An AIX assembler language program can be divided into *control sections* using the .CSECT assembler directive. Each control section has an associated storage mapping class that describes the kind of data it contains. Some of the most commonly used storage mapping classes are PR (executable instructions), RO (read-only data), RW (read/write data), and BS (uninitialized read/write data). AIX control sections combine some of the features of the SIC control sections and program blocks that we discussed in Section 2.3. One control section may consist of several different parts of the source program. These parts are gathered together by the assembler, as with SIC program blocks. The control sections themselves remain separate after assembly, and are handled independently by the loader or linkage editor.

The AIX assembler language provides a special type of control section called a *dummy section*. Data items included in a dummy section do not actually become part of the object program; they serve only to define labels within the section. Dummy sections are most commonly used to describe the layout of a record or table that is defined externally. The labels define symbols that can be used to address fields in the record or table (after an appropriate base register is established). AIX also provides *common blocks*, which are uninitialized blocks of storage that can be shared between independently assembled programs.

Linking of control sections can be accomplished using methods like the ones we discussed for SIC. The assembler directive .GLOBL makes a symbol available to the linker, and the directive .EXTERN declares that a symbol is defined in another source module. These directives are essentially the same as the SIC directives EXTDEF and EXTREF. Expressions that involve relocatable and external symbols are classified and handled using rules similar to those discussed in Sections 2.3.3 and 2.3.5.

The AIX assembler also provides a different method for linking control sections. By using assembler directives, the programmer can create a *table of contents* (TOC) for the assembled program. The TOC contains addresses of control sections and global symbols defined within the control sections. To refer to one of these symbols, the program retrieves the needed address from the TOC, and then uses that address to refer to the needed data item or procedure. (Some types of frequently used data items can be stored directly in the TOC for efficiency of retrieval.) If all references to external symbols are done in this way, then the TOC entries are the only parts of the program involved in relocation and linking when the program is loaded.

The AIX assembler itself has a two-pass structure similar to the one we discussed for SIC. However, there are some significant differences. The first pass of the AIX assembler writes a listing file that contains warnings and error messages. If errors are found during the first pass, the assembler terminates and does not continue to the second pass. In this case, the assembly listing contains only errors that could be detected during Pass 1.

If no errors are detected during the first pass, the assembler proceeds to Pass 2. The second pass reads the source program again, instead of using an intermediate file as we discussed for SIC. This means that location counter values must be recalculated during Pass 2. It also means that any warning messages that were generated during Pass 1 (but were not serious enough to terminate the assembly) are lost. The assembly listing will contain only errors and warnings that are generated during Pass 2.

Assembled control sections are placed into the object program according to their storage mapping class. Executable instructions, read-only data, and various kinds of debugging tables are assigned to an object program section named .TEXT. Read/write data and TOC entries are assigned to an object program section named .DATA. Uninitialized data is assigned to a section named .BSS. When the object program is generated, the assembler first writes all of the .TEXT control sections, followed by all of the .DATA control sections except for the TOC. The TOC is written after the other .DATA control sections. Relocation and linking operations are specified by entries in a relocation table, similar to the Modification records we discussed for SIC.

EXERCISES

Section 2.1

1. Apply the algorithm described in Fig. 2.4 to assemble the source program in Fig. 2.1. Your results should be the same as those shown in Figs. 2.2 and 2.3.

2. Apply the algorithm described in Fig. 2.4 to assemble the following SIC source program:

```
SUM       START     4000
FIRST     LDX       ZERO
          LDA       ZERO
LOOP      ADD       TABLE,X
          TIX       COUNT
          JLT       LOOP
          STA       TOTAL
          RSUB
TABLE     RESW      2000
COUNT     RESW      1
ZERO      WORD      0
TOTAL     RESW      1
          END       FIRST
```

3. As mentioned in the text, a number of operations in the algorithm of Fig. 2.4 are not explicitly spelled out. (One example would be scanning the instruction operand field for the modifier ",X".) List as many of these implied operations as you can, and think about how they might be implemented.

4. Suppose that you are to write a "disassembler"—that is, a system program that takes an ordinary object program as input and produces a listing of the source version of the program. What tables and data structures would be required, and how would they be used? How many passes would be needed? What problems would arise in recreating the source program?

