Chapter 2 – Assemblers

2.1 Basic Assembler Functions

- To show different assembler features, Fig 2.1 (Page 45) shows an assembler language program for the basic version of SIC.

```
Source statement

COPY     START   1000          COPY FILE FROM INPUT TO OUTPUT
FIRST    STL     RETADR        SAVE RETURN ADDRESS
CLOOP    JSUB    RDREC         READ INPUT RECORD
         LDA     LENGTH        TEST FOR EOF (LENGTH = 0)
         COMP    ZERO
         JEQ     ENDFIL        EXIT IF EOF FOUND
         JSUB    WRREC         WRITE OUTPUT RECORD
         J       CLOOP         LOOP
ENDFIL   LDA     EOF           INSERT END OF FILE MARKER
         STA     BUFFER
         LDA     THREE         SET LENGTH = 3
         STA     LENGTH
         JSUB    WRREC         WRITE EOF
         LDL     RETADR        GET RETURN ADDRESS
         RSUB                  RETURN TO CALLER
EOF      BYTE    C'EOF'
THREE    WORD    3
ZERO     WORD    0
RETADR   RESW    1
LENGTH   RESW    1             LENGTH OF RECORD
BUFFER   RESB    4096          4096-BYTE BUFFER AREA
.
.        SUBROUTINE TO READ RECORD INTO BUFFER
.
RDREC    LDX     ZERO          CLEAR LOOP COUNTER
         LDA     ZERO          CLEAR A TO ZERO
RLOOP    TD      INPUT         TEST INPUT DEVICE
         JEQ     RLOOP         LOOP UNTIL READY
         RD      INPUT         READ CHARACTER INTO REGISTER A
         COMP    ZERO          TEST FOR END OF RECORD (X'00')
         JEQ     EXIT          EXIT LOOP IF EOR
         STCH    BUFFER,X      STORE CHARACTER IN BUFFER
         TIX     MAXLEN        LOOP UNLESS MAX LENGTH
         JLT     RLOOP           HAS BEEN REACHED
EXIT     STX     LENGTH        SAVE RECORD LENGTH
         RSUB                  RETURN TO CALLER
INPUT    BYTE    X'F1'         CODE FOR INPUT DEVICE
MAXLEN   WORD    4096
.
.        SUBROUTINE TO WRITE RECORD FROM BUFFER
.
WRREC    LDX     ZERO          CLEAR LOOP COUNTER
WLOOP    TD      OUTPUT        TEST OUTPUT DEVICE
         JEQ     WLOOP         LOOP UNTIL READY
         LDCH    BUFFER,X      GET CHARACTER FROM BUFFER
         WD      OUTPUT        WRITE CHARACTER
         TIX     LENGTH        LOOP UNTIL ALL CHARACTERS
         JLT     WLOOP           HAVE BEEN WRITTEN
         RSUB                  RETURN TO CALLER
OUTPUT   BYTE    X'05'         CODE FOR OUTPUT DEVICE
         END     FIRST
```

Figure 2.1 Example of a SIC assembler language program.

- The mnemonic instructions used are those introduced in Section 1.3.1 and Appendix A.
- The following assembler directives are used in the program:
  1) START – Specify name and starting address for the program;
  2) END – Indicate the end of the source program and (optionally) specify the first executable instruction in the program;
  3) BYTE – Generate character or hexadecimal constant, occupying as many bytes as needed to represent the constant;
  4) WORD – Generate one-word integer constant;
  5) RESB – Reserve the indicated number of bytes for a data area;
  6) RESW – Reserve the indicated number of words for a data area.

2.1.1 A Simple SIC Assembler

- Fig 2.2 (Page 47) shows the same program as in Fig 2.1, with the generated object code for each statement.

Line	Loc	Source statement			Object code
5	1000	COPY	START	1000	
10	1000	FIRST	STL	RETADR	141033
15	1003	CLOOP	JSUB	RDREC	482039
20	1006		LDA	LENGTH	001036
25	1009		COMP	ZERO	281030
30	100C		JEQ	ENDFIL	301015
35	100F		JSUB	WRREC	482061
40	1012		J	CLOOP	3C1003
45	1015	ENDFIL	LDA	EOF	00102A
50	1018		STA	BUFFER	0C1039
55	101B		LDA	THREE	00102D
60	101E		STA	LENGTH	0C1036
65	1021		JSUB	WRREC	482061
70	1024		LDL	RETADR	081033
75	1027		RSUB		4C0000
80	102A	EOF	BYTE	C'EOF'	454F46
85	102D	THREE	WORD	3	000003
90	1030	ZERO	WORD	0	000000
95	1033	RETADR	RESW	1	
100	1036	LENGTH	RESW	1	
105	1039	BUFFER	RESB	4096	
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
125	2039	RDREC	LDX	ZERO	041030
130	203C		LDA	ZERO	001030
135	203F	RLOOP	TD	INPUT	E0205D
140	2042		JEQ	RLOOP	30203F
145	2045		RD	INPUT	D8205D
150	2048		COMP	ZERO	281030
155	204B		JEQ	EXIT	302057
160	204E		STCH	BUFFER,X	549039
165	2051		TIX	MAXLEN	2C205E
170	2054		JLT	RLOOP	38203F
175	2057	EXIT	STX	LENGTH	101036
180	205A		RSUB		4C0000
185	205D	INPUT	BYTE	X'F1'	F1
190	205E	MAXLEN	WORD	4096	001000
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
210	2061	WRREC	LDX	ZERO	041030
215	2064	WLOOP	TD	OUTPUT	E02079
220	2067		JEQ	WLOOP	302064
225	206A		LDCH	BUFFER,X	509039
230	206D		WD	OUTPUT	DC2079
235	2070		TIX	LENGTH	2C1036
240	2073		JLT	WLOOP	382064
245	2076		RSUB		4C0000
250	2079	OUTPUT	BYTE	X'05'	05
255			END	FIRST	

Figure 2.2 Program from Fig. 2.1 with object code.

- The translation of source program to object code requires us to accomplish the following functions:

  1) Convert mnemonic operation codes to their machine language equivalents – e.g., translates STL to 14 (line 10);

2) Convert symbolic operands to their equivalent machine addresses – e.g., translate RETADR to 1033 (line 10);

3) Build the machine instructions in the proper format;

4) Convert the data constants specified in the source program into their internal machine representations – e.g., translate EOF to 454F46 (line 80);

5) Write the object program and the assembly listing.

- Considering the statement of line 10, this instruction contains a *forward reference* – that is, a reference to a label (RETADR) that is defined later in the program.
- If we attempt to translate the program line by line, we will be unable to process this statement because we do not know the address that will be assigned to RETADR.
- Because of this, most assemblers make *two passes* over the source program.
- The *first pass* scans the source program for label definitions and assigns addresses.
- The *second pass* performs most of the actual translation.
- In addition to translating the instructions of the source program, the assembler must process statements called *assembler directives* (or *pseudo-instructions*).
- The assembler must write the generated object code onto some output device. This *object program* will later be loaded into memory for execution.
- The simple *object program* format contains three types of records: *Header*, *Text*, and *End*.
- The content of each record: shown at the bottom of Page 48 and at the top of Page 49.

Header record:

Col. 1	H
Col. 2–7	Program name
Col. 8–13	Starting address of object program (hexadecimal)
Col. 14–19	Length of object program in bytes (hexadecimal)

Text record:

Col. 1	T
Col. 2–7	Starting address for object code in this record(hexadecimal)
Col. 8–9	Length of object code in this record in bytes (hexadecimal)
Col. 10–69	Object code, represented in hexadecimal (2 columns per byte of object code)

- Fig 2.3 (Page 49) shows the object program corresponding to Fig 2.2, using this format.

```
HCOPY  ^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
```

Figure 2.3 Object program corresponding to Fig. 2.2.

- A general description of the functions of the two-pass assembler: see the top of Page 50.

2.1.2 Assembler Algorithm and Data Structures

- The simple assembler uses two major internal data structures: the Operation Code Table (OPTAB) and the Symbol Table (SYMTAB).

- **OPTAB** must contain (at least) the *mnemonic operation code* and its *machine language equivalent*. In more complex assemblers, this table also contains information about *instruction format* and *length*.

- During Pass 1, OPTAB is used to look up and validate operation codes in the source program.
- In Pass 2, it is used to translate the operation codes to machine language.

Pass 1 (define symbols):

1. Assign addresses to all statements in the program.
2. Save the values (addresses) assigned to all labels for use in Pass 2.
3. Perform some processing of assembler directives. (This includes processing that affects address assignment, such as determining the length of data areas defined by BYTE, RESW, etc.)

Pass 2 (assemble instructions and generate object program):

1. Assemble instructions (translating operation codes and looking up addresses).
2. Generate data values defined by BYTE, WORD, etc.
3. Perform processing of assembler directives not done during Pass 1.
4. Write the object program and the assembly listing.

