MODULE-1

Introduction:

Software is set of instructions or programs written to carry out certain task on digital computers. It is classified into system software and application software. System software consists of a variety of programs that support the operation of a computer. Application software focuses on an application or problem to be solved.

System software consists of a variety of programs that support the operation of a computer. Examples for system software are Operating system, compiler, assembler, macro processor, loader or linker, debugger, text editor etc.., These software's make it possible for the user to focus on an application or other problem to be solved, without needing to know the details of how the machine works internally.

Difference between System Software and Application Software:

S.No	System Software	Application Software
1	Machine Dependent.	Machine Independent.
2	Supports the operation and use of a computer.	Uses the computer as a tool to give solution of some problem.
3	Focuses on the computing system.	Focus is on the application.
4	Usually related to the machine architecture.	They are not related to the machine architecture.
5	Examples: Operating Systems, Assemblers, Interpreters, Compilers, Editors, Linkers, Loaders, Macro processors.	Examples: Banking Software, Airline Ticket Reservation System, Hospital Management System, etc..,

On the other hand, there are some aspects of system software that do not directly depend upon the type of computing system being supported.

For example,

- The general design and logic of an assembler is basically the same on most computers.
- Code optimization techniques used by compilers are independent of the target machine.
- The process of linking together independently assembled subprograms does not usually depend on the computer being used.

The Simplified Instructional Computer (SIC)

Simplified Instructional Computer (SIC) is a hypothetical computer that includes the hardware features most often found on real machines.

There are two versions of SIC,

➔ Standard model (SIC),

➔ Extension version (SIC/XE) (extra equipment or extra expensive).

The two versions have been designed to be upward compatible – that is an object program for the standard SIC machine will also execute properly on SIC/XE system.

SIC Machine Architecture:

Memory:

➔ Memory consists of 8-bit bytes; any 3 consecutive bytes form a word (24 bits). All addresses on SIC are byte addresses

➔ Words are addressed by the location of their lowest numbered byte.

➔ There are total of 32,768 (2^{15}) bytes in the computer memory.

Registers:

There are five registers, all of which have special uses. Each register is 24 bits in length. Their mnemonics, numbers and uses are given in the following table.

Mnemonic	Number	Special Use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; JSUB instruction stores the return address in this register
PC	8	Program counter; contains the address of the next instruction to be fetched for execution.
SW	9	Status word; contains a variety of information, including a Condition Code (CC)

Data Formats:

- Integers are stored as 24-bit binary numbers.
- 2’s complement representation is used for negative values.
- characters are stored using their 8-bit ASCII codes.
- There is no floating-point hardware on the standard version of SIC.

Instruction Formats:

All machine instructions on the standard version of SIC have the 24-bit format:

8	1	15
opcode	x	Address

Addressing Modes:

There are two addressing modes available, indicated by the setting of the ***x*** bit in the instruction.

Mode	Indication	Target Address Calculation
Direct	x=0	TA= address
Indexed	x=1	TA= address + (X)

Parentheses are used to indicate the contents of a register or a memory location. For example, (X) represents the contents of register X.

Direct Addressing Mode:

Example: LDA TEN LDA – 00

opcode	x	TEN
0 0 0 0 0 0 0 0	0	0 0 1 0 0 0 0 0 0 0 0 0 0 0 0

Effective Address (EA) = 1000

Content of the address 1000 is loaded to Accumulator.

Indexed Addressing Mode:

Example: STCH BUFFER, X

opcode	x	BUFFER
0 0 0 0 0 1 0 0	1	0 0 1 0 0 0 0 0 0 0 0 0 0 0 0

Effective Address (EA) = 1000 + [X]

= 1000 + content of the index register X

The Accumulator content, the character is loaded to the Effective address.

<u>Instruction Set:</u>

SIC provides, load and store instructions (LDA, LDX, STA, STX, etc.). Integer arithmetic operations: (ADD, SUB, MUL, DIV, etc.). All arithmetic operations involve register A and a word in memory, with the result being left in the register. Two instructions are provided for subroutine linkage. COMP compares the value in register A with a word in the memory, this instruction sets a condition code CC to indicate the result. There are conditional jump instructions: (JLT, JEQ, JGT), these instructions test the setting of CC and jump accordingly.

JSUB jumps to the subroutine placing the return address in register L, RSUB returns by jumpingto the address contained in register L.

Input and Output:

Input and Output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A (accumulator).

The Test Device (TD) instruction tests whether the addressed device is ready to send or receive a byte of data. Read Data (RD), Write Data (WD) are used for reading or writing the data.

Data movement and Storage Definition

LDA, STA, LDL, STL, LDX, STX (A- Accumulator, L – Linkage Register, X – Index Register), all uses 3-byte word. LDCH, STCH associated with characters uses 1 -byte. There are no memory-memory move instructions.

4 Storage definitions

- WORD → ONE-WORD CONSTANT
- RESW → ONE-WORD VARIABLE
- BYTE → ONE-BYTE CONSTANT
- RESB → ONE-BYTE VARIABLE

SIC/XE Architecture:

Memory:

→ Maximum memory available on a SIC/XE system is 1 Megabyte (2^{20} bytes). This increase in memory leads to a change in instruction formats and addressing modes.

Registers:

Additional B, S, T, and F registers are provided by SIC/XE, in addition to the registers of SIC

Mnemonic	Number	Special use
B	3	Base register; used for addressing
S	4	General working register-- no special use
T	5	General working register-- no special use
F	6	Floating-point accumulator(48 bits)

Data Formats:

- ➔ Integers are stored as 24-bit binary numbers.
- ➔ 2's complement representation is used for negative values.
- ➔ characters are stored using their 8-bit ASCII codes.

There is a 48-bit floating-point data type with the following format,

1	11	36
s	Exponent	fraction

The fraction is interpreted as a value between 0 and 1. If the exponent has value e and the fraction has value f, the absolute value of the number is represented is $f*2^{(e-1024)}$.

The sign of the floating-point number is indicated by the value of s(0=positive, 1=negative).A value of zero is represented by setting all bits to 0.

Instruction Formats:

The new set of instruction formats fro SIC/XE machine architecture are as follows.

Format 1 (1 byte): contains only operation code (straight from table).

Format 2 (2 bytes): First eight bits for operation code, next four for register 1 and following four for register 2.

The numbers for the registers go according to the numbers indicated at the registers section (ie, register T is replaced by hex 5, F is replaced by hex 6).

Format 3 (3 bytes): First 6 bits contain operation code, next 6 bits contain flags, last 12 bits contain displacement for the address of the operand. Operation code uses only 6 bits, thus the second hex digit will be affected by the values of the first two flags (n and i). The flags, in order, are: n, i, x, b, p, and e. Its functionality is explained in the next section. The last flag e indicates the instruction format (0 for 3 and 1 for 4).

Format 4 (4 bytes): same asformat 3 with an extra 2 hex digits (8 bits) for addresses that require more than 12 bits tobe represented.

Format 1 (1 byte)

8 bits
op

Format 2 (2 bytes)

8	4	4
op	R1	R2

Formats 1 and 2 are instructions do not reference memory at all

Format 3 (3 bytes)

6	1	1	1	1	1	1	12
op	n	i	x	b	p	e	displacement

Format 4 (4 bytes)

6	1	1	1	1	1	1	20
op	n	i	x	b	p	E	address

Addressing modes & Flag Bits:

Five possible addressing modes plus the combinations are as follows.

Direct (x, b, and p all set to 0): operand address goes as it is. n and i are both setto the same value, either 0 or 1. While in general that value is 1, if set to 0 for format 3we can assume that the rest of the flags (x, b, p, and e) are used as a part of the address of the operand, to makethe format compatible to the SIC format.

Relative (either b or p equal to 1 and the other one to 0): the address of the operand should be added to the current value stored at the B register (if b = 1) or to the value stored at the PC register (if p = 1).

Immediate (i = 1, n = 0): The operand value is already enclosed on the instruction (ie. lies on the last 12/20 bits of the instruction).

Indirect (i = 0, n = 1): The operand value points to an address that holds the address for the operand value.

Indexed (x = 1): value to be added to the value stored at the register x to obtain real address of the operand. This can be combined with any of the previous modes except immediate.

The various flag bits used in the above formats have the following meanings e = 0 means format 3, e = 1 means format 4. Bits x,b,p: Used to calculate the target address using relative, direct, and indexed.

Addressing Modes:

Bits i and n: how to use the target address b and p - both set to 0, disp field from format 3 instruction is taken to be the target address. For a format 4 bits b and p are normally set to 0, 20bit address is the target address x - x is set to 1, X register value is added for target address.

Calculation

i=1, n=0 Immediate addressing, TA: TA is used as the operand value, no memory reference.

i=0, n=1 Indirect addressing, ((TA)): The word at the TA is fetched. Value of TA is taken as theaddress of the operand value.

i=0, n=0 or i=1, n=1 Simple addressing, (TA):TA is taken as the address of the operand value.

Two new relative addressing modes are available for use with instructions assembled using format 3.

Mode Indication, Target address calculation

Base relative b=1,p=0

TA=(B)+ disp

Program-counter relative b=0,p=1, TA=(PC)+ disp

Instruction Set:

SIC/XE provides all of the instructions that are available on the standard version. In addition we have, Instructions to load and store the new registers LDB, STB, etc, Floating-point arithmetic operations, ADDF, SUBF, MULF, DIVF, Register move instruction : RMO, Register- to-register arithmetic operations, ADDR, SUBR, MULR,DIVR and, Supervisor call instruction : SVC.

Input and Output:

There are I/O channels that can be used to perform input and output while the CPU is executing other instructions. Allows overlap of computing and I/O, resulting in more efficient system operation. The instructions SIO, TIO, and HIO are used to start, test and halt the operation of I/O channels.

Basic Assembler Functions

- Convert mnemonic operation codes to their machine language equivalent
- Convert symbolic operands to their equivalent machine addresses
- Build the machine instructions in the proper format
- Convert the data constants specified in the source program into their machine representations
- Write the object program and the assembly listing.

Two Pass Assembler

Forward reference—a reference to a label that is defined later in the program, because of forward reference, most assembler make two pass over the source program. The first pass does little more than scan the source program for label definitions and assign address. The second pass performs most of the actual translation Assembler directives (or pseudo-instructions) provide instructions to the assembler itself

Pass 1 (define symbols)

- Assign addresses to all statements in the program
- Save the values (addresses) assigned to all labels
- Perform some processing of assembler directives

Pass 2 (assemble instructions and generate object program)

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

Algorithm for Pass 1 of Assembler

```
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
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
```

Algorithm for Pass 2 of Assembler

```
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
```

Assembler Data Structure and Variable

Two major data structures:

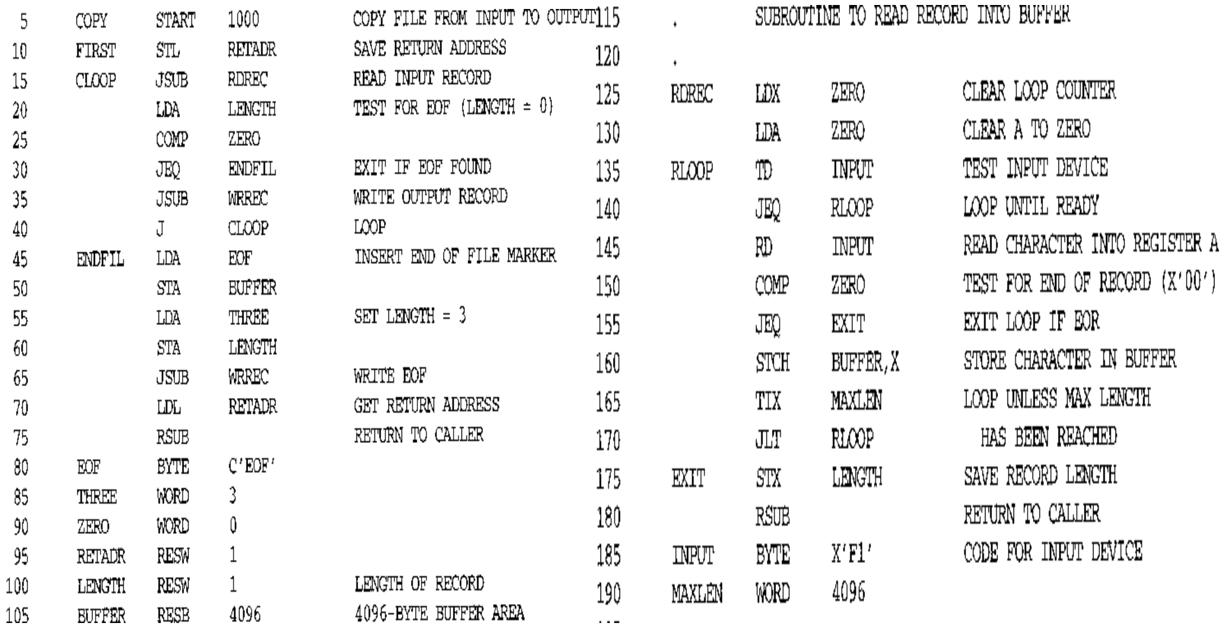
- Operation Code Table (OPTAB): is used to look up mnemonic operation codes and translate them to their machine language equivalents
- Symbol Table (SYMTAB): is used to store values (addresses) assigned to labels

