has all cross-references between subroutines resolved.

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

Link subroutines into the FTNIO.



Linkage Editors – Building Packages (Cont.)

- Linkage editors often allow the user to specify that external references are not to be resolved by automatic library search so as to reduce library space.
  - E.g., If 100 FORTAN programs using the above I/O routines have their external references resolved, this would mean that a total of 100 copies of FTNIO would be stored.
    - Thus external references between user-written routines would be resolved.
    - A linking loader could then be used to combine the linked user routines with FTNIO at execution time.
  - Because this process involves two separate linking operations, it would have more overheads, but save a lot of library space.
- In general, linkage editors ten to offer more *flexibility* and *control*, with a corresponding increase in complexity and overhead.



Dynamic Linking

- Linking types:
  - Linkage editors
    - Perform linking operations ***before the program is loaded*** for execution.
  - Linking loaders
    - Perform the linking operations ***at load time***.
  - Dynamic linking (*dynamic loading* or *load on call*)
    - Postpone the linking operations until ***execution time***.
- Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library.
  - E.g., 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 could be loaded into memory.
    - All programs currently in execution could be linked to this one copy.
- In an ***object-oriented system***, dynamic linking is often used for references to software objects.



Dynamic Linking (Cont.)

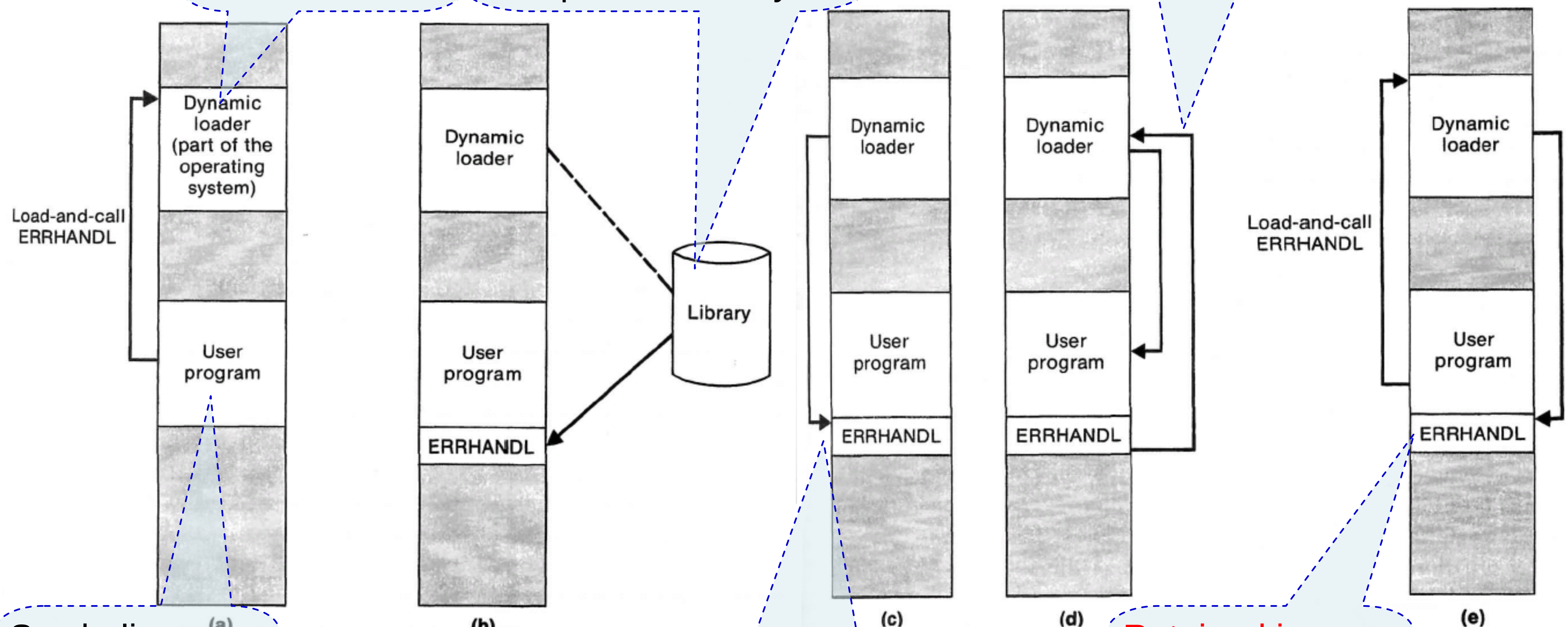
- Dynamic linking provides the ability to load the routines only when they are needed.
  - If the subroutines involved are large, or have many external references, this can result in substantial savings of *time* and *memory space*.
- Dynamic linking avoid the necessity of loading the entire library for each execution. E.g.,:
  - Suppose that a program uses only a few out of a large number of possible subroutines, but the exact routines needed can not be predicted until the program examines its input.

