Question Bank-II Internals

Module-I

1. **Define program block. How are they handled by an assembler? Explain with an example.**

Program blocks allow the generated machine instructions and data to appear in the object program in a different order by Separating blocks for storing code, data, stack, and larger data block.

Assembler Directive USE:

USE [blockname]

At the beginning, statements are assumed to be part of the unnamed (default) block. If no USE statements are included, the entire program belongs to this single block. Each program block may actually contain several separate segments of the source program. Assemblers rearrange these segments to gather together the pieces of each block and assign address. Separate the program into blocks in a particular order. Large buffer area is moved to the end of the object program. Program readability is better if data areas are placed in the source program close to the statements that reference them.

In the example below three blocks are used :

Default: executable instructions

CDATA: all data areas that are less in length

CBLKS: all data areas that consists of larger blocks of memory

Example Code

(default) block   Block number

0000	0	COPY	START	0		
0000	0	FIRST	STL	RETADR	172063	
0003	0	CLOOP	JSUB	RDREC	4B2021	
0006	0		LDA	LENGTH	032060	
0009	0		COMP	#0	290000	
000C	0		JEQ	ENDFIL	332006	
000F	0		JSUB	WRREC	4B203B	
0012	0		J	CLOOP	3F2FEE	
0015	0	ENDFIL	LDA	=C'EOF'	032055	
0018	0		STA	BUFFER	0F2056	
001B	0		LDA	#3	010003	
001E	0		STA	LENGTH	0F2048	
0021	0		JSUB	WRREC	4B2029	
0024	0		J	@RETADR	3E203F	
0000	1		USE	CDATA		CDATA block
0000	1	RETADR	RESW	1		
0003	1	LENGTH	RESW	1		
0000	2		USE	CBLKS		CBLKS block
0000	2	BUFFER	RESB	4096		
1000	2	BUFEND	EQU	*		
1000		MAXLEN	EQU	BUFEND-BUFFER		

```
                                              (default) block
0027    0    RDREC    USE
0027    0             CLEAR     X           B410
0029    0             CLEAR     A           B400
002B    0             CLEAR     S           B440
002D    0            +LDT       #MAXLEN     75101000
0031    0    RLOOP    TD        INPUT       E32038
0034    0             JEQ       RLOOP       332FFA
0037    0             RD        INPUT       DB2032
003A    0             COMPR     A,S         A004
003C    0             JEQ       EXIT        332008
003F    0             STCH      BUFFER,X    57A02F
0042    0             TIXR      T           B850
0044    0             JLT       RLOOP       3B2FEA
0047    0    EXIT     STX       LENGTH      13201F
004A    0             RSUB                  4F0000
0006    1             USE       CDATA       CDATA block
0006    1    INPUT    BYTE      X'F1'       F1
```

```
                                              (default) block
004D    0             USE
004D    0    WRREC    CLEAR     X           B410
004F    0             LDT       LENGTH      772017
0052    0    WLOOP    TD        =X'05'      E3201B
0055    0             JEQ       WLOOP       332FFA
0058    0             LDCH      BUFFER,X    53A016
005B    0             WD        =X'05'      DF2012
005E    0             TIXR      T           B850
0060    0             JLT       WLOOP       3B2FEF
0063    0             RSUB                  4F0000
0007    1             USE       CDATA       CDATA block
                      LTORG
0007    1    *       =C'EOF                 454F46
000A    1    *       =X'05'                 05
                      END       FIRST
```

Arranging code into program blocks:

Pass 1:

- A separate location counter for each program block is maintained.
- Save and restore LOCCTR when switching between blocks.
- At the beginning of a block, LOCCTR is set to 0.
- Assign each label an address relative to the start of the block.
- Store the block name or number in the SYMTAB along with the assigned relative address of the label
- Indicate the block length as the latest value of LOCCTR for each block at the end of Pass1
- Assign to each block a starting address in the object program by concatenating the program blocks in a particular order