Variable Location Counter (LOCCTR) is used to help the assignment of addresses. LOCCTR is initialized to the beginning address specified in the START statement. The length of the assembled instruction or data area to be generated is added to LOCCTR

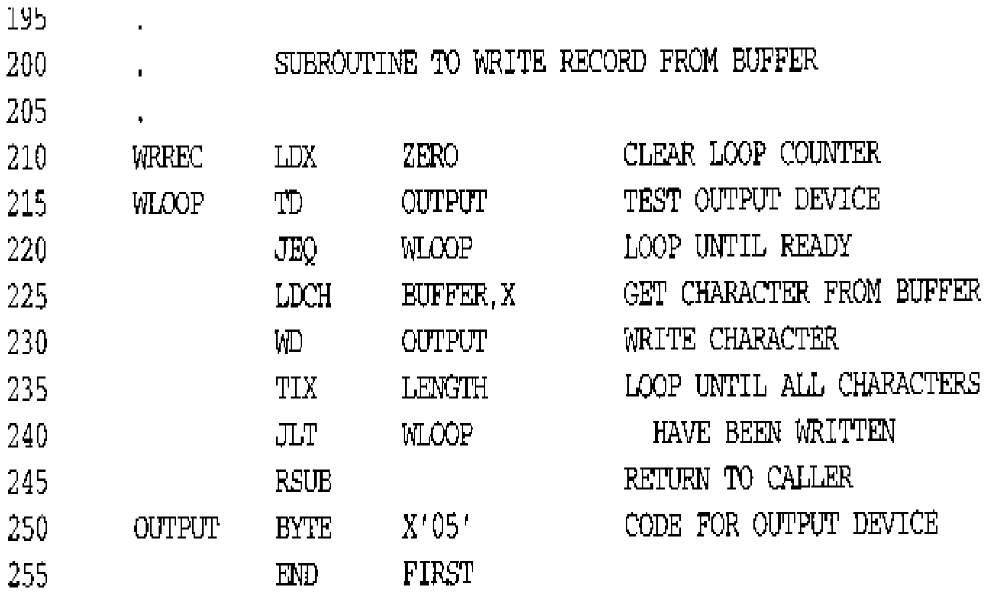
OPTAB must contain the mnemonic operation code and its machine language. In more complex assembler, it also contain information about instruction format and length For a machine that has instructions of different length, we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR

SYMTAB includes the name and value (address) for each label, together with flags to indicate error conditions. OPTAB and SYMTAB are usually organized as hash tables, with mnemonic operation code or label name as the key, for efficient retrieval

Example of a SIC Assembler LanguageProgram

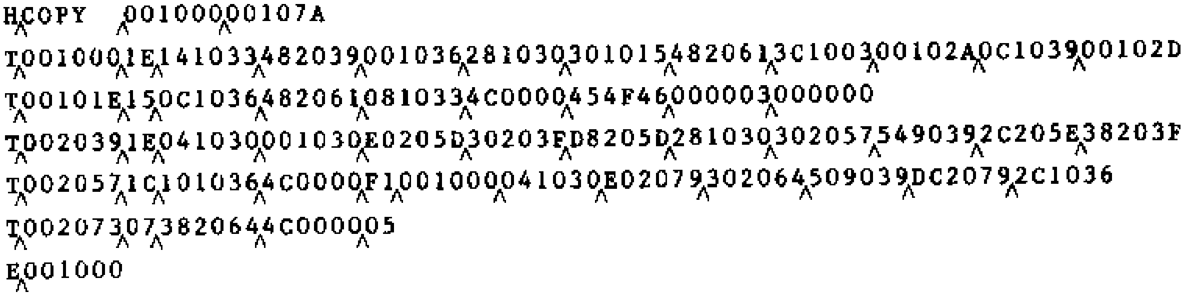


```
COPY     START   1000        COPY FILE FROM INPUT TO OUTPUT
FIRST    STL     RETADR      SAVE RETURN ADDRESS
CLOOP    JSUB    RDREC       READ INPUT RECORD
         LDA     LENGTH      TEST FOR EOF (LENGTH = 0)
         COMP    ZERO
         JEQ     ENDFIL      EXIT IF EOF FOUND
         JSUB    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
```

Object Program

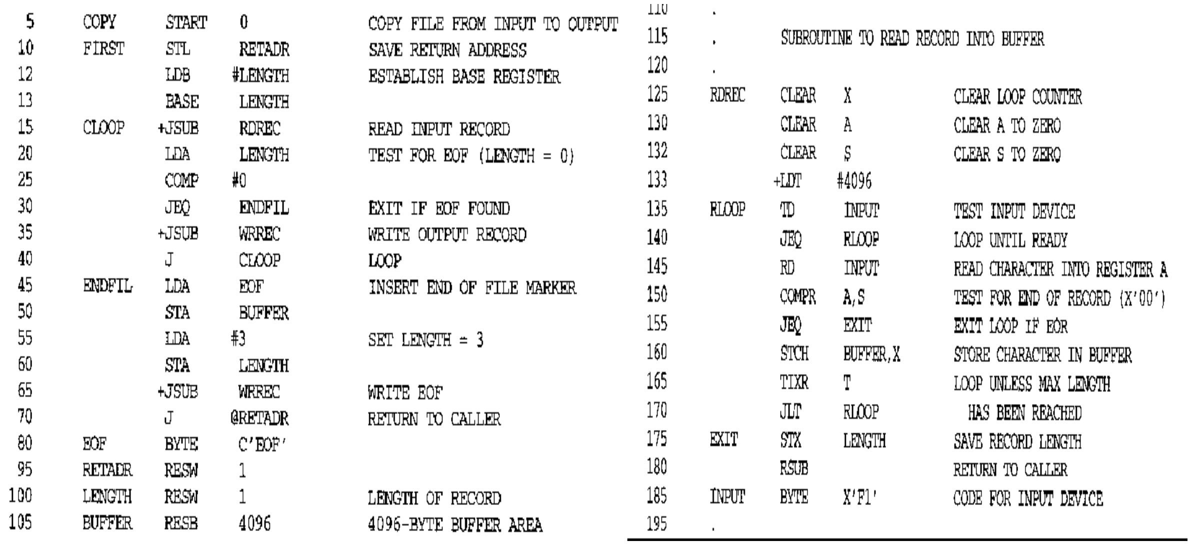


```
H^COPY  ^001000^00107A
T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^00102A^0C1039^00102D
T^00101E^15^0C1036^482061^081033^4C0000^454F46^000003^000000
T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^549039^2C205E^38203F
T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^509039^DC2079^2C1036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

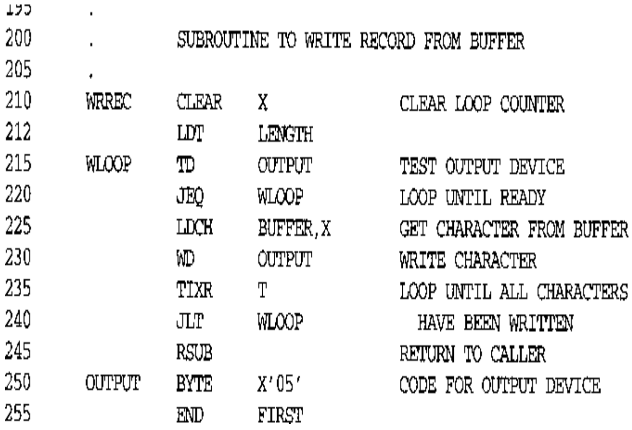
Machine-Dependent Assembler Features

- Indirect addressing is indicated by adding the prefix @ to the operand
- Immediate operands are denoted with the prefix #
- The assembler directive BASE is used in conjunction with base relative addressing
- The extended instruction format is specified with the prefix + added to the operation code
- Register-to-register instruction are faster than the corresponding register-to-memory operations because they are shorter and because they do not require another memory reference

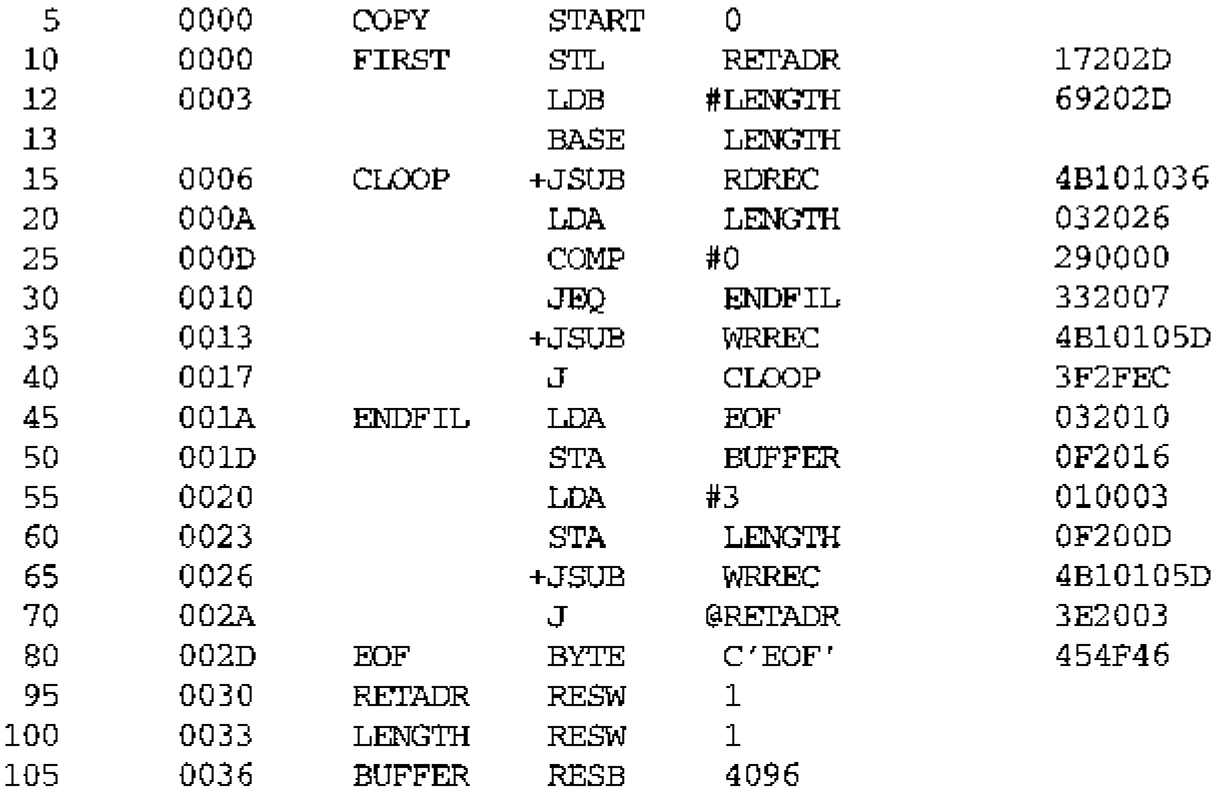
Program SIC/XE



