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 3.7 Relocatable program for a standard SIC machine.

```
HCOPY  000000001007A
T000000 1EFFC14003348103900003628003030001548106 13C0003000002A0C0039000002D
T00001E 15E000C003648106 1080033 4C0000454F46000003000000
T001039 1EFFC040030000030E0105D30103FD8105D280030301057548039 2C105E38103F
T001057 0A8001000364C0000F1001000
T001061 19FE0040030E01079301064508039DC10792C00363810644C000005
E000000
```

Figure 3.8 Object program with relocation by bit mask.

this mask is represented (in character form) as three hexadecimal digits. These characters are underlined for easier identification in the figure.

If the relocation bit corresponding to a word of object code is set to 1, the program's starting address is to be added to this word when the program is relocated. A bit value of 0 indicates that no modification is necessary. If a Text record contains fewer than 12 words of object code, the bits corresponding to unused words are set to 0. Thus the bit mask FFC (representing the bit string 111111111100) in the first Text record specifies that all 10 words of object code are to be modified during relocation. These words contain the instructions corresponding to lines 10 through 55 in Fig. 3.7. The mask E00 in the second Text record specifies that the first three words are to be modified. The remainder of the object code in this record represents data constants (and the RSUB instruction) and thus does not require modification.

The other Text records follow the same pattern. Note that the object code generated from 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 corresponds to a relocation bit. The assembled LDX instruction does require modification because of the direct address. However, if it were placed in the preceding Text record, it would not be properly aligned to correspond to a relocation bit because of the l-byte data value generated from line 185. Therefore, this instruction must begin a new Text record in the object program.

You should carefully examine the remainder of the object program in Fig. 3.8. Make sure you understand how the relocation bits are generated by the assembler and used by the loader. SIC relocation loader algorithm is shown in Fig. 3.9.

Some computers provide a 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

```
begin
   get PROGADDR from operating system
   while not end of input do
     begin
       read next record
       while record type ≠ 'E' do
       while record type = 'T'
          begin
             get length = second data
             mask bits(M) as third data
                For(i = 0, i < length, i++)
                       if M_i = 1 then
                         add PROGADDR at the location PROGADDR + specified
                              address
                       else
                         move object code from record to location PROGADDR +
                              specified address
             read next record
          end
     end
end
```

Figure 3.9 SIC relocation loader algorithm.

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. (We discuss this further when we study memory management in Chapter 6.) As the next section illustrates, however, the loader must still handle relocation of subprograms in connection with linking.

3.2.2 Program Linking

The basic concepts involved in program linking were introduced in Section 2.3.5. Before proceeding you may want to review that discussion and the examples in that section. In this section we consider more complex examples of external references between programs and examine the relationship between relocation and linking. The next section gives an algorithm for a linking and relocating loader.

Figure 2.15 in Section 2.3.5 showed a program made up of three control sections. These control sections could be assembled together (that is, in the same invocation of the assembler), or they could be assembled independently of one another. In either case, however, they would appear as separate segments of object code after assembly (see Fig. 2.17). The programmer has a natural

inclination to think of a program as a logical entity that combines all of the related control sections. From the loader's point of view, however, there is no such thing as a program in this sense—there are only control sections that are to be linked, relocated, and loaded. The loader has no way of knowing (and no need to know) which control sections were assembled at the same time.

Consider the three (separately assembled) programs in Fig. 3.10, each of which consists of a single control section. Each program contains a list of items (LISTA, LISTB, LISTC); the ends of these lists are marked by the labels ENDA, ENDB, ENDC. The labels on the beginnings and ends of the lists are external symbols (that is, they are available for use in linking). Note that each program contains exactly the same set of references to these external symbols. Three of these are instruction operands (REF1 through REF3), and the others are the values of data words (REF4 through REF8). In considering this example, we examine the differences in the way these identical expressions are handled within the three programs. This emphasizes the relationship between the relocation and linking processes. To focus on these issues, we have not attempted to make these programs appear realistic. All portions of the programs not involved in the relocation and linking process are omitted. The same applies to the generated object programs shown in Fig. 3.11.

Consider first the reference marked REF1. For the first program (PROGA), REF1 is simply a reference to a label within the program. It is assembled in the usual way as a program-counter relative instruction. No modification for relocation or linking is necessary. In PROGB, on the other hand, the same operand refers to an external symbol. The assembler uses an extended-format instruction with address field set to 00000. The object program for PROGB (see Fig. 3.11) contains a Modification record instructing the loader to add the value of the symbol LISTA to this address field when the program is linked. This reference is handled in exactly the same way for PROGC.

The reference marked REF2 is processed in a similar manner. For PROGA, the operand expression consists of an external reference plus a constant. The assembler stores the value of the constant in the address field of the instruction and a Modification record directs the loader to add to this field the value of LISTB. In PROGB, the same expression is simply a local reference and is assembled using a program-counter relative instruction with no relocation or linking required.

REF3 is an immediate operand whose value is to be the difference between ENDA and LISTA (that is, the length of the list in bytes). In PROGA, the assembler has all of the information necessary to compute this value. During the assembly of PROGB (and PROGC), however, the values of the labels are unknown. In these programs, the expression must be assembled as an external reference (with two Modification records) even though the final result will be an absolute value independent of the locations at which the programs are loaded.

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		

Figure 3.10 Sample programs illustrating linking and relocation.

```
Loc       Source statement                                  Object code

0000      PROGC   START   0
                  EXTDEF  LISTC,ENDC
                  EXTREF  LISTA,ENDA,LISTB,ENDB
                  .
                  .
                  .
0018      REF1    +LDA    LISTA                             03100000
001C      REF2    +LDT    LISTB+4                           77100004
0020      REF3    +LDX    #ENDA-LISTA                       05100000
                  .
                  .
                  .
0030      LISTC   EQU     *
                  .
                  .
0042      ENDC    EQU     *
0042      REF4    WORD    ENDA-LISTA+LISTC                  000030
0045      REF5    WORD    ENDC-LISTC-10                     000008
0048      REF6    WORD    ENDC-LISTC+LISTA-1                000011
004B      REF7    WORD    ENDA-LISTA-(ENDB-LISTB)           000000
004E      REF8    WORD    LISTB-LISTA                       000000
                  END
```

Figure 3.10 (*cont'd*)

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
RLISTB ENDB  LISTC ENDC
:
:
T0000200A03201D77100004050014
:
:
T0000540F000014FFFFF600003F000014FFFFC0
M000024 05+LISTB
M000054 06+LISTC
M000057 06+ENDC
M000057 06-LISTC
M00005A 06+ENDC
M00005A 06-LISTC
M00005A 06+PROGA
M00005D 06-ENDB
M00005D 06+LISTB
M000060 06+LISTB
M000060 06-PROGA
E000020
```

Figure 3.11 Object programs corresponding to Fig. 3.10.

```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
RLISTA ENDA  LISTC ENDC
.
.
T0000360B0310000077202705100000
.
.
T00007000F000000FFFFF6FFFFFFFFFFF0000060
M00003705+LISTA
M00003E05+ENDA
M00003E05-LISTA
M00007006+ENDA
M00007006-LISTA
M00007006+LISTC
M00007306+ENDC
M00007306-LISTC
M00007606+ENDC
M00007606-LISTC
M00007606+LISTA
M00007906+ENDA
M00007906-LISTA
M00007C06+PROGB
M00007C06-LISTA
E