Pass 2

- Calculate the address for each symbol relative to the start of the object program by adding the location of the symbol relative to the start of its block
- The starting address of this block

2. Define control section. How are they handled by an assembler? Explain with an example.

- A control section is a part of the program that maintains its identity after assembly; each control section can be loaded and relocated independently of the others. Different control sections are most often used for subroutines or other logical subdivisions.
- The programmer can assemble, load, and manipulate each of these control sections separately. Because of this, there should be some means for linking control sections together. For example, instructions in one control section may refer to the data or instructions of other control sections.
- Since control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. Such references between different control sections are called external references.
- The assembler generates the information about each of the external references that will allow the loader to perform the required linking. When a program is written using multiple control sections, the beginning of each of the control section is indicated by an assembler directive : CSECT.
- The Syntax is

  secname CSECT – separate location counter for each control section
- The external references are indicated by two assembler directives:
  EXTDEF (external Definition): It is the statement in a control section, names symbols that are defined in this section but may be used by other control sections.
  EXTREF (external Reference):
- The assembler must include proper information about the external references in the object program that will cause the loader to insert the proper value where they are required. It maintains two new records in the object code and a changed version of modification record.
- Define Record(EXTDEF):

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

  Refer record (EXTREF)

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

  Modification record

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

  A define record gives information about the external symbols that are defined in this control section, i.e., symbols named by EXTDEF.A refer record lists the symbols that are used as external references by the control section, i.e., symbols named by EXTREF. The new items in the modification record specify the modification to be performed: adding or subtracting the value of some external symbol. The symbol used for modification may be defined either in this control section or in another section.

```
0000    COPY      START    0
                  EXTDEF   BUFFER,BUFFEND,LENGTH
                  EXTREF   RDREC,WRREC
0000    FIRST     STL      RETADR                172027
0003    CLOOP    +JSUB     RDREC                 4B100000    Case 1
0007              LDA      LENGTH                032023
000A              COMP    #0                     290000
000D              JEQ      ENDFIL                332007
0010             +JSUB     WRREC                 4B100000
0014              J        CLOOP                 3F2FEC
0017    ENDFIL    LDA     =C'EOF'                032016
001A              STA      BUFFER                0F2016
001D              LDA     #3                     010003
0020              STA      LENGTH                0F200A
0023             +JSUB     WRREC                 4B100000
0027              J        @RETADR               3E2000
002A    RETADR    RESW     1
002D    LENGTH    RESW     1
                  LTORG
0030    *        =C'EOF'                         454F46
0033    BUFFER    RESB     4096
1033    BUFEND    EQU      *
1000    MAXLEN    EQU      BUFEND-BUFFER
```

```
0000    RDREC     CSECT
          .
          .       SUBROUTINE TO READ RECORD INTO BUFFER

                  EXTREF   BUFFER,LENGTH,BUFEND
0000              CLEAR    X                     B410
0002              CLEAR    A                     B400
0004              CLEAR    S                     B440
0006              LDT      MAXLEN                77201F
0009    RLOOP     TD       INPUT                 E3201B
000C              JEQ      RLOOP                 332FFA
000F              RD       INPUT                 DB2015
0012              COMPR    A,S                   A004
0014              JEQ      EXIT                  332009
0017             +STCH     BUFFER,X              57900000
001B              TIXR     T                     B850
001D              JLT      RLOOP                 3B2FE9
0020    EXIT     +STX      LENGTH                13100000
0024              RSUB                           4F0000
0027    INPUT     BYTE     X'F1'                 F1
0028    MAXLEN    WORD     BUFFEND-BUFFER        000000      Case 2
```

```
0000    WRREC     CSECT

          .
          .       SUBROUTINE TO WRITE RECORD FROM BUFFER

                  EXTREF   LENGTH,BUFFER
0000              CLEAR    X                     B410
0002             +LDT      LENGTH                77100000
0006    WLOOP     TD      =X'05'                 E32012
0009              JEQ      WLOOP                 332FFA
000C             +LDCH     BUFFER,X              53900000
0010              WD      =X'05'                 DF2008
0013              TIXR     T                     B850
0015              JLT      WLOOP                 3B2FEE
0018              RSUB                           4F0000
                  END      FIRST
001B    *        =X'05'                          05
```

Handling External Reference :

Case 1 :

0003 CLOOP +JSUB RDREC 4B100000

- The operand RDREC is an external reference

  The assembler has no idea where RDREC is

Inserts an address of zero
Can only use extended format to provide enough room (that is, relative addressing for external reference is invalid)
The assembler generates information for each external reference that will allow the loader to perform the required linking.