```
  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   .
```

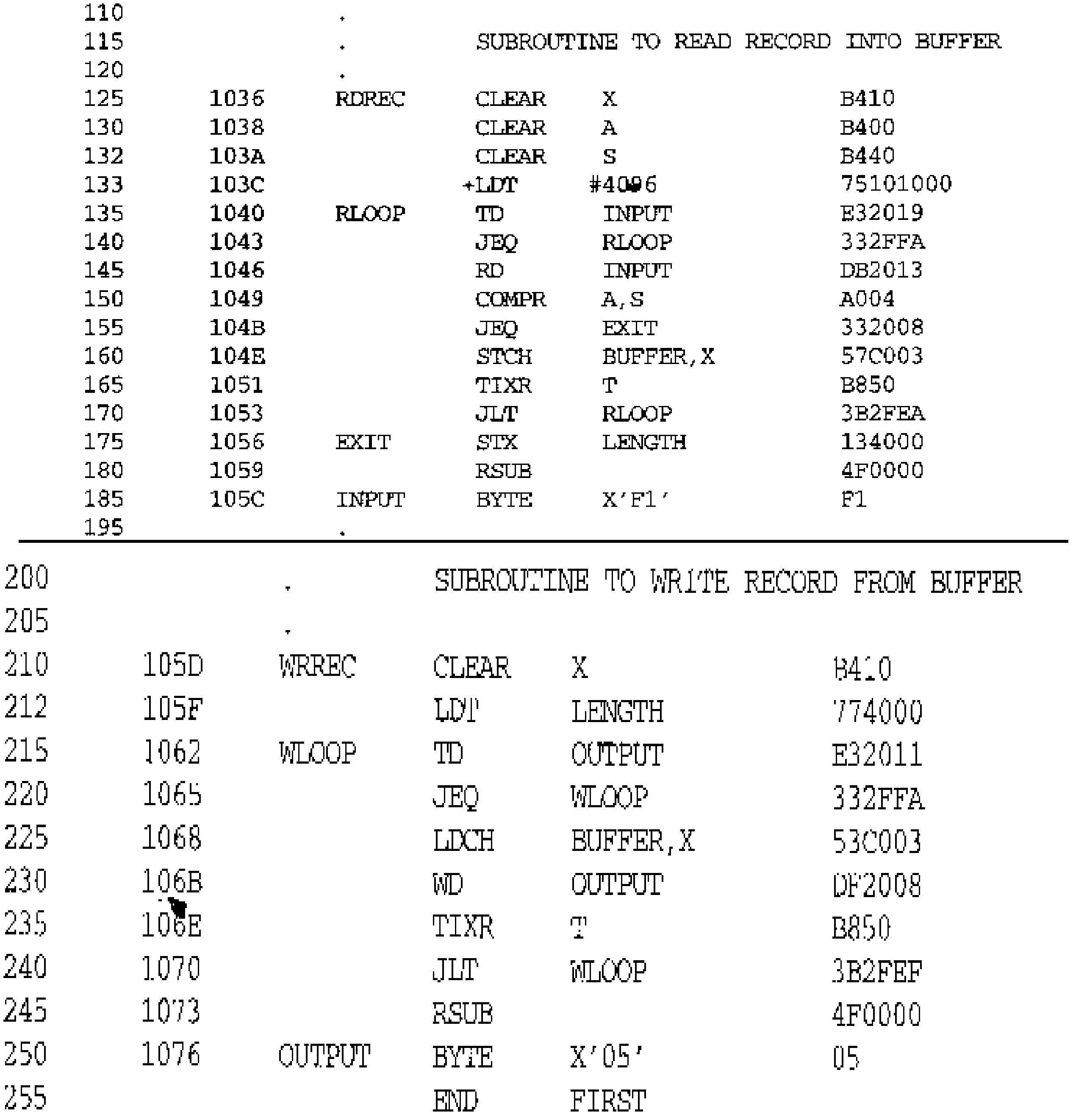


```
195       .
200       .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205       .
210     WRREC    CLEAR    X            CLEAR LOOP COUNTER
212              LDT      LENGTH
215     WLOOP    TD       OUTPUT       TEST OUTPUT DEVICE
220              JEQ      WLOOP        LOOP UNTIL READY
225              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
```



```
  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
```

SIC/XE Program with Object Code



```
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.

Object code sample calculation:

- Line 125: CLEAR=B4, r1=X=1, r2=0, obj=B410
- Line 133: LDT=74, n=0, i=1, op+ni=74+1=75, x=0, b=0, p=0, e=1àxbpe=1, #4096=01000, xbpe+address=101000, obj=75101000
- Line 160: STCH=54, n=1, i=1àni=3, op+ni=54+3=57, BUFFER=0036, B=0033, disp=BUFFER-B=003, x=1, b=1, p=0, e=0àxbpe=C, xbpe+disp=C003, obj=57C003

Program Relocation

The actual starting address of the program is not known until load time hence there may modification in the addresses as the assembler does not know the actual location where the program will be loaded. However, the assembler can identify to the loader those part 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.** Modification is not needed if operand is using program-counter relative or base relative addressing. The only parts of the program that require modification at load time are those that specified direct (as opposed to relative) addresses, which can be specified using modification record.

Modification record

Col. 1 M

Col. 2-7 Starting location of the address field to be modified, relative to the beginning of the program (Hex)

Col. 8-9 Length of the address field to be modified, in half-bytes (Hex)

Example of Program relocation

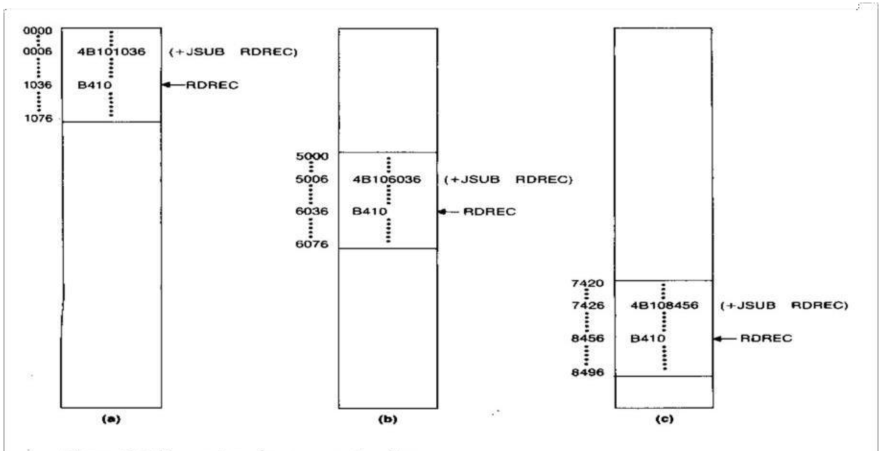
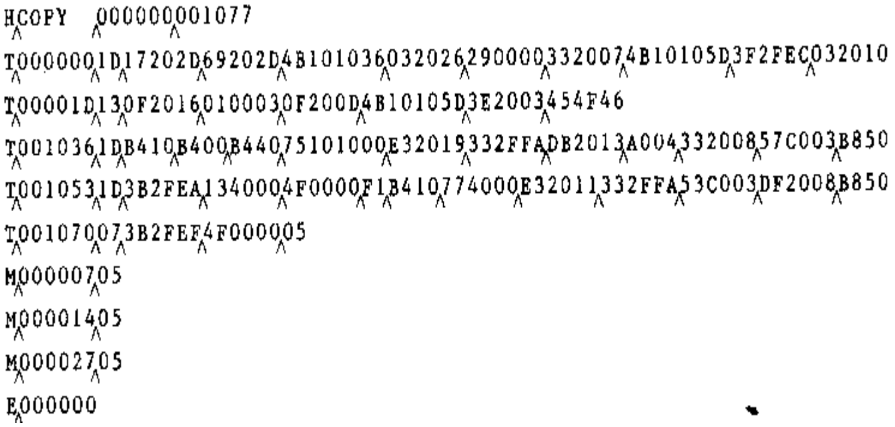


Figure 2.7 Examples of program relocation.

Object Program involving modification record



```
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
```

Machine-Independent Assembler Features

Literals:

To write the value of a constant operand as a part of the instruction that uses it, instead of having the constant defined elsewhere in the program and make up a label for it.Such an operand is called a literal.

A literal is identified with the prefix =, which is followed by a specification of the literal value Examples of literals in the statements:

```
45      001A ENDFIL             LDA    =C’EOF’    032010
215     1062    WLOOP       TD      =X’05’     E32011
```

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 the target address for the machine instruction. All of the literal operands used in the program are gathered together into one or more literal pools. Normally literals are placed into a pool at the end of the program. A LTORG statement creates a literal pool that contains all of the literal operands used since the previous LTORG. Most assembler recognizes duplicate literals: the same literal used in more than one place and store only one copy of the specified data value. LITTAB (literal 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 organized as a hash table, using literal name or value as the key.

Symbol-defining statements:

Most of the assembler provides assembler directive that allows the programmer to define symbols and specify their values. The directive generally used is EQU (for “equate”)

General form:

```
        Symbol          EQU             value
```

This statement defines the given symbol and assigns to it the value specified. The value can be constant or an expression.

example

```
        Line 133: +LDT          #4096
        MAXLEN                  EQU             4096
        +LDT            #MAXLEN
```

It is much easier to find and change the value of MAXLEN.

Another common use of EQU is defining mnemonics names for Registers

```
Example:        A               EQU             0
                X                EQU           1
```

The other common assembler directive that can be used to indirectly assign values to symbols is ORG.

General Form: ORG   value

Where value is constant or expression. When this statement is encountered during assembly of program, the assembler resets the location counter (LOCCTR) to the specified value. Since the values of symbols used as labels are taken from LOCCTR, the ORG statement will affect the values of all labels defined untill next ORG. example

```
STAB        RESB 1100
            ORG   STAB
SYMBOL      RESB  6
VALUE             RESW 1
FLAGS             RESW 2
            ORG   STAB+1100
```

The EQU and ORG statements pose restrictions :

In case EQU the symbols used on right hand side of the statement must have been defined previously. Example

```
ALPHA       RESW        1
BETA        EQU         ALPHA is valid
But
BETA        EQU         ALPHA
ALPHA       RESW        1     is not allowed
```

A similar restriction is posed by ORG statement i.e all symbols to define new loction counter value must have been previously defined.

Expressions

- Assembler allow arithmetic expressions formed according to the normal rules using the operator +, -, *, and /
- Individual terms in the expression may be constants, user-defined symbols, or special terms

Symbol	Type	Value
RETADR	R	0030
BUFFER	R	0036
BUFFEND	R	1036
MAXLEN	A	1000

- The most common such special term is the current value of the location counter (designed by *)
- Expressions are classified as either absolute expressions or relative expressions

Program block

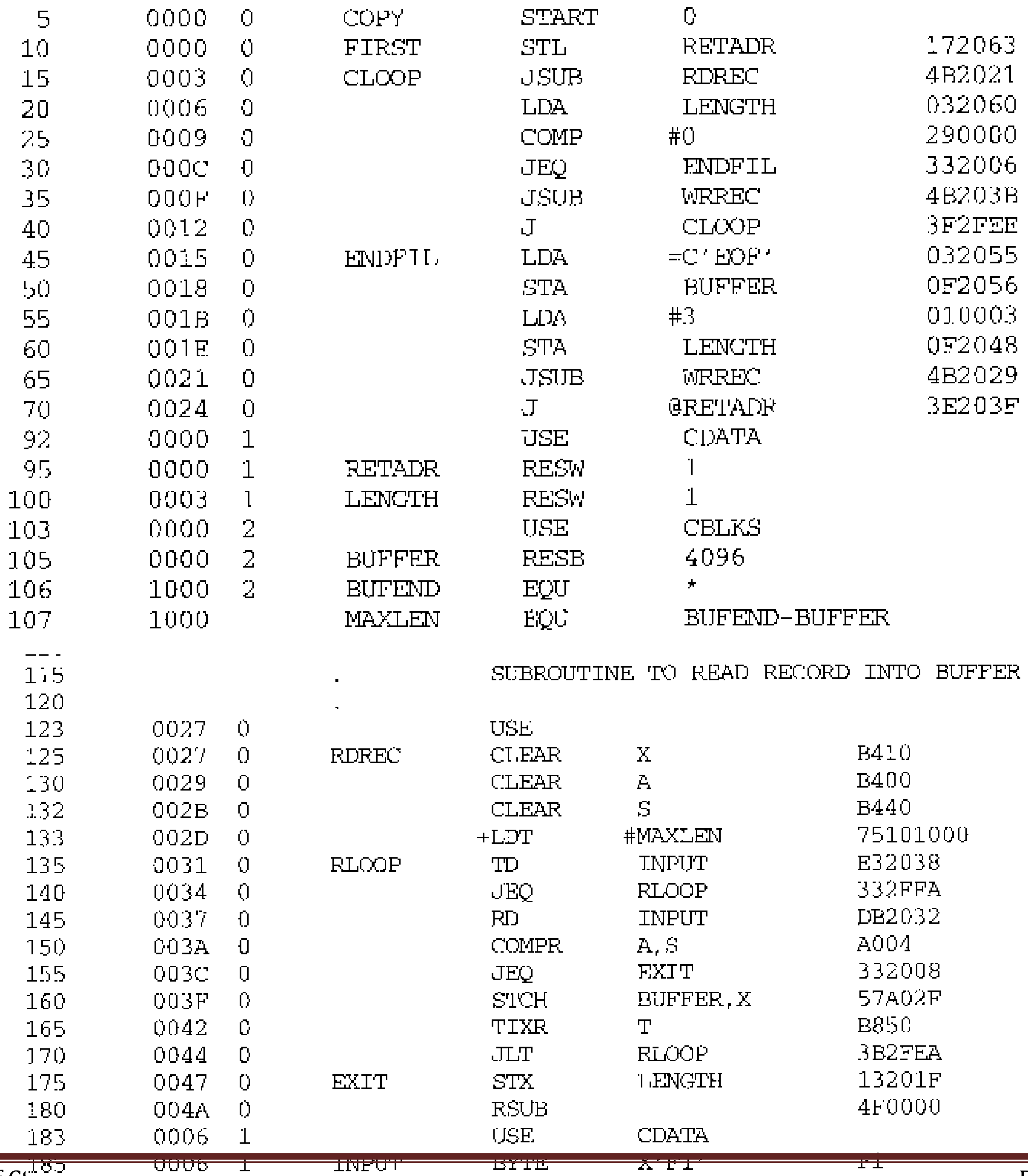
- Program blocks: segments of code that are rearranged within a single object unit
- Control sections: segments that are translated into independent object program units
- USE indicates which portions of the source program belong to the variousblocks