5. Many assemblers use free-format input. Labels must start in Column 1 of the source statement, but other fields (opcode, operands, comments) may begin in any column. The various fields are separated by blanks. How could our assembler logic be modified to allow this?

6. The algorithm in Fig. 2.4 provides for the detection of some assembly errors; however, there are many more such errors that might occur. List error conditions that might arise during the assembly of a SIC program. When and how would each type of error be detected, and what action should the assembler take for each?

7. Suppose that the SIC assembler language is changed to include a new form of the RESB statement, such as

```
                RESB   n'c'
```

   which reserves *n* bytes of memory and initializes all of these bytes to the character 'c'. For example, line 105 in Fig. 2.5 could be changed to

```
    BUFFER      RESB   4096'
```

   This feature could be implemented by simply generating the required number of bytes in Text records. However, this could lead to a large increase in the size of the object program—for example, the object program in Fig. 2.8 would be about 40 times its previous size. Propose a way to implement this new form of RESB without such a large increase in object program size.

8. Suppose that you have a two-pass assembler that is written according to the algorithm in Fig. 2.4. In the case of a duplicate symbol, this assembler would give an error message only for the second (i.e., duplicate) definition. For example, it would give an error message only for line 5 of the program that follows.

```
1        P3        START     1000
2                  LDA       ALPHA
.                  .         .
3                  STA       ALPHA
.                  .         .
4        ALPHA     RESW      1
.                  .         .
5        ALPHA     WORD      0
6                  END
```

Suppose that you want to change the assembler to give error messages for all definitions of a doubly defined symbol (e.g., lines 4 and 5), and also for all references to a doubly defined symbol (e.g., lines 2 and 3). Describe the changes you would make to accomplish this. In making this modification, you should change the existing assembler as little as possible.

9. Suppose that you have a two-pass assembler that is written according to the algorithm in Fig. 2.4. You want to change this assembler so that it gives a warning message for labels that are not referenced in the program, as illustrated by the following example.

```
P3       START     1000
         LDA       DELTA
         ADD       BETA
LOOP     STA       DELTA

Warning: label is never referenced

         RSUB
ALPHA    RESW      1

Warning: label is never referenced

BETA     RESW      1
DELTA    RESW      1
         END
```

The warning messages should appear in the assembly listing directly below the line that contains the unreferenced label, as shown above. Describe the changes you would make in the assembler to add this new diagnostic feature. In making this modification, you should change the existing assembler as little as possible.

Section 2.2

1. Could the assembler decide for itself which instructions need to be assembled using extended format? (This would avoid the necessity for the programmer to code + in such instructions.)

2. As we have described it, the BASE statement simply gives information to the assembler. The programmer must also write an instruction like LDB to load the correct value into the base register. Could the assembler automatically generate the LDB instruction from the BASE statement? If so, what would be the advantages and disadvantages of doing this?

3. Generate the object code for each statement in the following SIC/XE program:

```
SUM         START       0
FIRST       LDX         #0
            LDA         #0
           +LDB         #TABLE2
            BASE         TABLE2
LOOP        ADD          TABLE,X
            ADD          TABLE2,X
            TIX          COUNT
            JLT          LOOP
           +STA          TOTAL
            RSUB
COUNT       RESW         1
TABLE       RESW         2000
TABLE2      RESW         2000
TOTAL       RESW         1
            END          FIRST
```

4. Generate the complete object program for the source program given in Exercise 3.

5. Modify the algorithm described in Fig. 2.4 to handle all of the SIC/XE addressing modes discussed. How would these modifications be reflected in the assembler designs discussed in Chapter 8?

6. Modify the algorithm described in Fig. 2.4 to handle relocatable programs. How would these modifications be reflected in the assembler designs discussed in Chapter 8?

7. Suppose that you are writing a disassembler for SIC/XE (see Exercise 2.1.4). How would your disassembler deal with the various addressing modes and instruction formats?

8. Our discussion of SIC/XE Format 4 instructions specified that the 20-bit "address" field should contain the actual target address, and that addressing mode bits *b* and *p* should be set to 0. (That is, the instruction should contain a direct address—it should not use base relative or program-counter relative addressing.)