Case 2

0028 MAXLEN WORD BUFEND-BUFFER 000000

There are two external references in the expression, BUFEND and BUFFER.
The assembler inserts a value of zero
Passes information to the loader
Add to this data area the address of BUFEND
Subtract from this data area the address of BUFFER

The object program:

```
COPY
HCOPY  000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC WRREC
T0000001D1720274B1000000320232900003320074B1000003F2FEC0320160F2016
T00001D0D0100030F200A4B1000003E2000
T00003003454F46
M00000405+RDREC
M00001105+WRREC
M00002405+WRREC
E000000

RDREC
HRDREC 00000000002B
RBUFFERLENGTHBUFEND
T0000001DB410B400B44077201FE3201B332FFADB2015A00433200957900000B850
T00001D0E3B2FE9131000004F0000F1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFEND
M00002806-BUFFER      BUFEND - BUFFER
E
WRREC
HWRREC 00000000001C
RLENGTHBUFFER
T0000001CB41077100000E3201232FFA53900000DF2008B8503B2FEE4F000005
M00000305+LENGTH
M00000D05+BUFFER
E
```

3. Write an algorithm for One-Pass Assembler. Explain how forward reference problem is handled in One-Pass assembler.

There are two types of one-pass assemblers:

One that produces object code directly in memory for immediate execution (Load and-go assemblers).
The other type produces the usual kind of object code for later execution.

Load-and-Go Assembler:

- Load-and-go assembler generates their object code in memory for immediate execution.
- No object program is written out, no loader is needed.

- It is useful in a system with frequent program development and testing
- The efficiency of the assembly process is an important consideration.
- Programs are re-assembled nearly every time they are run; efficiency of the assembly process is an important consideration.

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		.			

Forward Reference in One-Pass Assemblers:

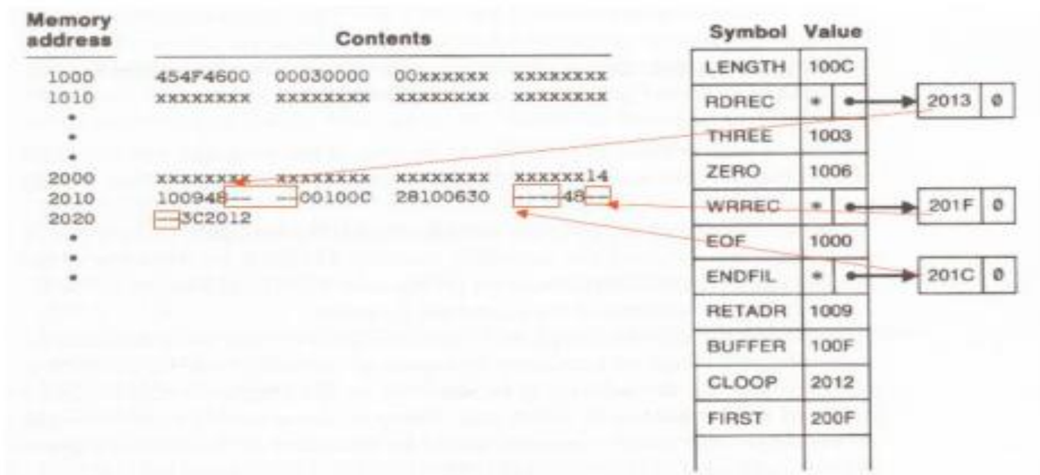
In load-and-Go assemblers when a forward reference is encountered:

- Omits the operand address if the symbol has not yet been defined
- Enters this undefined symbol into SYMTAB and indicates that it is undefined
- Adds the address of this operand address to a list of forward references associated with the SYMTAB entry
- When the definition for the symbol is encountered, scans the reference list and inserts the address.
- At the end of the program, reports the error if there are still SYMTAB entries indicated undefined symbols.
- For Load-and-Go assembler Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error

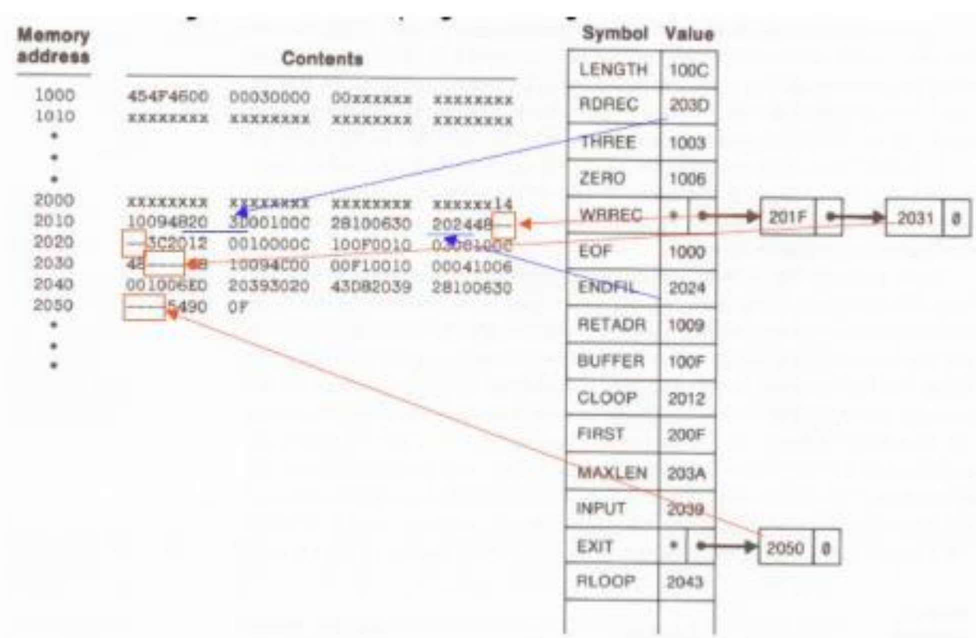
After Scanning line 40 of the program:

40 2021 J` CLOOP 302012

The status is that upto this point the symbol RREC is referred once at location 2013, ENDFIL at 201F and WRREC at location 201C. None of these symbols are defined. The figure shows that how the pending definitions along with their addresses are included in the symbol table.



The status after scanning line 160, which has encountered the definition of RDREC and ENDFIL is as given below:



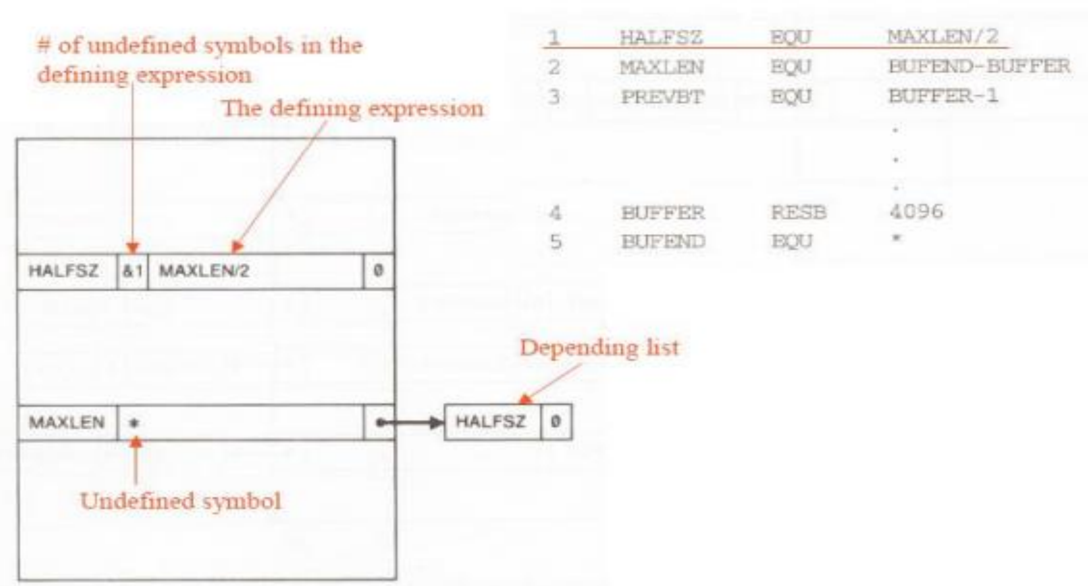
```
begin
  read first input line
  if OPCODE = 'START' then
    begin
      save #[OPERAND] as starting address
      initialize LOCCTR as starting address
      read next input line
    end (if START)
  else
    initialize LOCCTR to 0
while OPCODE ≠ 'END' do
    begin
      if there is not a comment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

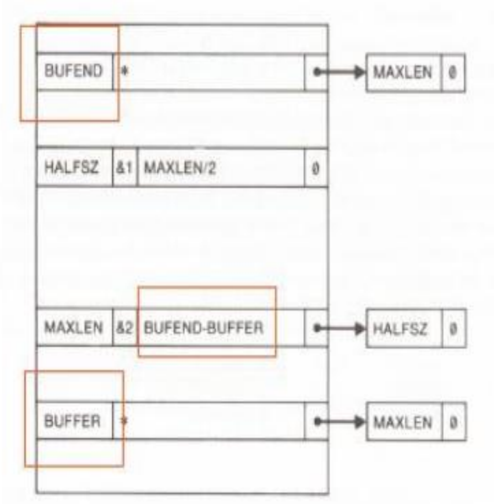
Figure 2.19(c) Algorithm for One pass assembler.

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

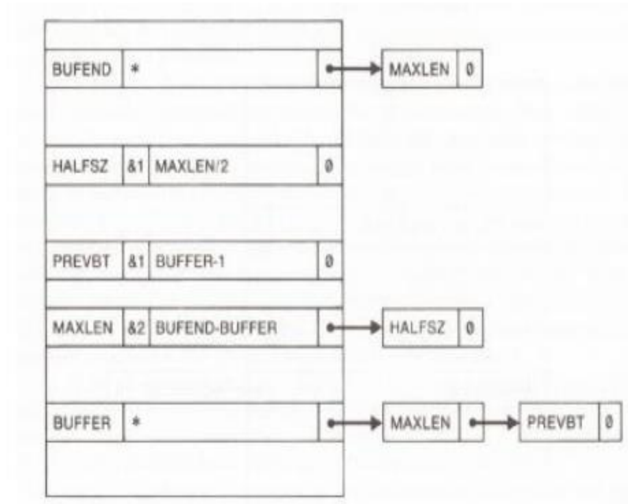
4. With suitable example, explain multi-pass assembler.

Multi-Pass Assembler Example Program

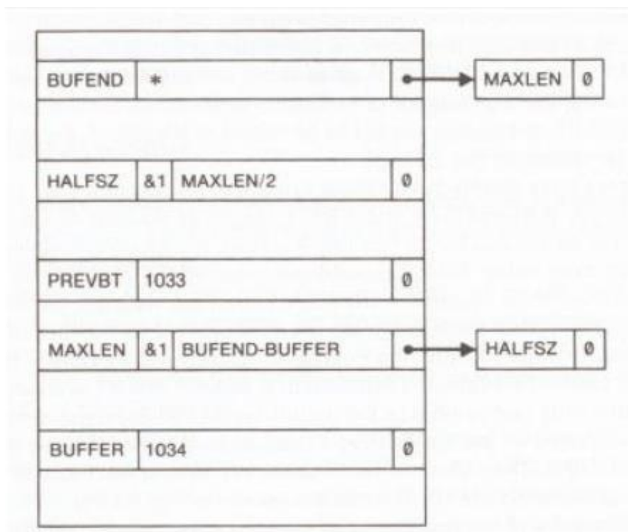




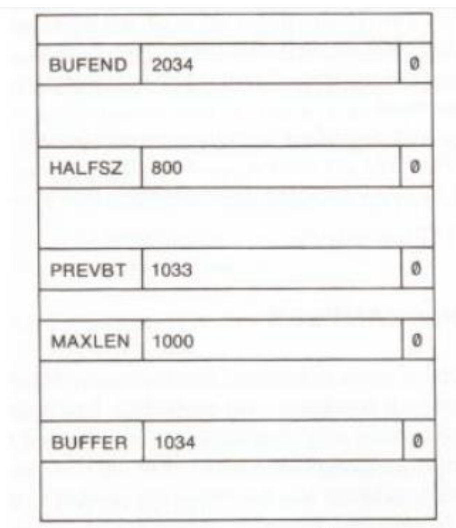
2 MAXLEN EQU BUFEND-BUFFER



3 PREVBT EQU BUFFER-1



4 BUFFER RESB 4096



5 BUFEND EQU *

5. **Mention the basic functions of a macro processor. Taking a suitable example, discuss the usage of various data structures in handling the macro definitions and macro expansions.**
    - A Macro represents a commonly used group of statements in the source programming language. A macro instruction (macro) is a notational convenience for the programmer. It allows the programmer to write shorthand version of a program (module programming)
    - The macro processor replaces each macro instruction with the corresponding group of source language statements (expanding)
    - Normally, it performs no analysis of the text it handles.
    - It does not concern the meaning of the involved statements during macro expansion.
    - The design of a macro processor generally is machine independent
    - Two new assembler directives are used in macro definition:
      MACRO: identify the beginning of a macro definition
      MEND: identify the end of a macro definition

- Prototype for the macro:

  Each parameter begins with '&'

  name MACRO parameters
  :
  :
  body
  :
  :
  MEND

  body: the statements that will be generated as the expansion of the macro

- Basic Macro Processor Functions:

  Macro Definition and Expansion:

  Figure shows the MACRO expansion. The left block shows the MACRO definition and the right block shows the expanded macro replacing the MACRO call with its block of executable instruction. M1 is a macro with two parameters D1 and D2. The MACRO stores the contents of register A in D1 and the contents of register B in D2. Later M1 is invoked with the parameters DATA1 and DATA2, Second time with DATA4 and DATA3. Every call of MACRO is expended with the executable statements.