Dynamic Linking through Operating System

OS checks its internal tables.

If necessary, the routine is loaded from the specified library.

Return to OS, and then back to the program.



Symbolic name of the routine to be called

Pass control from the OS to the called routine.

Retained in memory for the next call.



Bootstrap Loaders

- Given an idle computer with no program in memory, we need an absolute loader to bring up the first program.
  - With the machine empty and idle, we can simply specify the absolute address for whatever program (e.g., OS) that is first loaded because no program relocation is needed.
- On some systems, an absolute loader program is permanently resident in a read-only memory (ROM).
  - On some computers, the program is executed directly on the ROM.
  - On others, the program is copied from ROM to main memory and executed there.
- On some systems, a built-in hardware function (or a very short ROM program) reads a fixed-length record from some device into memory at a fixed location.
  - After the read operation is complete, control is transferred to the loaded record. This record contains machine instructions that load the absolute program.
  - If the loading process requires the reading of others, and these in turn can cause the reading of still more record, then this is called bootstrap. And the first record is called ***bootstrap loader***.



Implementation Examples



Examples

- MS-DOS Linker
- SunOS Linker
- Cray MPP Linker



MS-DOS Linker

- Most of MS-DOS compilers and assemblers (including MASM) produce object modules (.obj files), not executable programs.
  - Each object module contains a binary image of the translated instructions, data of the program, and structure of the program.
- MS-DOS LINK is a ***linkage editor*** that combines one or more object modules to produce a complete executable program (.exe files).
  - LINK can also combine the translated programs with other modules from object code libraries.



MS-DOS Object Module

- Record types in MS-DOS object module:

Record Types	Description
THEADR	Translator header
TYPDEF PUBDEF EXTDEF	External symbols and references
LNAMES SEGDEF GRPDEF	Segment definition and grouping
LEDATA LIDATA	Translated instructions and data
FIXUPP	Relocation and linking information
MODEND	End of object module



MS-DOS Object Module (Cont.)

- Module start and end
  - ***THEADR*** record (similar to **Header record** in SIC/XE)
    - Specify the name of the object module.
  - ***MODEND*** record (similar to **End record** in SIC/XE)
    - Mark the end of the modules and contain a reference to the entry point of the program.
- External symbols and references
  - ***PUBDEF*** record (similar to **Define record** in SIC/XE)
    - Contain a list of external symbols (i.e., public names) that are defined in this module.
  - ***EXTDEF*** record (similar to **Refer record** in SIC/XE)
    - Contain a list of external symbols that are referred to in this module.
  - ***TYPDEF*** record
    - Contain information about the data type designated by an external name.



MS-DOS Object Module (Cont.)

- Segment definition and grouping
  - ***SEGDEF*** record
    - Describe the segments in the module, including their name, length, and alignment.
  - ***GRPDEF*** record
    - Specify how these segments are combined into groups.
  - ***LNAMES*** record
    - Contain a list of all the segment and class names used in the programs.
    - Note: SEGDEF and GRPDEF records refer to a segment by giving the position of its name in the LNAMES records (similar to the "*reference number*" in SIC/XE).



MS-DOS Object Module (Cont.)

- Translated instructions and data
  - ***LEDATA*** record (similar to **Text record** in SIC/XE)
    - Contain translated *instruction* and *data* from the source program.
  - ***LIDATA*** record
    - Specify translated instructions and data that occur in a *repeating pattern*.
- Relocation and linking information
  - ***FIXUPP*** record (similar to **Modification record** of SIC/XE)
    - Use to *resolve external references*, and carry out *address modifications* that are associated with relocation and grouping of segments within the program.
    - Must immediately follow the LEDATA or LIDATA record to which it applies.



MS-DOS LINK

- LINK performs its processing in two passes:
  - ***Pass 1***
    - Compute a *starting address* for each *segment* in the program.
      - ***Segments*** are placed into the executable program in the same order of processing ***SEGDEF records***.
      - ***Segments*** from different object modules are combined if they have the *same segment name and class*.
      - Segments with the same class, but different names, are concatenated.
    - Control a ***symbol table*** that associates an address with each *segment* and each *external symbol*.
      - If unresolved external symbols remain after all object modules have been processed, LINK searches the specified *libraries*. (similar to automatic ***library search*** in SIC/XE)

MS-DOS LINK (Cont.)