Block name	Block number	Address	Length
(default)	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

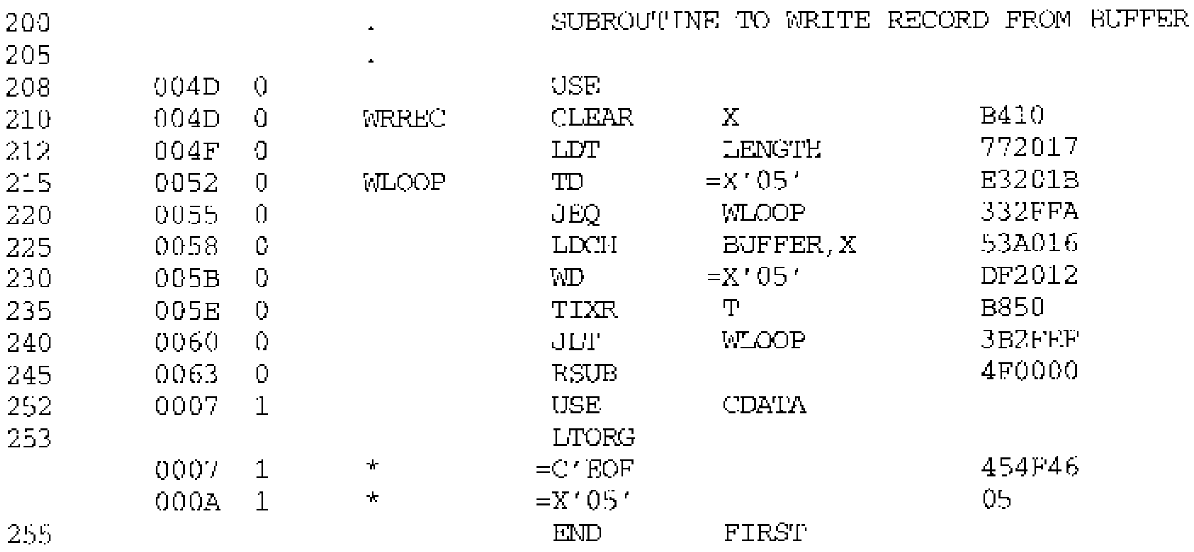
- Because the large buffer area is moved to the end of the object program, we no longer need to used extended format instructions
- Program readability is improved if the definition of data areas are placed in the source program close to the statements that reference them

- 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

Object code calculation for program blocks

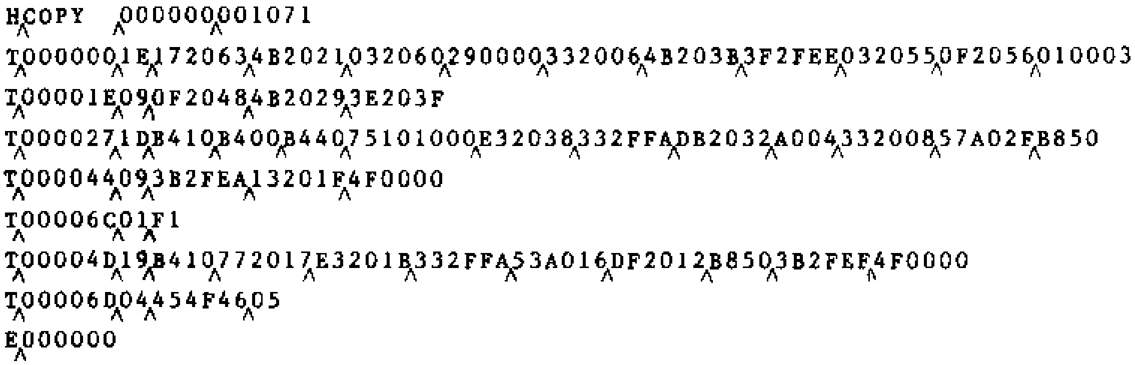


```
  5   0000  0    COPY      START    0
 10   0000  0    FIRST     STL      RETADR          172063
 15   0003  0    CLOOP     JSUB     RDREC           4B2021
 20   0006  0              LDA      LENGTH          032060
 25   0009  0              COMP    #0               290000
 30   000C  0              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
 ---
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
```

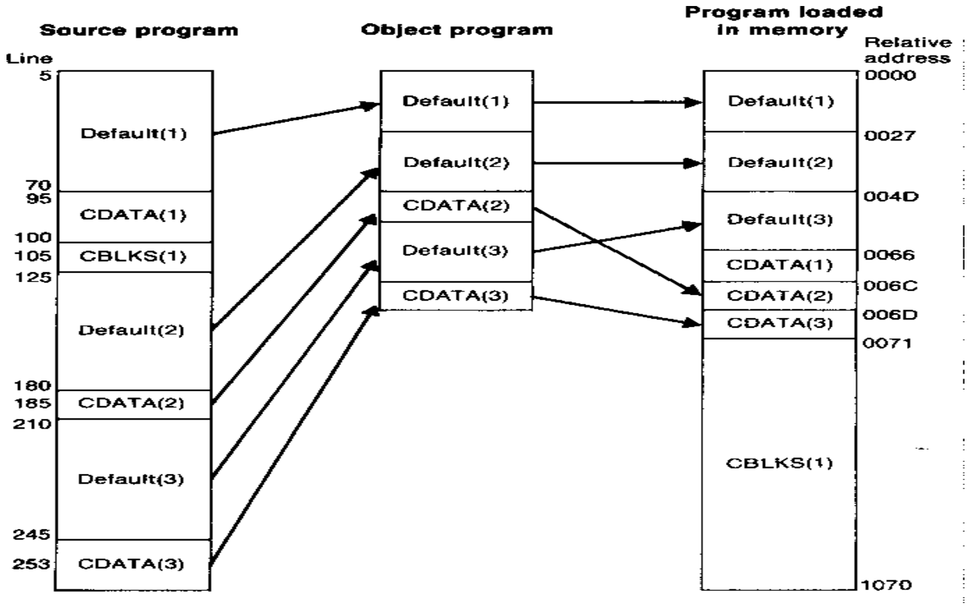


```
200                        .          SUBROUTINE TO WRITE RECORD FROM BUFFER
205                        .
208     004D  0                       USE
210     004D  0     WRREC             CLEAR      X                B410
212     004F  0                       LDT        LENGTH           772017
215     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
```

Object Program for program blocks



```
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
```

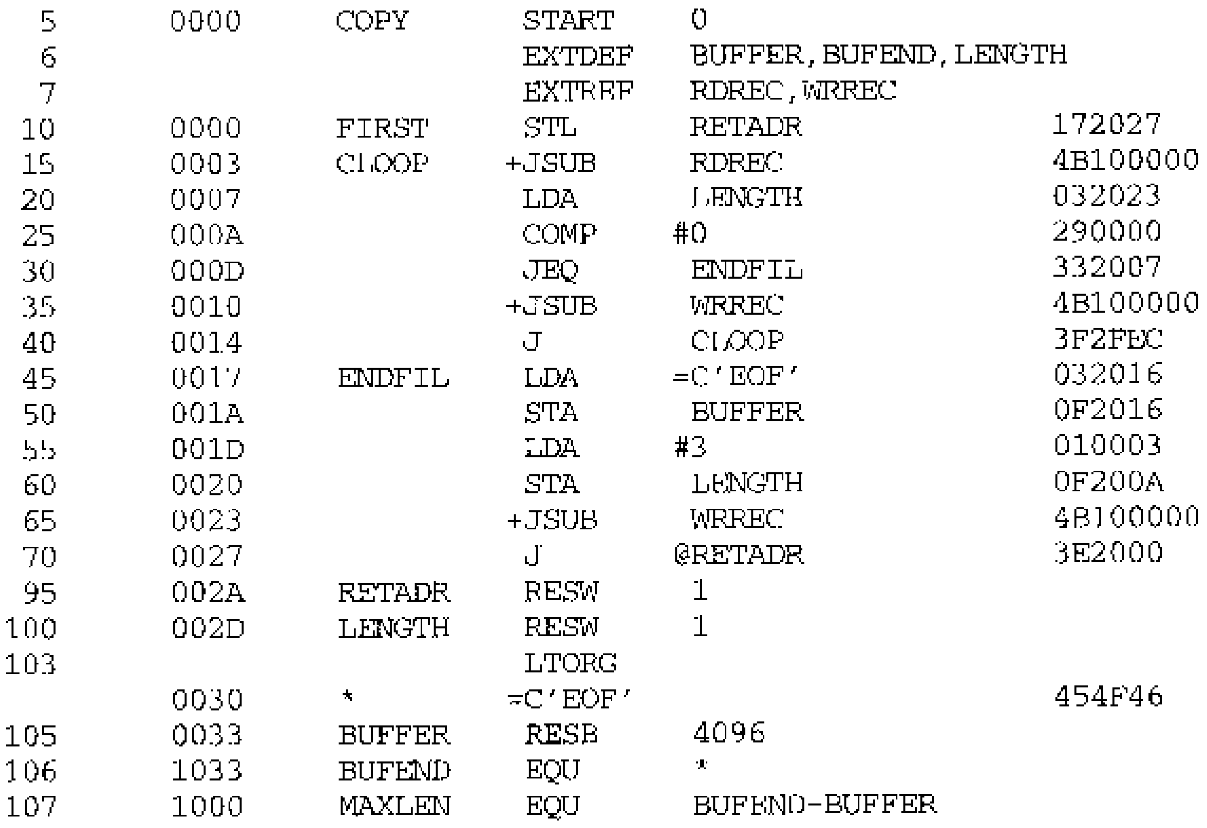


Control sections

- References between control sections are called external references
- The assembler generates information for each external reference that will allow the loader to perform the required linking
- The EXTDEF (external definition) statement in a control section names symbol, called external symbols, that are define in this section and may be used by other sections
- The EXTREF (external reference) statement names symbols that are used in this control section and are defined elsewhere
- Define record (D)
  - Col. 2-7 Name of external symbol defined in this control section
  - Col. 8-13 Relative address of symbol within this control section (Hex)
  - Col. 14-73 Repeat information in Col. 2-13 for other external symbols
- Refer record (R)
  - Col. 2-7 Name of external symbol referred to in this control section
  - Col. 8-73 Names of other external reference symbols
- Modification record (revised : M)
  - Col. 2-7 Starting address of the field to be modified, relative to the beginning of

