```
M00000705
M00001405
M00002705
```

to

```
M00000705+COPY
M00001405+COPY
M00002705+COPY
```

In this way, exactly the same mechanism can be used for program relocation and for program linking. There are more examples in the next chapter.

The existence of multiple control sections that can be relocated independently of one another makes the handling of expressions slightly more complicated. Our earlier definitions required that all of the relative terms in an expression be paired (for an absolute expression), or that all except one be paired (for a relative expression). We must now extend this restriction to specify that both terms in each pair must be relative within the same control section. The reason is simple—if the two terms represent relative locations in the same control section, their difference is an absolute value (regardless of where the control section is located). On the other hand, if they are in different control sections, their difference has a value that is unpredictable (and therefore probably useless). For example, the expression

```
BUFEND-BUFFER
```

has as its value the length of BUFFER in bytes. On the other hand, the value of the expression

```
RDREC-COPY
```

is the difference in the load addresses of the two control sections. This value depends on the way run-time storage is allocated; it is unlikely to be of any use whatsoever to an application program.

When an expression involves external references, the assembler cannot in general determine whether or not the expression is legal. The pairing of relative terms to test legality cannot be done without knowing which of the terms occur in the same control sections, and this is unknown at assembly time. In such a case, the assembler evaluates all of the terms it can, and combines these to form an initial expression value. It also generates Modification records so the loader can finish the evaluation. The loader can then check the expression for errors. We discuss this further in Chapter 3 when we examine the design of a linking loader.

2.4 ASSEMBLER DESIGN OPTIONS

In this section we discuss two alternatives to the standard two-pass assembler logic. Section 2.4.1 describes the structure and logic of one-pass assemblers. These assemblers are used when it is necessary or desirable to avoid a second pass over the source program. Section 2.4.2 introduces the notion of a multi-pass assembler, an extension to the two-pass logic that allows an assembler to handle forward references during symbol definition.

2.4.1 One-Pass Assemblers

In this section we examine the structure and design of one-pass assemblers. As we discussed in Section 2.1, the main problem in trying to assemble a program in one pass involves forward references. Instruction operands often are symbols that have not yet been defined in the source program. Thus the assembler does not know what address to insert in the translated instruction.

It is easy to eliminate forward references to data items; we can simply require that all such areas be defined in the source program before they are referenced. This restriction is not too severe. The programmer merely places all storage reservation statements at the start of the program rather than at the end. Unfortunately, forward references to labels on instructions cannot be eliminated as easily. The logic of the program often requires a forward jump—for example, in escaping from a loop after testing some condition. Requiring that the programmer eliminate all such forward jumps would be much too restrictive and inconvenient. Therefore, the assembler must make some special provision for handling forward references. To reduce the size of the problem, many one-pass assemblers do, however, prohibit (or at least discourage) forward references to data items.

There are two main types of one-pass assembler. One type produces object code directly in memory for immediate execution; the other type produces the usual kind of object program for later execution. We use the program in Fig. 2.18 to illustrate our discussion of both types. This example is the same as in Fig. 2.2, with all data item definitions placed ahead of the code that references them. The generated object code shown in Fig. 2.18 is for reference only; we will discuss how each type of one-pass assembler would actually generate the object program required.

We first discuss one-pass assemblers that generate their object code in memory for immediate execution. No object program is written out, and no loader is needed. This kind of *load-and-go* assembler is useful in a system that is oriented toward program development and testing. A university computing system for student use is a typical example of such an environment. In such

```
Line   Loc          Source statement              Object code

  0    1000    COPY      START    1000
  1    1000    EOF       BYTE     C'EOF'             454F46
  2    1003    THREE     WORD     3                  000003
  3    1006    ZERO      WORD     0                  000000
  4    1009    RETADR    RESW     1
  5    100C    LENGTH    RESW     1
  6    100F    BUFFER    RESB     4096
  9            .
 10    200F    FIRST     STL      RETADR             141009
 15    2012    CLOOP     JSUB     RDREC              48203D
 20    2015              LDA      LENGTH             00100C
 25    2018              COMP     ZERO               281006
 30    201B              JEQ      ENDFIL             302024
 35    201E              JSUB     WRREC              482062
 40    2021              J        CLOOP              302012
 45    2024    ENDFIL    LDA      EOF                001000
 50    2027              STA      BUFFER             0C100F
 55    202A              LDA      THREE              001003
 60    202D              STA      LENGTH             0C100C
 65    2030              JSUB     WRREC              482062
 70    2033              LDL      RETADR             081009
 75    2036              RSUB                        4C0000
110            .
115            .         SUBROUTINE TO READ RECORD INTO BUFFER
120            .
121    2039    INPUT     BYTE     X'F1'              F1
122    203A    MAXLEN    WORD     4096               001000
124            .
125    203D    RDREC     LDX      ZERO               041006
130    2040              LDA      ZERO               001006
135    2043    RLOOP     TD       INPUT              E02039
140    2046              JEQ      RLOOP              302043
145    2049              RD       INPUT              D82039
150    204C              COMP     ZERO               281006
155    204F              JEQ      EXIT               30205B
160    2052              STCH     BUFFER,X           54900F
165    2055              TIX      MAXLEN             2C203A
170    2058              JLT      RLOOP              382043
175    205B    EXIT      STX      LENGTH             10100C
180    205E              RSUB                        4C0000
195            .
200            .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205            .
206    2061    OUTPUT    BYTE     X'05'              05
207            .
210    2062    WRREC     LDX      ZERO               041006
215    2065    WLOOP     TD       OUTPUT             E02061
220    2068              JEQ      WLOOP              302065
225    206B              LDCH     BUFFER,X           50900F
230    206E              WD       OUTPUT             DC2061
235    2071              TIX      LENGTH             2C100C
240    2074              JLT      WLOOP              382065
245    2077              RSUB                        4C0000
255                      END      FIRST
```