However, it would be possible to use program-counter relative addressing with Format 4. In that case, the "address" field would actually contain a displacement, and bit *p* would be set to 1. For example, the instruction on line 15 in Fig. 2.6 could be assembled as

```
0006  CLOOP  +JSUB   RDREC        4B30102C
```

(using program-counter relative addressing with displacement 102C).

What would be the advantages (if any) of assembling Format 4 instructions in this way? What would be the disadvantages (if any)? Are there any situations in which it would *not* be possible to assemble a Format 4 instruction using program-counter relative addressing?

9. Our Modification record format is well suited for SIC/XE programs because all address fields in instructions and data words fall neatly into half-bytes. What sort of Modification record could we use if this were not the case (that is, if address fields could begin anywhere within a byte and could be of any length)?

10. Suppose that we made the program in Fig. 2.1 a relocatable program. This program is written for the *standard* version of SIC, so all operand addresses are actual addresses, and there is only one instruction format. Nearly every instruction in the object program would need to have its operand address modified at load time. This would mean a large number of Modification records (more than doubling the size of the object program). How could we include the required relocation information without this large increase in object program size?

11. Suppose that you are writing an assembler for a machine that has *only* program-counter relative addressing. (That is, there are no direct-addressing instruction formats and no base relative addressing.) Suppose that you wish to assemble an instruction whose operand is an absolute address in memory—for example,

```
LDA   100
```

to load register A from address (hexadecimal) 100 in memory. How might such an instruction be assembled in a relocatable program? What relocation operations would be required?

12. Suppose that you are writing an assembler for a machine on which the length of an assembled instruction depends upon the type of the operand. Consider, for example, the following three fragments of code:

a.
```
        ADD   ALPHA
         .
         .
ALPHA   DC     I(3)
```

b.
```
        ADD   ALPHA
         .
         .
ALPHA   DC     F(3.1)
```

c.
```
        ADD   ALPHA
         .
         .
ALPHA   DC     D(3.14159)
```

In case (a), ALPHA is an integer operand; the ADD instruction generates 2 bytes of object code. In case (b), ALPHA is a single-precision floating-point operand; the ADD instruction generates 3 bytes of object code. In case (c), ALPHA is a double-precision floating-point operand; the ADD instruction generates 4 bytes of object code.

What special problems does such a machine present for an assembler? Briefly describe how you would solve these problems—that is, how your assembler for this machine would be different from the assembler structure described in Section 2.1.

Section 2.3

1. Write an algorithm for SIC/XE assembler.
2. Modify the algorithm described in Fig. 2.4 to handle literals.
3. In the program of Fig. 2.9, could we have used literals on lines 135 and 145? Why might we prefer *not* to use a literal here?
4. With a minor extension to our literal notation, we could write the instruction on line 55 of Fig. 2.9 as

   ```
   LDA    =W'3'
   ```

   specifying as the literal operand a word with the value 3. Would this be a good idea?
5. Immediate operands and literals are both ways of specifying an operand value in a source statement. What are the advantages and disadvantages of each? When might each be preferable to the other?

6. Suppose that you have a two-pass SIC/XE assembler that does not support literals. Now you want to modify the assembler to handle literals. However, you want to place the literal pool at the *beginning* of the assembled program, not at the end as is commonly done. (You do not have to worry about LTORG statements—your assembler should always place all literals in a pool at the beginning of the program.) Describe how you could accomplish this. If possible, you should do so without adding another pass to the assembler. Be sure to describe any data structures that you may need, and explain how they are used in the assembler.

7. Suppose we made the following changes to the program in Fig. 2.9:

   a. Delete the LTORG statement on line 93.

   b. Change the statement on line 45 to +LDA... .

   c. Change the operands on lines 135 and 145 to use literals (and delete line 185).

   Show the resulting object code for lines 45, 135, 145, 215, and 230. Also show the literal pool with addresses and data values. Note: you do not need to retranslate the entire program to do this.