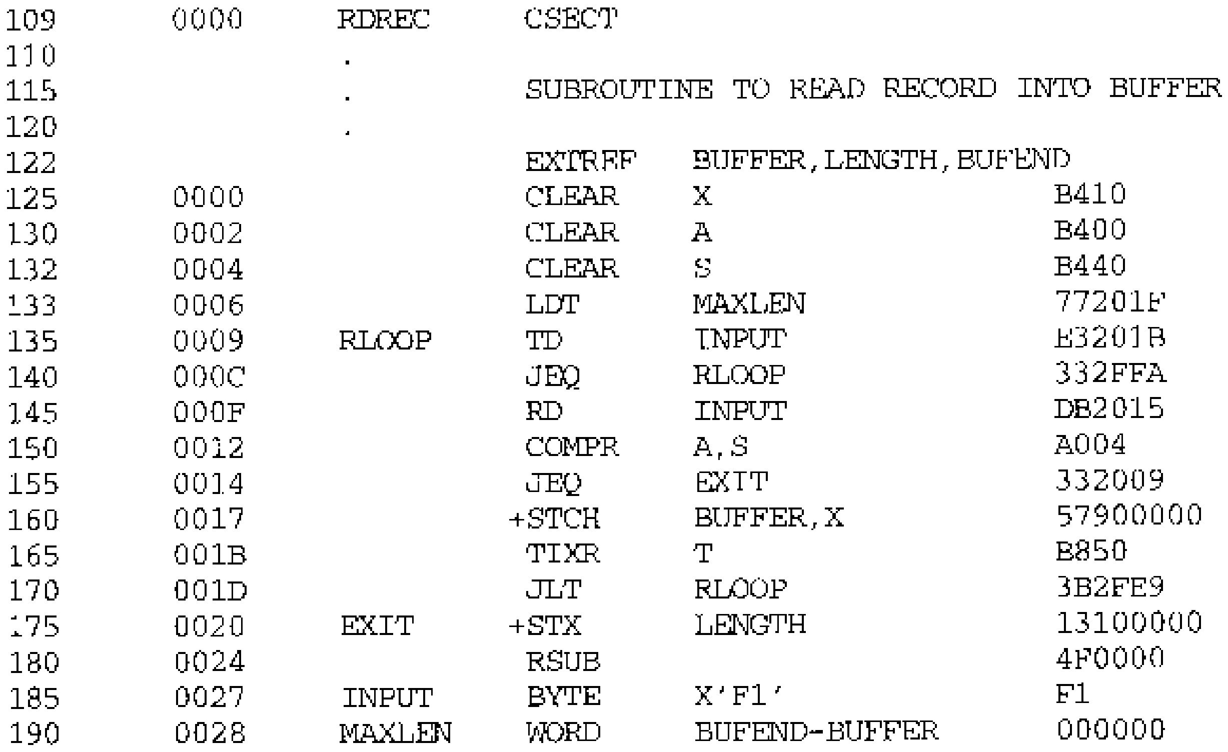
the control section (Hex)

- Col. 8-9 Length of the field to be modified, in half-bytes (Hex)
- Col. 10 Modification flag (+ or -)
- Col. 11-16 External symbol whose value is to be added to or subtracted from the indicated field

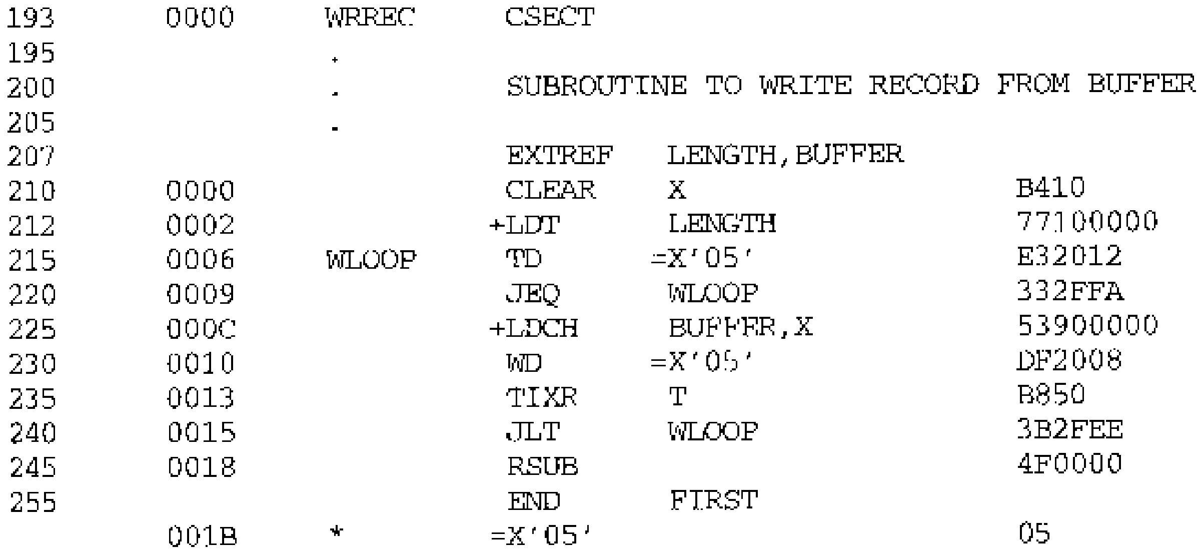


```
  5    0000    COPY     START    0
  6                     EXTDEF   BUFFER,BUFEND,LENGTH
  7                     EXTREF   RDREC,WRREC
 10    0000    FIRST    STL      RETADR                 172027
 15    0003    CLOOP   +JSUB     RDREC                  4B100000
 20    0007             LDA      LENGTH                 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
```

Object code for control sections

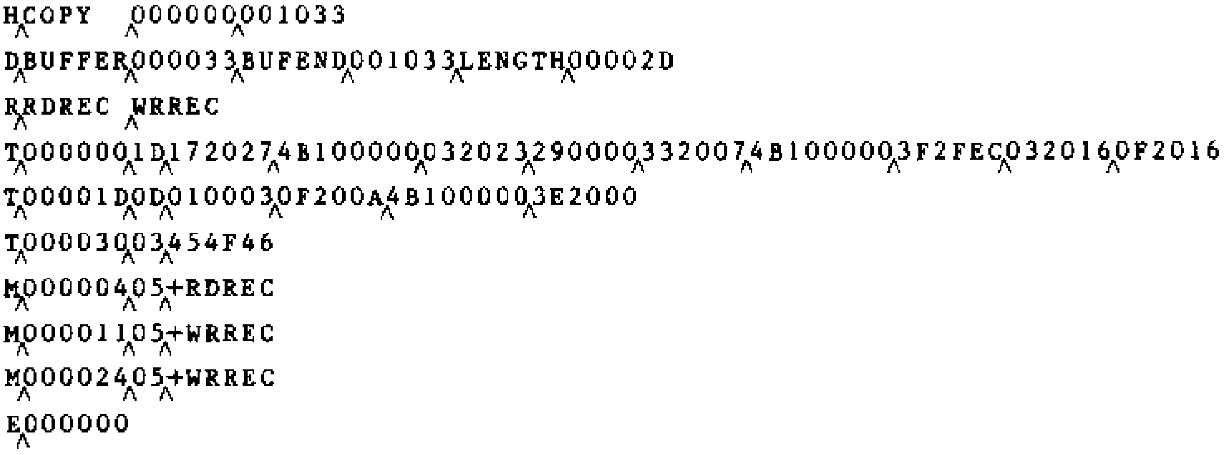


```
109    0000    RDREC    CSECT
110            .
115            .        SUBROUTINE TO READ RECORD INTO BUFFER
120            .
122                     EXTREF   BUFFER,LENGTH,BUFEND
125    0000             CLEAR    X                      B410
130    0002             CLEAR    A                      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
```

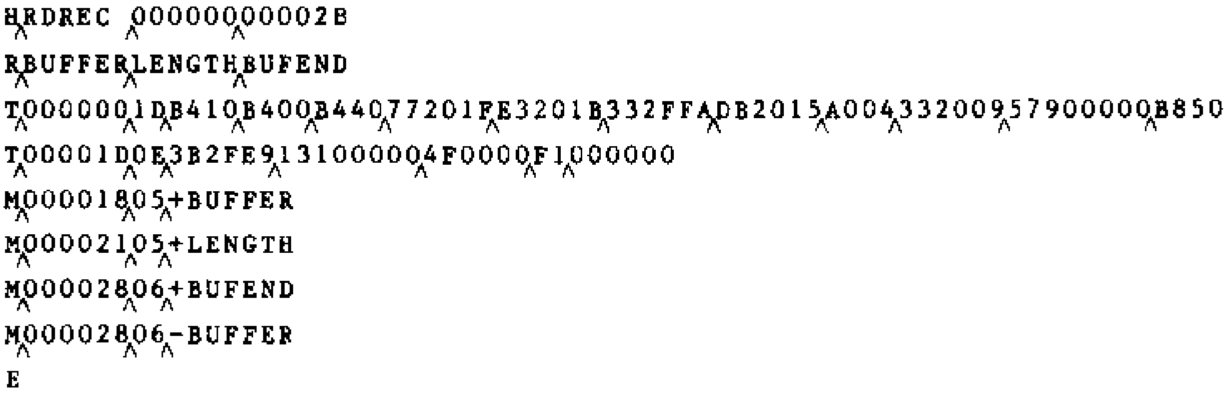


```
193    0000    WRREC    CSECT
195            .
200            .        SUBROUTINE TO WRITE RECORD FROM BUFFER
205            .
207                     EXTREF   LENGTH,BUFFER
210    0000             CLEAR    X                      B410
212    0002            +LDT      LENGTH                 77100000
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
```

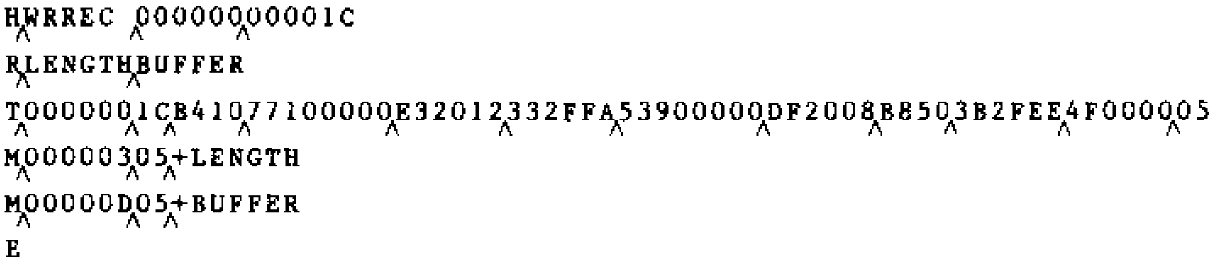
Object program for control section



```
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^4F0000^05
M^000003^05^+LENGTH
M^00000D^05^+BUFFER
E
```