– ***Pass 2***
  - Extract the translated instructions and data from modules.
  - Build an image of the ***executable program*** in memory because the executable program is organized by segment, not the order of object modules.
    - Building a memory image is the most efficient way to handle rearrangements caused by combining and concatenating segments.
    - If the available memory is not enough, use a temporary disk file.
  - Process each LEDATA and LIDATA record with FIXUPP records.
    - Placing the binary data from records into the memory image at locations that reflect the segment addresses computed in Pass 1.
    - *Repeated data specified in LIDATA records is expanded at this time.*
  - Resolve relocations and external references.
    - Relocation operations that involve ***the starting address of a segment*** are added to ***a table of segment fixups***.
      - » This table is used to perform segment relocation when the program is loaded for execution.



MS-DOS LINK (Cont.)

- After the memory image is complete, it is written as an executable (.exe) file. This file contains a header that contains
  - The table of segment fixups
  - Information about memory requirements and entry points
  - Initial contents for registers CS and SP.

SunOS Linkers

- SunOS provides two different linkers:
  - ***Link-editor***:
    - It is invoked in the process of *compiling* a program.
    - It takes object modules (produced by assemblers or compilers) and combines them to produce a single output module.
  - ***Run-time link***:
    - It is invoked at *execution* time to bind dynamic executables and shared objects.
    - It determines what shared objects are required , and ensures that these objects are included.
    - It also inspects whether the share objects have the dependency on other shard objects.



SunOS Linkers – Link Editor

- The output module of the link editor could be one of the following types:
  - ***Relocatable object module***
    - Suitable for further link-editing
  - ***Static executable***
    - All symbolic references bound and ready to run
  - ***Dynamic executable***
    - Some symbolic references may need to be bound at run time.
  - ***Shared object***
    - This provides services that can be bound at run time to other *dynamic executables*.



SunOS Linkers – Link Editor (Cont.)

- An ***object module*** contains one or more ***sections***, which represent the instructions and data areas from the source program.
  - Each section has a set of attributes (e.g., "*executable*" and "*writable*").
  - The object module also includes
    - ***A list of the relocation and linking operations*** that need to be performed, and
    - ***A symbol table*** that describes the symbols used in these operations.



SunOS Linkers – Link Editor (Cont.)

- The link-editor begins by reading the input files of object modules .
  - Sections from the input files that have *the same attributes are concatenated* to form new sections within the output file.
  - The symbol tables from the input files are processed to match *symbol definitions and references*, and *relocation and linking operations* within the output file.
- The linker normally generates *a new symbol table*, and *a new set of relocation instruction*, within the output file.
  - For symbols that must be bound at run time.
  - For relocation operations that must be performed when loaded.
- Relocation and linking operations are specified using a set of *processor-specific codes*.
  - Processor-specific codes describe ***the size of the field that is to be modified***, and ***the calculation that must be performed***.
    - Relocation codes for different machines (e.g., SPARC and x86) are different.



SunOS Linkers – Link Editor (Cont.)

- ***Symbolic references*** from the input files (that do not have matching definitions) are processed by referring to *archives* and *shared objects*.
  - An ***archive*** is a collection of relocatable object modules.
    - A ***directory*** stored with the archive associates *symbol names* with *the object modules* that contain their definitions.
    - Selected modules from an archive are automatically included to resolve symbolic references.
  - A ***shared object*** is an indivisible unit that was generated by a previous link-edit operation.
    - When the link-editor encounters a reference to a symbol defined in a shared object, the entire contents of the shared object become a *logical part of the output file*.
    - All symbols defined in the object are made available to the link-editing process.
    - The shared object is not physically included in the output file.
      - The actual inclusion is deferred until run time.
      - The link-editor only records the dependency on the shared object.



SunOS Linkers – Run-Time Linker

- The run-time linker determines what shared objects are required , and ensures that these objects are included.
- After the necessary shared objects are included, the linker performs relocation and linking operations.
  - The operations are specified in ***the relocation and linking sections*** of the *dynamic executable* and *shared objects*.
    - They bind symbols to the actual memory addresses.
    - ***Binding of data references*** is performed before the control is passed to the executable program.
    - ***Binding of procedure calls*** is normally deferred until the program is in execution. This is called ***lazy binding***:
      - When a procedure is called for the first time, the linker looks up the actual address of the called procedure and insert it ***a procedure linkage table***.
      - The subsequent calls will go directly to the called procedure through this table.

