Chapter 3

Loaders and Linkers

As we have seen, an object program contains translated instructions and data values from the source program, and specifies addresses in memory where these items are to be loaded. Our discussions in Chapter 2 introduced the following three processes:

1. *Loading*, which brings the object program into memory for execution.
2. *Relocation*, which modifies the object program so that it can be loaded at an address different from the location originally specified (see Section 2.2.2).
3. *Linking*, which combines two or more separate object programs and supplies the information needed to allow references between them (see Section 2.3.5).

A *loader* is a system program that performs the loading function. Many loaders also support relocation and linking. Some systems have a *linker* (or *linkage editor*) to perform the linking operations and a separate loader to handle relocation and loading. In most cases all the program translators (i.e., assemblers and compilers) on a particular system produce object programs in the same format. Thus one system loader or linker can be used regardless of the original source programming language.

In this chapter we study the design and implementation of loaders and linkers. For simplicity we often use the term *loader* in place of *loader and/or linker*. Because the processes of assembly and loading are closely related, this chapter is similar in structure to the preceding one. Many of the same examples used in our study of assemblers are carried forward in this chapter. During our discussion of assemblers, we studied a number of features and capabilities that are of concern to both the assembler and the loader. In the present chapter we encounter many of the same concepts again. This time, of course, we are primarily concerned with the operation of the loader; however, it is important to remember the close connections between program translation and loading.

As in the preceding chapter, we begin by discussing the most basic soft-

execution. Section 3.1 presents the design of an *absolute loader* and illustrates its operation. Such a loader might be found on a simple SIC machine that uses the sort of assembler described in Section 2.1.

Section 3.2 examines the issues of relocation and linking from the loader's point of view. We consider some possible alternatives for object program representation and examine how these are related to issues of machine architecture. We also present the design of a *linking loader*, a more advanced type of loader that is typical of those found on most modern computing systems.

Section 3.3 presents a selection of commonly encountered loader features that are not directly related to machine architecture. As before, our purpose is not to cover all possible options, but to introduce some of the concepts and techniques most frequently found in loaders.

Section 3.4 discusses alternative ways of accomplishing loader functions. We consider the various times at which relocation and linking can be performed, and the advantages and disadvantages associated with each. In this context we study linkage editors (which perform linking before loading) and dynamic linking schemes (which delay linking until execution time).

Finally, in Section 3.5 we briefly discuss some examples of actual loaders and linkers. As before, we are primarily concerned with aspects of each piece of software that are related to hardware or software design decisions.

3.1 BASIC LOADER FUNCTIONS

In this section we discuss the most fundamental functions of a loader—bringing an object program into memory and starting its execution. You are probably already familiar with how these basic functions are performed. This section is intended as a review to set the stage for our later discussion of more advanced loader functions. Section 3.1.1 discusses the functions and design of an absolute loader and gives the outline of an algorithm for such a loader. Section 3.1.2 presents an example of a very simple absolute loader for SIC/XE, to clarify the coding techniques that are involved.

3.1.1 Design of an Absolute Loader

We consider the design of an absolute loader that might be used with the sort of assembler described in Section 2.1. The object program format used is the same as that described in Section 2.1.1. An example of such an object program is shown in Fig. 3.1(a).

Because our loader does not need to perform such functions as linking and program relocation, its operation is very simple. All functions are accom-


```
HCOPY  00100000107A
T001000 1E 141033 482039 001036 281030 301015 482061 3C1003 00102A 0C1039 00102D
T00101E 15 0C1036 482061 081033 4C0000 454F46 000003 000000
T002039 1E 041030 001030 E0205D 30203F D8205D 281030 302057 549039 2C205E 38203F
T002057 1C 101036 4C0000 F1 001000 041030 E02079 302064 509039 DC2079 2C1036
T002073 07 382064 4C0000 05
E001000
```