Figure 2.18 Sample program for a one-pass assembler.

a system, a large fraction of the total workload consists of program translation. Because programs are re-assembled nearly every time they are run, efficiency of the assembly process is an important consideration. A load-and-go assembler avoids the overhead of writing the object program out and reading it back in. This can be accomplished with either a one- or a two-pass assembler. However, a one-pass assembler also avoids the overhead of an additional pass over the source program.

Because the object program is produced in memory rather than being written out on secondary storage, the handling of forward references becomes less difficult. The assembler simply generates object code instructions as it scans the source program. If an instruction operand is a symbol that has not yet been defined, the operand address is omitted when the instruction is assembled. The symbol used as an operand is entered into the symbol table (unless such an entry is already present). This entry is flagged to indicate that the symbol is undefined. The address of the operand field of the instruction that refers to the undefined symbol is added to a list of forward references associated with the symbol table entry. When the definition for a symbol is encountered, the forward reference list for that symbol is scanned (if one exists), and the proper address is inserted into any instructions previously generated.

An example should help to make this process clear. Figure 2.19(a) shows the object code and symbol table entries as they would be after scanning line 40 of the program in Fig. 2.18. The first forward reference occurred on line 15. Since the operand (RDREC) was not yet defined, the instruction was assembled with no value assigned as the operand address (denoted in the figure by ----). RDREC was then entered into SYMTAB as an undefined symbol (indicated by *); the address of the operand field of the instruction (2013) was inserted in a list associated with RDREC. A similar process was followed with the instructions on lines 30 and 35.

Now consider Fig. 2.19(b), which corresponds to the situation after scanning line 160. Some of the forward references have been resolved by this time, while others have been added. When the symbol ENDFIL was defined (line 45), the assembler placed its value in the SYMTAB entry; it then inserted this value into the instruction operand field (at address 201C) as directed by the forward reference list. From this point on, any references to ENDFIL would not be forward references, and would not be entered into a list. Similarly, the definition of RDREC (line 125) resulted in the filling in of the operand address at location 2013. Meanwhile, two new forward references have been added: to WRREC (line 65) and EXIT (line 155). You should continue tracing through this process to the end of the program to show yourself that all of the forward references will be filled in properly. At the end of the program, any SYMTAB entries that are still marked with * indicate undefined symbols. These should be flagged by the assembler as errors.

```
Memory
address                 Contents

1000     454F4600  00030000  00xxxxxx  xxxxxxxx
1010     xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
  •
  •
  •
2000     xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxx14
2010     100948--  --00100C  28100630  ------48--
2020     --3C2012
  •
  •
  •
```

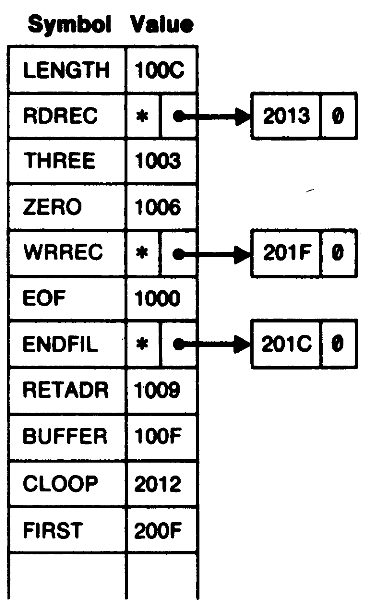


Figure 2.19(a) Object code in memory and symbol table entries for the program in Fig. 2.18 after scanning line 40.

When the end of the program is encountered, the assembly is complete. If no errors have occurred, the assembler searches SYMTAB for the value of the symbol named in the END statement (in this case, FIRST) and jumps to this location to begin execution of the assembled program. The algorithm for one pass assembler is shown in Fig. 2.19(c).

We used an absolute program as our example because, for a load-and-go assembler, the actual address must be known at assembly time. Of course it is not necessary for this address to be specified by the programmer; it might be assigned by the system. In either case, however, the assembly process would be the same—the location counter would be initialized to the actual program starting address.