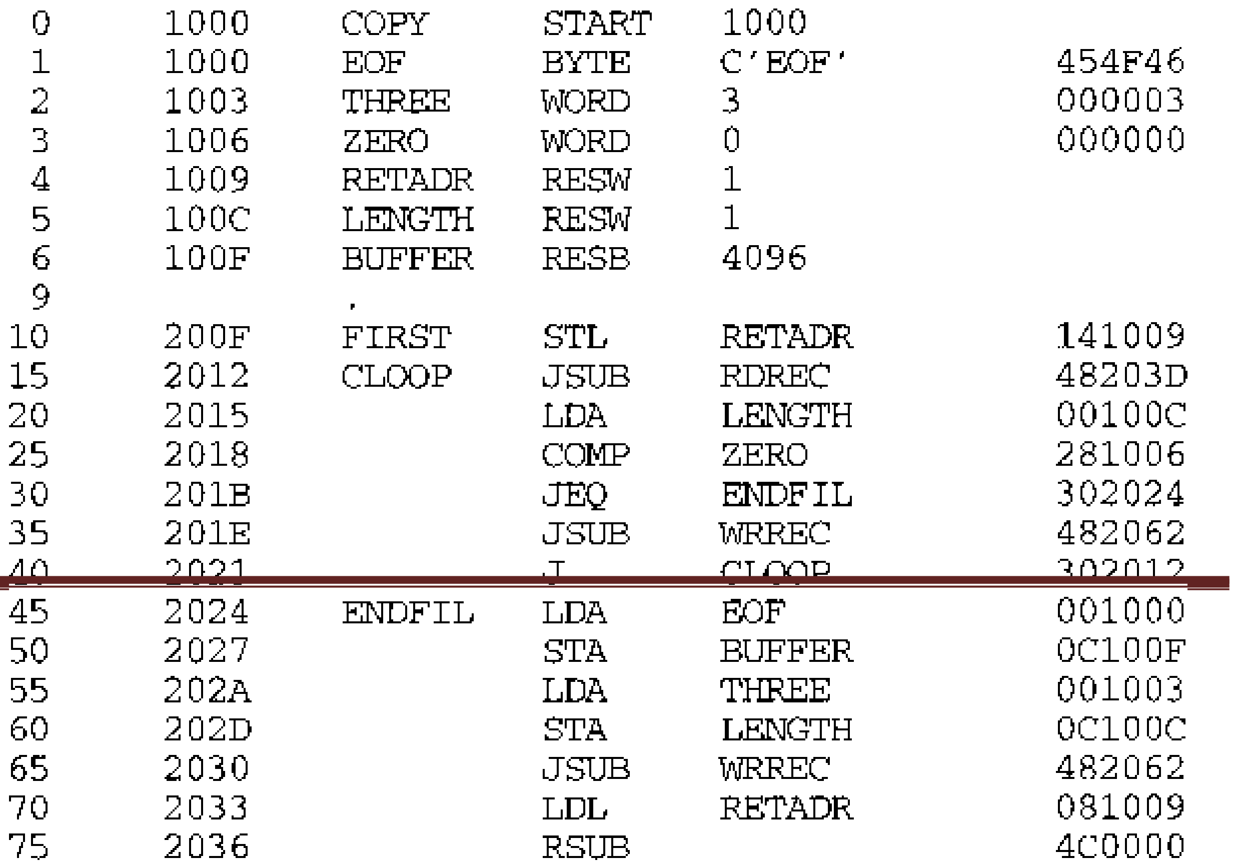
Assembler Design Options

One-Pass Assemblers

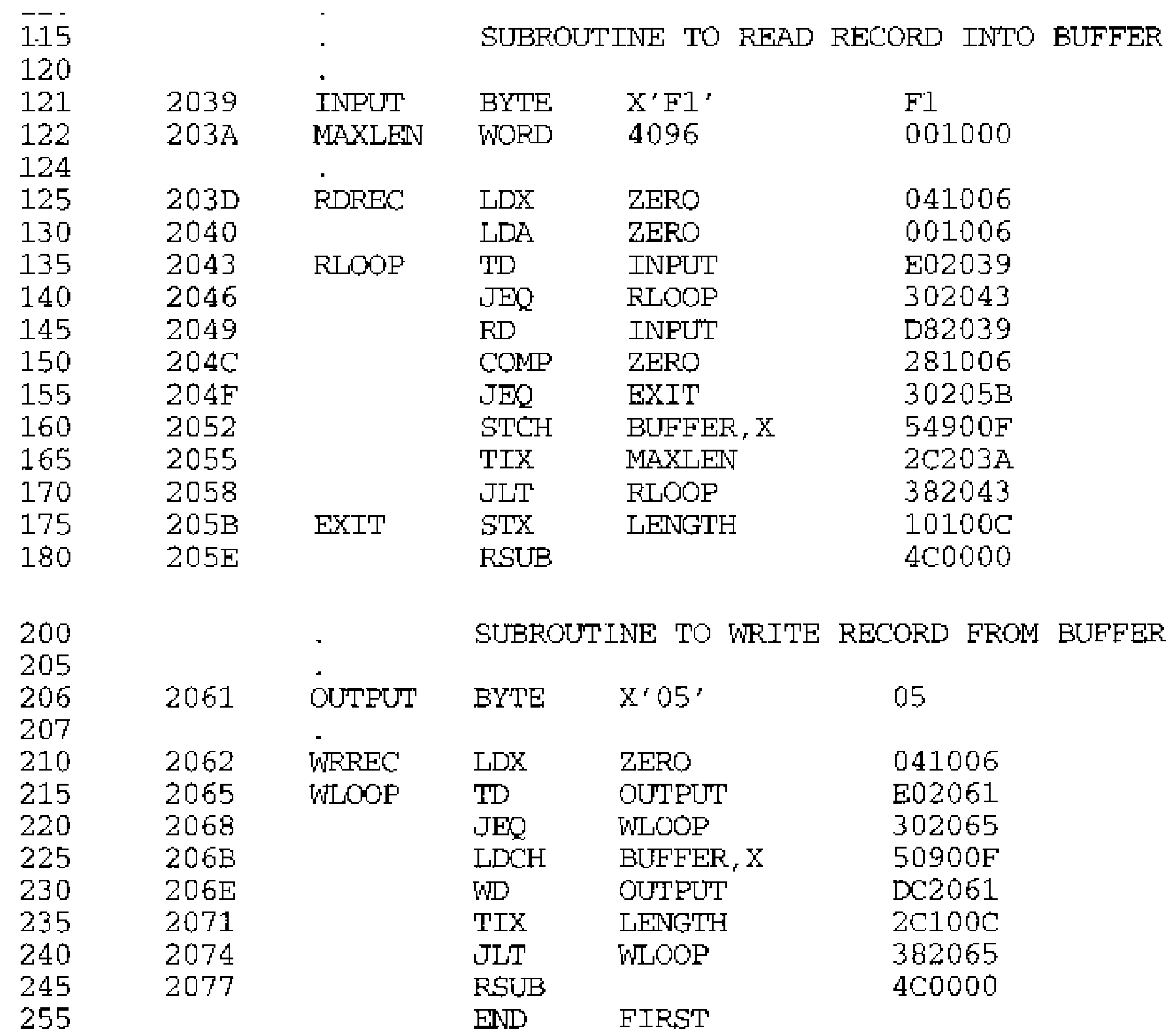
- Eliminate forward references: require that all such areas be defined in the source program before they are referenced
- One-pass assembler:
    - □ Generate their object code in memory for immediate execution
    - □ Load-and-go assembler is useful in a system that is oriented toward program development and testing
- Handle Forward Reference

- The symbol used as an operand is entered into the symbol table
- This entry is flagged to indicate that the symbol is undefined
- The address of the operand field of the instruction that refers to 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, and the proper address is inserted into any instructions previously generated

Sample Program for One-Pass assembler

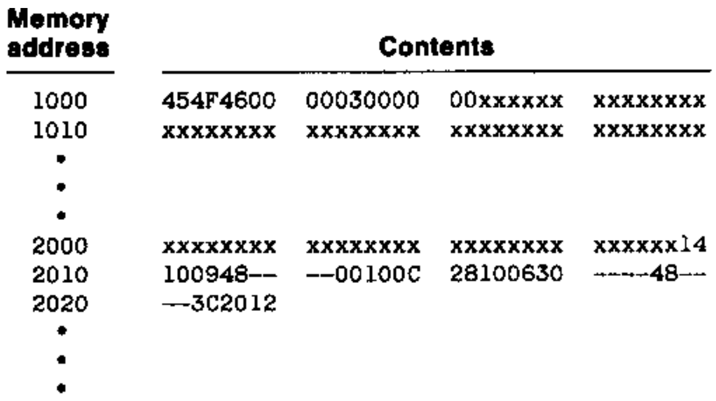


```
  0    1000    COPY     START   1000
  1    1000    EOF      BYTE    C'EOF'       454F46
  2    1003    THREE    WORD    3            000003
  3    1006    ZERO     WORD    0            000000
  4    1009    RETADR   RESW    1
  5    100C    LENGTH   RESW    1
  6    100F    BUFFER   RESB    4096
  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
```

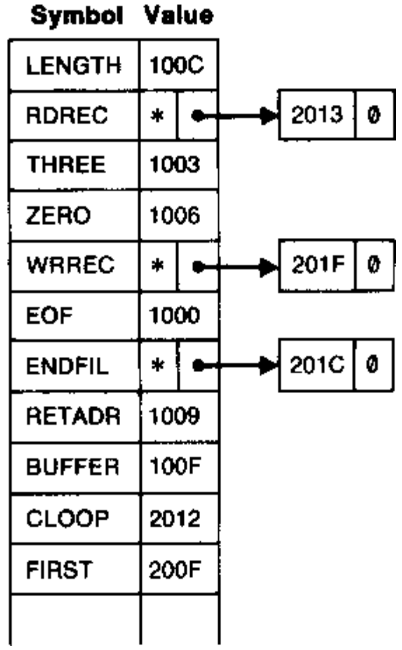


```
115                .         SUBROUTINE TO READ RECORD INTO BUFFER
120                .
121      2039      INPUT     BYTE      X'F1'            F1
122      203A      MAXLEN    WORD      4096             001000
124                .
125      203D      RDREC     LDX       ZERO             041006
130      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

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
```

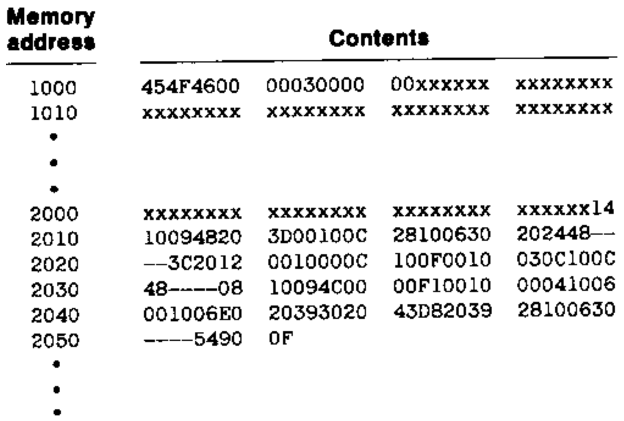
Example of Handling Forward Reference object code



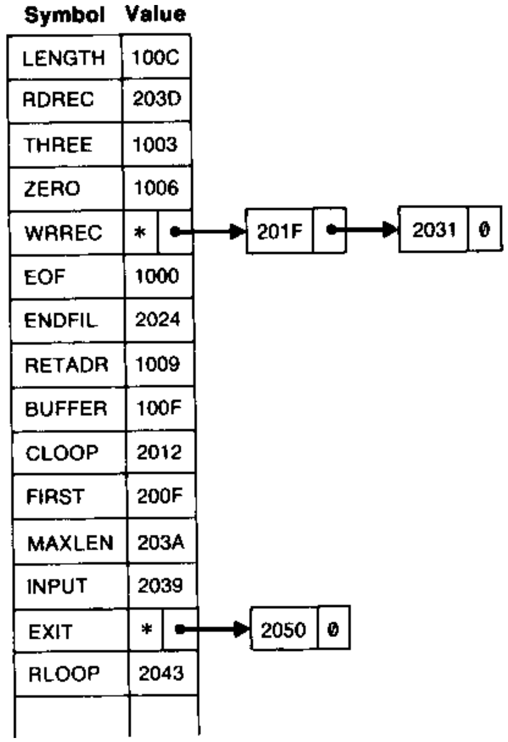
```
Memory
address                      Contents
 1000     454F4600  00030000  00xxxxxx  xxxxxxxx
 1010     xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
   •
   •
   •
 2000     xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxx14
 2010     100948--  --00100C  28100630  ----48--
 2020     --3C2012
   •
   •
   •
```



Symbol	Value		
LENGTH	100C		
RDREC	*	2013	0
THREE	1003		
ZERO	1006		
WRREC	*	201F	0
EOF	1000		
ENDFIL	*	201C	0
RETADR	1009		
BUFFER	100F		
CLOOP	2012		
FIRST	200F		



```
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
   •
   •
   •
```



Symbol	Value				
LENGTH	100C				
RDREC	203D				
THREE	1003				
ZERO	1006				
WRREC	*	201F		2031	0
EOF	1000				
ENDFIL	2024				
RETADR	1009				
BUFFER	100F				
CLOOP	2012				
FIRST	200F				
MAXLEN	203A				
INPUT	2039				
EXIT	*	2050	0		
RLOOP	2043				