HPROGC 000000000051
DLISTC 000030ENDC  000042
RLISTA ENDA  LISTB ENDB
.
.
T0000180C0310000077100004105100000
.
.
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
```

Figure 3.11 (*cont'd*)

The remaining references illustrate a variety of other possibilities. The general approach taken is for the assembler to evaluate as much of the expression as it can. The remaining terms are passed on to the loader via Modification records. To see this, consider REF4. The assembler for PROGA can

evaluate all of the expression in REF4 except for the value of LISTC. This results in an initial value of (hexadecimal) 000014 and one Modification record. However, the same expression in PROGB contains no terms that can be evaluated by the assembler. The object code therefore contains an initial value of 000000 and three Modification records. For PROGC, the assembler can supply the value of LISTC relative to the beginning of the program (but not the actual address, which is not known until the program is loaded). The initial value of this data word contains the relative address of LISTC (hexadecimal 000030). Modification records instruct the loader to add the beginning address of the program (i.e., the value of PROGC), to add the value of ENDA, and to subtract the value of LISTA. Thus the expression in REF4 represents a simple external reference for PROGA, a more complicated external reference for PROGB, and a combination of relocation and external references for PROGC.

You should work through references REF5 through REF8 for yourself to be sure you understand how the object code and Modification records in Fig. 3.11 were generated.

Figure 3.12(a) shows these three programs as they might appear in memory after loading and linking. PROGA has been loaded starting at

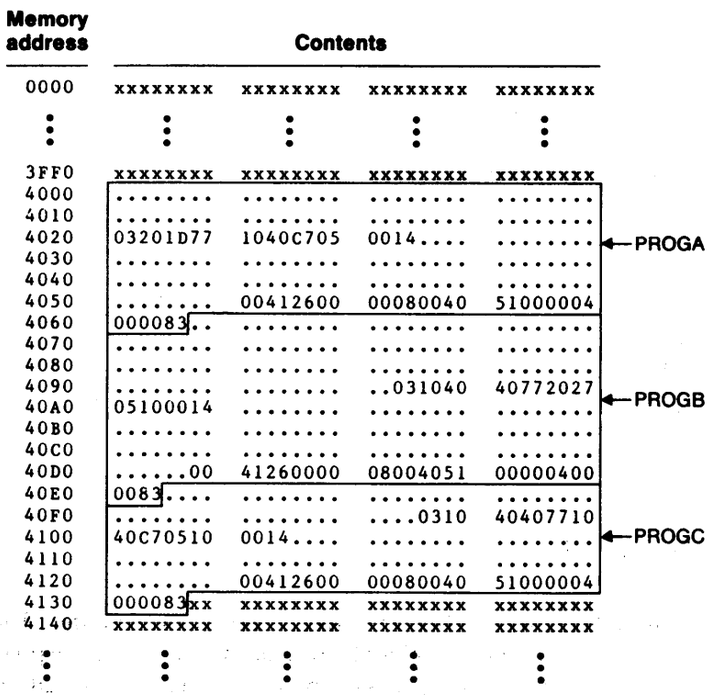


Figure 3.12(a) Programs from Fig. 3.10 after linking and loading.

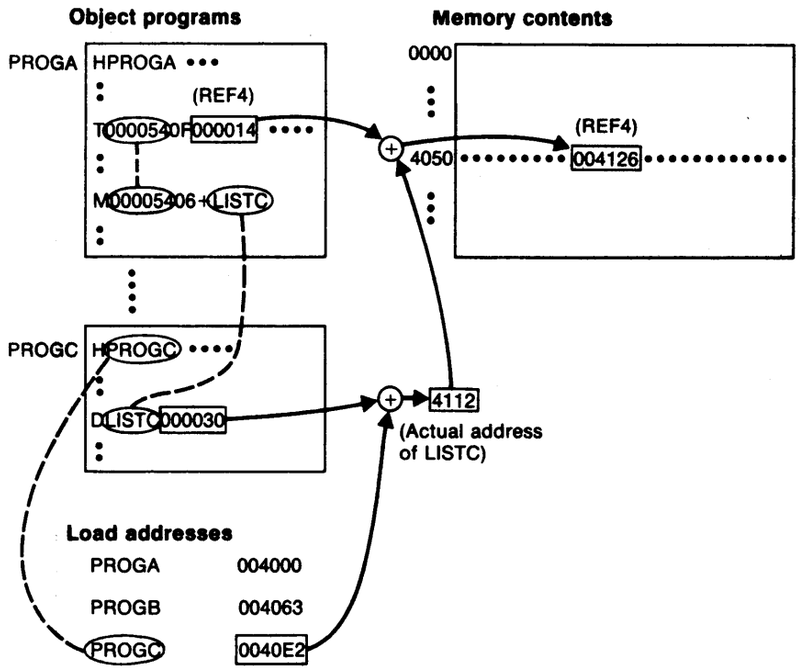


Figure 3.12(b) Relaxation and linking operations performed on REF4 from PROGA.

address 4000, with PROGB and PROGC immediately following. Note that each of REF4 through REF8 has resulted (after relocation and linking is performed) in the same value in each of the three programs. This is as it should be, since the same source expression appeared in each program.

For example, the value for reference REF4 in PROGA is located at address 4054 (the beginning address of PROGA plus 0054, the relative address of REF4 within PROGA). Figure 3.12(b) shows the details of how this value is computed. The initial value (from the Text record) is 000014. To this is added the address assigned to LISTC, which is 4112 (the beginning address of PROGC plus 30). In PROGB, the value for REF4 is located at relative address 70 (actual address 40D3). To the initial value (000000), the loader adds the values of ENDA (4054) and LISTC (4112), and subtracts the value of LISTA (4040). The result, 004126, is the same as was obtained in PROGA. Similarly, the computation for REF4 in PROGC results in the same value. The same is also true for each of the other references REF5 through REF8.

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) instructions. In these cases it is the *target addresses* that are the same. For example, in PROGA the reference REF1 is a program-counter relative instruction 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. No relocation is necessary for this instruction since the program counter will always contain the actual (not relative) address of the next instruction. We could also think of this process as automatically providing the needed relocation at execution time through the target address calculation. In PROGB, on the other hand, reference REF1 is an extended format instruction that contains a direct (actual) address. This address, after linking, is 4040—the same as the target address for the same reference in PROGA.

You should work through the details of the other references to see that the target addresses (for REF2 and REF3) or the data values (for REF5 through REF8) are the same in each of the three programs. You do not need to worry about how these calculations are actually performed by the loader because the algorithm and data structures for doing this are discussed in the next section. It is important, however, that you *understand* the calculations to be performed, and that you are able to carry out the computations by hand (following the instructions that are contained in the object programs).

3.2.3 Algorithm and Data Structures for a Linking Loader

Now we are ready to present an algorithm for a linking (and relocating) loader. We use Modification records for relocation so that the linking and relocation functions are performed using the same mechanism. As mentioned previously, this type of loader is often found on machines (like SIC/XE) whose relative addressing makes relocation unnecessary for most instructions.

The algorithm for a linking loader is considerably more complicated than the absolute loader algorithm discussed in Section 3.1. The input to such a loader consists of a set of object programs (i.e., 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 (that is,

until the later control section is read). Thus a linking loader usually makes two passes over its input, just as an assembler does. In terms of general function, the two passes of a linking loader are quite similar to the two passes of an assembler: Pass 1 assigns addresses to all external symbols, and Pass 2 performs the actual loading, relocation, and linking.

The main data structure needed for our linking loader is an external symbol table ESTAB. This table, which is analogous to SYMTAB in our assembler algorithm, is used to store the name and address of each external symbol in the set of control sections being loaded. The table also often indicates in which control section the symbol is defined. A hashed organization is typically used for this table. Two other important variables are PROGADDR (program load address) and CSADDR (control section address). PROGADDR is the beginning address in memory where the linked program is to be loaded. Its value is supplied to the loader by the operating system. (In Chapter 6 we discuss how PROGADDR might be generated within the operating system.) CSADDR contains the starting address assigned to the control section currently being scanned by the loader. This value is added to all relative addresses within the control section to convert them to actual addresses.

The algorithm itself is presented in Fig. 3.13. As we discuss this algorithm, you may find it useful to refer to the example of loading and linking in the preceding section (Figs. 3.11 and 3.12).

During the first pass [Fig. 3.13(a)], the loader is concerned only with Header and Define record types in the control sections. 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. 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. 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 was saved from the Header record) is added to CSADDR. This calculation gives the starting address for the next control section in sequence.

At the end of Pass 1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each. Many loaders include as an option the ability to print a *load map* that shows these symbols and their addresses. This information is often useful in program debugging. For the example in Figs. 3.11 and 3.12, such a load map might look like the following. This is essentially the same information contained in ESTAB 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	

Pass 1:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR {for first control section}
while not end of input do
   begin
      read next input record {Header record for control section}
      set CSLTH to control section length
      search ESTAB for control section name
      if found then
         set error flag {duplicate external symbol}
      else
         enter control section name into ESTAB with value CSADDR
      while record type ≠ 'E' do
         begin
            read next input record
            if record type = 'D' then
               for each symbol in the record do
                  begin
                     search ESTAB for symbol name
                     if found then
                        set error flag (duplicate external symbol)
                     else
                        enter symbol into ESTAB with value
                           (CSADDR + indicated address)
                  end {for}
         end {while ≠ 'E'}
      add CSLTH to CSADDR {starting address for next control section}
   end {while not EOF}
end {Pass 1}
```

Figure 3.13(a) Algorithm for Pass 1 of a linking loader.

Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
   begin
      read next input record {Header record}
      set CSLTH to control section length
      while record type ≠ 'E' do
         begin
            read next input record
            if record type = 'T' then
               begin
                  {if object code is in character form, convert
                     into internal representation}
                  move object code from record to location
                     (CSADDR + specified address)
               end {if 'T'}
            else if record type = 'M' then
               begin
                  search ESTAB for modifying symbol name
                  if found then
                     add or subtract symbol value at location
                        (CSADDR + specified address)
                  else
                     set error flag (undefined external symbol)
               end {if 'M'}
         end {while ≠ 'E'}
      if an address is specified {in End record} then
         set EXECADDR to (CSADDR + specified address)
      add CSLTH to CSADDR
   end {while not EOF}
jump to location given by EXECADDR {to start execution of loaded program}
end {Pass 2}
```

Figure 3.13(b) Algorithm for Pass 2 of a linking loader.

Pass 2 of our loader [Fig. 3.13(b)] performs the actual loading, relocation, and linking of the program. CSADDR is used in the same way 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). 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.

The last step performed by the loader is usually the transferring of control to the loaded program to begin execution. (On some systems, the address

where execution is to begin is simply passed back to the operating system. The user must then 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. Our loader takes this as the transfer point to begin execution. If more than one control section specifies a transfer address, the loader arbitrarily uses the last one encountered. If no control section contains a transfer address, the loader uses the beginning of the linked program (i.e., PROGADDR) as the transfer point. This convention is typical of those found in most linking loaders. Normally, a 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. (See Fig. 2.17 for an example of this.)

You should apply this algorithm (by hand) to load and link the object programs in Fig. 3.11. If PROGADDR is taken to be 4000, the result should be the same as that shown in Fig. 3.12.

This algorithm can be made more efficient if a slight change is made in the object program format. This modification involves assigning a *reference number* to each external symbol referred to in a control section. This reference number is used (instead of the symbol name) in Modification records.

Suppose we always assign the reference number 01 to the control section name. The other external reference symbols may be assigned numbers as part of the Refer record for the control section. Figure 3.14 shows the object