(a) Object program

Memory address	Contents				
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	
0FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
1000	14103348	20390010	36281030	30101548	
1010	20613C10	0300102A	0C103900	102D0C10	
1020	36482061	0810334C	0000454F	46000003	
1030	000000xx	xxxxxxxx	xxxxxxxx	xxxxxxxx	← COPY
⋮	⋮	⋮	⋮	⋮	
2030	xxxxxxxx	xxxxxxxx	xx041030	001030E0	
2040	205D3020	3FD8205D	28103030	20575490	
2050	392C205E	38203F10	10364C00	00F10010	
2060	00041030	E0207930	20645090	39DC2079	
2070	2C103638	20644C00	0005xxxx	xxxxxxxx	
2080	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	

(b) Program loaded in memory

Figure 3.1 Loading of an absolute program.

program has been presented for loading (and that it will fit into the available memory). As each Text record is read, the object code it contains is moved to the indicated address in memory. When the End record is encountered, the loader jumps to the specified address to begin execution of the loaded program. Figure 3.1(b) shows a representation of the program from Fig. 3.1(a) after loading. The contents of memory locations for which there is no Text record are shown as *xxxx*. This indicates that the previous contents of these locations remain unchanged.

Figure 3.2 shows an algorithm for the absolute loader we have discussed. Although this process is extremely simple, there is one aspect that deserves comment. In our object program, each byte of assembled code is given its

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

Figure 3.2 Algorithm for an absolute loader.

operation code for an STL instruction would be represented by the *pair of characters* "1" and "4". When these are read by the loader (as part of the object program), they will occupy *two* bytes of memory. In the instruction as loaded for execution, however, this operation code must be stored in a *single* byte with *hexadecimal value* 14. Thus each pair of bytes from the object program record must be packed together into one byte during loading. It is very important to realize that in Fig. 3.1(a), each printed character represents one *byte* of the object program record. In Fig. 3.1(b), on the other hand, each printed character represents one *hexadecimal digit* in memory (i.e., a half-byte).

This method of representing an object program is inefficient in terms of both space and execution time. Therefore, most machines store object programs in a *binary* form, with each byte of object code stored as a single byte in the object program. In this type of representation, of course, a byte may contain any binary value. We must be sure that our file and device conventions do not cause some of the object program bytes to be interpreted as control characters. For example, the convention described in Section 2.1—indicating the end of a record with a byte containing hexadecimal 00—would clearly be unsuitable for use with a binary object program.

Obviously object programs stored in binary form do not lend themselves well to printing or to reading by human beings. Therefore, we continue to use character representations of object programs in our examples in this book.

3.1.2 A Simple Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute

program to be run by the computer—usually an operating system. (Bootstrap loaders are discussed in more detail in Section 3.4.3.) In this section, we examine a very simple bootstrap loader for SIC/XE. In spite of its simplicity, this program illustrates almost all of the logic and coding techniques that are used in an absolute loader.

Figure 3.3 shows the source code for our bootstrap loader. The bootstrap itself begins at address 0 in the memory of the machine. It loads the operating system (or some other program) starting at address 80. Because this loader is used in a unique situation (the initial program load for the system), the program to be loaded can be represented in a very simple format. Each byte of object code to be loaded is represented on device F1 as two hexadecimal digits (just as it is in a Text record of a SIC object program). However, there is no Header record, End record, or control information (such as addresses or lengths). The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80. After all of the object code from device F1 has been loaded, the bootstrap jumps to address 80, which begins the execution of the program that was loaded.

Much of the work of the bootstrap loader is performed by the subroutine GETC. This subroutine reads one character from device F1 and converts it from the ASCII character code to the value of the hexadecimal digit that is represented by that character. For example, the ASCII code for the character "0" (hexadecimal 30) is converted to the numeric value 0. Likewise, the ASCII codes for "1" through "9" (hexadecimal 31 through 39) are converted to the numeric values 1 through 9, and the codes for "A" through "F" (hexadecimal 41 through 46) are converted to the values 10 through 15. This is accomplished by subtracting 48 (hexadecimal 30) from the character codes for "0" through "9", and subtracting 55 (hexadecimal 37) from the codes for "A" through "F". The subroutine GETC jumps to address 80 when an end-of-file (hexadecimal 04) is read from device F1. It skips all other input characters that have ASCII codes less than hexadecimal 30. This causes the bootstrap to ignore any control bytes (such as end-of-line) that are read.