One-pass assemblers that produce object programs as output are often used on systems where external working-storage devices (for the intermediate file between the two passes) are not available. Such assemblers may also be useful when the external storage is slow or is inconvenient to use for some other reason. One-pass assemblers that produce object programs follow a slightly different procedure from that previously described. Forward references are entered into lists as before. Now, however, when the definition of a symbol is encountered, instructions that made forward references to that

```
Memory
address                       Contents
1000      454F4600  00030000  00xxxxxx  xxxxxxxx
1010      xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
  •
  •
  •
2000      xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxx14
2010      10094820  3D00100C  28100630  202448——
2020      ——3C2012  0010000C  100F0010  030C100C
2030      48————08  10094C00  00F10010  00041006
2040      001006E0  20393020  43D82039  28100630
2050      ————5490  0F
  •
  •
  •
```

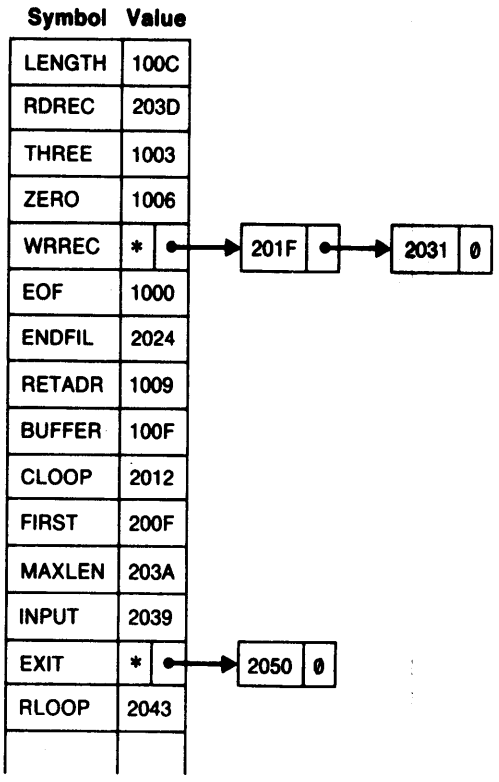


Figure 2.19(b) Object code in memory and symbol table entries for the program in Fig. 2.18 after scanning line 160.

symbol may no longer be available in memory for modification. In general, they will already have been written out as part of a Text record in the object program. In this case the assembler must generate another Text record with the correct operand address. When the program is loaded, this address will be inserted into the instruction by the action of the loader.

Figure 2.20 illustrates this process. The second Text record contains the object code generated from lines 10 through 40 in Fig. 2.18. The operand addresses for the instructions on lines 15, 30, and 35 have been generated as 0000. When the definition of ENDFIL on line 45 is encountered, the assembler generates the third Text record. This record specifies that the value 2024 (the address of ENDFIL) is to be loaded at location 201C (the operand address field of the JEQ instruction on line 30). When the program is loaded, therefore, the value 2024 will replace the 0000 previously loaded. The other forward references in the program are handled in exactly the same way. In effect, the services of the loader are being used to complete forward

```
begin
  read first input line
  if OPCODE = 'START' then
    begin
      save #[OPERAND] as starting address
      initialize LOCCTR as starting address
      read next input line
    end {if START}
  else
    initialize LOCCTR to 0
while OPCODE ≠ 'END' do
    begin
      if there is not a comment line then
        begin
          if there is a symbol in the LABEL field then
            begin
              search SYMTAB for LABEL
                if found then
              begin
                  if symbol value as null
                  set symbol value as LOCCTR and search
                     the linked list with the corresponding
                     operand
                  PTR addresses and generate operand
                     addresses as corresponding symbol
                     values
                  set symbol value as LOCCTR in symbol
                     table and delete the linked list
              end
              else
                insert (LABEL, LOCCTR) into SYMTAB
            end
              search OPTAB for OPCODE
                if found then
                  begin
                     search SYMTAB for OPERAND address
                if found then
                  if symbol value not equal to null then
                    store symbol value as OPERAND address
                  else
                    insert at the end of the linked list
                      with a node with address as LOCCTR
                  else
                    insert (symbol name, null)
```

Figure 2.19(c) Algorithm for One pass assembler.

```
                  add 3 to LOCCTR
                end
                else if OPCODE = 'WORD' then
                  add 3 to LOCCTR & convert comment to
                    object code
                else if OPCODE = 'RESW' then
                  add 3 #[OPERAND] to LOCCTR
                else if OPCODE = 'RESB' then
                  add #[OPERAND] to LOCCTR
                else if OPCODE = 'BYTE' then
                  begin
                    find length of constant in bytes
                    add length to LOCCTR
                    convert constant to object code
                  end
              if object code will not fit into current
                text record then
                begin
                  write text record to object program
                  initialize new text record
                end
              add object code to Text record
            end
          write listing line
          read next input line
        end
      write last Text record to object program
      write End record to object program
      write last listing line
    end {Pass 1}
```

Figure 2.19(c) *(cont'd)*

references that could not be handled by the assembler. Of course, the object program records must be kept in their original order when they are presented to the loader.