```
HPROGA 000000000063
DLISTA 00004CENDA  000054
R02LISTB 03ENDB  04LISTC 05ENDC
:
T0000200A03201D771000040500 14
:
T000054 0E 000014 FFFFF6 00003E 000014 FFFFC0
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 3.14 Object programs corresponding to Fig. 3.10 using reference numbers for code modification. (Reference numbers are underlined for easier reading.)

```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
R02LISTA 03ENDA  04LISTC 05ENDC
.
.
T0000360B03100000772027 05100000
.
.
T000070 0F000000FFFFF6FFFFFEFFFFF0000060
M00003705+02
M00003E05+03
M00003E05-02
M00007006+03
M00007006-02
M00007006+04
M00007306+05
M00007306-04
M00007606+05
M00007606-04
M00007606+02
M00007906+03
M00007906-02
M00007C06+01
M00007C06-02
E


HPROGC 000000000051
DLISTC 000030ENDC  000042
R02LISTA 03ENDA  04LISTB 05ENDB
.
.
T0000180C03100000771000040 5100000
.
.
T000042 0F000030000008000011000000000000
M00001905+02
M00001D05+04
M00002105+03
M00002105-02
M00004206+03
M00004206-02
M00004206+01
M00004806+02
M00004B06+03
M00004B06-02
M00004B06-05
M00004B06+04
M00004E06+04
M00004E06-02
E
```

Figure 3.14 (*cont'd*)

programs from Fig. 3.11 with this change. The reference numbers are underlined in the Refer and Modification records for easier reading. The common use of a technique such as this is one reason we included Refer records in our object programs. You may have noticed that these records were not used in the algorithm of Fig. 3.13.

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 values for code modification can then be obtained by simply indexing into an array of these values. You are encouraged to develop an algorithm that includes this technique, together with any additional data structures you may require.

3.3 MACHINE-INDEPENDENT LOADER FEATURES

In this section we discuss some loader features that are not directly related to machine architecture and design. Loading and linking are often thought of as operating system service functions. The programmer's connection with such services is not as direct as it is with, for example, the assembler during program development. Therefore, most loaders include fewer different features (and less varied capabilities) than are found in a typical assembler.

Section 3.3.1 discusses the use of an automatic library search process for handling external references. This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded. The routines are automatically retrieved from a library as they are needed during linking.

Section 3.3.2 presents some common options that can be selected at the time of loading and linking. These include such capabilities as specifying alternative sources of input, changing or deleting external references, and controlling the automatic processing of external references.

3.3.1 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. This feature allows the programmer to use subroutines from one or more libraries (for example, mathematical or statistical routines) almost as if they were a part of the programming language. 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 subroutine names as external references in the source program. On some systems, this feature is referred to as *automatic library call.* We use the term

library search to avoid confusion with the call feature found in most programming languages.

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 symbols from each Refer record into the symbol table (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 Pass 1, the symbols in ESTAB that remain undefined represent *unresolved* external references. The loader searches the library or libraries specified for routines that contain the definitions of these symbols, and processes the subroutines found by this search exactly as if they had been part of the primary input stream.

Note that the subroutines fetched from a library in this way may themselves contain external references. It is therefore necessary to repeat the library search process until all references are resolved (or until no further resolution can be made). If unresolved external references remain after the library search is completed, these must be treated as errors.

The process just described allows the programmer 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 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 Pass 1 of the loader, SQRT would already be defined, so it would not be included in any library search that might be necessary.

The libraries to be searched by the loader ordinarily contain assembled or compiled versions of the subroutines (that is, object programs). It is possible to search these libraries by scanning the Define records for all of the object programs on the library, but this might be quite inefficient. 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 to be callable by more than one name (using different entry points), both names are entered into the directory. The object program itself, of course, is only stored 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.

The process of library search has been discussed as the resolution of a call to a subroutine. Obviously the same technique applies equally well to the resolution of external references to data items.

3.3.2 Loader Options

Many loaders allow the user to specify options that modify the standard processing described in Section 3.2. In this section we discuss some typical loader options and give examples of their use. 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 the assembler or compiler retains these commands as a part of the object program.

We discuss loader options in this section as though they were specified using a command language, but there are other possibilities. 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 to the loader when it is invoked. The implementation of such options is, of course, the same regardless of the means used to select them.

One typical loader option allows the selection of alternative sources of input. For example, 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.

Other commands allow the user to delete external symbols or entire control sections. It may also be possible to change external references within the programs being loaded and linked. For example, the command

```
DELETE  csect-name
```

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

```
CHANGE  name1,name2
```

might cause the external symbol *name1* to be changed to *name2* wherever it appears in the object programs. An illustration of the use of such commands is given in the following example.

Consider the source program in Fig. 2.15 and the corresponding object program in Fig. 2.17. There is a main program (COPY) that uses two subprograms (RDREC and WRREC); each of these is a separate control section. If RDREC and WRREC are designed only for use with COPY, it is likely that the three control sections will be assembled at the same time. This means that the three control sections of the object program will appear in the same file (or as part of the same library member).

Suppose now 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. It would probably be desirable to change the source program of COPY to use these utility routines. As a temporary measure, however, a sequence of loader commands could be used to make this change without reassembling the program. This might be done, for example, to test the utility routines before the final conversion is made.

Suppose that a file containing the object programs in Fig. 2.17 is the primary loader input with the loader commands

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

These commands would direct the loader to include control sections READ and WRITE from the library UTLIB, and 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. The result would be exactly the same as if the source program for COPY had been changed to use READ and WRITE. You are encouraged to think for yourself about how the loader might handle such commands to perform the specified processing.

Another common loader option involves the automatic inclusion of library routines to satisfy external references (as described in the preceding section). Most loaders allow the user to specify alternative libraries to be searched, using a statement such as

```
LIBRARY   MYLIB
```

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

Loaders that perform automatic library search to satisfy external references often allow the 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 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  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.

It is also 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. It is often desirable to postpone the resolution of external references in such a case. In Section 3.4.1 we discuss linkage editors that perform this sort of function.

Another common option involves output from the loader. In Section 3.2.3 we gave an example of a load map that might be generated during the loading process. Through control statements the user can often 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 external symbol.

Loaders often include a variety of other options. One such option is the ability to specify the location at which execution is to begin (overriding any information given in the object programs). 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).

3.4 LOADER DESIGN OPTIONS

In this section we discuss some common alternatives for organizing the loading functions, including relocation and linking. Linking loaders, as described in

Section 3.2.3, perform all linking and relocation at load time. We discuss two alternatives to this: linkage editors, which perform linking prior to load time, and dynamic linking, in which the linking function is performed at execution time.

Section 3.4.1 discusses linkage editors, which are found on many computing systems 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 or library 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.

Section 3.4.2 introduces dynamic linking, which uses facilities of the operating system to load and link subprograms at the time they are first called. By delaying the linking process in this way, additional flexibility can be achieved. However, this approach usually involves more overhead than does a linking loader.

In Section 3.4.3 we discuss bootstrap loaders. Such loaders can be used to run stand-alone programs independent of the operating system or the system loader. They can also be used to load the operating system or the loader itself into memory.

3.4.1 Linkage Editors

The essential difference between a linkage editor and a linking loader is illustrated in Fig. 3.15. 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 memory for execution. A *linkage editor,* on the other hand, produces a linked version of the program (often called a *load module* or an *executable image*), which is written to a file or library for later execution.

When the user is ready to run the linked program, a simple relocating loader can be used to load the program into memory. The only object code modification necessary is the addition of an actual load address to relative values within the program. The 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. 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

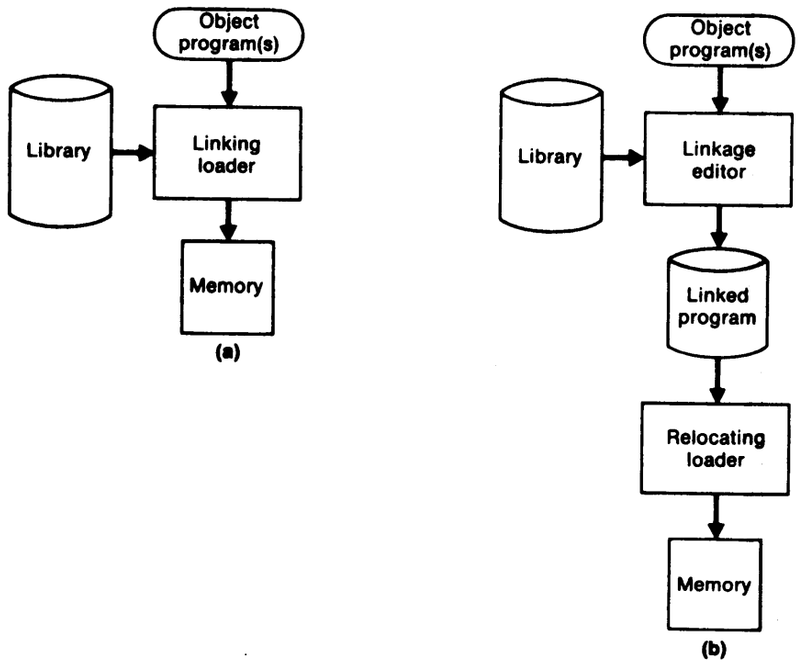


Figure 3.15 Processing of an object program using (a) linking loader and (b) linkage editor.

once (when the program is link edited). In contrast, a linking loader searches libraries and resolves external references every time the program is executed.