The main loop of the bootstrap keeps the address of the next memory location to be loaded in register X. GETC is used to read and convert a pair of characters from device F1 (representing 1 byte of object code to be loaded). These two hexadecimal digit values are combined into a single byte by shifting the first one left 4 bit positions and adding the second to it. The resulting byte is stored at the address currently in register X, using a STCH instruction that refers to location 0 using indexed addressing. The TIXR instruction is then used to add 1 to the value in register X. (Because we are not interested in the result of the comparison performed by TIXR, register X is also used as the second operand for this instruction.)

```
BOOT     START   0          BOOTSTRAP LOADER FOR SIC/XE
.
. THIS BOOTSTRAP READS OBJECT CODE FROM DEVICE F1 AND ENTERS IT
. INTO MEMORY STARTING AT ADDRESS 80 (HEXADECIMAL). AFTER ALL OF
. THE CODE FROM DEVF1 HAS BEEN SEEN ENTERED INTO MEMORY, THE
. BOOTSTRAP EXECUTES A JUMP TO ADDRESS 80 TO BEGIN EXECUTION OF
. THE PROGRAM JUST LOADED.  REGISTER X CONTAINS THE NEXT ADDRESS
. TO BE LOADED.
.
          CLEAR   A       CLEAR REGISTER A TO ZERO
          LDX     #128    INITIALIZE REGISTER X TO HEX 80
LOOP      JSUB    GETC    READ HEX DIGIT FROM PROGRAM BEING LOADED
          RMO     A,S     SAVE IN REGISTER S
          SHIFTL  S,4     MOVE TO HIGH-ORDER 4 BITS OF BYTE
          JSUB    GETC    GET NEXT HEX DIGIT
          ADDR    S,A     COMBINE DIGITS TO FORM ONE BYTE
          STCH    0,X     STORE AT ADDRESS IN REGISTER X
          TIXR    X,X     ADD 1 TO MEMORY ADDRESS BEING LOADED
          J       LOOP    LOOP UNTIL END OF INPUT IS REACHED
.
. SUBROUTINE TO READ ONE CHARACTER FROM INPUT DEVICE AND
. CONVERT IT FROM ASCII CODE TO HEXADECIMAL DIGIT VALUE. THE
. CONVERTED DIGIT VALUE IS RETURNED IN REGISTER A. WHEN AN
. END-OF-FILE IS READ, CONTROL IS TRANSFERRED TO THE STARTING
. ADDRESS (HEX 80).
.
GETC      TD      INPUT   TEST INPUT DEVICE
          JEQ     GETC    LOOP UNTIL READY
          RD      INPUT   READ CHARACTER
          COMP    #4      IF CHARACTER IS HEX 04 (END OF FILE),
          JEQ      80         JUMP TO START OF PROGRAM JUST LOADED
          COMP    #48     COMPARE TO HEX 30 (CHARACTER '0')
          JLT      GETC   SKIP CHARACTERS LESS THAN '0'
          SUB     #48     SUBTRACT HEX 30 FROM ASCII CODE
          COMP    #10     IF RESULT IS LESS THAN 10, CONVERSION IS
          JLT      RETURN    COMPLETE. OTHERWISE, SUBTRACT 7 MORE
          SUB     #7        (FOR HEX DIGITS 'A' THROUGH 'F')
          RSUB            RETURN TO CALLER
RETURN
INPUT     BYTE     X'F1'  CODE FOR INPUT DEVICE
          END      LOOP
```

Figure 3.3 Bootstrap loader for SIC/XE.

You should work through the execution of this bootstrap routine by hand [illegible] of sample input, keeping track of the exact contents of all

For simplicity, the bootstrap routine in Fig. 3.3 does not do any error checking it assumes that its input is correct. You are encouraged to think about the different kinds of error conditions that might arise during the loading, and how these could be handled.