```
Source
M1      MACRO    &D1, &D2
        STA      &D1
        STB      &D2
        MEND
   .
M1 DATA1, DATA2
   .
M1 DATA4, DATA3
```

```
Expanded source
   .
   .
   .
        STA    DATA1
        STB    DATA2
   .
        STA    DATA4
        STB    DATA3
   .
```

The statement M1 DATA1, DATA2 is a macro invocation statements that gives the name of the macro instruction being invoked and the arguments (M1 and M2) to be used in expanding. A macro invocation is referred as a Macro Call or Invocation.

Macro Expansion:

The program with macros is supplied to the macro processor. Each macro invocation statement will be expanded into the statement s that form the body of the macro, with the arguments from the macro invocation substituted for the parameters in the macro prototype. During the expansion, the macro definition statements are deleted since they are no longer needed. The arguments and the parameters are associated with one another according to their positions. The first argument in the macro matches with the first parameter in the macro prototype and so on. After macro processing the expanded file can become the input for the Assembler. The Macro Invocation statement is considered as comments and the statement generated from expansion is treated exactly as though they had been written directly by the

The data structures required are:

DEFTAB (Definition Table)

- Stores the macro definition including macro prototype and macro body
- Comment lines are omitted
- References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.

NAMTAB (Name Table)

- Stores macro names
- Serves as an index to DEFTAB - Pointers to the beginning and the end of the macro definition (DEFTAB).

ARGTAB (Argument Table)

- Stores the arguments according to their positions in the argument list.
- As the macro is expanded the arguments from the Argument table are substituted for the corresponding parameters in the macro body.

Example: Consider the following Example

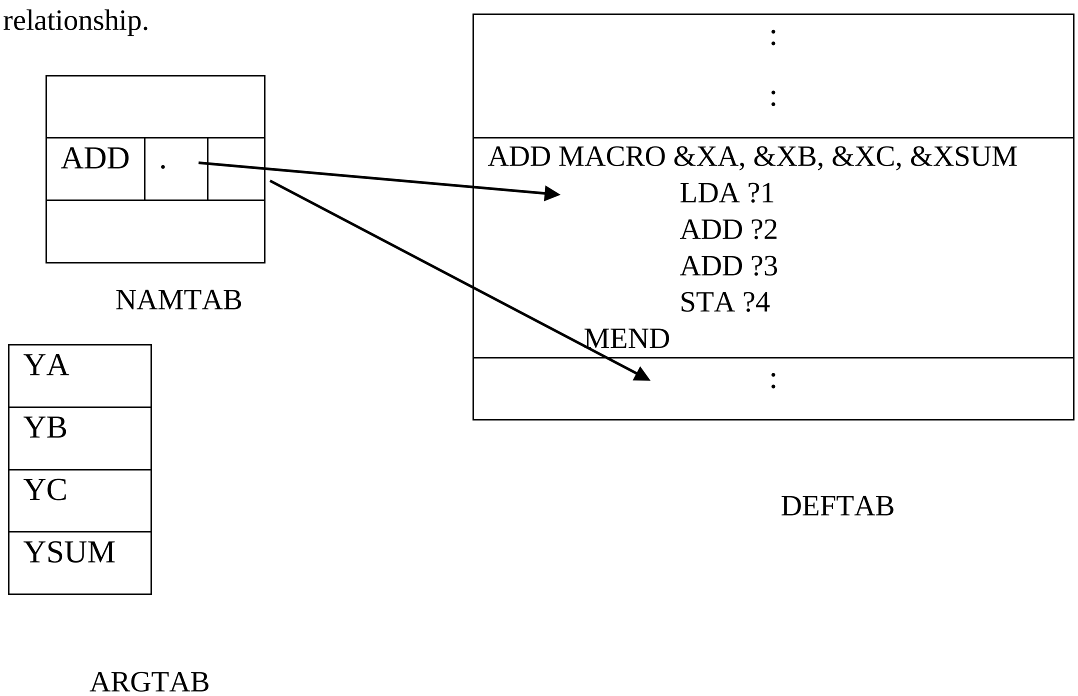
Suppose a Macro ADD has been defined in this way:

```
ADD MACRO &XA, &XB, &XC, &XSUM
        LDA &XA
        ADD &XB
        ADD &XC
        STA &XSUM
MEND
```

Now suppose, this macro has been called in the main procedure using the statement

```
        :
        :
ADD YA, YB, YC, YSUM
        :
        :
```

Then, the figure below shows the different data structures described and their relationship.



The above figure shows the portion of the contents of the table during the processing of the program. The definition of ADD is stored in DEFTAB, with an entry in NAMTAB having the pointers to the beginning and the end of the definition. The arguments referred by the instructions are denoted by the their positional notations. For example, LDA ?1 The above instruction loads the accumulator whose number is given by the parameter &XA. In the instruction this is replaced by its positional value ?1.

6. Write an algorithm for macro definition and macro expansion.

Algorithms

```
begin {macro processor}
        EXPANDINF := FALSE
        while OPCODE ≠ 'END' do
                begin
                        GETLINE
                        PROCESSLINE
                end {while}
end {macro processor}
```