In this section we considered only simple one-pass assemblers that handled absolute programs. Instruction operands were assumed to be single symbols, and the assembled instructions contained the actual (not relative) addresses of the operands. More advanced assembler features such as literals were not allowed. You are encouraged to think about ways of removing some of these restrictions (see the Exercises for this section for some suggestions).

```
H^COPY  ^001000^00107A
T^001000^09^454F46^000003^000000
T^00200F^15^141009^480000^00100C^281006^300000^480000^3C2012
T^00201C^02^2024
T^002024^19^001000^0C100F^001003^0C100C^480000^081009^4C0000^F1^001000
T^002013^02^203D
T^00203D^1E^041006^001006^E02039^302043^D82039^281006^300000^54900F^2C203A^382043
T^002050^02^205B
T^00205B^07^10100C^4C0000^05
T^00201F^02^2062
T^002031^02^2062
T^002062^18^041006^E02061^302065^50900F^DC2061^2C100C^382065^4C0000
E^00200F
```

Figure 2.20 Object program from one-pass assembler for program in Fig. 2.18.

2.4.2 Multi-Pass Assemblers

In our discussion of the EQU assembler directive, we required that any symbol used on the right-hand side (i.e., in the expression giving the value of the new symbol) be defined previously in the source program. A similar requirement was imposed for ORG. As a matter of fact, such a restriction is normally applied to all assembler directives that (directly or indirectly) define symbols.

The reason for this is the symbol definition process in a two-pass assembler. Consider, for example, the sequence

```
ALPHA   EQU    BETA
BETA    EQU    DELTA
DELTA   RESW   1
```

The symbol BETA cannot be assigned a value when it is encountered during the first pass because DELTA has not yet been defined. As a result, ALPHA cannot be evaluated during the second pass. This means that any assembler that makes only two sequential passes over the source program cannot resolve such a sequence of definitions.

Restrictions such as prohibiting forward references in symbol definition are not normally a serious inconvenience for the programmer. As a matter of fact, such forward references tend to create difficulty for a person reading the program as well as for the assembler. Nevertheless, some assemblers are designed to eliminate the need for such restrictions. The general solution is a multi-pass assembler that can make as many passes as are needed to process the definitions of symbols. It is not necessary for such an assembler to make more than two passes over the entire program. Instead, the portions of the program that involve forward references in symbol definition are saved during Pass 1. Additional passes through these stored definitions are made as the assembly progresses. This process is followed by a normal Pass 2.

There are several ways of accomplishing the task outlined above. The method we describe involves storing those symbol definitions that involve forward references in the symbol table. This table also indicates which symbols are dependent on the values of others, to facilitate symbol evaluation.

Figure 2.21(a) shows a sequence of symbol-defining statements that involve forward references; the other parts of the source program are not important for our discussion, and have been omitted. The following parts of Fig. 2.21 show information in the symbol table as it might appear after processing each of the source statements shown.

Figure 2.21(b) displays symbol table entries resulting from Pass 1 processing of the statement

```
HALFSZ   EQU   MAXLEN/2
```

MAXLEN has not yet been defined, so no value for HALFSZ can be computed. The defining expression for HALFSZ is stored in the symbol table in place of its value. The entry &1 indicates that one symbol in the defining expression is undefined. In an actual implementation, of course, this definition might be stored at some other location. SYMTAB would then simply contain a pointer to the defining expression. The symbol MAXLEN is also entered in the symbol table, with the flag * identifying it as undefined. Associated with this entry is a list of the symbols whose values depend on MAXLEN (in this case, HALFSZ). (Note the similarity to the way we handled forward references in a one-pass assembler.)

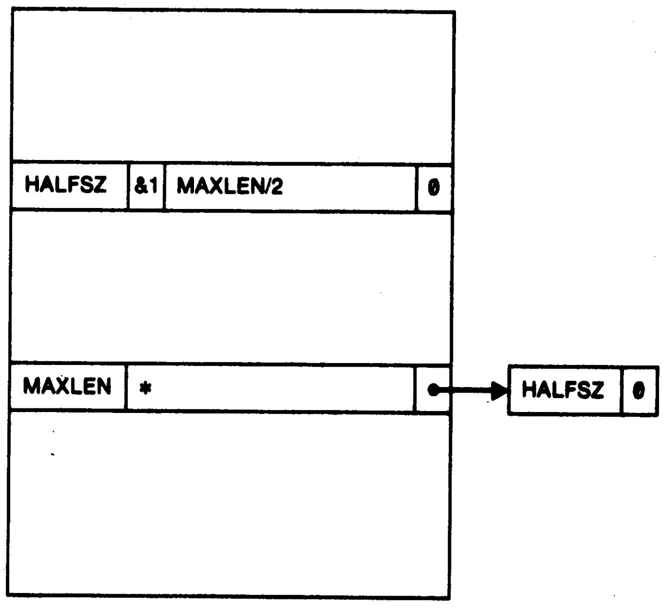
The same procedure is followed with the definition of MAXLEN [see Fig. 2.21(c)]. In this case there are two undefined symbols involved in the definition: BUFEND and BUFFER. Both of these are entered into SYMTAB with lists indicating the dependence of MAXLEN upon them. Similarly, the definition of PREVBT causes this symbol to be added to the list of dependencies on BUFFER [as shown in Fig. 2.21(d)].