- OPTAB is usually organized as a hash table, with mnemonic operation code as the key. In most cases, OPTAB is a static table – that is, entries are not normally added to or deleted from it.
- **SYMTAB** includes the *name and value (address) for each label* in the source program, together with *flags to indicate error condition* (e.g., a symbol defined in two different places).
- During Pass 1, labels are entered into SYMTAB as they are encountered in the source program, along with their assigned addresses (from LOCCTR).
- During Pass 2, symbols used as operands are looked up in SYMTAB to obtain the addresses to be inserted in the assembled instruction.
- SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval.
- A Location Counter (**LOCCTR**) is used to be a variable

and help in the assignment of addresses. Whenever a label in the source program is read, the current value of LOCCTR gives the address to be associated with that label.

- There is certain information (such as *location counter values* and *error flags for statements*) that can or should be communicated between the two passes. For this reason, Pass 1 usually writes an *inter-mediate file* that contains each source statement together with its assigned address, error indicators, etc. This file is used as the input to Pass 2.
- Figures 2.4 (a) and (b) (Page 53~54) show the logic flow of the two passes of our assembler.

Pass 1:

```
begin
   read first input line
   if OPCODE = 'START' then
      begin
         save #[OPERAND] as starting address
         initialize LOCCTR to starting address
         write line to intermediate file
         read next input line
      end {if START}
   else
      initialize LOCCTR to 0
   while OPCODE ≠ 'END' do
      begin
         if this 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
                        set error flag (duplicate symbol)
                     else
                        insert (LABEL,LOCCTR) into SYMTAB
                  end {if symbol}
               search OPTAB for OPCODE
               if found then
                  add 3 (instruction length) to LOCCTR
               else if OPCODE = 'WORD' then
                  add 3 to LOCCTR
               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
                  end {if BYTE}
               else
                  set error flag (invalid operation code)
            end {if not a comment}
         write line to intermediate file
         read next input line
      end {while not END}
   write last line to intermediate file
   save (LOCCTR - starting address) as program length
end {Pass 1}
```

Figure 2.4(a) Algorithm for Pass 1 of assembler.

Pass 2:

```
begin
   read first input line {from intermediate file}
   if OPCODE = 'START' then
      begin
         write listing line
         read next input line
      end {if START}
   write Header record to object program
   initialize first Text record
   while OPCODE ≠ 'END' do
      begin
         if this is not a comment line then
            begin
               search OPTAB for OPCODE
               if found then
                  begin
                     if there is a symbol in OPERAND field then
                        begin
                           search SYMTAB for OPERAND
                           if found then
                              store symbol value as operand address
                           else
                              begin
                                 store 0 as operand address
                                 set error flag (undefined symbol)
                              end
                        end {if symbol}
                     else
                        store 0 as operand address
                     assemble the object code instruction
                  end {if opcode found}
               else if OPCODE = 'BYTE' or 'WORD' then
                  convert constant to object code
               if object code will not fit into the current Text record then
                  begin
                     write Text record to object program
                     initialize new Text record
                  end
               add object code to Text record
            end {if not comment}
         write listing line
         read next input line
      end {while not END}
   write last Text record to object program
   write End record to object program
   write last listing line
end {Pass 2}
```

Figure 2.4(b) Algorithm for Pass 2 of assembler.

- The source lines input to this algorithm is assumed in a fixed format with fields LABEL, OPCODE, and OPERAND. If one of these fields contains a character string that represents a number, we denote its numeric value with the prefix # (for example, #[OPERAND]).

2.2 Machine-Dependent Assembler Features

- Fig 2.5 shows the example program from Fig 2.1 by SIC/XE instruction set.

```
Line     Source statement

  5    COPY     START   0          COPY FILE FROM INPUT TO OUTPUT
 10    FIRST    STL     RETADR     SAVE RETURN ADDRESS
 12             LDB     #LENGTH    ESTABLISH BASE REGISTER
 13             BASE    LENGTH
 15    CLOOP   +JSUB    RDREC      READ INPUT RECORD
 20             LDA     LENGTH     TEST FOR EOF (LENGTH = 0)
 25             COMP    #0
 30             JEQ     ENDFIL     EXIT IF EOF FOUND
 35            +JSUB    WRREC      WRITE OUTPUT RECORD
 40             J       CLOOP      LOOP
 45    ENDFIL   LDA     EOF        INSERT END OF FILE MARKER
 50             STA     BUFFER
 55             LDA     #3         SET LENGTH = 3
 60             STA     LENGTH
 65            +JSUB    WRREC      WRITE EOF
 70             J       @RETADR    RETURN TO CALLER
 80    EOF      BYTE    C'EOF'
 95    RETADR   RESW    1
100    LENGTH   RESW    1          LENGTH OF RECORD
105    BUFFER   RESB    4096       4096-BYTE BUFFER AREA
110    .
115    .        SUBROUTINE TO READ RECORD INTO BUFFER
120    .
125    RDREC    CLEAR   X          CLEAR LOOP COUNTER
130             CLEAR   A          CLEAR A TO ZERO
132             CLEAR   S          CLEAR S TO ZERO
133            +LDT     #4096
135    RLOOP    TD      INPUT      TEST INPUT DEVICE
140             JEQ     RLOOP      LOOP UNTIL READY
145             RD      INPUT      READ CHARACTER INTO REGISTER A
150             COMPR   A,S        TEST FOR END OF RECORD (X'00')
155             JEQ     EXIT       EXIT LOOP IF EOR
160             STCH    BUFFER,X   STORE CHARACTER IN BUFFER
165             TIXR    T          LOOP UNLESS MAX LENGTH
170             JLT     RLOOP        HAS BEEN REACHED
175    EXIT     STX     LENGTH     SAVE RECORD LENGTH
180             RSUB               RETURN TO CALLER
185    INPUT    BYTE    X'F1'      CODE FOR INPUT DEVICE
195    .
200    .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205    .
210    WRREC    CLEAR   X          CLEAR LOOP COUNTER
212             LDT     LENGTH
215    WLOOP    TD      OUTPUT     TEST OUTPUT DEVICE
220             JEQ     WLOOP      LOOP UNTIL READY
225             LDCH    BUFFER,X   GET CHARACTER FROM BUFFER
230             WD      OUTPUT     WRITE CHARACTER
235             TIXR    T          LOOP UNTIL ALL CHARACTERS
240             JLT     WLOOP        HAVE BEEN WRITTEN
245             RSUB               RETURN TO CALLER
250    OUTPUT   BYTE    X'05'      CODE FOR OUTPUT DEVICE
255             END     FIRST
```

Figure 2.5 Example of a SIC/XE program.

- *Prefix* to operands: @ - indirect addressing; # - immediate operands; + - extended instruction format.
- Instructions that refer to memory are normally assembled using either the *program-counter relative* or the *base relative* mode. The assemble directive BASE (Fig 2.5, line 13) is used in conjunction with base relative addressing.
- The main differences between Fig 2.5 (SIC/XE) and Fig 2.1 (SIC) involve the use of *register-to-register* instructions (lines 150, 165). In addition, *immediate* addressing and *indirect* addressing have been used as much as possible (lines 25, 55, and 70).

2.2.1 Instruction Formats and Addressing Modes

- Fig 2.6 shows the object code generated for each statement in the program of Fig 2.5.

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 2.6 Program from Fig. 2.5 with object code.

- Key points of this subsection: the translation of the source program, and the handling of different *instruction formats* and different *addressing modes*.
- Note that the START statement (assembler directive) specifies a beginning program address of 0.

- Translation of *register-to-register instructions* (such as CLEAR – line 125, COMPR – line 150): The assembler must simply convert the mnemonic operation code to machine language (using OPTAB) and change each register mnemonic to its numeric equivalent.
- *Register-to-memory instructions*: assembled using either *program-counter relative* or *base relative addressing*; The assembler must, in either case, calculate a *displacement* to be assembled as part of the object instruction. Note that

  a) When the displacement is added to the contents of the program counter (PC) or the base register (B), the correct target address must be computed.

  b) The resulting displacement must be small enough to fit in the 12-bit field in the instruction. This means that the displacement must be between 0 and 4095 (for base relative mode) or between –2048 and +2047 (for program-counter relative mode).
- If *neither program-counter relative nor base relative addressing* can be used (because the displacements are too large), then the 4-byte extended instruction format (20-bit displacement) must be used.
- Example:

  15 0006 CLOOP +JSUB RDREC 4B10**1036**

  (bit ***e*** set to 1 to indicate extended instruction format)
- Note that programmer *must* specify the *extended format* by using the prefix + (line 15).

  If extended format is not specified, the assembler first attempts to translate the instruction using *program-counter relative* addressing.

If this is not possible (out of range), the assembler then attempts to use *base relative* addressing.

If neither form is applicable and the extended format is not specified, then the instruction cannot be properly assembled and the assembler must generate an error message.

- Example: the displacement calculation for program-counter relative and base relative addressing mode -

  A typical example of program-counter relative assembly:

  10 0000 FIRST STL RETADR 17202D

  1) Note that the program counter is advanced *after* each instruction is fetched and *before* it is executed.
  2) While STL is executed, PC will contain the address of the *next* instruction (0003), where RETADR (line 95) is assigned the address 0030.
  3) The *displacement* we need in the instruction is 30 – 3 = 2D, that is, *target address* = (PC) + disp = 3 + 2D = 30.
  4) Note that bit ***p*** = 1 to indicate PC relative addressing, making the last 2 bytes of the instruction 202D.