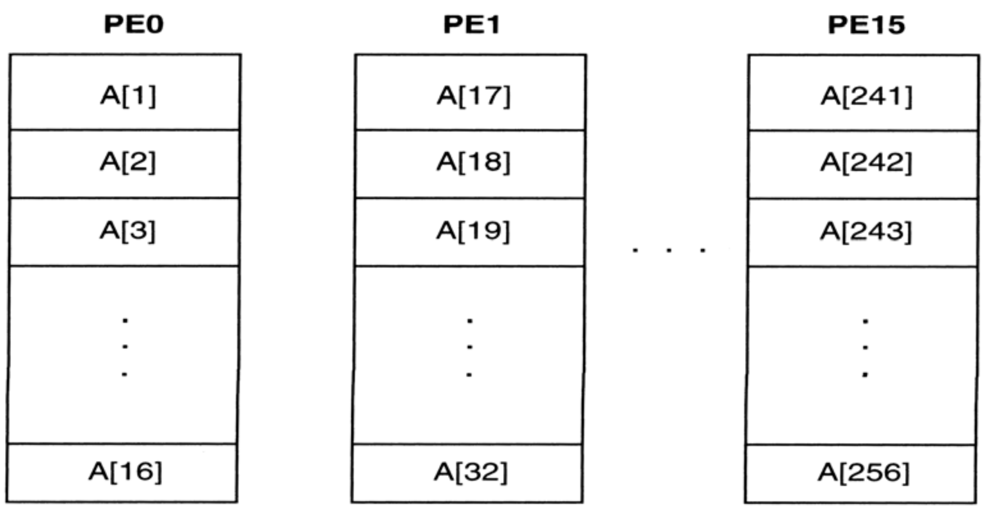
SunOS Linkers – Run-Time Linker (Cont.)

- During execution, a program can *dynamically bind to new shared objects* by requesting the same services of the linker by inserting the actual address to the ***procedure linkage table***.
  - This feature allows a program to choose between a number of shared objects.
  - This feature reduces the amount of overhead required for starting a program and the amount of required memory.



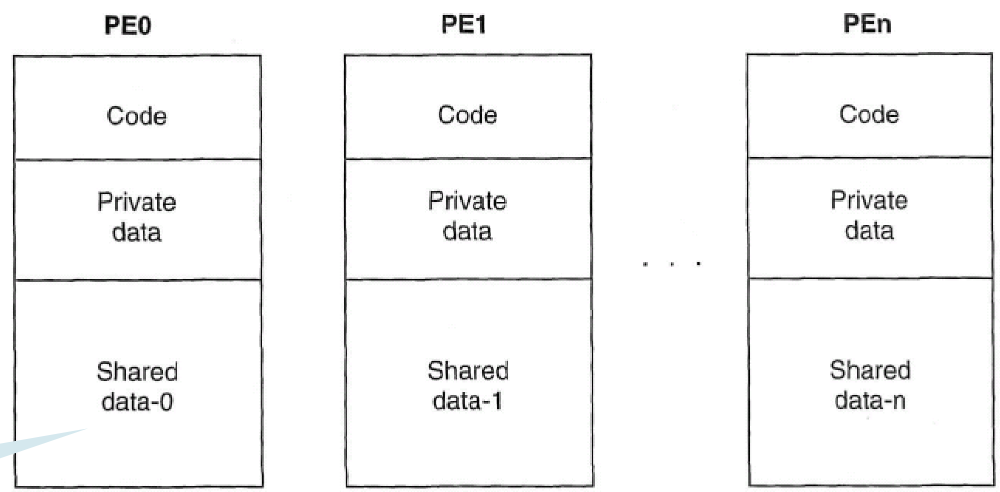
Cray MPP Linker

- Cray MPP architecture
  - A T3E system contains a large number of processing elements (PEs).
  - Each PE has its own local memory. (Faster)
  - A PE can access the memory of all other PEs. (called *remote memory*). (Slower)
- An application program on a T3E system is normally allocated a *partition* that consists of several PEs.
- The work to be done by the program is divided between the PEs in the partition.
  - One common way is to distribute the elements of an array among the PEs.
  - This kind of data sharing and work between PEs can be specified in a program.



Cray MPP Linker (Cont.)

- Shared data and private data
  - Data divided among a number of PEs is called ***shared data***.
  - Data that are not shared among PEs are called ***private data***.
- In most cases, private data is replicated on each PE in the partition.
- When a program is loaded, each PE gets
  - A copy of the executable code for the program,
  - Its private data, and
  - Its portion of the share data.



Shared data-i indicates the portion assigned to PEi



Cray MPP Linker (Cont.)

- The MPP linker organizes blocks of code or data from the object programs into ***lists***.
  - The blocks on a given list all share some common property, e.g., *executable code*, *private data*, or *shared data*.
  - The blocks on each list are *collected together*, *assigned addresses*, and *performed with relocation/linking operations*.
- The linker then writes an executable file that contains the relocated and linked blocks. This executable file also specifies the number of required PEs.
- The distribution of shared data depends on the number of PEs in the partition.
  - If the number of PEs in the partition is specified at compile time, it cannot be overridden later.
  - If the partition size is not specified at compile time, there are two possibilities:
    - 1. The linker creates an executable file specifying a fixed number of PEs.
    - 2. The linker allows the partition size to be chosen at run time (*plastic executable*).
      - A plastic executable file must contain a copy of *all relocatable object modules* and *all linker directives needed to the final executable*.