3.2 MACHINE-DEPENDENT LOADER FEATURES

The absolute loader described in Section 3.1 is certainly simple and efficient; however, this scheme has several potential disadvantages. One of the most obvious is the need for the programmer to specify (when the program is assembled) the actual address at which it will be loaded into memory. If we are considering a very simple computer with a small memory (such as the standard version of SIC), this does not create much difficulty. There is only room to run one program at a time, and the starting address for this single user program is known in advance. On a larger and more advanced machine (such as SIC/XE), the situation is not quite as easy. We would often like to run several independent programs together, sharing memory (and other system resources) between them. This means that we do not know in advance where a program will be loaded. Efficient sharing of the machine requires that we write relocatable programs instead of absolute ones.

Writing absolute programs also makes it difficult to use subroutine libraries efficiently. Most such libraries (for example, scientific or mathematical packages) contain many more subroutines than will be used by any one program. To make efficient use of memory, it is important to be able to select and load exactly those routines that are needed. This could not be done effectively if all of the subroutines had preassigned absolute addresses.

In this section we consider the design and implementation of a more complex loader. The loader we present is one that is suitable for use on a SIC/XE system and is typical of those that are found on most modern computers. This loader provides for program relocation and linking, as well as for the simple loading functions described in the preceding section. As part of our discussion, we examine the effect of machine architecture on the design of the loader.

The need for program relocation is an indirect consequence of the change to larger and more powerful computers. The way relocation is implemented in a loader is also dependent upon machine characteristics. Section 3.2.1 discusses these dependencies by examining different implementation techniques and the circumstances in which they might be used.

Section 3.2.2 examines program linking from the loader's point of view. Linking is not a machine-dependent function in the sense that relocation is; however, the same implementation techniques are often used for these two

some of the routines being linked together. (See, for example, the previous discussion concerning the use of subroutine libraries.) For these reasons we discuss linking together with relocation in this section.

Section 3.2.3 discusses the data structures used by a typical linking (and relocating) loader, and gives a description of the processing logic involved. The algorithm presented here serves as a starting point for discussion of some of the more advanced loader features in the following sections.

3.2.1 Relocation

Loaders that allow for program relocation are called *relocating loaders* or *relative loaders*. The concept of program relocation was introduced in Section 2.2.2; you may want to briefly review that discussion before reading further. In this section we discuss two methods for specifying relocation as part of the object program.

The first method we discuss is essentially the same as that introduced in Chapter 2. A Modification record is used to describe each part of the object code that must be changed when the program is relocated. (The format of the Modification record is given in Section 2.3.5.) Figure 3.4 shows a SIC/XE program we use to illustrate this first method of specifying relocation. The program is the same as the one in Fig. 2.6; it is reproduced here for convenience. Most of the instructions in this program use relative or immediate addressing. The only portions of the assembled program that contain actual addresses are the extended format instructions on lines 15, 35, and 65. Thus these are the only items whose values are affected by relocation.

Figure 3.5 displays the object program corresponding to the source in Fig. 3.4. Notice that there is one Modification record for each value that must be changed during relocation (in this case, the three instructions previously mentioned). Each Modification record specifies the starting address and length of the field whose value is to be altered. It then describes the modification to be performed. In this example, all modifications add the value of the symbol COPY, which represents the starting address of the program (Fig. 3.6). More examples of relocation specified in this manner appear in the next section when we examine the relationship between relocation and linking.

The Modification record scheme is a convenient means for specifying program relocation; however, it is not well suited for use with all machine architectures. Consider, for example, the program in Fig. 3.7. This is a relocatable program written for the standard version of SIC. The important difference between this example and the one in Fig. 3.4 is that the standard SIC machine

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

```
HCOPY  000000001077
T0000001D17202D69202D4B10103603202629000033200 74B10105D3F2FEC032010
T00001D130F20160100030F200D4B10105D3E2003454F46
T001036 1DB410B400B44075101000E32019332FFADB2013A004332008 57C003B850
T001053 1D3B2FEA1340004F0000F1B410774000E32011332FFA 53C003DF2008B850
T00107007 3B2FEF4F000005
M00000705+COPY
M00001405+COPY
M00002705+COPY
E000000
```

Figure 3.5 Object program with relocation by Modification records.

```
begin
   get PROGADDR from operating system
   while not end of input do
      begin
         read next record
         while record type ≠ 'E' do
            begin
               read next input record
               while record type = 'T' then
                  begin
                     move object code from record to location
                        ADDR + specified address
                  end
               while record type = 'M'
                  add PROGADDR at the location PROGADDR +
                     specified address
            end
      end
end
```

Figure 3.6 SIC/XE relocation loader algorithm.