8. Assume that the symbols ALPHA and BETA are labels in a source program. What is the difference between the following two sequences of statements?

```
a.          LDA   ALPHA-BETA

b.          LDA   ALPHA
            SUB   BETA
```

9. What is the difference between the following sequences of statements?

```
a.          LDA   #3

b. THREE    EQU    3
            .
            .
            LDA   #THREE

c. THREE    EQU    3
            .
            .
            LDA    THREE
```

10. Modify the algorithm described in Fig. 2.4 to handle multiple control sections.

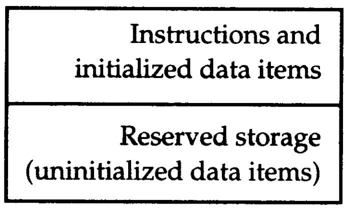
11. Suppose all the features we described in Section 2.3 were to be implemented in an assembler. How would the symbol table required be different from the one discussed in Section 2.1?

12. Which of the features described in Section 2.3 would create additional problems in the writing of a disassembler (see Exercise 2.1.4)? Describe these problems, and discuss possible solutions.

13. When different control sections are assembled together, some references between them could be handled by the assembler (instead of being passed on to the loader). In the program of Fig. 2.15, for example, the expression on line 190 could be evaluated directly by the assembler because its symbol table contains all of the required information. What would be the advantages and disadvantages of doing this?

14. In the program of Fig. 2.11, suppose we used only two program blocks: the default block and CBLKS. Assume that the data items in CDATA are to be included in the default block. What changes in the source program would accomplish this? Show the object program (corresponding to Fig. 2.13) that would result.

15. Suppose that for some reason it is desirable to separate the parts of an assembler language program that require initialization (e.g., instructions and data items defined with WORD or BYTE) from the parts that do not require initialization (e.g., storage reserved with RESW or RESB). Thus, when the program is loaded for execution it should look like



Suppose that it is considered too restrictive to require the programmer to perform this separation. Instead, the assembler should take the source program statements in whatever order they are written, and automatically perform the rearrangement as described above.

Describe a way in which this separation of the program could be accomplished by a two-pass assembler.

16. Suppose LENGTH is defined as in the program of Fig. 2.9. What would be the difference between the following sequences of statements?

a.
```
        LDA     LENGTH
        SUB     #1
```
b.
```
        LDA     LENGTH-1
```

17. Referring to the definitions of symbols in Fig. 2.10, give the value, type, and intuitive meaning (if any) of each of the following expressions:

    a. `BUFFER-FIRST`

    b. `BUFFER+4095`

    c. `MAXLEN-1`

    d. `BUFFER+MAXLEN-1`

    e. `BUFFER-MAXLEN`

    f. `2*LENGTH`

    g. `2*MAXLEN-1`

    h. `MAXLEN-BUFFER`

    i. `FIRST+BUFFER`

    j. `FIRST-BUFFER+BUFEND`

18. In the program of Fig. 2.9, what is the advantage of writing (on line 107)

```
MAXLEN   EQU    BUFEND-BUFFER
```

    instead of

```
MAXLEN   EQU    4096  ?
```

19. In the program of Fig. 2.15, could we change line 190 to

```
MAXLEN   EQU    BUFEND-BUFFER
```

    and line 133 to

```
         +LDT   #MAXLEN
```

    as we did in Fig. 2.9?

20. The assembler could simply assume that any reference to a symbol not defined within a control section is an external reference. This change would eliminate the need for the EXTREF statement. Would this be a good idea?

21. How could an assembler that allows external references avoid the need for an EXTDEF statement? What would be the advantages and disadvantages of doing this?

22. The assembler could automatically use extended format for instructions whose operands involve external references. This would eliminate the need for the programmer to code + in such statements. What would be the advantages and disadvantages of doing this?

23. On some systems, control sections can be composed of several different parts, just as program blocks can. What problems does this pose for the assembler? How might these problems be solved?

24. Assume that the symbols RDREC and COPY are defined as in Fig. 2.15. According to our rules, the expression

    ```
    RDREC-COPY
    ```

    would be illegal (that is, the assembler and/or the loader would reject it). Suppose that for some reason the program really needs the value of this expression. How could such a thing be accomplished without changing the rules for expressions?