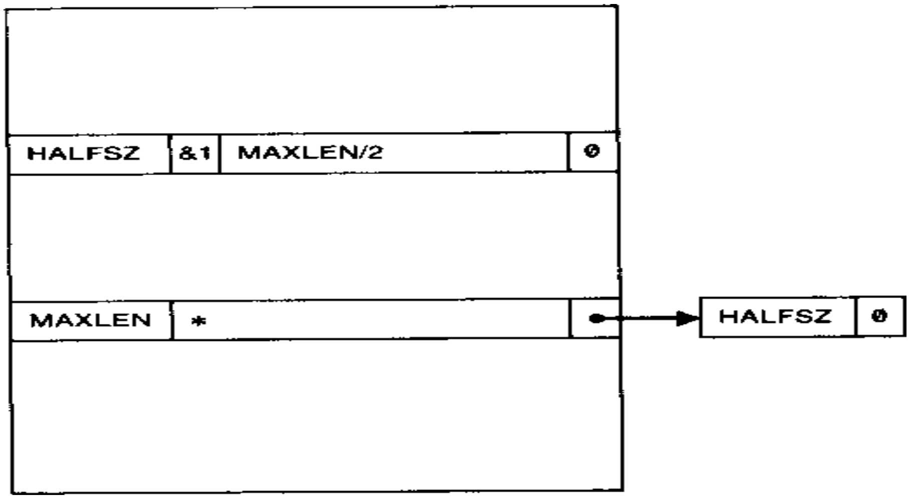
Multi-Pass Assemblers

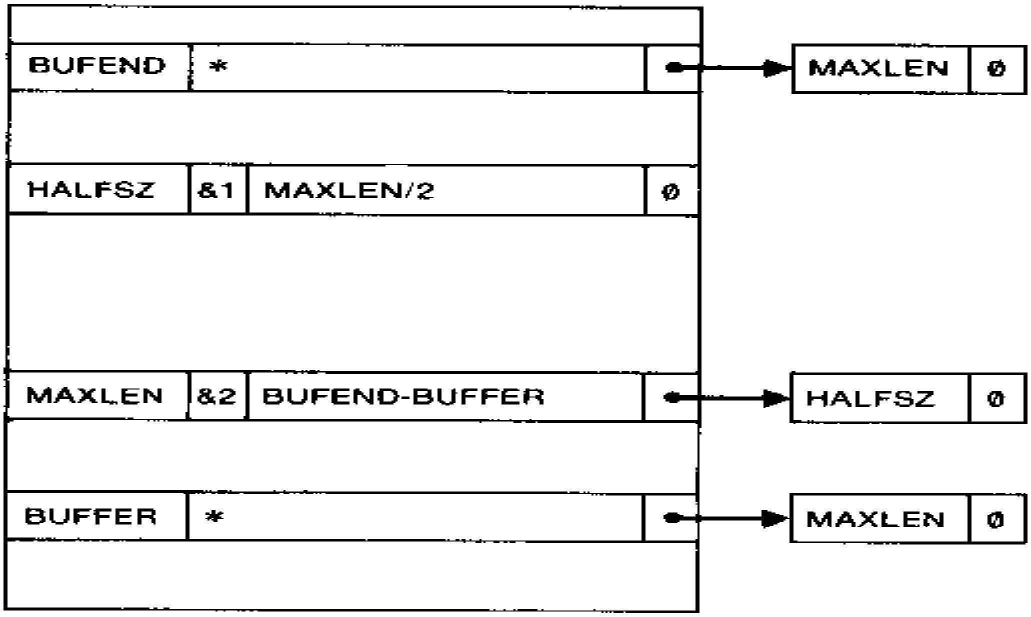
Any assembler that makes only two sequential passes over the source program cannot resolve such a sequence of definitions.

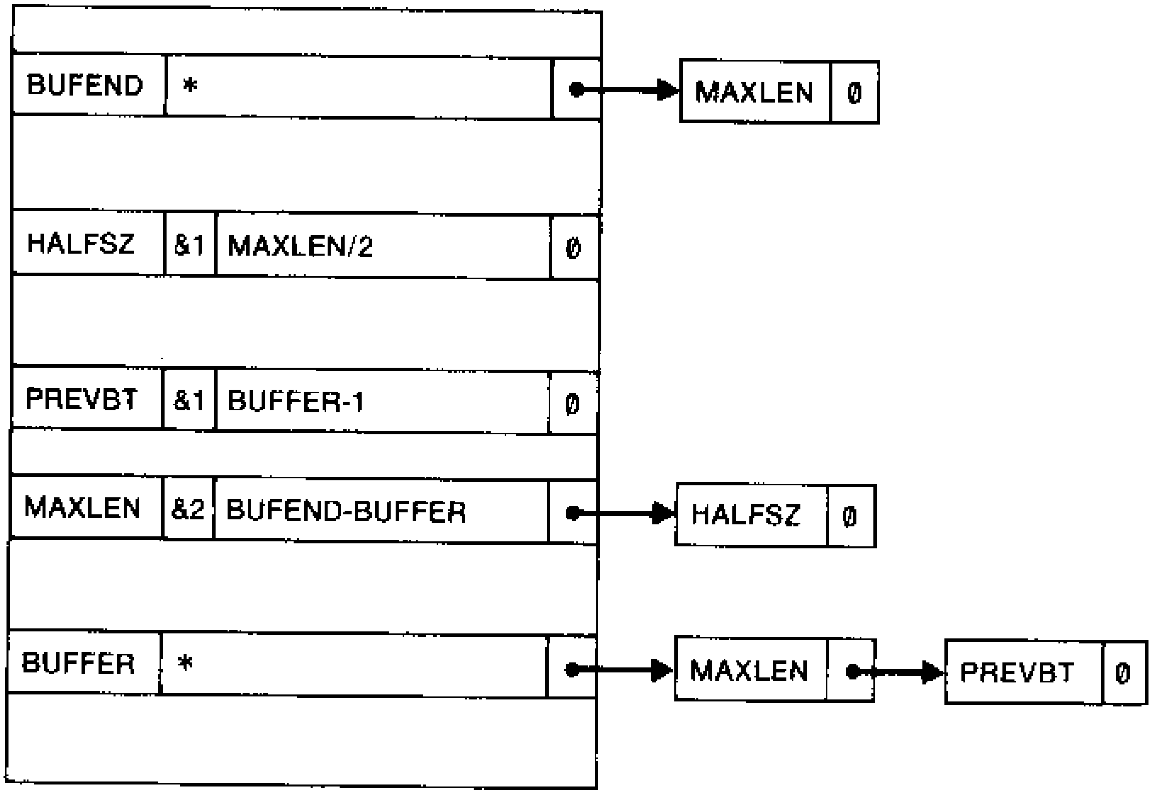
Restrictions such as prohibiting forward references in symbol definition are not normally a serious inconvenience for the programmer. some assemblers are designed to eliminate the need for restrictions.

The general solution is a multi pass assembler that can make as many passes as are needed to process the definition of symbols.

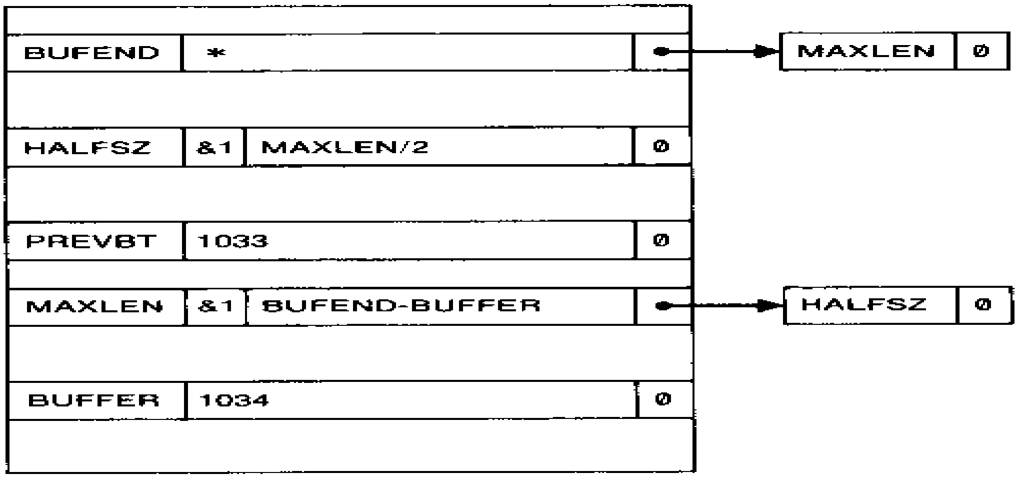
```
HALFSZ      EQU       MAXLEN/2

MAXLEN      EQU       BUFFEND-BUFFER

PREVBT      EQU       BUFFER-1

……….

BUFFER      RESB  4096

BUFFEND     EQU       *
```



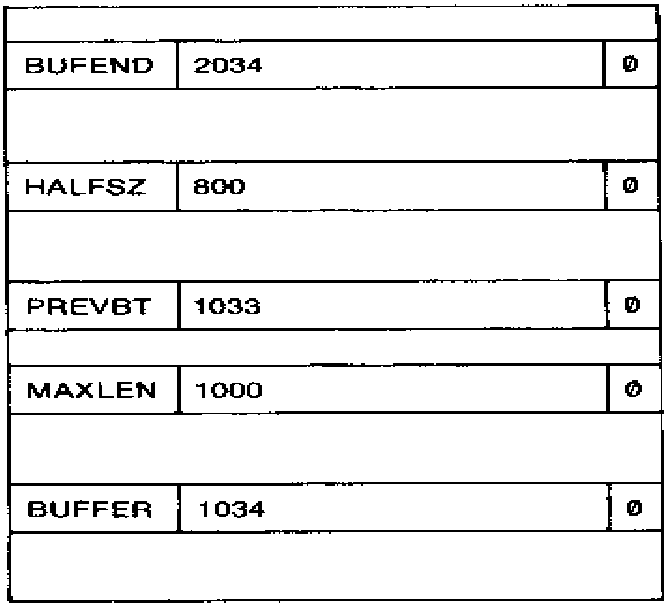

BUFEND
*
MAXLEN
Ø
HALFSZ
&1
MAXLEN/2
Ø
MAXLEN
&2
BUFEND-BUFFER
HALFSZ
Ø
BUFFER
*
MAXLEN
Ø


BUFEND
*
MAXLEN
Ø
HALFSZ
&1
MAXLEN/2
Ø
PREVBT
&1
BUFFER-1
Ø
MAXLEN
&2
BUFEND-BUFFER
HALFSZ
Ø
BUFFER
*
MAXLEN
PREVBT
Ø

Let us assume that when line 4 is read, the location counter contains the hexadecimal value 1034.this value is stored as the value of BUFFER.



BUFEND	*		•
HALFSZ	&1	MAXLEN/2	Ø
PREVBT	1033		Ø
MAXLEN	&1	BUFEND-BUFFER	•
BUFFER	1034		Ø



BUFEND	2034	Ø
HALFSZ	800	Ø
PREVBT	1033	Ø
MAXLEN	1000	Ø
BUFFER	1034	Ø

Introduction

The Source Program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution. This conversion either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies addresses in primary memory where these items are to be loaded for execution.

This contains the following three processes, and they are,

Loading - which allocates memory location and brings the object program into memory for execution - (Loader)

Linking- which combines two or more separate object programs and supplies the information needed to allow references between them - (Linker)

Relocation - which modifies the object program so that it can be loaded at an address different from the location originally specified - (Linking Loader)

Basic Loader Functions

A loader is a system program that performs the loading function. It brings object program into memory and starts its execution. The role of loader is as shown in the figure 3.1. In figure 3.1 translator may be assembler/complier, which generates the object program and later loaded to the memory by the loader for execution. In figure 3.2 the translator is specifically an assembler, which generates the object loaded, which becomes input to the loader. The figure 3.3 shows the role of both loader and linker.