So far we have simply been saving symbol definitions for later processing. The definition of BUFFER on line 4 lets us begin evaluation of some of these symbols. Let us assume that when line 4 is read, the location counter contains the hexadecimal value 1034. This address is stored as the value of BUFFER. The assembler then examines the list of symbols that are dependent on BUFFER. The symbol table entry for the first symbol in this list (MAXLEN) shows that it depends on two currently undefined symbols; therefore, MAXLEN cannot be evaluated immediately. Instead, the &2 is changed to &1 to show that only one symbol in the definition (BUFEND) remains undefined. The other symbol in the list (PREVBT) can be evaluated because it depends only on BUFFER. The value of the defining expression for PREVBT is calculated and stored in SYMTAB. The result is shown in Fig. 2.21(e).

The remainder of the processing follows the same pattern. When BUFEND is defined by line 5, its value is entered into the symbol table. The list associated with BUFEND then directs the assembler to evaluate MAXLEN, and

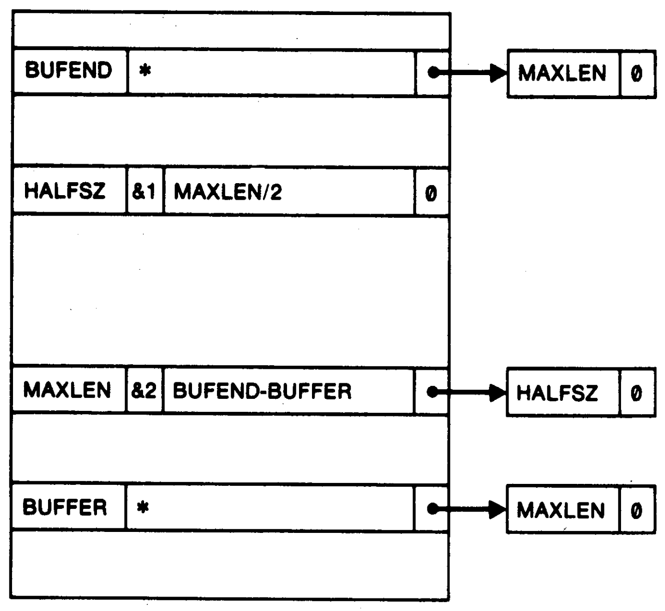
```
1    HALFSZ    EQU     MAXLEN/2
2    MAXLEN    EQU     BUFEND-BUFFER
3    PREVBT    EQU     BUFFER-1
                       .
                       .
                       .
4    BUFFER    RESB    4096
5    BUFEND    EQU     *
```

(a)

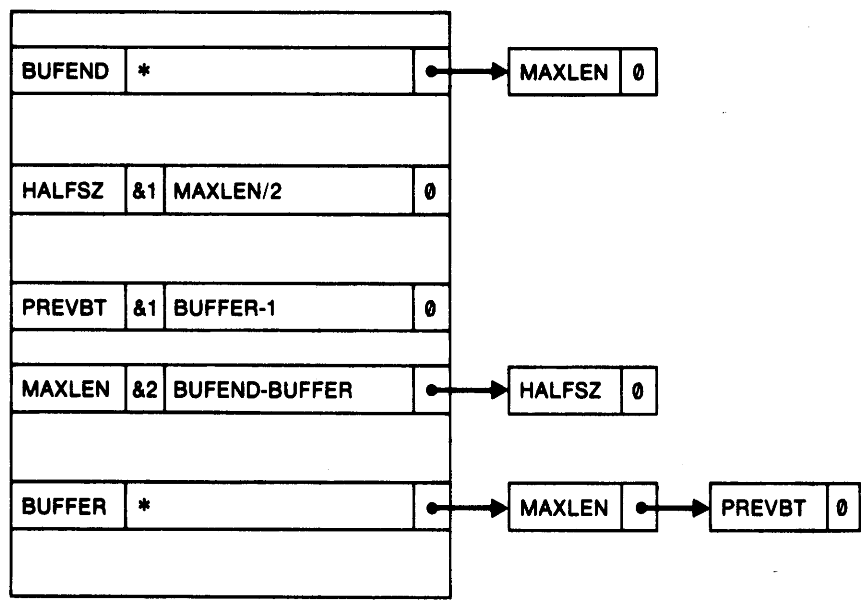


(b)

Figure 2.21 Example of multi-pass assembler operation.