Sometimes, however, a program is reassembled for nearly every execution. This situation might occur in a program development and testing environment (for example, student programs). It also occurs when a program is used so infrequently that it is not worthwhile to store the assembled version in a library. In such cases 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 a bit mask. Even though all linking has been performed, information concerning external references is often retained in the linked program. This allows subsequent relinking 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 be 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 an object program for execution. Consider, for example, a program (PLANNER) that uses a large number of subroutines. Suppose that one subroutine (PROJECT) used by the program is changed to correct an error or to improve efficiency. After the 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 to accomplish this. The command language is similar to that discussed in Section 3.3.2.

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

Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. This can be useful when dealing with subroutine libraries that support high-level programming languages. In a typical implementation of FORTRAN, for example, 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 deblock records, and to encode and decode data items according to format specifications. There are a large number of cross-references 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-references 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 like the following:

```
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 of 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-references between subroutines resolved, these linkages would not be reprocessed when each user's program is linked. The result would be a much more efficient linkage editing operation for each program and a considerable overall savings for the system.

Linkage editors often allow the user to specify that external references are not to be resolved by automatic library search. Suppose, for example, that 100 FORTRAN programs using the I/O routines described above were to be stored on a library. If all external references were resolved, this would mean that a total of 100 copies of FTNIO would be stored. If library space were an important resource, this might be highly undesirable. Using commands like those discussed in Section 3.3.2, the user could specify that no library search be performed during linkage editing. Thus only the 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 require slightly more overhead; however, it would result in a large savings inlibrary space.

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

3.4.2 Dynamic Linking

Linkage editors perform linking operations before the program is loaded for execution. Linking loaders perform these same operations at load time. In this section we discuss 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 example, 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 this one copy, instead of linking a separate copy into each object program.

In an object-oriented system, dynamic linking is often used for references to software objects. This allows the implementation of the object and its methods 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, as discussed previously. (See Section 8.4 for an introduction to object-oriented programming and design.)

Dynamic linking also offers some other advantages over the other types of linking we have discussed. Suppose, for example, that 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 any 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, for example, with a program that allows its user to interactively call any of the subroutines of a large mathematical and statistical library. Input data could be supplied by the user, and results could be displayed at the terminal. In this case, all of the library subroutines could potentially be needed, but only a few will actually be used in any one execution. Dynamic linking avoids the necessity of loading the entire library for each execution. As a matter of fact, dynamic linking may make it unnecessary for the program even to know the possible set of subroutines that might be used. The subroutine name might simply be treated as another input item.

There are a number of different mechanisms that can be used to accomplish the actual loading and linking of a called subroutine. Figure 3.16 illustrates a method 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 a part of the loader that is kept in memory during execution of the program.

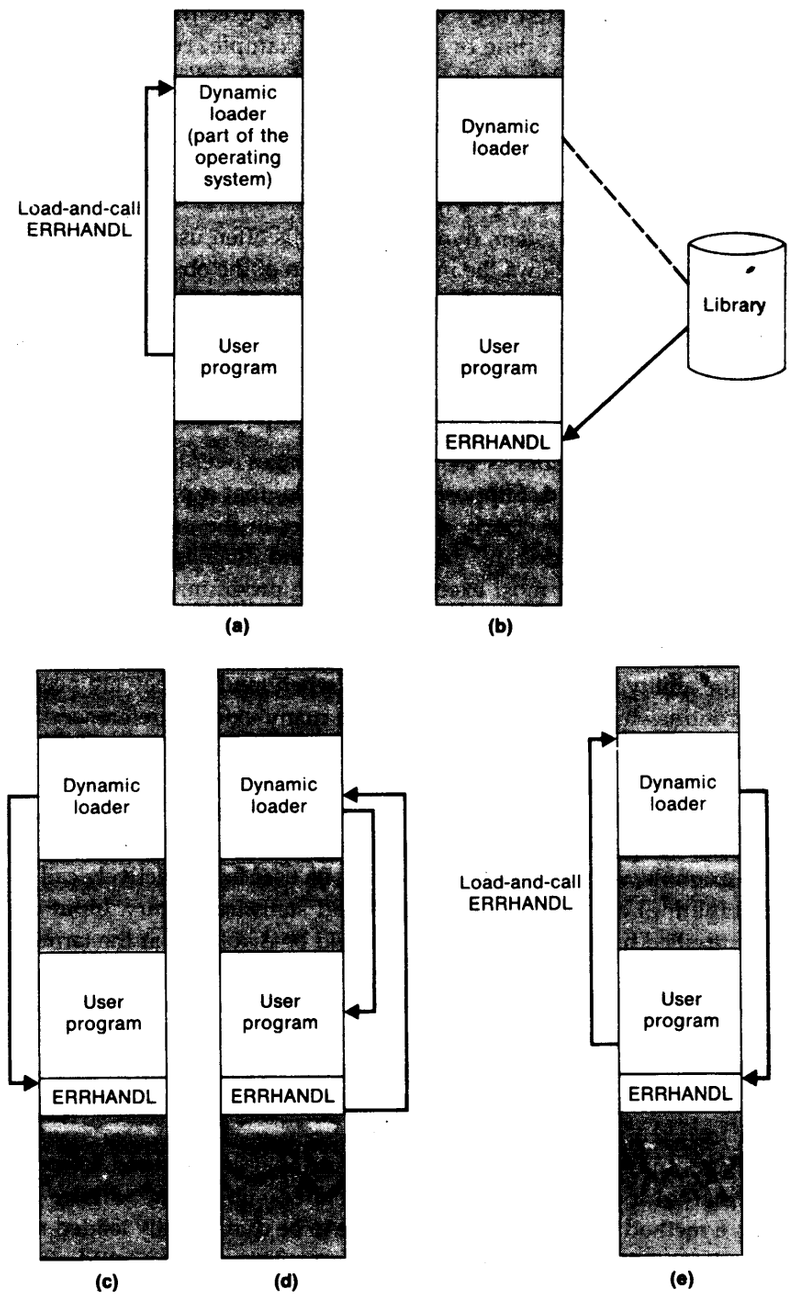


Figure 3.16 Loading and calling of a subroutine using dynamic linking.

Instead of executing a JSUB instruction that refers to an external symbol, the program makes a load-and-call service request to the operating system. The parameter of this request is the symbolic name of the routine to be called. [See Fig. 3.16(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 as shown in Fig. 3.16(b). Control is then passed from the operating system to the routine being called [Fig. 3.16(c)].

When the called subroutine completes its processing, it returns to its caller (that is, to the operating system routine that handles the load-and-call service request). The operating system then returns control to the program that issued the request. This process is illustrated in Fig. 3.16(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, this is not always done immediately. Sometimes 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 a subroutine is still in memory, a second call to it may not require another load operation. Control may simply be passed from the dynamic loader to the called routine, as shown in Fig. 3.16(e).

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, as we have discussed. It also requires more overhead since the operating system must intervene in the calling process. In later chapters we see other examples of delayed binding. In those examples, too, delayed binding gives more capabilities at a higher cost.

3.4.3 Bootstrap Loaders

In our discussions of loaders we have neglected to answer one important question: How is the loader itself loaded into memory? Of course, we could say that the operating system loads the loader; however, we are then left with the same question with respect to the operating system. More generally, the question is this: Given an idle computer with no program in memory, how do we get things 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, this program will be the 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 inconvenient and error-prone to be a good solution to the problem.

On some computers, an absolute loader program is permanently resident in a read-only memory (ROM). When some hardware signal occurs (for example, the 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 machines do not have such read-only storage. In addition, it can be 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 (or a very short ROM program) that reads a fixed-length record from some device into memory at a fixed location. The particular device to be used can often be selected via console switches. After the read operation is complete, control is automatically 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 than can be read in a single record, this first record causes the reading of others, and these in turn can cause the reading of still more records—hence the term *bootstrap*. The first record (or records) is generally referred to as a *bootstrap loader*. (A simple example of such a bootstrap loader was given in Section 3.1.2.) Such a loader is added to the beginning of all object programs that are to be loaded into an empty and idle system. This includes, for example, the operating system itself and all stand-alone programs that are to be run without an operating system.

3.5 IMPLEMENTATION EXAMPLES

In this section we briefly examine linkers and loaders for actual computers. As in our previous discussions, we make no attempt to give a full description of the linkers and loaders used as examples. Instead we concentrate on any particularly interesting or unusual features, and on differences between these implementations and the more general model discussed earlier in this chapter. We also point out areas in which the linker or loader design is related to the assembler design or to the architecture and characteristics of the machine.

The loader and linker examples we discuss are for the Pentium, SPARC, and T3E architectures. You may want to review the descriptions of these architectures in Chapter 1, and the related assembler examples in Section 2.5.

3.5.1 MS-DOS Linker

This section describes some of the features of the Microsoft MS-DOS linker for Pentium and other x86 systems. Further information can be found in Simrin (1991) and Microsoft (1988).

Most MS-DOS compilers and assemblers (including MASM) produce object modules, not executable machine language programs. By convention, these object modules have the file name extension .OBJ. Each object module contains a binary image of the translated instructions and data of the program. It also describes the structure of the program (for example, the grouping of segments and the use of external references in the program).

MS-DOS LINK is a linkage editor that combines one or more object modules to produce a complete executable program. By convention, this executable program has the file name extension .EXE. LINK can also combine the translated programs with other modules from object code libraries, as we discussed previously.

Figure 3.17 illustrates a typical MS-DOS object module. There are also several other possible record types (such as comment records), and there is some flexibility in the order of the records.

The THEADR record specifies the name of the object module. The MODEND record marks the end of the module, and can contain a reference to the entry point of the program. These two records generally correspond to the Header and End records we discussed for SIC/XE.

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

Figure 3.17 MS-DOS object module.

The PUBDEF record contains a list of the external symbols (called public names) that are defined in this object module. The EXTDEF record contains a list of the external symbols that are referred to in this object module. These records are similar in function to the SIC/XE Define and Refer records. Both PUBDEF and EXTDEF can contain information about the data type designated by an external name. These types are defined in the TYPDEF record.

SEGDEF records describe the segments in the object module, including their name, length, and alignment. GRPDEF records specify how these segments are combined into groups. (See Section 2.5.1 for a discussion of the use of segmentation in the MASM assembler.) The LNAMES record contains a list of all the segment and class names used in the program. SEGDEF and GRPDEF records refer to a segment by giving the position of its name in the LNAMES record. (This approach to specifying names is similar to the "reference number" technique described near the end of Section 3.2.3.)

LEDATA records contain translated instructions and data from the source program, similar to the SIC/XE Text record. LIDATA records specify translated instructions and data that occur in a repeating pattern. (See Exercise 2.1.7.)

FIXUPP records are used to resolve external references, and to carry out address modifications that are associated with relocation and grouping of segments within the program. This is similar to the function performed by the SIC/XE Modification records. However, FIXUPP records are substantially more complex, because of the more complicated object program structure. A FIXUPP record must immediately follow the LEDATA or LIDATA record to which it applies.