- Another example of PC relative addressing:

  40 0017 J CLOOP 3F2FEC

  The operand address (CLOOP=0006); during instruction execution, the PC=001A. Thus the displacement = 6 – 1A = -14 (using 2's complement for negative number in a 12-bit field = FEC).

- The displacement calculation process for *base relative addressing* is much the same as for *PC relative addressing*. The main difference is that *the assembler* *knows what the contents of the PC will be at execution time*. On the other hand, *the base register is under control of the programmer*.

- Therefore, the programmer must tell the assembler what the base register will contain during execution of the program so that the assembler can compute displacements. This is done in our example with the assembler directive BASE (line 13).
- In some case, the programmer can use another assembler directive NOBASE to inform the assembler that the contents of the base register can no longer be relied upon for addressing.
- Example for base relative assembly:

  160 104E STCH BUFFER,X 57C003

  1) According to the BASE statement, register B = 0033 (the address of LENGTH) during execution.
  2) The address BUFFER is 0036.
  3) Thus the displacement in the instruction must be 36-33=3.
  4) Note that bits ***x*** and ***b*** are set to 1 to indicate indexed and base relative addressing.
- *Immediate addressing* mode: the assembly of instruction with immediate addressing is to convert the immediate operand to its internal representation and insert it into the instruction. Example:

  55 0020 LDA #3 010003

  1) The operand stored in the instruction is 003.
  2) Bit ***i*** = 1 to indicate immediate addressing.
- Another example:

133 103C +LDT #4096 75101000

1) In this case, the operand (4096) is too large to fit into the 12-bit displacement field, so the extended instruction format is called for. (If the operand were too large even for this 20-bit address field, immediate addressing could not be used.)

- A different way of using immediate addressing is shown in the instruction

  12 0003 LDB #LENGTH 69202D

  1) The immediate operand is the *symbol* LENGTH.

  2) Since the *value* of this symbol is the *address* assigned to it, this immediate instruction has the effect of loading register B with the address of LENGTH.

  3) Note that we have combined PC relative addressing with immediate addressing. (PC = 0006, LENGTH = 0033, disp = 0033 – 0006 = 002D)

- The mixed usage of different address mode is allowed. For example, line 70 shows a statement that combines PC relative and indirect addressing.

2.2.2 Program Relocation

- The program we considered in Section 2.1 is an example of an *absolute program* (or absolute assembly). The program must be loaded at address 1000 (specified at assembly time) in order to execute properly.

- Example: 55 101B LDA THREE 00102D.
  In the object program (Fig 2.3), this statement is translated as 00102D, specifying that register A is to be loaded from memory address 102D.

- Suppose we attempt to load and execute the program at address 2000 instead of address 1000. If we do this, address 102D will not contain the value that we expect.

- In reality, the assembler does not know the *actual location* where the program will be loaded. However, *the assembler can identify* for the loader *those parts of the object program that need modification*. An object program that contains the information necessary to perform this kind of modification is called a *relocatable* program.

- Fig 2.7 shows different places (0000, 5000, 7420) for locating a program. For example, in the instruction "+JSUB RDREC", the address of RDREC is 1036(0000), 6036(5000), 8456(7420). *How to modify the address of RDREC according to different relocating address?*

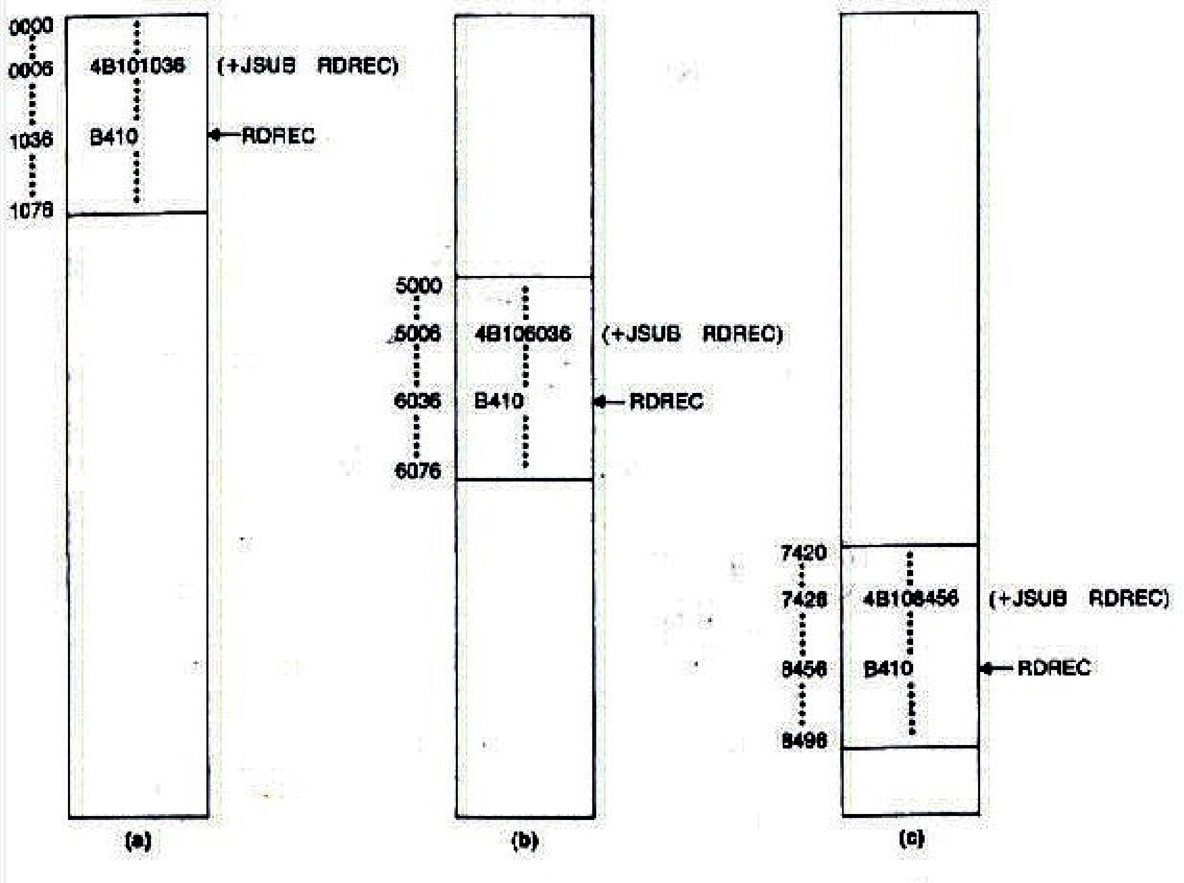


Figure 2.7 Examples of program relocation.

- The solution to the relocation problem:

  1) When the assembler generates the object code for JSUB instruction, it will insert the address of RDREC

relative to the start of the program. (This is the reason we initialized the location counter to 0 for the assembly.)

2) The assembler will also produce a command for the loader, instructing it to *add* the beginning address of the program to the address field in the JSUB instruction at load time.

- A modification record has the format shown in P.64.
- Note that the ***length*** field of a modification record is stored in half-bytes (rather than byte) because the address field to be modified may not occupy an integral number of bytes. For example, the address field in the +JSUB occupies 20 bits.
- The ***starting location*** field of a modification record is the location of the byte containing the leftmost bits of the address field to be modified. If this address field occupies an odd number of half-bytes, it is assumed to begin in the middle of the first byte at the starting location.
- Example: the modification record for the +JSUB instruction would be "M00000705". This record specifies that the beginning address of the program is to be added to a field that begins at address 000007 (relative to the start of the program) and is 5 half-bytes in length.

  Thus in the assembled instruction 4B101036, the first 12 bits (4B1) will remain unchanged. The ***program load address*** will be added to the last 20 bits (01036) to produce the correct operand address.
- In Fig 2.6, only lines 35 and 65 need to be relocated. The rest of the instructions in the program need not be modified when the program is loaded.

  In some cases, this is because the instruction operand is not a memory address at all (e.g., CLEAR R or LDA #3).

In other cases, no modification is needed because the operand is specified using PC relative or base relative addressing.

- Obviously, the only parts of the program that require modification at load time are those that specify ***direct*** (as opposed to *relative*) ***addresses***.
- Fig 2.8 shows the complete object program corresponding to the source program of Fig 2.5.

```
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
M^000014^05
M^000027^05
E^000000
```

Figure 2.8 Object program corresponding to Fig. 2.6.

2.3 Machine-Independent Assembler Features

- Key points of this section: *the implementation of literals within an assembler*, *two assembler directives* (EQU and ORG), *the use of expressions in assembler language*, *program blocks* and *control sections*.