This would require 31 Modification records, which results in an object program more than twice as large as the one in Fig. 3.5.

On a machine that primarily uses direct addressing and has a fixed instruction format, it is often more efficient to specify relocation using a different technique. Figure 3.8 shows this method applied to our SIC program example. There are no Modification records. The Text records are the same as before except that there is a *relocation bit* associated with each word of object code. Since all SIC instructions occupy one word, this means that there is one reloc

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	

```
HCOPY  000000001078
T0000001EFFC1400334810390000362800303000154810613C0003000002A0C0039000002D
T00001E15E000C0036481061080033 4C0000454F46000003000000
T001039 1EFFC040030000030E0105D30103FD8105D280030301057548039 2C105E38103F
T0010570A8001000364C0000F1001000
T00106119FE0040030E01079301064508039DC10792C0036381064 4C0000005
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 1-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 elim[inates] the loader to perform program relocation. For


```
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

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 [illegible] (with two Modification records) even though the final result will be an

Loc		Source statement		Object code
0000	PROGA	START	0	
		EXTDEF	LISTA,ENDA	
		EXTREF	LISTB,ENDB,LISTC,ENDC	
		.		
		.		
		.		
0020	REF1	LDA	LISTA	03201D
0023	REF2	+LDT	LISTB+4	77100004
0027	REF3	LDX	#ENDA-LISTA	050014
		.		
		.		
		.		
0040	LISTA	EQU	*	
		.		
		.		
0054	ENDA	EQU	*	
0054	REF4	WORD	ENDA-LISTA+LISTC	000014
0057	REF5	WORD	ENDC-LISTC-10	FFFFF6
005A	REF6	WORD	ENDC-LISTC+LISTA-1	00003F
005D	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	000014
0060	REF8	WORD	LISTB-LISTA	FFFFC0
		END	REF1	

Loc		Source statement		Object code
0000	PROGB	START	0	
		EXTDEF	LISTB,ENDB	
		EXTREF	LISTA,ENDA,LISTC,ENDC	
		.		
		.		
		.		
0036	REF1	+LDA	LISTA	03100000
003A	REF2	LDT	LISTB+4	772027
003D	REF3	+LDX	#ENDA-LISTA	05100000
		.		
		.		
		.		
0060	LISTB	EQU	*	
		.		
		.		
0070	ENDB	EQU	*	
0070	REF4	WORD	ENDA-LISTA+LISTC	000000
0073	REF5	WORD	ENDC-LISTC-10	FFFFF6
0076	REF6	WORD	ENDC-LISTC+LISTA-1	FFFFFF
0079	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	FFFFF0
007C	REF8	WORD	LISTB-LISTA	000060
		END		

```
Loc                Source statement                          Object code

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

Figure 3.10 (*cont'd*)

```
HPROGA 000000000063
DLISTA 000040ENDA  000054
RLISTB ENDB  LISTC ENDC
.
.
T0000200A03201D77100004050014
.
.
T0000540F000014FFFFF600003F000014FFFFC0
M00002405+LISTB
M00005406+LISTC
M00005706+ENDC
M00005706-LISTC
M00005A06+ENDC
M00005A06-LISTC
M00005A06+PROGA
M00005D06-ENDB
M00005D06+LISTB
M00006006+LISTB
M00006006-PROGA
```



```
HPROGB 00000000007F
DLISTB 000060ENDB  000070
RLISTA ENDA  LISTC ENDC
.
.
T0000360B03100000772027051000000
.
.
T0000700F000000FFFFF6FFFFFFFFFF0000060
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
.
.
T0000180C03100000771000040510000O
.
.
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 Modifica-