LINK performs its processing in two passes, following a similar approach to that described in Section 3.2.3. Pass 1 computes a starting address for each segment in the program. In general, segments are placed into the executable program in the same order that the SEGDEF records are processed. However, in some cases segments from different object modules that have the same segment name and class are combined. Segments with the same class, but different names, are concatenated. The starting address initially associated with a segment is updated during Pass 1 as these combinations and concatenations are performed.

Pass 1 constructs a symbol table that associates an address with each segment (using the LNAMES, SEGDEF, and GRPDEF records) and each external symbol (using the EXTDEF and PUBDEF records). If unresolved external symbols remain after all object modules have been processed, LINK searches the specified libraries as described in Section 3.3.1.

During Pass 2, LINK extracts the translated instructions and data from the object modules, and builds an image of the executable program in memory.

It does this because the executable program is organized by segment, not by the order of the object modules. Building a memory image is the most efficient way to handle the rearrangements caused by combining and concatenating segments. If there is not enough memory available to contain the entire executable image, LINK uses a temporary disk file in addition to all of the available memory.

Pass 2 of LINK processes each LEDATA and LIDATA record along with the corresponding FIXUPP record (if there is one). It places the binary data from LEDATA and LIDATA records into the memory image at locations that reflect the segment addresses computed during Pass 1. (Repeated data specified in LIDATA records is expanded at this time.) Relocations within a segment (caused by combining or grouping segments) are performed, and external references are resolved. Relocation operations that involve the starting address of a segment are added to a table of segment fixups. This table is used to perform relocations that reflect the actual segment addresses when the program is loaded for execution.

After the memory image is complete, LINK writes it to the executable (.EXE) file. This file also includes a header that contains the table of segment fixups, information about memory requirements and entry points, and the initial contents for registers CS and SP.

3.5.2 SunOS Linkers

This section describes some of the features of the SunOS linkers for SPARC systems. Further information can be found in Sun Microsystems (1994b).

SunOS actually provides two different linkers, called the *link-editor* and the *run-time linker.* The link-editor is most commonly invoked in the process of compiling a program. It takes one or more object modules produced by assemblers and compilers, and combines them to produce a single output module. This output module may be one of the following types:

1. A *relocatable object module,* suitable for further link-editing.
2. A *static executable,* with all symbolic references bound and ready to run.
3. A *dynamic executable,* in which some symbolic references may need to be bound at run time.
4. A *shared object,* which provides services that can be bound at run time to one or more dynamic executables.

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, such as "executable" and "writeable." (See Section 2.5.2 for a discussion of how sections are defined in an assembler language program.) 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.

The SunOS link-editor begins by reading the object modules (or other files) that are presented to it to process. 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 are performed. The linker normally generates a new symbol table, and a new set of relocation instructions, within the output file. These represent symbols that must be bound at run time, and relocations that must be performed when the program is loaded.

Relocation and linking operations are specified using a set of processor-specific codes. These codes describe the size of the field that is to be modified, and the calculation that must be performed. Thus, the set of codes reflects the instruction formats and addressing modes that are found on a particular machine. For example, there are 24 different relocation codes that are used on SPARC systems. SunOS linker implementations on x86 systems use a different set of 11 codes.

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, as described in Section 3.3.1.

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. However, the shared object is not physically included in the output file. Instead, the link-editor records the dependency on the shared object. The actual inclusion of the shared object is deferred until run time. (This is an example of the dynamic linking approach we discussed in Section 3.4.2. In this case, the use of dynamic linking allows several executing programs to share one copy of a shared object.)

The SunOS run-time linker is used to bind dynamic executables and shared objects at execution time. The linker determines what shared objects

are required by the dynamic executable, and ensures that these objects are included. It also inspects the shared objects to detect and process any additional dependencies on other shared objects.

After it locates and includes the necessary shared objects, the linker performs relocation and linking operations to prepare the program for execution. These operations are specified in the relocation and linking sections of the dynamic executable and shared objects. They bind symbols to the actual memory addresses at which the segments are loaded. Binding of data references is performed before control is passed to the executable program. Binding of procedure calls is normally deferred until the program is in execution. During link-editing, calls to globally defined procedures are converted to references to a procedure linkage table. When a procedure is called for the first time, control is passed via this table to the run-time linker. The linker looks up the actual address of the called procedure and inserts it into the linkage table. Thus subsequent calls will go directly to the called procedure, without intervention by the linker. This process is sometimes referred to as *lazy binding*.

The run-time linker also provides an additional level of flexibility. During execution, a program can dynamically bind to new shared objects by requesting the same services of the linker that we have just described. This feature allows a program to choose between a number of shared objects, depending on the exact services required. It also reduces the amount of overhead required for starting a program. If a shared object is not needed during a particular run, it is not necessary to bind it at all. These advantages are similar to those that we discussed for dynamic linking in Section 3.4.2.

3.5.3 Cray MPP Linker

This section describes some of the features of the MPP linker for Cray T3E systems. Further information can be found in Cray Research (1995b).

As we discussed in Chapter 1, a T3E system contains a large number of processing elements (PEs). Each PE has its own local memory. In addition, any PE can access the memory of all other PEs (this is sometimes referred to as *remote memory*). However, the fastest access time always results from a PE accessing its own local memory.

An application program on a T3E system is normally allocated a *partition* that consists of several PEs. (It is possible to run a program in a partition of one PE, but this does not take advantage of the parallel architecture of the machine.) The work to be done by the program is divided between the PEs in the partition. One common method for doing this is to distribute the elements of an array among the PEs. For example, if a partition consists of

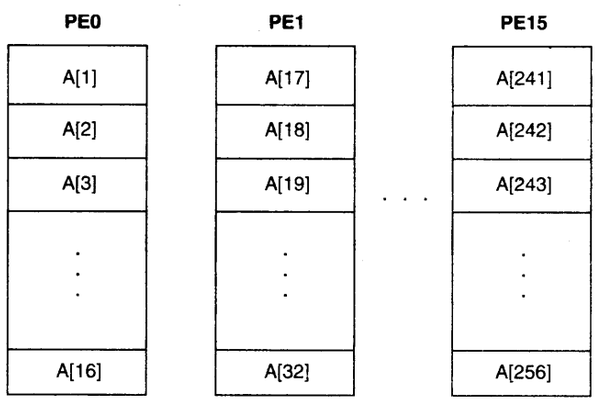


Figure 3.18 Example of data shared between PEs.

16 PEs, the elements of a one-dimensional array might be distributed as shown in Fig. 3.18.

The processing of such an array can also be divided among the PEs. Suppose, for example, that the program contains a loop that processes all 256 array elements. PE0 could execute this loop for subscripts 1 through 16, PE1 could execute the loop for subscripts 17 through 32, and so on. In this way, all of the PEs would share in the array processing, with each PE handling the array elements from its own local memory. Section 6.5.4 discusses some of the operating system functions that are used to support the parallel operation of PEs.

Data that is divided among a number of PEs, as in the example just discussed, is called *shared data.* Data that is not shared in this way is called *private data.* In most cases, private data is replicated on each PE in the partition—that is, each PE has its own copy. It is also possible for a PE to have private data items that exist only in its own local memory.

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 shared data. There are a number of possible arrangements of these items, but the overall situation can be visualized as shown in Fig. 3.19. In this diagram, *shared data-i* indicates the portion of the shared data that is assigned to PEi.

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—for example, executable code, private data, or shared data. The blocks on each list are collected together, an address is assigned to each block, and relocation and linking operations are performed. The linker then

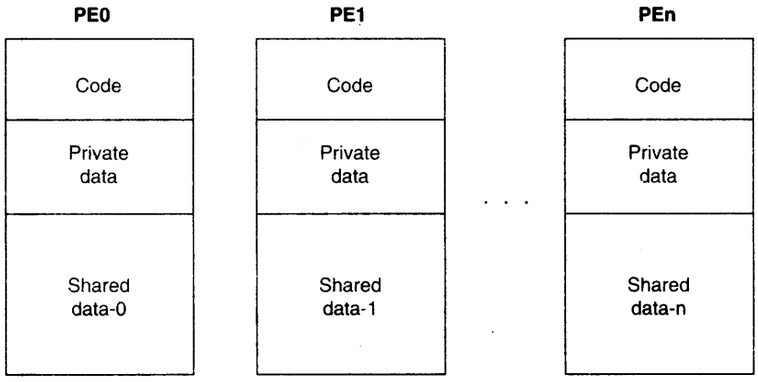


Figure 3.19 T3E program loaded on multiple PEs.

writes an executable file that contains the relocated and linked blocks. This executable file also specifies the number of PEs required and other control information.

Notice that the distribution of shared data depends on the number of PEs in the partition. For example, if the partition in Fig. 3.18 contained only 8 PEs, each PE would receive 32 elements of the shared array. 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. The linker can create an executable file that is targeted for a fixed number of PEs, or one that allows the partition size to be chosen at run time. This latter type is called a *plastic* executable. A plastic executable file must contain a copy of all relocatable object modules, and all linker directives that are needed to produce the final executable. Thus, a plastic executable is often considerably larger than one targeted for a fixed number of PEs.

EXERCISES

Section 3.1

1. Define a binary object program format for SIC and write an absolute loader (in SIC assembler language) to load programs in this format.
2. Describe a method for performing the packing required when loading an object program such as that in Fig. 3.1(a), which uses character

representation of assembled code. How could you implement this method in SIC assembler language?

3. What would be the advantages and disadvantages of writing a loader using a high-level programming language? What problems might you encounter, and how might these be solved?

3.2 Section Exercises

1. Modify the algorithm given in Fig. 3.13 to use the bit-mask approach to relocation. Linking will still be performed using Modification records.

2. Suppose that a computer primarily uses direct addressing, but has several different instruction formats. What problems does this create for the relocation-bit approach to program relocation? How might these problems be solved?

3. Apply the algorithm described in Fig. 3.13 to link and load the object programs in Fig. 3.11. Compare your results with those shown in Fig. 3.12.

4. Assume that PROGA, PROGB, and PROGC are the same as in Fig. 3.10. Show how the object programs would change (including Text and Modification records) if the following statements were added to each program:

```
REF9    WORD    LISTC
REF10   WORD    LISTB-3
REF11   WORD    LISTA+LISTB
REF12   WORD    ENDC-LISTC-100
REF13   WORD    LISTA-LISTB-ENDA+ENDB
```

5. Apply the algorithm described in Fig. 3.13 to link and load the revised object programs you generated in Exercise 4.

6. Using the methods outlined in Chapter 8, develop a modular design for a relocating and linking loader.

7. Extend the algorithm in Fig. 3.13 to include the detection of improper external reference expressions as suggested in the text. (See Section 2.3.5 for the set of rules to be applied.) What problems arise in performing this kind of error checking?

8. Modify the algorithm in Fig. 3.13 to use the reference-number technique for code modification that is described in Section 3.2.3.

9. Suppose that you are implementing an assembler and loader and want to allow *absolute-valued* external symbols. For example, one control section might contain the statements

```
          EXTDEF   MAXLEN
          .
          .
          .
MAXLEN    EQU      4096
```

   and other control sections could refer to the value of MAXLEN as an external symbol. Describe a way of implementing this new feature, including any needed changes in the loader logic and object program format.

10. Suppose that you have been given the task of writing an "unloader"—that is, a piece of software that can take the image of a program that has been loaded and write out an object program that could later be loaded and executed. The computer system uses a relocating loader, so the object program you produce must be capable of being loaded at a location in memory that is different from where your unloader took it. What problems do you see that would prevent you from accomplishing this task?

11. Suppose that you are given two images of a program as it would appear after loading at two *different* locations in memory. Assume that the images represent the program *after* it is loaded and relocated, but *before* any of the program's instructions are actually executed. Describe how this information could be used to accomplish the "unloading" task mentioned in Exercise 10.

12. Some loaders have used an indirect linking scheme. To use such a technique with SIC/XE, the assembler would generate a list of pointer words from the EXTREF directive (one pointer word for each external reference symbol). Modification records would direct the loader to insert the external symbol addresses into the corresponding words in the pointer list. External references would then be accomplished with indirect addressing using these pointers. Thus, for example, an instruction like

```
          LDA      XYZ
```

   (where XYZ is an external reference) would be assembled as if it were

```
          LDA      @PXYZ
```

where PXYZ is the pointer word containing the address of XYZ. What would be the advantages and disadvantages of using such a method?

13. Suggest a design for a *one-pass* linking loader. What restrictions (if any) would be required? What would be the advantages and disadvantages of such a one-pass loader?

14. Some programming languages allow data items to be placed in common areas. There may be more than one common area (with different names) in a source program. We may think of each common area as being a separate control section in the object program.

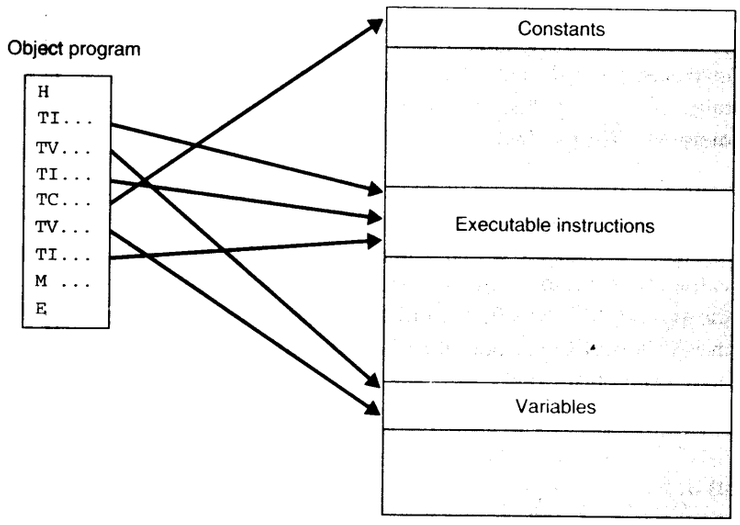
    When object programs are linked and loaded, all of the common areas with the same name are assigned the same starting address in memory. (These common areas may be of different lengths in the different programs declaring them.) This assignment of memory establishes an equivalence between the variables that were declared in common by the different programs. Any data value stored into a common area by one program is thus available to the others.

    How might the loader handle such common areas? (Suggest modifications to the algorithm of Fig. 3.13 that will perform the necessary processing.)

15. Suppose that you have a one-pass assembler that produces object code directly in memory, as described in Section 2.4. This assembler was designed to assemble and run only one control section. Now you want to change it so that it can assemble and run a program that consists of several different control sections (as illustrated in Fig. 2.15).

    Describe the changes you would make to implement this new capability. Your modified assembler should still run in one pass, and should still produce object code in memory, without using any other files.

16. Suppose that a relocatable SIC/XE program is to be loaded in three different parts. One part contains the assembled instructions of the program (LDA, JSUB, etc.). Another part contains the data variables used in the program (which are defined by RESW, RESB, BYTE, and WORD). The third part contains data constants (which are defined by a new assembler directive named CONST).



In the object program, the assembled instructions are contained in type TI records, the variables in type TV records, and the constants in type TC records. (These new record types take the place of the normal Text records in the object program.) The three parts of the object program will be loaded into separate areas of memory, as illustrated above. The starting address for each of the three segments of the program will be supplied to the loader at the time the program is being loaded.

Describe how the assembler could separate the object program into TI, TV, and TC records as described above. Describe how the loader would use the information in these records in loading the program.

17. Consider an extended version of SIC/XE that has a new register R. The contents of R cannot be accessed or changed by the user program. When a program is loaded, however, the loader sets register R so that it contains the starting address of the program. For simplicity, assume that this version of SIC has no program-counter or base relative addressing—thus, all instructions that refer to memory must use Format 4.

Each time the program refers to an address in memory, the contents of register R are automatically added into the target address calculation.

Suppose, for example, that an assembled instruction specifies an address of 800 (hexadecimal). If R contains 5000, executing this instruction would actually refer to memory address 5800. If R contains 8000, executing the same instruction would actually refer to memory address 8800.

Consider the control sections shown in Fig. 3.10. Assume that these control sections are being loaded and linked at the addresses shown in Fig. 3.12; thus the loader will set register R to the value 4000. What value should appear in the External Symbol Table of the loader for the symbol LISTB? What should the instruction labeled REF2 in control section PROGC look like after all loading and linking operations have been performed?

Section 3.3

1. Modify the algorithm in Fig. 3.13 to include automatic library search to resolve external references. You may assume that the details of library access are handled by operating system service routines.
2. Modify the algorithm in Fig. 3.13 to implement CHANGE, DELETE, and INCLUDE directives as described in Section 3.3.2. If you need to place any restrictions on the use of these commands, be sure to state what they are.
3. Suppose that the loader is to produce a listing that shows not only the addresses assigned to external symbols, but also the cross-references between control sections in the program being loaded. What information might be useful in such a listing? Briefly describe how you might implement this feature and include a description of any data structures needed.

Section 3.4

1. Define a module format suitable for representing linked programs produced by a linkage editor. Assume that the linked program is not to be reprocessed by the linkage editor. Describe an algorithm for a relocating loader that would be suitable for the loading of linked programs in this format.

2. Define a module format suitable for representing linked programs produced by a linkage editor. This format should allow for the loading of the linked program by a one-pass relocating loader, as in Exercise 1. However, it should also allow for the linked program to be reprocessed by the linkage editor. Describe how your format allows for both one-pass loading and relinking.

3. Consider the following possibilities for the storage, linking, and execution of a user's program:

   a. Store the source program only; reassemble the program and use a linking loader each time it is to be executed.

   b. Store the source and object versions of the program; use a linking loader each time the program is to be executed.

   c. Store the source program and the linked version with external references to library subroutines left unresolved. Use a linking loader each time the program is to be executed.

   d. Store the source program and the linked version with all external references resolved. Use a relocating loader each time the program is to be executed.

   e. Store the source program and a linked version that has all external references resolved and all relocation performed. Use an absolute loader each time the program is to be executed.

   Under what circumstances might each of these approaches be appropriate? Assume that no changes are required in the source program from one execution to the next.

4. Dynamic linking, as described in Section 3.4.2, works for transfers of control only. How could the implementation be extended so that data references could also cause dynamic loading to occur?

5. Suppose that routines that are brought into memory by dynamic loading need not be removed until the termination of the main program. Suggest a way to improve the efficiency of dynamic linking by making it unnecessary for the operating system to be involved in the transfer of control after the routine is loaded.

6. Suppose that it may be necessary to remove from memory routines that were dynamically loaded (to reuse the space). Will the method that you suggested in Exercise 5 still work? What problems arise, and how might they be solved?

7. What kinds of errors might occur during bootstrap loading? What action should the bootstrap loader take for such errors? Modify the SIC/XE bootstrap loader shown in Fig. 3.3 to include such error checking.
8. Compose merits and demerits of all the loading schemes.

Section 3.5

1. Consider the description of the VAX architecture in Section 1.4.1. What characteristics would you expect to find in a VAX linker and loader?
2. Consider the description of the PowerPC architecture in Section 1.5.2, and the description of the PowerPC assembler in Section 2.5.3. What characteristics would you expect to find in a PowerPC linker and loader?

Chapter 4

Macro Processors

In this chapter we study the design and implementation of macro processors. A *macro instruction* (often abbreviated to *macro*) is simply a notational convenience for the programmer. A macro represents a commonly used group of statements in the source programming language. The macro processor replaces each macro instruction with the corresponding group of source language statements. This is called *expanding* the macros. Thus macro instructions allow the programmer to write a shorthand version of a program, and leave the mechanical details to be handled by the macro processor.

For example, suppose that it is necessary to save the contents of all registers before calling a subprogram. On SIC/XE, this would require a sequence of seven instructions (STA, STB, etc.). Using a macro instruction, the programmer could simply write one statement like SAVEREGS. This macro instruction would be expanded into the seven assembler language instructions needed to save the register contents. A similar macro instruction (perhaps named LOADREGS) could be used to reload the register contents after returning from the subprogram.

The functions of a macro processor essentially involve the substitution of one group of characters or lines for another. Except in a few specialized cases, the macro processor performs no analysis of the text it handles. The design and capabilities of a macro processor may be influenced by the *form* of the programming language statements involved. However, the *meaning* of these statements, and their translation into machine language, are of no concern whatsoever during macro expansion. This means that the design of a macro processor is not directly related to the architecture of the computer on which it is to run.

The most common use of macro processors is in assembler language programming. We use SIC assembler language examples to illustrate most of the concepts being discussed. However, macro processors can also be used with high-level programming languages, operating system command languages, etc. In addition, there are general-purpose macro processors that are not tied to any particular language. In the later sections of this chapter, we briefly discuss these more general uses of macros.

Section 4.1 introduces the basic concepts of macro processing, including macro definition and expansion. We also present an algorithm for a simple macro processor. Section 4.2 discusses extended features that are commonly found in macro processors. These features include the generation of unique labels within macro expansions, conditional macro expansion, and the use of keyword parameters in macros. All these features are machine-independent. Because the macro processor is not directly related to machine architecture, this chapter contains no section on machine-dependent features.

Section 4.3 describes some macro processor design options. One of these options (recursive macro expansion) involves the internal structure of the macro processor itself. The other options are concerned with how the macro processor is related to other pieces of system software such as assemblers or compilers.

Finally, Section 4.4 briefly presents three examples of actual macro processors. One of these is a macro processor designed for use by assembler language programmers. Another is intended to be used with a high-level programming language. The third is a general-purpose macro processor, which is not tied to any particular language. Additional examples may be found in the references cited throughout this chapter.

4.1 BASIC MACRO PROCESSOR FUNCTIONS

In this section we examine the fundamental functions that are common to all macro processors. Section 4.1.1 discusses the processes of macro definition, invocation, and expansion with substitution of parameters. These functions are illustrated with examples using the SIC/XE assembler language. Section 4.1.2 presents a one-pass algorithm for a simple macro processor together with a description of the data structures needed for macro processing. Later sections in this chapter discuss extensions to the basic capabilities introduced in this section.

4.1.1 Macro Definition and Expansion

Figure 4.1 shows an example of a SIC/XE program using macro instructions. This program has the same functions and logic as the sample program in Fig. 2.5; however, the numbering scheme used for the source statements has been changed.

This program defines and uses two macro instructions, RDBUFF and WRBUFF. The functions and logic of the RDBUFF macro are similar to those of the RDREC subroutine in Fig. 2.5; likewise, the WRBUFF macro is similar

Line **Source statement**