2.3.1 Literals

- It is often convenient for the programmer to be able to write the value of a constant operand as a part of the instruction that uses it. The program in Fig 2.9 illustrates the use of literals and the object code generated for the statements of this program is shown in Fig 2.10.
- Note that a literal is identified with the prefix =, which followed by a specification of the literal value. Example:

  45 001A ENDFIL LDA =C'EOF' 032010

  specifies a 3-byte operand with value 'EOF'.
- It is important to understand the difference between a *literal* and *immediate* operand.
  1. With *immediate addressing*, the operand value is assembled as part of the machine instruction.
  2. With a *literal*, the assembler generates the specified value as a constant at some other memory location. The *address* of this generated constant is used as target address for the machine instruction. For instance, see line 45 and 55 in Fig 2.10 (P. 69).
- All of the literal operands used in a program are gathered together into one or more *literal pools*. Normally literals are placed into a pool at the *end* of the program. (See Fig 2.10)
- In some cases, it is desirable to place literals into a pool at some other location in the object program. To allow this,

we introduce the assembler directive LTORG (line 93 in Fig 2.9).

```
Line    Source statement

  5     COPY     START   0               COPY FILE FROM INPUT TO OUTPUT
 10     FIRST    STL     RETADR          SAVE RETURN ADDRESS
 13              LDB     #LENGTH         ESTABLISH BASE REGISTER
 14              BASE    LENGTH
 15     CLOOP   +JSUB    RDREC           READ INPUT RECORD
 20              LDA     LENGTH          TEST FOR EOF (LENGTH = 0)
 25              COMP    #0
 30              JEQ     ENDFIL          EXIT IF EOF FOUND
 35             +JSUB    WRREC           WRITE OUTPUT RECORD
 40              J       CLOOP           LOOP
 45     ENDFIL   LDA     =C'EOF'         INSERT END OF FILE MARKER
 50              STA     BUFFER
 55              LDA     #3              SET LENGTH = 3
 60              STA     LENGTH
 65             +JSUB    WRREC           WRITE EOF
 70              J       @RETADR         RETURN TO CALLER
 93              LTORG
 95     RETADR   RESW    1
100     LENGTH   RESW    1               LENGTH OF RECORD
105     BUFFER   RESB    4096            4096-BYTE BUFFER AREA
106     BUFEND   EQU     *
107     MAXLEN   EQU     BUFEND-BUFFER   MAXIMUM RECORD LENGTH
110     .
115     .        SUBROUTINE TO READ RECORD INTO BUFFER
120     .
125     RDREC    CLEAR   X               CLEAR LOOP COUNTER
130              CLEAR   A               CLEAR A TO ZERO
132              CLEAR   S               CLEAR S TO ZERO
133             +LDT     #MAXLEN
135     RLOOP    TD      INPUT           TEST INPUT DEVICE
140              JEQ     RLOOP           LOOP UNTIL READY
145              RD      INPUT           READ CHARACTER INTO REGISTER A
150              COMPR   A,S             TEST FOR END OF RECORD (X'00')
155              JEQ     EXIT            EXIT LOOP IF EOR
160              STCH    BUFFER,X        STORE CHARACTER IN BUFFER
165              TIXR    T               LOOP UNLESS MAX LENGTH
170              JLT     RLOOP             HAS BEEN REACHED
175     EXIT     STX     LENGTH          SAVE RECORD LENGTH
180              RSUB                    RETURN TO CALLER
185     INPUT    BYTE    X'F1'           CODE FOR INPUT DEVICE
195     .
200     .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205     .
210     WRREC    CLEAR   X               CLEAR LOOP COUNTER
212              LDT     LENGTH
215     WLOOP    TD      =X'05'          TEST OUTPUT DEVICE
220              JEQ     WLOOP           LOOP UNTIL READY
225              LDCH    BUFFER,X        GET CHARACTER FROM BUFFER
230              WD      =X'05'          WRITE CHARACTER
235              TIXR    T               LOOP UNTIL ALL CHARACTERS
240              JLT     WLOOP             HAVE BEEN WRITTEN
245              RSUB                    RETURN TO CALLER
255              END     FIRST
```

Figure 2.9 Program demonstrating additional assembler features.

1. When the assembler encounters a LTORG statement, it creates a literal pool that contains all of the literal operands used since the previous LTORG (or the beginning of the program).
2. This literal pool is placed in the object program at the location where the LTORG directive was encountered (Fig 2.10).

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
10	0000	FIRST	STL	RETADR	17202D
13	0003		LDB	#LENGTH	69202D
14			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	=C'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
93			LTORG		
	002D	*	=C'EOF'		454F46
95	0030	RETADR	RESW	1	
100	0033	LENGTH	RESW	1	
105	0036	BUFFER	RESB	4096	
106	1036	BUFEND	EQU	*	
107	1000	MAXLEN	EQU	BUFEND-BUFFER	
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	#MAXLEN	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 BUFFE		
205		.			
210	105D	WRREC	CLEAR	X	B410
212	105F		LDT	LENGTH	774000
215	1062	WLOOP	TD	=X'05'	E32011
220	1065		JEQ	WLOOP	332FFA
225	1068		LDCH	BUFFER,X	53C003
230	106B		WD	=X'05'	DF2008
235	106E		TIXR	T	B850
240	1070		JLT	WLOOP	3B2FEF
245	1073		RSUB		4F0000
255			END	FIRST	
	1076	*	=X'05'		05

Figure 2.10 Program from Fig. 2.9 with object code.

3. Of course, literals placed in a pool by LTORG will not be repeated in the pool at the end of the program.

- If we had not used the LTORG statement on line 93, the literal =C'EOF' would be placed in the pool at the end of the program.
- Most assemblers recognize duplicate literals – that is, the same literal used in more than one place in the program – and store only one copy of the specified data value. For example, the literal =X'05' is used in our program on lines 215 and 230.
- How to find the duplicate literals? The easiest way to recognize duplicate literals is by comparison of the character strings defining them (the string =X'05').
- The basic data structure that assembler handles literal operands is *literal table* LITTAB. For each literal used, this table contains the *literal name*, the *operand value* and *length*, and the *address assigned to the operand* when it is placed in a literal pool.
- LITTAB is often organized as a hash table, using the literal name or value as the key. During pass 1, the assembler searches LITTAB for the specified literal name (or value).

  If the literal is already present in the table, no action is needed.

  If it is not present, the literal is added to LITTAB (leaving the address unassigned).
- During pass 2, the operand address for use in generating object code is obtained by searching LITTAB for each literal operand encountered.

2.3.2 Symbol-Defining Statements

- The user-defined symbols in assembler language programs appear as *labels* on instructions or data areas. The *value* of such a label is the *address assigned to the statement* on which it appears.
- Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their value. The assembler directive generally used is EQU. The general form:

  symbol EQU value

  *This statement define the given symbol (enters it into SYMBOL) and assigns to it the value specified.
- One common use of EQU is to establish symbolic names that can be used for improved readability in place of numeric values.

  +LDT +4096 → MAXLEN EQU 4096

  +LDT #MAXLEN

  When the assembler encounters the EQU statement, it enters MAXLEN into SYMTAB (with value 4096).
- Another common use of EQU is in defining *mnemonic names* for *registers*. For example:

  A EQU 0
  X EQU 1
  L EQU 2

  These statements cause the symbols A, X, L, ,,, to be entered into SYMBOL with their corresponding values 0, 1, 2, …
- Another common assembler directive 'ORG': its form is

  ORG value

where *value* is a *constant* or an *expression* involving constants and previously defined symbols.

- When this statement is encountered during assembly of a program, the assembler resets its *location counter* (LOCCTR) to the specified value. Since the values of symbols are taken from LOCCTR, the ORG statement will affect the values of all labels defined until the next ORG.

- **Example**: To define a table STAB, the content of the table is as follows:

  SYMBOL field – 6-byte, VALUE field – one-word, FLAGS field – 2-byte.

Using Indexed Addressing:			Using ORG:		
Reserve space			*Use LOCCTR to address fields*		
STAB	RESB	1100	STAB	RESB	1100
Refer to each field				ORG	STAB
SYMBOL	EQU	STAB	SYMBOL	RESB	6
VALUE	EQU	STAB+6	VALUE	RESW	1
FLAGS	EQU	STAB+9	FLAGS	RESB	2
Ex: *To fetch the VALUE field*				ORG	STAB+1100
	LDA	VALUE, X	(*Last ORG sets LOCCTR back)		

- Notice that two-pass assembler design requires that all symbols be defined during Pass 1. Example:

ALPHA	RESW	1	BETA	EQU	ALPHA
BETA	EQU	ALPHA	ALPHA	RESW	1

(*BETA cannot be assigned a value)

- Another example: the sequence of statements cannot be resolved by an ordinary two-pass assembler regardless of how the work is divided between passes.

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

2.3.3 Expressions

- Most assemblers allow the use of *expressions*. Each such expression must be evaluated by the assembler to produce a single operand address or value.
- Expressions are classified as either *absolute* expressions or *relative* expressions.

  Relative: *means relative to the beginning of the program*. *Labels* on instructions and data areas, and *references* to the location counter value, are relative terms.

  Absolute: *means independent of program location*. A constant is an absolute term.