25. We discussed a large number of assembler directives, and many more could be implemented in an actual assembler. Checking for them one at a time using comparisons might be quite inefficient. How could we use a table, perhaps similar to OPTAB, to speed recognition and handling of assembler directives? (Hint: the answer to this problem may depend upon the language in which the assembler itself is written.)

26. Other than the listing of the source program with generated object code, what assembler outputs might be useful to the programmer? Suggest some optional listings that might be generated and discuss any data structures or algorithms involved in producing them.

Section 2.4

1. The process of fixing up a few forward references should involve less overhead than making a complete second pass of the source program. Why don't all assemblers use the one-pass technique for efficiency?

2. Suppose we wanted our assembler to produce a cross-reference listing for all symbols used in the program. For the program of Fig. 2.5, such a listing might look like

Symbol	Defined on line	Used on lines
COPY	5	
FIRST	10	255
CLOOP	15	40
ENDFIL	45	30
EOF	80	45
RETADR	95	10,70
LENGTH	100	12,13,20,60,175,212
.		
.		

How might this be done by the assembler? Indicate changes to the logic and tables discussed in Section 2.1 that would be required.

3. Could a one-pass assembler produce a relocatable object program and handle external references? Describe the processing logic that would be involved and identify any potential difficulties.
4. How could literals be implemented in a one-pass assembler?
5. We discussed one-pass assemblers as though instruction operands could only be single symbols. How could a one-pass assembler handle an instruction like

```
JEQ     ENDFIL+3
```

   where ENDFIL has not yet been defined?
6. Outline the logic flow for a simple one-pass load-and-go assembler.
7. Using the methods outlined in Chapter 8, develop a modular design for a one-pass assembler that produces object code in memory.
8. Suppose that an instruction involving a forward reference is to be assembled using program-counter relative addressing. How might this be handled by a one-pass assembler?
9. The process of fixing up forward references in a one-pass assembler that produces an object program is very similar to the linking process described in Section 2.3.5. Why didn't we just use Modification records to fix up the forward references?
10. How could we extend the methods of Section 2.4.2 to handle forward references in ORG statements?
11. Write an algorithm for a multipass assembler.

Section 2.5

1. Consider the description of the VAX architecture in Section 1.4.1. What characteristics would you expect to find in a VAX assembler?
2. Consider the description of the T3E architecture in Section 1.5.3. What characteristics would you expect to find in a T3E assembler?

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 software function—in this case, loading an object program into memory for 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 accomplished in a single pass. The Header record is checked to verify that the correct

```
H^COPY  ^001000^00107A
T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^00102A^0C1039^00102D
T^00101E^15^0C1036^482061^081033^4C0000^454F46^000003^000000
T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^549039^2C205E^38203F
T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^509039^DC2079^2C1036
T^002073^07^382064^4C0000^05
E^001000
```

(a) Object program

```
Memory
address                     Contents
0000     xxxxxxxx   xxxxxxxx   xxxxxxxx   xxxxxxxx
0010     xxxxxxxx   xxxxxxxx   xxxxxxxx   xxxxxxxx
  •          •          •          •          •
  •          •          •          •          •
  •          •          •          •          •
0FF0     xxxxxxxx   xxxxxxxx   xxxxxxxx   xxxxxxxx
1000     14103348   20390010   36281030   30101548
1010     20613C10   0300102A   0C103900   102D0C10
1020     36482061   0810334C   0000454F   46000003
1030     000000xx   xxxxxxxx   xxxxxxxx   xxxxxxxx  ←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 using its hexadecimal representation in character form. For example, the machine

```
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 loader, called a *bootstrap loader*, is executed. This bootstrap loads the first 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')
RETURN   RSUB                RETURN TO CALLER
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 with several bytes of sample input, keeping track of the exact contents of all registers and memory locations as you go. This will help you become familiar with the machine-level details of how loading is performed.

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 functions. In addition, the process of linking usually involves relocation of 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 does not use relative addressing. In this program the addresses in all the instructions except RSUB must be modified when the program is relocated.

```
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
```

Figure 3.4 Example of a SIC/XE program (from Fig. 2.6).

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

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 relocation bit for each possible instruction. The relocation bits are gathered together into a *bit mask* following the length indicator in each Text record. In Fig. 3.8