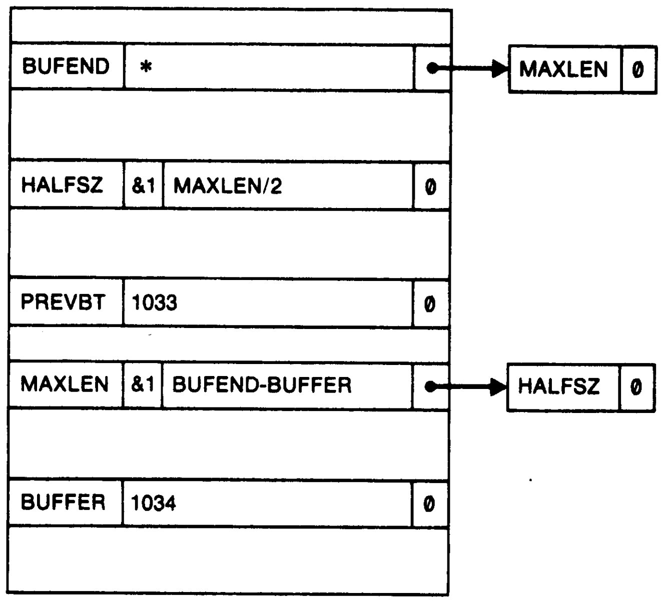


(c)

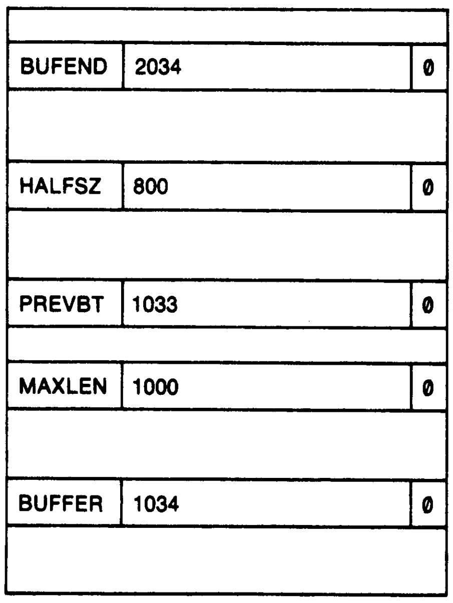


(d)

Figure 2.21 (*cont'd*)



(e)



(f)

Figure 2.21 (*cont'd*)

entering a value for MAXLEN causes the evaluation of the symbol in its list (HALFSZ). As shown in Fig. 2.21(f), this completes the symbol definition process. If any symbols remained undefined at the end of the program, the assembler would flag them as errors.

The procedure we have just described applies to symbols defined by assembler directives like EQU. You are encouraged to think about how this method could be modified to allow forward references in ORG statements as well.

2.5 IMPLEMENTATION EXAMPLES

We discussed many of the most common assembler features in the preceding sections. However, the variety of machines and assembler languages is very great. Most assemblers have at least some unusual features that are related to machine architecture or language design. In this section we discuss three examples of assemblers for real machines. We are obviously unable to give a full description of any of these in the space available. Instead we focus on some of the most interesting or unusual features of each assembler. We are also particularly interested in areas where the assembler design differs from the basic algorithm and data structures described earlier.

The assembler examples we discuss are for the Pentium (x86), SPARC, and PowerPC architectures. You may want to review the descriptions of these architectures in Chapter 1 before proceeding.

2.5.1 MASM Assembler

This section describes some of the features of the Microsoft MASM assembler for Pentium and other x86 systems. Further information about MASM can be found in Barkakati (1992).

As we discussed in Section 1.4.2, the programmer of an x86 system views memory as a collection of segments. An MASM assembler language program is written as a collection of segments. Each segment is defined as belonging to a particular class, corresponding to its contents. Commonly used classes are CODE, DATA, CONST, and STACK.

During program execution, segments are addressed via the x86 segment registers. In most cases, code segments are addressed using register CS, and stack segments are addressed using register SS. These segment registers are automatically set by the system loader when a program is loaded for execution. Register CS is set to indicate the segment that contains the starting label specified in the END statement of the program. Register SS is set to indicate the last stack segment processed by the loader.

Data segments (including constant segments) are normally addressed using DS, ES, FS, or GS. The segment register to be used can be specified explicitly by the programmer (by writing it as part of the assembler language instruction). If the programmer does not specify a segment register, one is selected by the assembler.

By default, the assembler assumes that all references to data segments use register DS. This assumption can be changed by the assembler directive ASSUME. For example, the directive

```
ASSUME  ES:DATASEG2
```

tells the assembler to assume that register ES indicates the segment DATASEG2. Thus, any references to labels that are defined in DATASEG2 will be assembled using register ES. It is also possible to collect several segments into a group and use ASSUME to associate a segment register with the group.

Registers DS, ES, FS and GS must be loaded by the program before they can be used to address data segments. For example, the instructions

```
MOV     AX,DATASEG2
MOV     ES,AX
```

would set ES to indicate the data segment DATASEG2. Notice the similarities between the ASSUME directive and the BASE directive we discussed for SIC/XE. The BASE directive tells a SIC/XE assembler the contents of register B; the programmer must provide executable instructions to load this value into the register. Likewise, ASSUME tells MASM the contents of a segment register; the programmer must provide instructions to load this register when the program is executed.