```
 5     COPY      START   0               COPY FILE FROM INPUT TO OUTPUT
10     RDBUFF    MACRO   &INDEV,&BUFADR,&RECLTH
15     .
20     .        MACRO TO READ RECORD INTO BUFFER
25     .
30               CLEAR   X               CLEAR LOOP COUNTER
35               CLEAR   A
40               CLEAR   S
45               +LDT    #4096           SET MAXIMUM RECORD LENGTH
50               TD      =X'&INDEV'      TEST INPUT DEVICE
55               JEQ     *-3             LOOP UNTIL READY
60               RD      =X'&INDEV'      READ CHARACTER INTO REG A
65               COMPR   A,S             TEST FOR END OF RECORD
70               JEQ     *+11            EXIT LOOP IF EOR
75               STCH    &BUFADR,X       STORE CHARACTER IN BUFFER
80               TIXR    T               LOOP UNLESS MAXIMUM LENGTH
85               JLT     *-19              HAS BEEN REACHED
90               STX     &RECLTH         SAVE RECORD LENGTH
95               MEND
100    WRBUFF    MACRO   &OUTDEV,&BUFADR,&RECLTH
105    .
110    .        MACRO TO WRITE RECORD FROM BUFFER
115    .
120              CLEAR   X               CLEAR LOOP COUNTER
125              LDT     &RECLTH
130              LDCH    &BUFADR,X       GET CHARACTER FROM BUFFER
135              TD      =X'&OUTDEV'     TEST OUTPUT DEVICE
140              JEQ     *-3             LOOP UNTIL READY
145              WD      =X'&OUTDEV'     WRITE CHARACTER
150              TIXR    T               LOOP UNTIL ALL CHARACTERS
155              JLT     *-14              HAVE BEEN WRITTEN
160              MEND
165    .
170    .        MAIN PROGRAM
175    .
180    FIRST     STL     RETADR          SAVE RETURN ADDRESS
190    CLOOP     RDBUFF  F1,BUFFER,LENGTH READ RECORD INTO BUFFER
195              LDA     LENGTH          TEST FOR END OF FILE
200              COMP    #0
205              JEQ     ENDFIL          EXIT IF EOF FOUND
210              WRBUFF  05,BUFFER,LENGTH WRITE OUTPUT RECORD
215              J       CLOOP           LOOP
220    ENDFIL    WRBUFF  05,EOF,THREE    INSERT EOF MARKER
225              J       @RETADR
230    EOF       BYTE    C'EOF'
235    THREE     WORD    3
240    RETADR    RESW    1
245    LENGTH    RESW    1               LENGTH OF RECORD
250    BUFFER    RESB    4096            4096-BYTE BUFFER AREA
255              END     FIRSTZ
```

Figure 4.1 Use of macros in a SIC/XE program.

to the WRREC subroutine. The definitions of these macro instructions appear in the source program following the START statement.

Two new assembler directives (MACRO and MEND) are used in macro definitions. The first MACRO statement (line 10) identifies the beginning of a macro definition. The symbol in the label field (RDBUFF) is the name of the macro, and the entries in the operand field identify the *parameters* of the macro instruction. In our macro language, each parameter begins with the character &, which facilitates the substitution of parameters during macro expansion. The macro name and parameters define a pattern or *prototype* for the macro instructions used by the programmer. Following the MACRO directive are the statements that make up the *body* of the macro definition (lines 15 through 90). These are the statements that will be generated as the expansion of the macro. The MEND assembler directive (line 95) marks the end of the macro definition. The definition of the WRBUFF macro (lines 100 through 160) follows a similar pattern.

The main program itself begins on line 180. The statement on line 190 is a *macro invocation* statement that gives the name of the macro instruction being invoked and the *arguments* to be used in expanding the macro. (A macro invocation statement is often referred to as a *macro call*. To avoid confusion with the call statements used for procedures and subroutines, we prefer to use the term *invocation*. As we shall see, the processes of macro invocation and subroutine call are quite different.) You should compare the logic of the main program in Fig. 4.1 with that of the main program in Fig. 2.5, remembering the similarities in function between RDBUFF and RDREC and between WRBUFF and WRREC.

The program in Fig. 4.1 could be supplied as input to a macro processor. Figure 4.2 shows the output that would be generated. The macro instruction definitions have been deleted since they are no longer needed after the macros are expanded. Each macro invocation statement has been expanded into the statements that form the body of the macro, with the arguments from the macro invocation substituted for the parameters in the macro prototype. The arguments and parameters are associated with one another according to their positions. The first argument in the macro invocation corresponds to the first parameter in the macro prototype, and so on. In expanding the macro invocation on line 190, for example, the argument F1 is substituted for the parameter &INDEV wherever it occurs in the body of the macro. Similarly, BUFFER is substituted for &BUFADR, and LENGTH is substituted for &RECLTH.

Lines 190a through 190m show the complete expansion of the macro invocation on line 190. The comment lines within the macro body have been deleted, but comments on individual statements have been retained. Note that the macro invocation statement itself has been included as a comment line. This serves as documentation of the statement written by the programmer.

Line	Source statement			
5	COPY	START	0	COPY FILE FROM INPUT TO OUTPUT
180	FIRST	STL	RETADR	SAVE RETURN ADDRESS
190	.CLOOP	RDBUFF	F1,BUFFER,LENGTH	READ RECORD INTO BUFFER
190a	CLOOP	CLEAR	X	CLEAR LOOP COUNTER
190b		CLEAR	A	
190c		CLEAR	S	
190d		+LDT	#4096	SET MAXIMUM RECORD LENGTH
190e		TD	=X'F1'	TEST INPUT DEVICE
190f		JEQ	*-3	LOOP UNTIL READY
190g		RD	=X'F1'	READ CHARACTER INTO REG A
190h		COMPR	A,S	TEST FOR END OF RECORD
190i		JEQ	*+11	EXIT LOOP IF EOR
190j		STCH	BUFFER,X	STORE CHARACTER IN BUFFER
190k		TIXR	T	LOOP UNLESS MAXIMUM LENGTH
190l		JLT	*-19	HAS BEEN REACHED
190m		STX	LENGTH	SAVE RECORD LENGTH
195		LDA	LENGTH	TEST FOR END OF FILE
200		COMP	#0	
205		JEQ	ENDFIL	EXIT IF EOF FOUND
210		WRBUFF	05,BUFFER,LENGTH	WRITE OUTPUT RECORD
210a		CLEAR	X	CLEAR LOOP COUNTER
210b		LDT	LENGTH	
210c		LDCH	BUFFER,X	GET CHARACTER FROM BUFFER
210d		TD	=X'05'	TEST OUTPUT DEVICE
210e		JEQ	*-3	LOOP UNTIL READY
210f		WD	=X'05'	WRITE CHARACTER
210g		TIXR	T	LOOP UNTIL ALL CHARACTERS
210h		JLT	*-14	HAVE BEEN WRITTEN
215		J	CLOOP	LOOP
220	.ENDFIL	WRBUFF	05,EOF,THREE	INSERT EOF MARKER
220a	ENDFIL	CLEAR	X	CLEAR LOOP COUNTER
220b		LDT	THREE	
220c		LDCH	EOF,X	GET CHARACTER FROM BUFFER
220d		TD	=X'05'	TEST OUTPUT DEVICE
220e		JEQ	*-3	LOOP UNTIL READY
220f		WD	=X'05'	WRITE CHARACTER
220g		TIXR	T	LOOP UNTIL ALL CHARACTERS
220h		JLT	*-14	HAVE BEEN WRITTEN
225		J	@RETADR	
230	EOF	BYTE	C'EOF'	
235	THREE	WORD	3	
240	RETADR	RESW	1	
245	LENGTH	RESW	1	LENGTH OF RECORD
250	BUFFER	RESB	4096	4096-BYTE BUFFER AREA
255		END	FIRST	

Figure 4.2 Program from Fig. 4.1 with macros expanded.

The label on the macro invocation statement (CLOOP) has been retained as a label on the first statement generated in the macro expansion. This allows the programmer to use a macro instruction in exactly the same way as an assembler language mnemonic. The macro invocations on lines 210 and 220 are expanded in the same way. Note that the two invocations of WRBUFF specify different arguments, so they produce different expansions.

After macro processing, the expanded file (Fig. 4.2) can be used as input to the assembler. The macro invocation statements will be treated as comments, and the statements generated from the macro expansions will be assembled exactly as though they had been written directly by the programmer.

A comparison of the expanded program in Fig. 4.2 with the program in Fig. 2.5 shows the most significant differences between macro invocation and subroutine call. In Fig. 4.2, the statements from the body of the macro WRBUFF are generated twice: lines 210a through 210h and lines 220a through 220h. In the program of Fig. 2.5, the corresponding statements appear only once: in the subroutine WRREC (lines 210 through 240). In general, the statements that form the expansion of a macro are generated (and assembled) each time the macro is invoked. Statements in a subroutine appear only once, regardless of how many times the subroutine is called.

Note also that our macro instructions have been written so that the body of the macro contains no labels. In Fig. 4.1, for example, line 140 contains the statement "JEQ *–3" and line 155 contains "JLT *–14." The corresponding statements in the WRREC subroutine (Fig. 2.5) are "JEQ WLOOP" and "JLT WLOOP," where WLOOP is a label on the TD instruction that tests the output device. If such a label appeared on line 135 of the macro body, it would be generated twice—on lines 210d and 220d of Fig. 4.2. This would result in an error (a duplicate label definition) when the program is assembled. To avoid duplication of symbols, we have eliminated labels from the body of our macro definitions.

The use of statements like "JLT *–14" is generally considered to be a poor programming practice. It is somewhat less objectionable within a macro definition; however, it is still an inconvenient and error-prone method. In Section 4.2.2 we discuss ways of avoiding this problem.

4.1.2 Macro Processor Algorithm and Data Structures

It is easy to design a two-pass macro processor in which all macro definitions are processed during the first pass, and all macro invocation statements are expanded during the second pass. However, such a two-pass macro processor would not allow the body of one macro instruction to contain definitions of other macros (because all macros would have to be defined during the first pass before any macro invocations were expanded).

Such definitions of macros by other macros can be useful in certain cases. Consider, for example, the two macro instruction definitions in Fig. 4.3. The body of the first macro (MACROS) contains statements that define RDBUFF, WRBUFF, and other macro instructions for a SIC system (standard version). The body of the second macro instruction (MACROX) defines these same macros for a SIC/XE system. A program that is to be run on a standard SIC system could invoke MACROS to define the other utility macro instructions. A program for a SIC/XE system could invoke MACROX to define these same macros in their XE versions. In this way, the same program could run on either a standard SIC machine or a SIC/XE machine (taking advantage of the