  Note: A symbol whose value is given by EQU (or some similar assembler directive) may be either an absolute term or a relative term depending on the expression used to define its value.
- If relative terms occur in pairs and the terms in each such pair have opposite signs, then the resulting expressions are *absolute expressions*. None of the relative terms may enter into a multiplication or division operation.
- A relative expression is one in which all of the relative terms except one can be paired as described above; the remaining unpaired relative term must have a positive sign.
- Example: 107 MAXLEN EQU BUFEND－BUFFER

  both BUFEND and BUFFER are *relative terms*, each representing an address within the program. However, the expression represents an *absolute value*: the *difference* between the two addresses.
- Example: BUFEND ＋ BUFFER, 100 － BUFFER, or 3×BUFFER represent neither absolute values nor

locations within the program. Because such expressions are very unlikely to be of any use, they are considered errors.

- To determine the type of an expression, we must keep track of the *types* of all symbols defined in the program. (See page 77 example symbol table) With this information, the assembler can easily determine the type of each expression used as an operand and generate Modification records in the object program for relative values.

2.3.4 Program Blocks

- *Program blocks* are referred to be segments of code that are rearranged within a single object program unit, and *control sections* (appeared in next subsection) to be segments that are translated into independent object program units.
- Fig 2.11 shows our example program, as it might be written using program blocks. Three blocks are used: The *first* (unnamed) program block contains the executable instructions of the program. The *second* (named CDATA) contains all data areas that are a few words or less in length. The *third* (named CBLKS) contains all data areas that consist of larger blocks of memory.

```
Line        Source statement

  5    COPY      START    0              COPY FILE FROM INPUT TO OUTPUT
 10    FIRST     STL      RETADR         SAVE RETURN ADDRESS
 15    CLOOP     JSUB     RDREC          READ INPUT RECORD
 20              LDA      LENGTH         TEST FOR EOF (LENGTH = 0)
 25              COMP     #0
 30              JEQ      ENDFIL         EXIT IF EOF FOUND
 35              JSUB     WRREC          WRITE OUTPUT RECORD
 40              J        CLOOP          LOOP
 45    ENDFIL    LDA      =C'EOF'        INSERT END OF FILE MARKER
 50              STA      BUFFER
 55              LDA      #3             SET LENGTH = 3
 60              STA      LENGTH
 65              JSUB     WRREC          WRITE EOF
 70              J        @RETADR        RETURN TO CALLER
 92              USE      CDATA
 95    RETADR    RESW     1
100    LENGTH    RESW     1              LENGTH OF RECORD
103              USE      CBLKS
105    BUFFER    RESB     4096           4096-BYTE BUFFER AREA
106    BUFEND    EQU      *              FIRST LOCATION AFTER BUFFER
107    MAXLEN    EQU      BUFEND-BUFFER  MAXIMUM RECORD LENGTH
110    .
115    .         SUBROUTINE TO READ RECORD INTO BUFFER
120    .
123              USE
125    RDREC     CLEAR    X              CLEAR LOOP COUNTER
130              CLEAR    A              CLEAR A TO ZERO
132              CLEAR    S              CLEAR S TO ZERO
133             +LDT      #MAXLEN
135    RLOOP     TD       INPUT          TEST INPUT DEVICE
140              JEQ      RLOOP          LOOP UNTIL READY
145              RD       INPUT          READ CHARACTER INTO REGISTER A
150              COMPR    A,S            TEST FOR END OF RECORD (X'00')
155              JEQ      EXIT           EXIT LOOP IF EOR
160              STCH     BUFFER,X       STORE CHARACTER IN BUFFER
165              TIXR     T              LOOP UNLESS MAX LENGTH
170              JLT      RLOOP            HAS BEEN REACHED
175    EXIT      STX      LENGTH         SAVE RECORD LENGTH
180              RSUB                    RETURN TO CALLER
183              USE      CDATA
185    INPUT     BYTE     X'F1'          CODE FOR INPUT DEVICE
195    .
200    .         SUBROUTINE TO WRITE RECORD FROM BUFFER
205    .
208              USE
210    WRREC     CLEAR    X              CLEAR LOOP COUNTER
212              LDT      LENGTH
215    WLOOP     TD       =X'05'         TEST OUTPUT DEVICE
220              JEQ      WLOOP          LOOP UNTIL READY
225              LDCH     BUFFER,X       GET CHARACTER FROM BUFFER
230              WD       =X'05'         WRITE CHARACTER
235              TIXR     T              LOOP UNTIL ALL CHARACTERS
240              JLT      WLOOP            HAVE BEEN WRITTEN
245              RSUB                    RETURN TO CALLER
252              USE      CDATA
253              LTORG
255              END      FIRST
```

Figure 2.11 Example of a program with multiple program blocks.

- The assembler directive USE indicates which portions of the source program belong to the various blocks.

The beginning of program	begins Default block (unnamed)
Line 92	signals the beginning of CDATA
Line 103	begins the CBLK block
Line 123	resumes Default block
Line 183	resumes CDATA
Line 208	resumes Default block
Line 252	resumes CDATA

- The assembler accomplishes this logical rearrangement of code by maintaining, during Pass 1, a *separate location counter* for each program block. Thus each *label* in the program is assigned an address that is relative to the start of the block that contains it.
- At the end of Pass 1, the latest value of the location counter for each block indicates the length of that block. The assembler can then assign to each block a starting address in the object program (beginning with relative location 0).
- For code generation during Pass 2, the assembler needs the address for each symbol relative to the start of the object program (not the start of an individual program block). This is easily found from the information in SYMTAB. The assembler simply adds ***the location of the symbol***, relative to the start of its block, to ***the assigned block starting address***.
- Fig 2.12 shows this process applied to our sample program. Notice that the symbol MAXLEN (line 107) is shown without a block number. It is an absolute symbol.

```
  5   0000  0   COPY     START   0
 10   0000  0   FIRST    STL     RETADR        172063
 15   0003  0   CLOOP    JSUB    RDREC         4B2021
 20   0006  0            LDA     LENGTH        032060
 25   0009  0            COMP    #0            290000
 30   000C  0            JEQ     ENDFIL        332006
 35   000F  0            JSUB    WRREC         4B203B
 40   0012  0            J       CLOOP         3F2FEE
 45   0015  0   ENDFIL   LDA     =C'EOF'       032055
 50   0018  0            STA     BUFFER        0F2056
 55   001B  0            LDA     #3            010003
 60   001E  0            STA     LENGTH        0F2048
 65   0021  0            JSUB    WRREC         4B2029
 70   0024  0            J       @RETADR       3E203F
 92   0000  1            USE     CDATA
 95   0000  1   RETADR   RESW    1
100   0003  1   LENGTH   RESW    1
103   0000  2            USE     CBLKS
105   0000  2   BUFFER   RESB    4096
106   1000  2   BUFEND   EQU     *
107   1000      MAXLEN   EQU     BUFEND-BUFFER
110             .
115             .        SUBROUTINE TO READ RECORD INTO BUFFER
120             .
123   0027  0            USE
125   0027  0   RDREC    CLEAR   X             B410
130   0029  0            CLEAR   A             B400
132   002B  0            CLEAR   S             B440
133   002D  0           +LDT     #MAXLEN       75101000
135   0031  0   RLOOP    TD      INPUT         E32038
140   0034  0            JEQ     RLOOP         332FFA
145   0037  0            RD      INPUT         DB2032
150   003A  0            COMPR   A,S           A004
155   003C  0            JEQ     EXIT          332008
160   003F  0            STCH    BUFFER,X      57A02F
165   0042  0            TIXR    T             B850
170   0044  0            JLT     RLOOP         3B2FEA
175   0047  0   EXIT     STX     LENGTH        13201F
180   004A  0            RSUB                  4F0000
183   0006  1            USE     CDATA
185   0006  1   INPUT    BYTE    X'F1'         F1
195             .
200             .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205             .
208   004D  0            USE
210   004D  0   WRREC    CLEAR   X             B410
212   004F  0            LDT     LENGTH        772017
215   0052  0   WLOOP    TD      =X'05'        E3201B
220   0055  0            JEQ     WLOOP         332FFA
225   0058  0            LDCH    BUFFER,X      53A016
230   005B  0            WD      =X'05'        DF2012
235   005E  0            TIXR    T             B850
240   0060  0            JLT     WLOOP         3B2FEF
245   0063  0            RSUB                  4F0000
252   0007  1            USE     CDATA
253                      LTORG
      0007  1   *        =C'EOF'               454F46
      000A  1   *        =X'05'                05
255                      END     FIRST
```

Figure 2.12 Program from Fig. 2.11 with object code.

- See page 80 for the table constructed by assemblers at the end of Pass 1. This table contains the starting addresses and lengths for all blocks.

- Example: 0006 0 LDA LENGTH 032060
  SYMTAB shows the value of the operand (LENGTH) as relative location 0003 within program block 1 (CDATA). The starting address for CDATA is 0066. Thus the desired target address for this instruction is 0003+0066=0069.

- We can see that the separation of the program into blocks as considerably reduced our addressing problems. Because the large buffer area is moved to the end of the object program, we no longer need to use extended format instructions on lines 15, 35, and 65.