Jump instructions are assembled in two different ways, depending on whether the target of the jump is in the same code segment as the jump instruction. A *near jump* is a jump to a target in the same code segment; a *far jump* is a jump to a target in a different code segment. A near jump is assembled using the current code segment register CS. A far jump must be assembled using a different segment register, which is specified in an instruction prefix. The assembled machine instruction for a near jump occupies 2 or 3 bytes (depending upon whether the jump address is within 128 bytes of the current instruction). The assembled instruction for a far jump requires 5 bytes.

Forward references to labels in the source program can cause problems. For example, consider a jump instruction like

```
JMP     TARGET
```

If the definition of the label TARGET occurs in the program before the JMP instruction, the assembler can tell whether this is a near jump or a far jump.

However, if this is a forward reference to TARGET, the assembler does not know how many bytes to reserve for the instruction.

By default, MASM assumes that a forward jump is a near jump. If the target of the jump is in another code segment, the programmer must warn the assembler by writing

```
        JMP      FAR PTR TARGET
```

If the jump address is within 128 bytes of the current instruction, the programmer can specify the shorter (2-byte) near jump by writing

```
        JMP      SHORT TARGET
```

If the JMP to TARGET is a far jump, and the programmer does not specify FAR PTR, a problem occurs. During Pass 1, the assembler reserves 3 bytes for the jump instruction. However, the actual assembled instruction requires 5 bytes. In the earlier versions of MASM, this caused an assembly error (called a phase error). In later versions of MASM, the assembler can repeat Pass 1 to generate the correct location counter values.

Notice the similarities between the far jump and the forward references in SIC/XE that require the use of extended format instructions.

There are also many other situations in which the length of an assembled instruction depends on the operands that are used. For example, the operands of an ADD instruction may be registers, memory locations, or immediate operands. Immediate operands may occupy from 1 to 4 bytes in the instruction. An operand that specifies a memory location may take varying amounts of space in the instruction, depending upon the location of the operand.

This means that Pass 1 of an x86 assembler must be considerably more complex than Pass 1 of a SIC assembler. The first pass of the x86 assembler must analyze the operands of an instruction, in addition to looking at the operation code. The operation code table must also be more complicated, since it must contain information on which addressing modes are valid for each operand.

Segments in an MASM source program can be written in more than one part. If a SEGMENT directive specifies the same name as a previously defined segment, it is considered to be a continuation of that segment. All of the parts of a segment are gathered together by the assembly process. Thus, segments can perform a similar function to the program blocks we discussed for SIC/XE.

References between segments that are assembled together are automatically handled by the assembler. External references between separately assembled modules must be handled by the linker. The MASM directive PUBLIC has approximately the same function as the SIC/XE directive EXTDEF. The MASM directive EXTRN has approximately the same function as EXTREF. We will consider the action of the linker in more detail in the next chapter.

The object program from MASM may be in several different formats, to allow easy and efficient execution of the program in a variety of operating environments. MASM can also produce an instruction timing listing that shows the number of clock cycles required to execute each machine instruction. This allows the programmer to exercise a great deal of control in optimizing timing-critical sections of code.

2.5.2 SPARC Assembler

This section describes some of the features of the SunOS SPARC assembler. Further information about this assembler can be found in Sun Microsystems (1994a).

A SPARC assembler language program is divided into units called *sections.* The assembler provides a set of predefined section names. Some examples of these are

.TEXT	Executable instructions
.DATA	Initialized read/write data
.RODATA	Read-only data
.BSS	Uninitialized data areas

I is also possible to define other sections, specifying section attributes such as "executable" and "writeable."

The programmer can switch between sections at any time in the source program by using assembler directives. The assembler maintains a separate location counter for each named section. Each time the assembler switches to a different section, it also switches to the location counter associated with that section. In this way, sections are similar to the program blocks we discussed for SIC. However, references between different sections are resolved by the linker, not by the assembler.

By default, symbols used in a source program are assumed to be local to that program. (However, a section may freely refer to local symbols defined in another section of the same program.) Symbols that are used in linking separately assembled programs may be declared to be either *global* or *weak.* A global symbol is either a symbol that is defined in the program and made accessible to others, or a symbol that is referenced in a program and defined externally. (Notice that this combines the functions of the EXTDEF and EXTREF directives we discussed for SIC.) A weak symbol is similar to a global symbol. However, the definition of a weak symbol may be overridden by a global symbol with the same name. Also, weak symbols may remain undefined when the program is linked, without causing an error.

The object file written by the SPARC assembler contains translated versions of the segments of the program and a list of relocation and linking operations that need to be performed. References between different segments of the same program are resolved when the program is linked. The object program also includes a symbol table that describes the symbols used during relocation and linking (global symbols, weak symbols, and section names).