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

Memory address	Contents				
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	
3FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
4000	....	....	....	....	
4010	....	....	....	....	
4020	03201D77	1040C705	0014....	....	←PROGA
4030	....	....	....	....	
4040	....	....	....	....	
4050	....	00412600	00080040	51000004	
4060	000083..	....	....	....	
4070	....	....	....	....	
4080	....	....	....	....	
4090	....	....	..031040	40772027	←PROGB
40A0	05100014	....	....	....	
40B0	....	....	....	....	
40C0	....	....	....	....	
40D0	00	41260000	08004051	00000400	
40E0	0083....	....	....	....	
40F0	....	....	0310	40407710	←PROGC
4100	40C70510	0014....	....	....	
4110	....	....	....	....	
4120	....	00412600	00080040	51000004	
4130	000083xx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
4140	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	

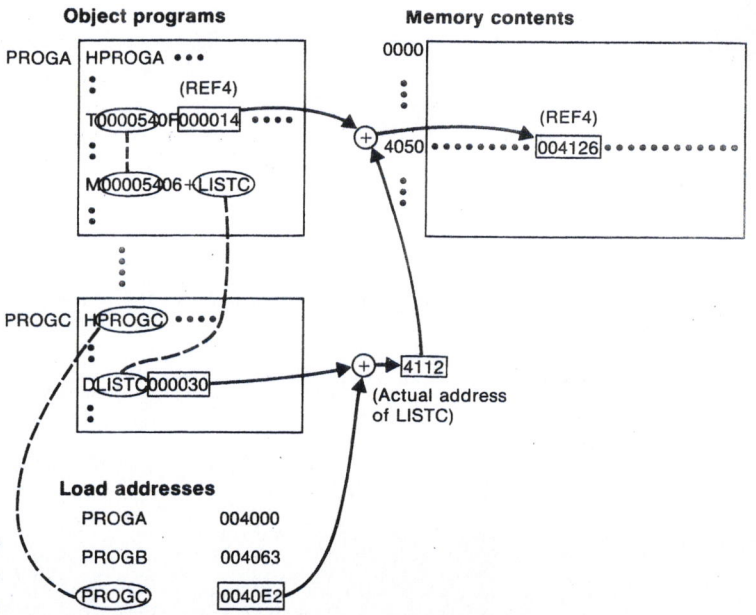


Figure 3.12(b) Relaxation and linking operations performed on REF4 from PROGA.

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}
```


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.

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 000040ENDA  000054
R02LISTB 03ENDB  04LISTC 05ENDC
.
.
T0000200A03201D771000040500 14
.
.
T0000540F000014FFFFF600003F000014FFFFC0
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 refer- [illegible] underlined


```
HPROGB 000000000007F
DLISTB 000060ENDB  000070
R02LISTA 03ENDA  04LISTC 05ENDC
.
.
T000036 0B 03100000772027 05100000
.
.
T000070 0F 000000FFFFF6FFFFFEFFFFF0000060
M000037 05+02
M00003E 05+03
M00003E 05-02
M000070 06+03
M000070 06-02
M000070 06+04
M000073 06+05
M000073 06-04
M000076 06+05
M000076 06-04
M000076 06+02
M000079 06+03
M000079 06-02
M00007C 06+01
M00007C 06-02
E
```