```
1   MACROS    MACRO       {Defines SIC standard version macros}
2   RDBUFF    MACRO       &INDEV,&BUFADR,&RECLTH
              .
              .           {SIC  standard version}
              .
3             MEND        {End of RDBUFF}
4   WRBUFF    MACRO       &OUTDEV,&BUFADR,&RECLTH
              .
              .           {SIC standard version}
              .
5             MEND        {End of WRBUFF}
              .
              .
              .
6             MEND        {End of MACROS}
```

(a)

```
1   MACROX    MACRO       {Defines  SIC/XE macros}
2   RDBUFF    MACRO       &INDEV,&BUFADR,&RECLTH
              .
              .           {SIC/XE version}
              .
3             MEND        {End of RDBUFF}
4   WRBUFF    MACRO       &OUTDEV,&BUFADR,&RECLTH
              .
              .           {SIC/XE version}
              .
5             MEND        {End of WRBUFF}
              .
              .
              .
6             MEND        {End of MACROX}
```

(b)

Figure 4.3 Example of the definition of macros within a macro body.

extended features). The only change required would be the invocation of either MACROS or MACROX. It is important to understand that *defining* MACROS or MACROX does not define RDBUFF and the other macro instructions. These definitions are processed only when an invocation of MACROS or MACROX is *expanded*.

A one-pass macro processor that can alternate between macro definition and macro expansion is able to handle macros like those in Fig. 4.3. In this section we present an algorithm and a set of data structures for such a macro processor. Because of the one-pass structure, the definition of a macro must appear in the source program before any statements that invoke that macro. This restriction does not create any real inconvenience for the programmer. In fact, a macro invocation statement that preceded the definition of the macro would be confusing for anyone reading the program.

There are three main data structures involved in our macro processor. The macro definitions themselves are stored in a definition table (DEFTAB), which contains the macro prototype and the statements that make up the macro body (with a few modifications). Comment lines from the macro definition are not entered into DEFTAB because they will not be part of the macro expansion. References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments. The macro names are also entered into NAMTAB, which serves as an index to DEFTAB. For each macro instruction defined, NAMTAB contains pointers to the beginning and end of the definition in DEFTAB.

The third data structure is an argument table (ARGTAB), which is used during the expansion of macro invocations. When a macro invocation statement is recognized, the arguments are stored in ARGTAB according to their position in the argument list. As the macro is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body.

Figure 4.4 shows portions of the contents of these tables during the processing of the program in Fig. 4.1. Figure 4.4(a) shows the definition of RDBUFF stored in DEFTAB, with an entry in NAMTAB identifying the beginning and end of the definition. Note the positional notation that has been used for the parameters: the parameter &INDEV has been converted to *?1* (indicating the first parameter in the prototype), &BUFADR has been converted to *?2*, and so on. Figure 4.4(b) shows ARGTAB as it would appear during expansion of the RDBUFF statement on line 190. For this invocation, the first argument is F1, the second is BUFFER, etc. This scheme makes substitution of macro arguments much more efficient. When the *?n* notation is recognized in a line from DEFTAB, a simple indexing operation supplies the proper argument from ARGTAB.

The macro processor algorithm itself is presented in Fig. 4.5. The procedure DEFINE, which is called when the beginning of a macro definition is recognized, makes the appropriate entries in DEFTAB and NAMTAB. EXPAND is