- Fig 2.13 shows the object program corresponding to Fig 2.11. It does not matter that the Text records of the object program are not in sequence by address; the loader will simply load the object code from each record at the indicated address.

```
H^COPY  ^000000^001071
T^000000^1E^172063^4B2021^032060^290000^332006^4B203B^3F2FEE^032055^0F2056^010003
T^00001E^09^0F2048^4B2029^3E203F
T^000027^1D^B410^B400^B440^75101000^E32038^332FFA^DB2032^A004^332008^57A02F^B850
T^000044^09^3B2FEA^13201F^4F0000
T^00006C^01^F1
T^00004D^19^B410^772017^E3201B^332FFA^53A016^DF2012^B850^3B2FEF^4F0000
T^00006D^04^454F46^05
E^000000
```

Figure 2.13 Object program corresponding to Fig. 2.11.

- Fig 2.14 traces the blocks of the example program through this process of assembly and loading.

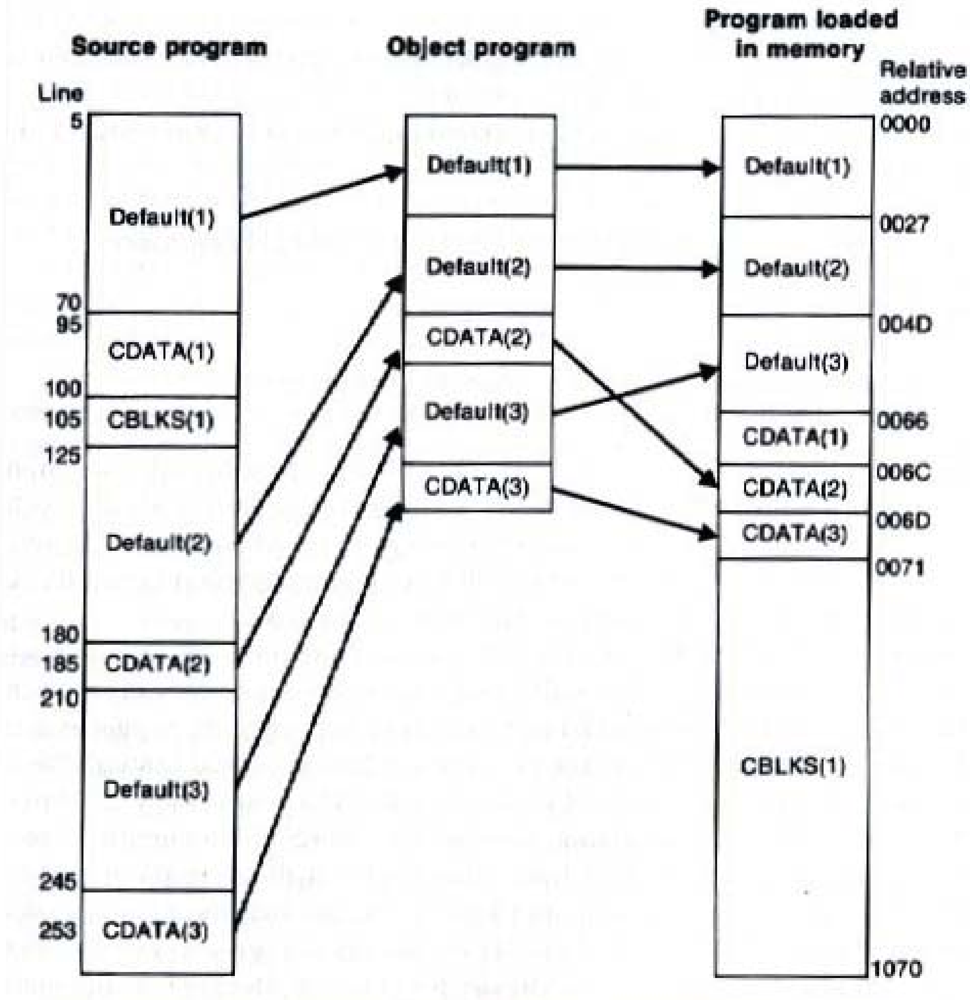


Figure 2.14 Program blocks from Fig. 2.11 traced through the assembly and loading processes.

2.3.5 Control Sections and Program Linking

- A *control section* is a part of the program that maintains its identity after assembly; each such control section can be *loaded* and *relocated* independently of the others. Different control sections are most often used for subroutines or other logical subdivisions of a program.

- *Control sections* differ from *program blocks* in that they are handled separately by the assembler.

- Fig 2.15 shows three control sections: The *first* section continues (from COPY) till the CSECT statement on line 109. (CSECT signals the start of a new control section named RDREC – 2nd control section.) Similarly, CSECT on line 193 begins WRREC – 3rd control section. The assembler establishes a separate location counter (beginning at 0) for each control section, just as it does for program blocks.

Line	Loc	Source statement			Object code
5	0000	COPY	START	0	
6			EXTDEF	BUFFER,BUFEND,LENGTH	
7			EXTREF	RDREC,WRREC	
10	0000	FIRST	STL	RETADR	172027
15	0003	CLOOP	+JSUB	RDREC	4B100000
20	0007		LDA	LENGTH	032023
25	000A		COMP	#0	290000
30	000D		JEQ	ENDFIL	332007
35	0010		+JSUB	WRREC	4B100000
40	0014		J	CLOOP	3F2FEC
45	0017	ENDFIL	LDA	=C'EOF'	032016
50	001A		STA	BUFFER	0F2016
55	001D		LDA	#3	010003
60	0020		STA	LENGTH	0F200A
65	0023		+JSUB	WRREC	4B100000
70	0027		J	@RETADR	3E2000
95	002A	RETADR	RESW	1	
100	002D	LENGTH	RESW	1	
103			LTORG		
	0030	*	=C'EOF'		454F46
105	0033	BUFFER	RESB	4096	
106	1033	BUFEND	EQU	*	
107	1000	MAXLEN	EQU	BUFEND-BUFFER	
109	0000	RDREC	CSECT		
110		.			
115		.	SUBROUTINE TO READ RECORD INTO BUFFER		
120		.			
122			EXTREF	BUFFER,LENGTH,BUFEND	
125	0000		CLEAR	X	B410
130	0002		CLEAR	A	B400
132	0004		CLEAR	S	B440
133	0006		LDT	MAXLEN	77201F
135	0009	RLOOP	TD	INPUT	E3201B
140	000C		JEQ	RLOOP	332FFA
145	000F		RD	INPUT	DB2015
150	0012		COMPR	A,S	A004
155	0014		JEQ	EXIT	332009
160	0017		+STCH	BUFFER,X	57900000
165	001B		TIXR	T	B850
170	001D		JLT	RLOOP	3B2FE9
175	0020	EXIT	+STX	LENGTH	13100000
180	0024		RSUB		4F0000
185	0027	INPUT	BYTE	X'F1'	F1
190	0028	MAXLEN	WORD	BUFEND-BUFFER	000000
193	0000	WRREC	CSECT		
195		.			
200		.	SUBROUTINE TO WRITE RECORD FROM BUFFER		
205		.			
207			EXTREF	LENGTH,BUFFER	
210	0000		CLEAR	X	B410
212	0002		+LDT	LENGTH	77100000
215	0006	WLOOP	TD	=X'05'	E32012
220	0009		JEQ	WLOOP	332FFA
225	000C		+LDCH	BUFFER,X	53900000
230	0010		WD	=X'05'	DF2008
235	0013		TIXR	T	B850
240	0015		JLT	WLOOP	3B2FEE
245	0018		RSUB		4F0000
255			END	FIRST	
	001B	*	=X'05'		05

Figure 2.16 Program from Fig. 2.15 with object code.

- Fig 2.15 shows the use of two assembler directives to identify such references: EXTDEF (external definition) and EXTREF (external reference).

  The EXTDEF statement in a control section names symbols, called *external symbols*, that are defined in this control section and may be used by other sections.

  Control section names do not need to be named in an EXTDEF statement because they are automatically considered to be external symbols.

  The EXTREF statement names symbols that are used in this control section and are defined elsewhere.

- Fig 2.16 shows the generated object code for each statement in the program. Example:

  0003 CLOOP +JSUB RDREC 4B100000
  The operand RDREC is named in the EXTREF statement for the control section, so this is an external reference.

  0017 +STCH BUFFER,X 57900000
  This instruction makes an external reference BUFFER. The instruction is assembled using extended format with an address of zero.

- The assembler must remember (via entries in SYMTAB) in *which control section* a symbol is defined. For example, note the handling difference between line 107 and line 190. The symbols BUFEND and BUFFER are defined in the same control section with the EQU statement on line 107. Thus, the value of the expression can be calculated immediately by the assembler. *This could not be done for line 190*; BUFEND and BUFFER are defined in another control section, so their values are unknown at assembly time.

- *The assembler must include information in the object program that will cause the loader to insert the proper values where they are required*. We need two new record types (Define and Refer) in the object program.

Define record:

Col. 1	D
Col. 2–7	Name of external symbol defined in this control section
Col. 8–13	Relative address of symbol within this control section (hexadecimal)
Col. 14–73	Repeat information in Col. 2–13 for other external symbols

Refer record:

Col. 1	R
Col. 2–7	Name of external symbol referred to in this control section
Col. 8–73	Names of other external reference symbols

- A <u>Define</u> record gives information about external symbols that are defined in this control section – that is, symbols named by EXTDEF. (The record format see page 89)
- A Refer record lists symbols that are used as external reference by the control section – that is, symbols named by EXTREF. (The record format see page 89)
- In addition, a revised Modification record is also shown in page 89.