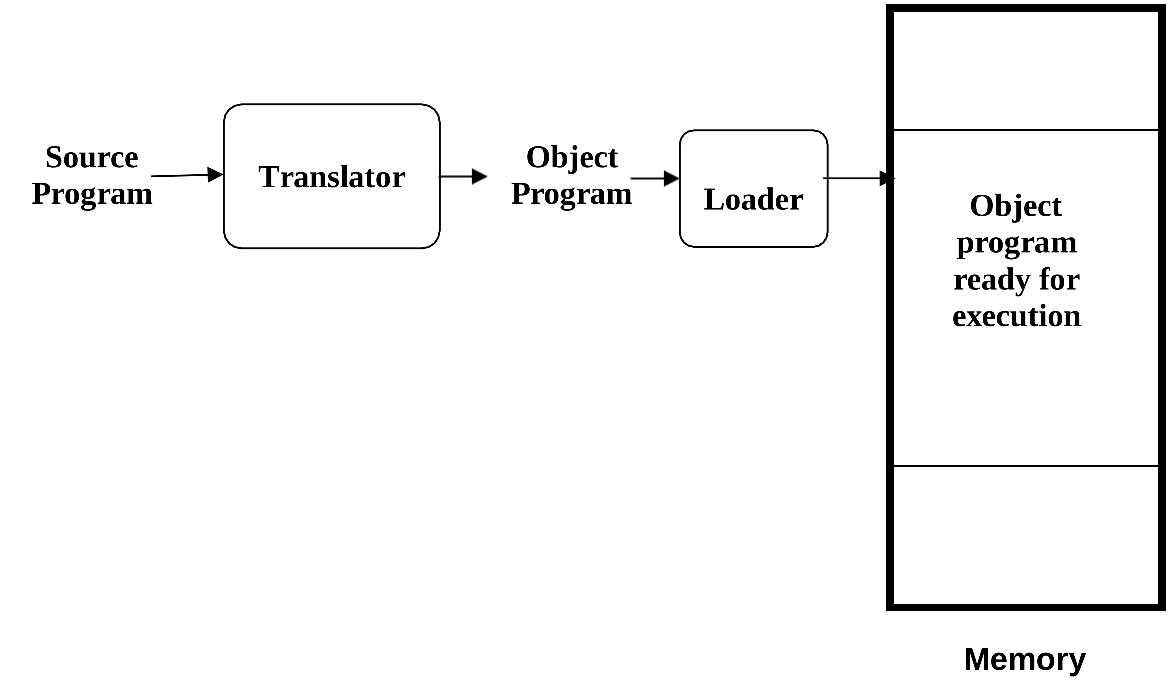


Figure 3.1 : The Role of Loader

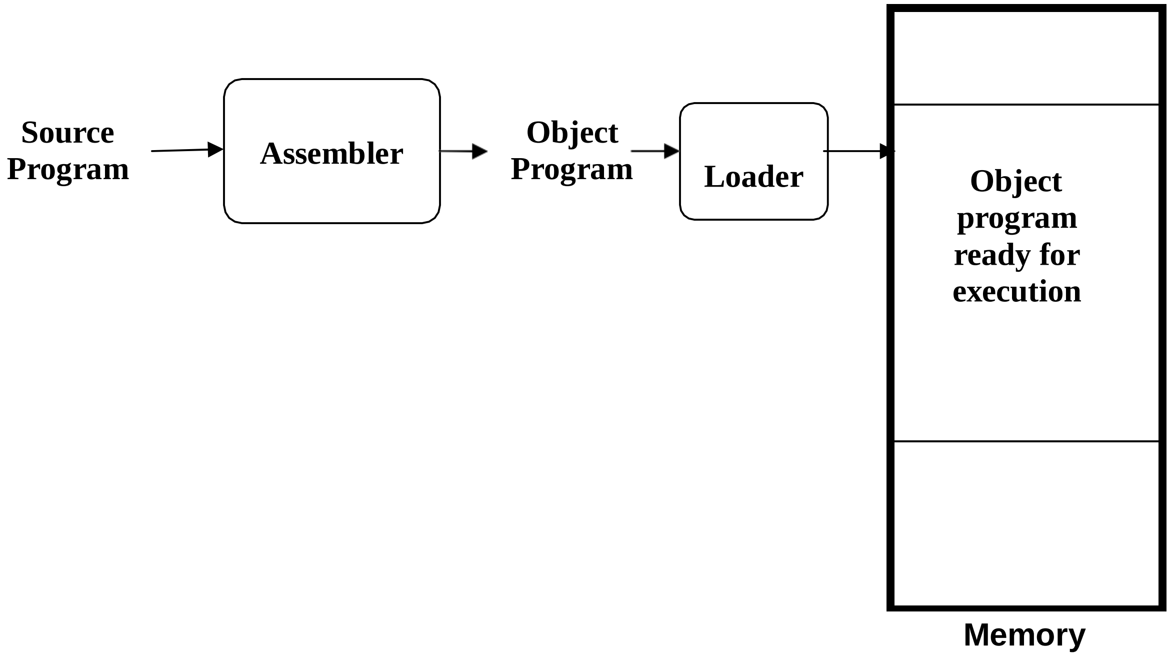
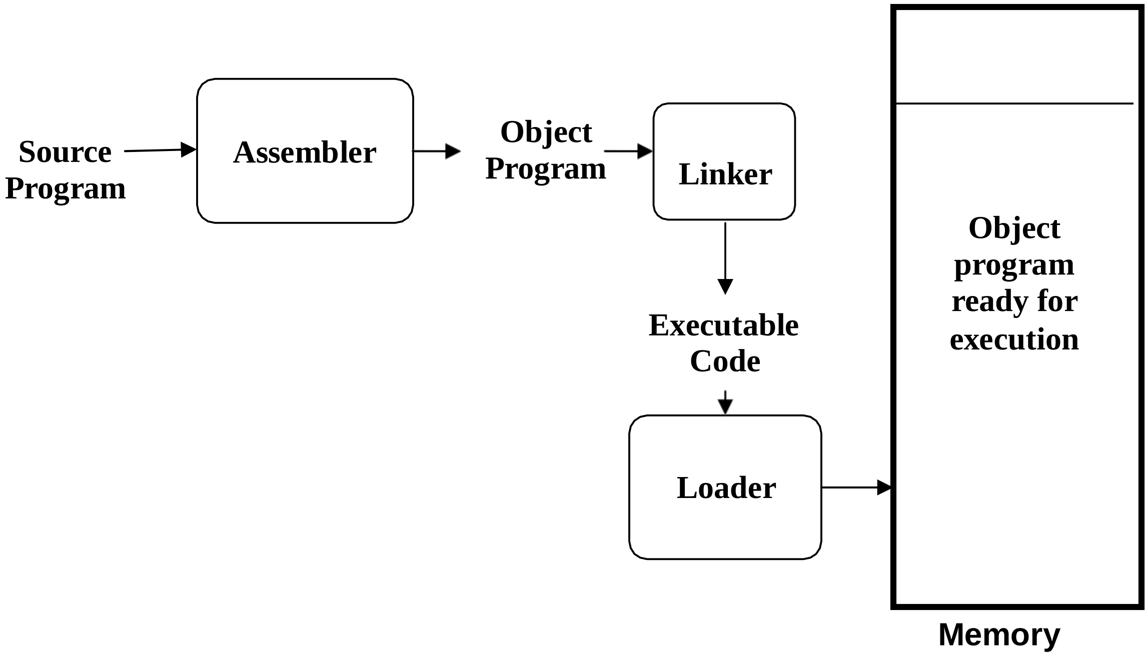


Figure 3.2: The Role of Loader with Assembler



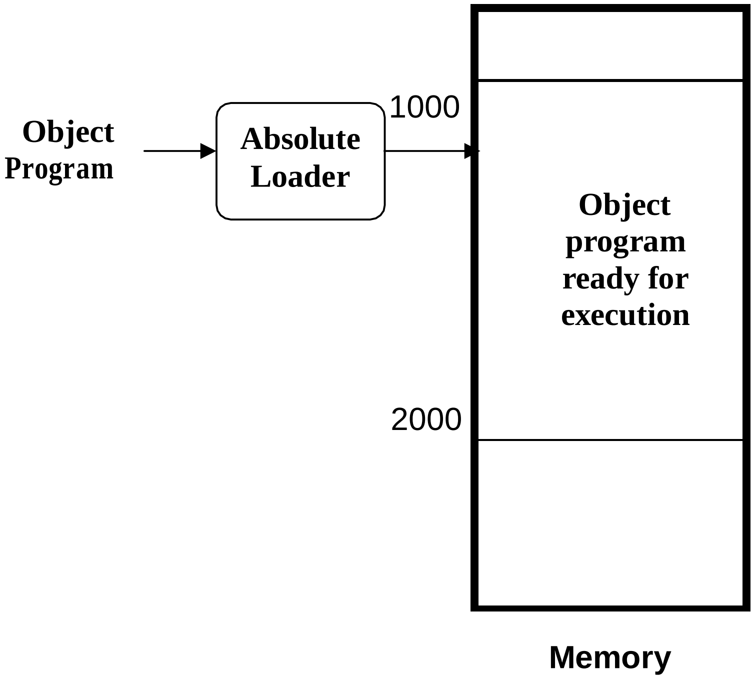
The Role of both Loader and Linker

Type of Loaders

The different types of loaders are, absolute loader, bootstrap loader, relocating loader (relative loader), and, direct linking loader. The following sections discuss the functions and design of all these types of loaders.

Absolute Loader

The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program. The role of absolute loader is as shown in the figure above The advantage of absolute loader is simple and efficient. But the disadvantages are, the need for programmer to specify the actual address, and, difficult to use subroutine libraries.



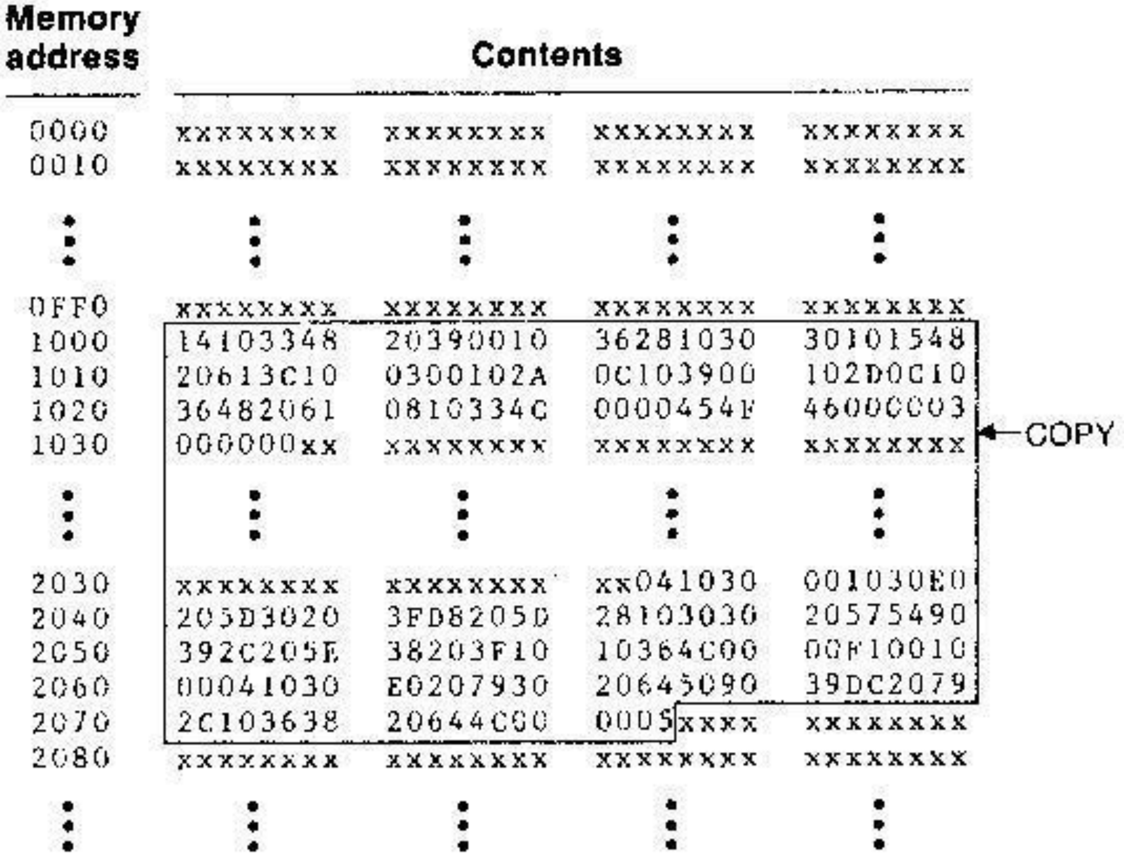
The Role of Absolute Loader

The algorithm for this type of loader is given here. The object program and, the object program loaded into memory by the absolute loader are also shown. Each byte of assembled code is given using its hexadecimal representation in character form. Easy to read by human beings. Each byte of object code is stored as a single byte. Most machine store object programs in a binary form, and we must be sure that our file and device conventions do not cause some of the program bytes to be interpreted as control characters.

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

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

(a) Object program



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

(b) Program loaded in memory

A Simple Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer -- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 0x80. No header record or control information, the object code is consecutive bytes of memory.

The algorithm for the bootstrap loader is as follows

```
Begin
X=0x80 (the address of the next memory location to be loaded
Loop
            A←GETC (and convert it from the ASCII character
           code to the value of the hexadecimal digit)
            save the value in the high-order 4 bits of S
            A←GETC
            combine the value to form one byte A← (A+S)
            store the value (in A) to the address in register X
            X←X+1
End
```

It uses a subroutine GETC, which is

```
GETC      A←read one character
           if A=0x04 then jump to 0x80
           if A<48 then GETC
           A ← A-48 (0x30)
           if A<10 then return
           A ← A-7
           return
```