```
HPROGC 000000000051
DLISTC 000030ENDC  000042
R02LISTA 03ENDA  04LISTB 05ENDB
.
.
T000018 0C 03100000771000040 5100000
.
.
T000042 0F 000030000008000011000000000000
M000019 05+02
M00001D 05+04
M000021 05+03
M000021 05-02
M000042 06+03
M000042 06-02
M000042 06+01
M000048 06+02
M00004B 06+03
M00004B 06-02
M00004B 06-05
M00004B 06+04
M00004E 06+04
M00004E 06-02
E
```

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

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

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.

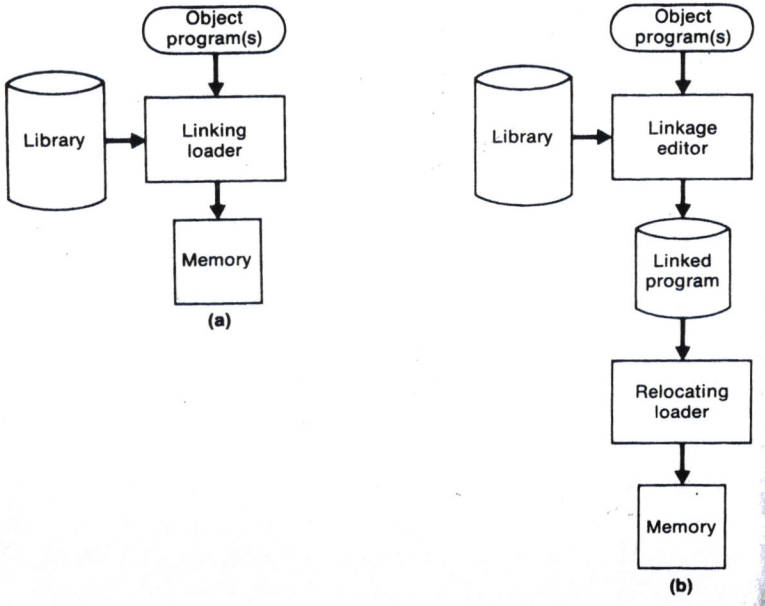


Figure 3.15 Processing of an object program using (a) linking loader and (b) linkage editor.

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

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   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

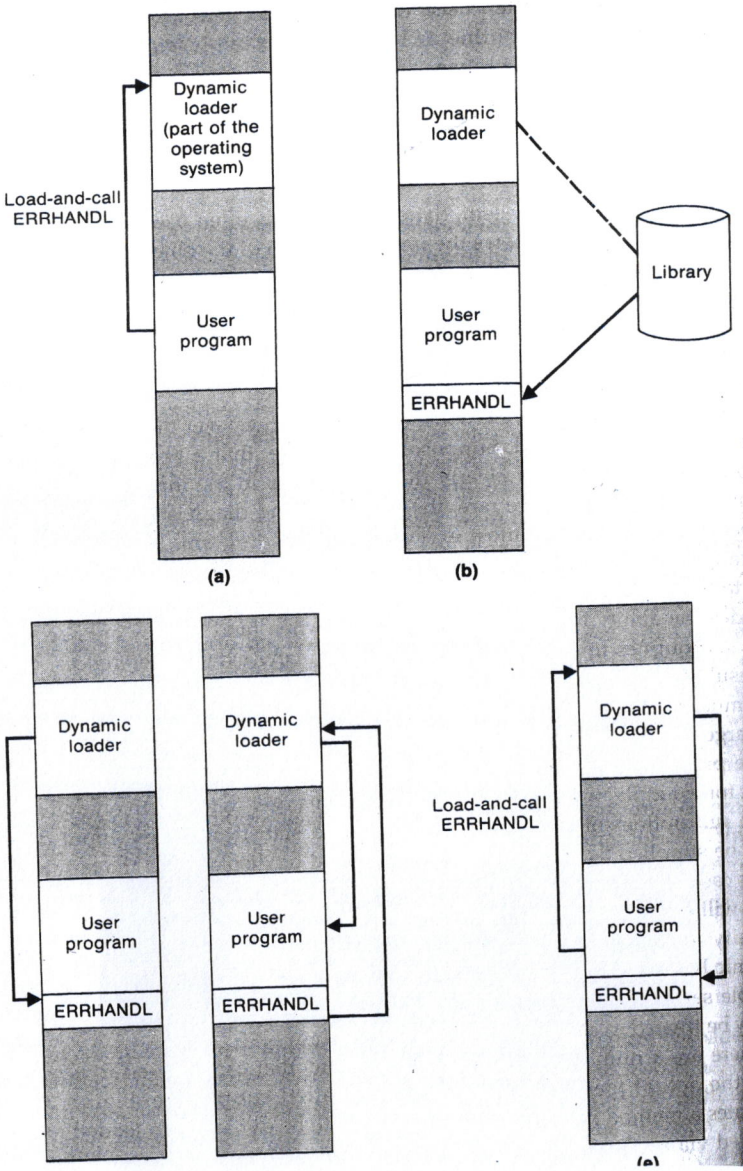
Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library. For example, run-time support routines for a high-level language like C could be stored in a *dynamic link library*. A single copy of the routines in this library could be loaded into the memory 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



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

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

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