SPARC assembler language has an unusual feature that is directly related to the machine architecture. As we discussed in Section 1.5.1, SPARC branch instructions (including subroutine calls) are *delayed branches*. The instruction immediately following a branch instruction is actually executed before the branch is taken. For example, in the instruction sequence

```
        CMP     %L0, 10
        BLE     LOOP
        ADD     %L2, %L3, %L4
```

the ADD instruction is executed *before* the conditional branch BLE. This ADD instruction is said to be in the *delay slot* of the branch; it is executed regardless of whether or not the conditional branch is taken.

To simplify debugging, SPARC assembly language programmers often place NOP (no-operation) instructions in delay slots when a program is written. The code is later rearranged to move useful instructions into the delay slots. For example, the instruction sequence illustrated above might originally have been

```
LOOP:   .
        .
        .
        ADD     %L2, %L3, %L4
        CMP     %L0, 10
        BLE     LOOP
        NOP
```

Moving the ADD instruction into the delay slot would produce the version discussed earlier. (Notice that the CMP instruction could not be moved into the delay slot, because it sets the condition codes that must be tested by the BLE.)

However, there is another possibility. Suppose that the original version of the loop had been

```
LOOP:   ADD     %L2, %L3, %L4
        .
        .
        CMP     %L0, 10
        BLE     LOOP
        NOP
```

Now the ADD instruction is logically the *first* instruction in the loop. It could still be moved into the delay slot, as previously described. However, this would create a problem. On the last execution of the loop, the ADD instruction (which is the beginning of the next loop iteration) should not be executed.

The SPARC architecture defines a solution to this problem. A conditional branch instruction like BLE can be *annulled.* If a branch is annulled, the instruction in its delay slot is executed if the branch is taken, but *not* executed if the branch is not taken. Annulled branches are indicated in SPARC assembler language by writing ",A" following the operation code. Thus the loop just discussed could be rewritten as

```
LOOP: .
      .
      .
      CMP       %L0, 10
      BLE,A     LOOP
      ADD       %L2, %L3, %L4
```

The SPARC assembler provides warning messages to alert the programmer to possible problems with delay slots. For example, a label on an instruction in a delay slot usually indicates an error. A segment that ends with a branch instruction (with nothing in the delay slot) is also likely to be incorrect. Before the branch is executed, the machine will attempt to execute whatever happens to be stored at the memory location immediately following the branch.

2.5.3 AIX Assembler

This section describes some of the features of the AIX assembler for PowerPC and other similar systems. Further information about this assembler can be found in IBM (1994b).

The AIX assembler includes support for various models of PowerPC microprocessors, as well as earlier machines that implement the original POWER architecture. The programmer can declare which architecture is being used with the assembler directive .MACHINE. The assembler automatically checks for POWER or PowerPC instructions that are not valid for the specified environment. When the object program is generated, the assembler includes a flag that indicates which processors are capable of running the program. This flag depends on which instructions are actually used in the program, not on the .MACHINE directive. For example, a PowerPC program that contains only instructions that are also in the original POWER architecture would be executable on either type of system.

As we discussed in Section 1.5.2, PowerPC load and store instructions use a base register and a displacement value to specify an address in memory. Any of the general-purpose registers (except GPR0) can be used as a base register. Decisions about which registers to use in this way are left to the programmer. In a long program, it is not unusual to have several different base registers in use at the same time. The programmer specifies which registers are available for use as base registers, and the contents of these registers, with the .USING assembler directive. This is similar in function to the BASE statement in our SIC/XE assembler language. Thus the statements

```
.USING      LENGTH,1
.USING      BUFFER,4
```

would identify GPR1 and GPR4 as base registers. GPR1 would be assumed to contain the address of LENGTH, and GPR4 would be assumed to contain the address of BUFFER. As with SIC/XE, the programmer must provide instructions to place these values into the registers at execution time. Additional .USING statements may appear at any point in the program. If a base register is to be used later for some other purpose, the programmer indicates with the .DROP statement that this register is no longer available for addressing purposes.

This additional flexibility in register usage means more work for the assembler. A *base register table* is used to remember which of the general-purpose registers are currently available as base registers, and what base addresses they contain. Processing a .USING statement causes an entry to be made in this table (or an existing entry to be modified); processing a .DROP statement removes the corresponding table entry. For each instruction whose operand is an address in memory, the assembler scans the table to find a base register that can be used to address that operand. If more than one register can be used, the assembler selects the base register that results in the smallest signed displacement. If no suitable base register is available, the instruction cannot be assembled. The process of displacement calculation is the same as we described for SIC/XE.

The AIX assembler language also allows the programmer to write base registers and displacements explicitly in the source program. For example, the instruction

```
L    2,8(4)
```

specifies an operand address that is 8 bytes past the address contained in GPR4. This form of addressing may be useful when some register is known to contain the starting address of a table or data record, and the programmer wishes to refer to a fixed location within that table or record. The assembler simply inserts the specified values into the object code instruction: in this case base register GPR4 and displacement 8. The base register table is not involved, and the register used in this way need not have appeared in a .USING statement.