Modification record (revised):

Col. 1	M
Col. 2–7	Starting address of the field to be modified, relative to the beginning of the control section (hexadecimal)
Col. 8–9	Length of the field to be modified, in half-bytes (hexadecimal)
Col. 10	Modification flag (+ or –)
Col. 11–16	External symbol whose value is to be added to or subtracted from the indicated field

- Fig 2.17 shows the object program corresponding to the source in Fig 2.16. Notice that there is a separate set of object program records for each control section.

```
H^COPY  ^000000^001033
D^BUFFER^000033^BUFEND^001033^LENGTH^00002D
R^RDREC ^WRREC
T^000000^1D^172027^4B100000^032023^290000^332007^4B100000^3F2FEC^032016^0F2016
T^00001D^0D^010003^0F200A^4B100000^3E2000
T^000030^03^454F46
M^000004^05^+RDREC
M^000011^05^+WRREC
M^000024^05^+WRREC
E^000000

H^RDREC ^000000^00002B
R^BUFFER^LENGTH^BUFEND
T^000000^1D^B410^B400^B440^77201F^E3201B^332FFA^DB2015^A004^332009^57900000^B850
T^00001D^0E^3B2FE9^13100000^4F0000^F1^000000
M^000018^05^+BUFFER
M^000021^05^+LENGTH
M^000028^06^+BUFEND
M^000028^06^-BUFFER
E

H^WRREC ^000000^00001C
R^LENGTH^BUFFER
T^000000^1C^B410^77100000^E32012^332FFA^53900000^DF2008^B850^3B2FEE^4F000005
M^000003^05^+LENGTH
M^00000D^05^+BUFFER
E
```

Figure 2.17 Object program corresponding to Fig. 2.15.

- Example: The address field for the JSUB on line 15 begins at relative address 0004. Its initial value in the object program is zero. The Modification record 'M00000405+RDREC' in control section COPY specifies that the address of RDREC is to be added to this field, thus producing the correct machine instruction for execution.

- Example: The handling of line 190. The value of this word is to be BUFEND−BUFFER, where both BUFEND and BUFFER are defined in another control section. The assembler generates an initial value of *zero* for this word. The last two Modification records in RDREC direct that the address of BUFEND be added to this field, and the address of BUFFER be subtracted from it. This computation, performed at load time, results in the desired value for the data word.

2.4 Assembler Design Options

2.4.1 One-Pass Assemblers

- The main problem in trying to assemble a program in one pass involves *forward references* and *forward jump* (page 93).
- There are *two* main types of *one-pass assembler*. One type produces object code directly in memory for immediate execution (*load-and-go assembler*); the other type produces the usual kind of object program for later execution.
- Fig 2.18 shows a sample program for a one-pass assembler.

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.

- Fig 2.19(a) shows <u>the object code</u> and <u>symbol table entries</u> as they would be <u>after scanning line 40 of the program in Fig 2.18</u>.

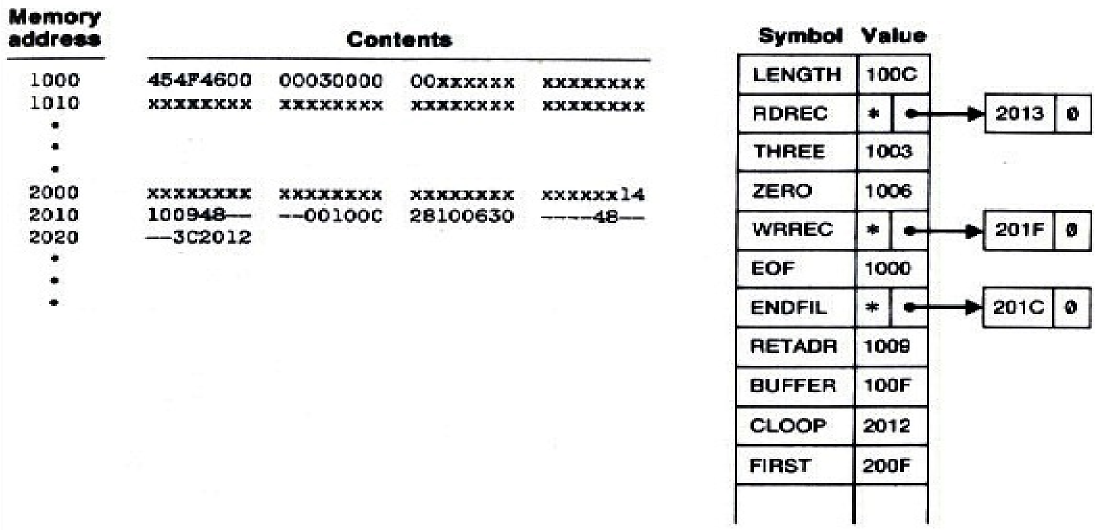


Figure 2.19(a) Object code in memory and symbol table entries for the program in Fig. 2.18 after scanning line 40.

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 by ----).

RDREC was then entered into SYMTAB as an undefined symbol (indicated by *); the address of the operand field (2013) of the instruction 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.

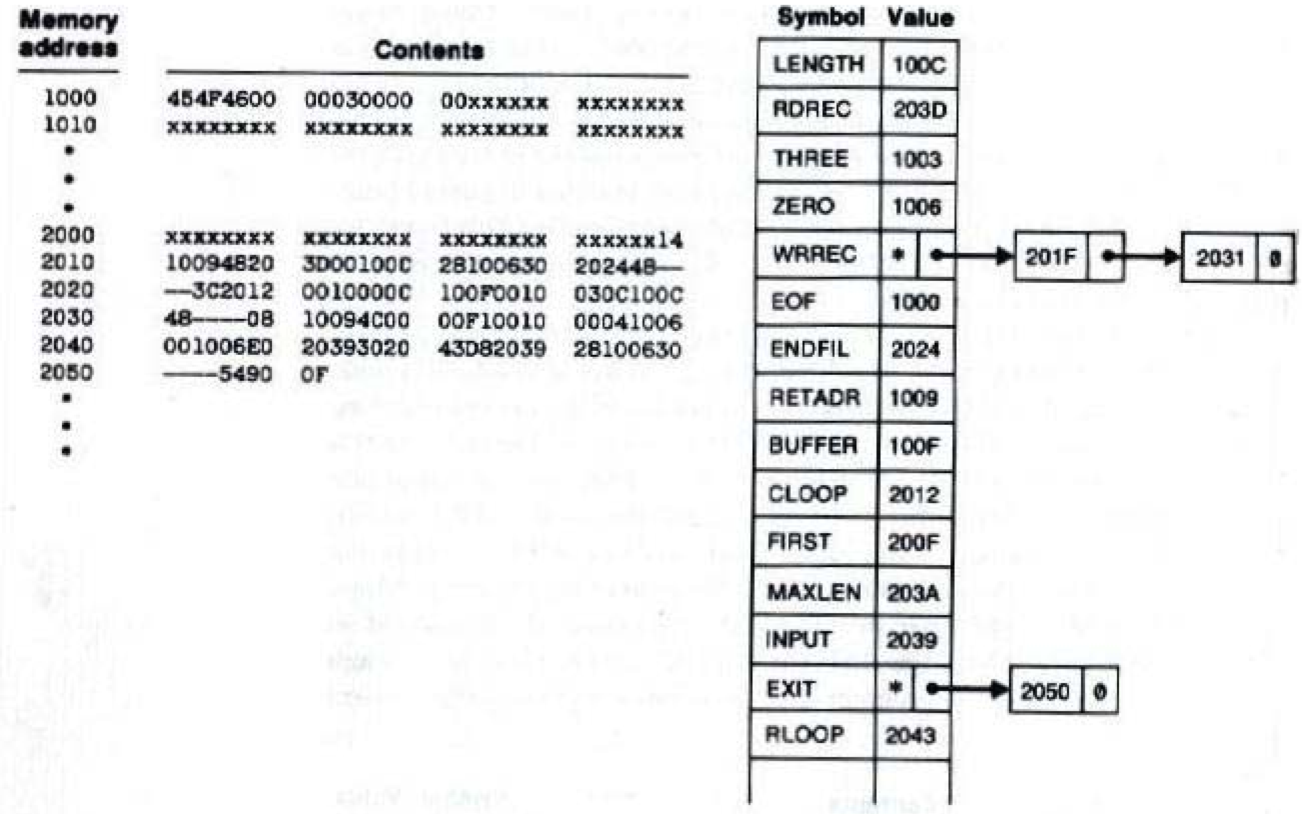


Figure 2.19(b) Object code in memory and symbol table entries for the program in Fig. 2.18 after scanning line 160.

By this time, <u>some</u> of the forward references (ENDFIL, line 45 and RDREC, line 125) have <u>been resolved</u>, while <u>others</u> (EXIT, line 175 and WRREC, line 210) have <u>been added</u>.

When the symbol ENDFIL was defined (known), <u>the assembler placed its value in the SYMTAB entry</u>; it then inserted this value into <u>the instruction operand field (at address 201C)</u> <u>as directed by the *forward reference list*</u>. From this point on, any references to ENDFIL would not be forward references, and would not be entered into a list.

- At the end of the program, any SYMTAB entries that are still marked with * indicate *undefined symbols*. <u>These should be flagged by the assembler as errors</u>.
- One-pass assemblers that produce object programs follow a slightly different procedure from that previously described.

1) *Forward references* are entered into *lists* as before.

2) When the definition of a symbol is encountered, instructions that made forward references to that symbol may no longer 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*.

3) When the program is loaded, this address will be inserted into the instruction by the action of the loader.

- Fig 2.20 illustrates the above process.

```
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.

The 2nd Text record contains that 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 ENDFIL on line 45 is encountered, the assembler generates the 3rd 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 JEQ on line 30).

When the program is loaded, the value 2024 will replace the 0000 previously loaded.

- Note that in this section, we considered only simple one-pass assemblers that handled *absolute* programs.

2.4.2 Multi-Pass Assemblers

- Consider the program sequence

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

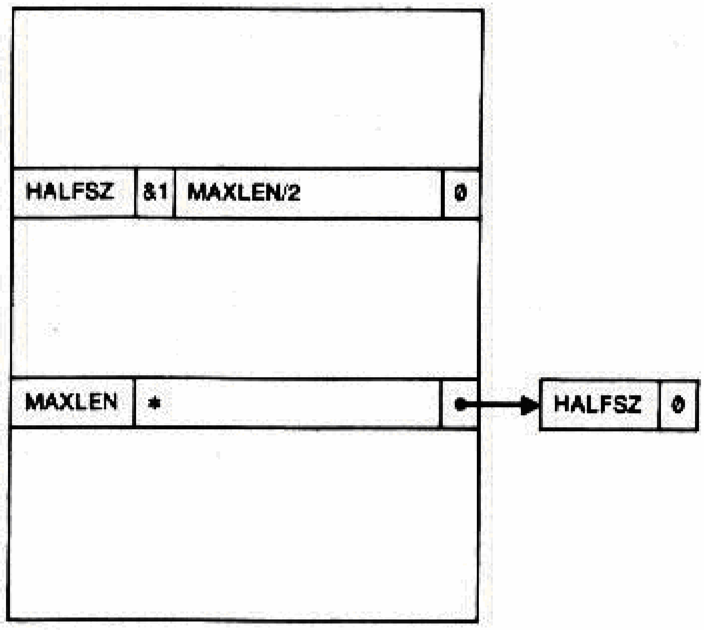
Note that any assembler that makes only two sequential passes over the source program cannot resolve such a sequence of definitions.

- The general solution is a multi-pass assembler that can make as many passes as are needed to process the definitions of symbols.
- Fig 2.21(a) shows a sequence of symbol-defining statements that involve forward references.

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

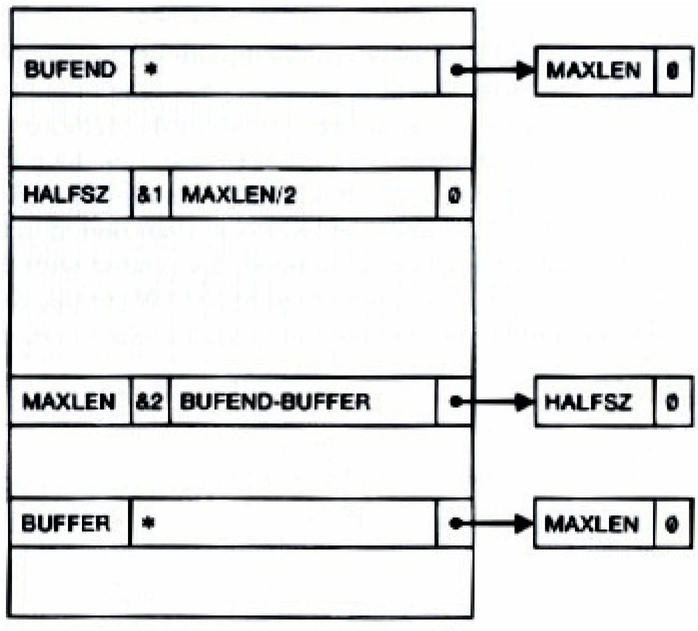
(a)

Fig 2.21(b) displays symbol table entries resulting from Pass 1 processing of the statement. The entry &1 indicates that one symbol in the defining expression is undefined.



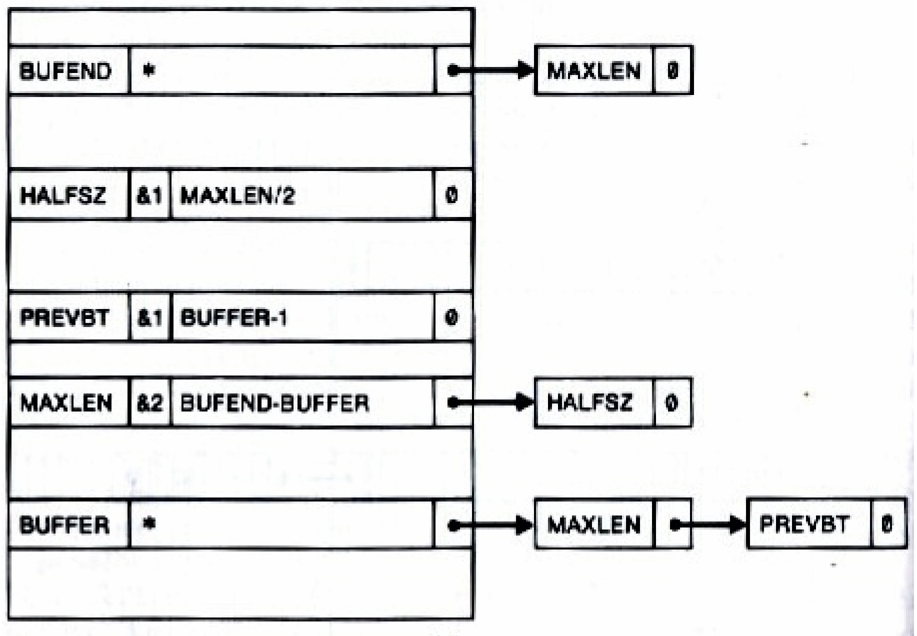
(b)

Fig 2.21(c) shows two undefined symbols involved in the definition: BUFEND and BUFFER.



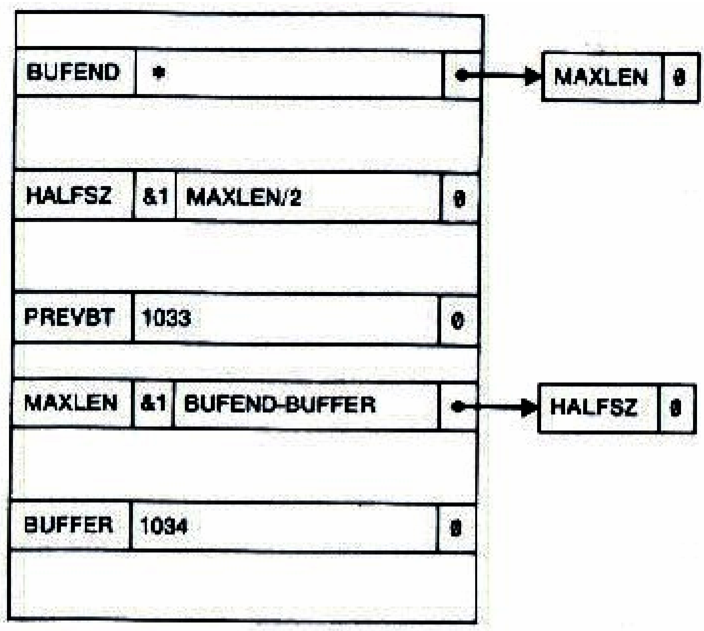
(c)

Fig 2.21(d) shows a new undefined symbol PREVBT (dependent on BUFFER) is added.



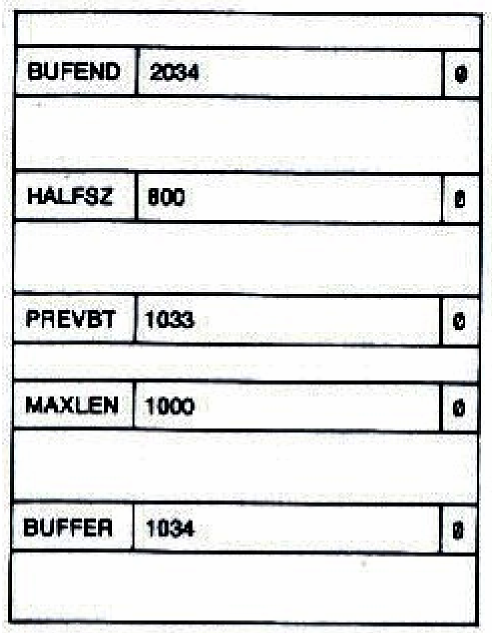
(d)

Fig 2.21(e) shows that when BUFFER is encountered, PREVBT can be determined accordingly.



(e)

Fig 2.21(f) shows that when BUFEND is defined, MAXLEN and HALFSZ can be determined accordingly.



(f)

2.5 Implementation Examples

(Skip)