```
Procedure PROCESSLINE
        begin
          search MAMTAB for OPCODE
          if found then
                EXPAND
          else if OPCODE = 'MACRO' then
                DEFINE
          else write source line to expanded file
        end {PRCOESSOR}
```

```
Procedure DEFINE
        begin
             enter macro name into NAMTAB
             enter macro prototype into DEFTAB
             LEVEL  :- 1
             while LEVEL > do
                 begin
                       GETLINE
                       if this is not a comment line then
                         begin
                            substitute positional notation for parameters
                            enter line into DEFTAB
                            if OPCODE = 'MACRO' then
                                LEVEL := LEVEL +1
                            else if OPCODE = 'MEND' then
                                LEVEL := LEVEL – 1
                         end {if not comment}
                 end {while}
              store in NAMTAB pointers to beginning and end of definition
        end {DEFINE}
```

```
Procedure EXPAND
    begin
            EXPANDING := TRUE
            get first line of macro definition {prototype} from DEFTAB
            set up arguments from macro invocation in ARGTAB
            while macro invocation to expanded file as a comment
            while not end of macro definition do
                begin
                        GETLINE
                        PROCESSLINE
                end {while}
            EXPANDING := FALSE
    end {EXPAND}

Procedure GETLINE
    begin
            if EXPANDING then
               begin
                  get next line of macro definition from DEFTAB
                  substitute arguments from ARGTAB for positional notation
               end {if}
            else
               read next line from input file
     end {GETLINE}
```

MODULE –II

7. **What is a loader? What are its advantages and disadvantages? Explain the bootstrap loader with algorithm or source program.**

Loader is an utility program which takes object code as input, prepares it for execution and loads the executable code into the memory.

Advantages

1. When source program is executed an object program gets generated. So there is no need to retranslate the program each time.
2. The source program can be written with multiple programs and multiple languages.
3. Some loaders are simple to implement.

Disadvantages

1. If the program is modified it has to be retranslated.
2. Some portion of the memory is occupied by the loader.
3. Loader has to load the object code as indicated by the operating system. This requires loader to perform relocation, linking and loading functions.

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
```

The Source Program:

```
BOOT    START   0       BOOTSTRAP LOADER FOR SIC/XE
.
. THIS BOOTSTRAP READS OBJECT CODE FROM DEVICE F1 AND ENTERS IT
. INTO MEMORY STARTING AT ADDRESS 80 (HEXADECIMAL). AFTER ALL OF
. THE CODE FROM DEVF1 HAS BEEN SEEN ENTERED INTO MEMORY, THE
. BOOTSTRAP EXECUTES A JUMP TO ADDRESS 80 TO BEGIN EXECUTION OF
. THE PROGRAM JUST LOADED.  REGISTER X CONTAINS THE NEXT ADDRESS
. TO BE LOADED.
.
        CLEAR   A       CLEAR REGISTER A TO ZERO
        LDX     #128    INITIALIZE REGISTER X TO HEX 80
LOOP    JSUB    GETC    READ HEX DIGIT FROM PROGRAM BEING LOADED
        RMO     A,S     SAVE IN REGISTER S
        SHIFTL  S,4     MOVE TO HIGH-ORDER 4 BITS OF BYTE
        JSUB    GETC    GET NEXT HEX DIGIT
        ADDR    S,A     COMBINE DIGITS TO FORM ONE BYTE
        STCH    0,X     STORE AT ADDRESS IN REGISTER X
        TIXR    X,X     ADD 1 TO MEMORY ADDRESS BEING LOADED
        J       LOOP    LOOP UNTIL END OF INPUT IS REACHED
```

```
.
. SUBROUTINE TO READ ONE CHARACTER FROM INPUT DEVICE AND
. CONVERT IT FROM ASCII CODE TO HEXADECIMAL DIGIT VALUE. THE
. CONVERTED DIGIT VALUE IS RETURNED IN REGISTER A. WHEN AN
. END-OF-FILE IS READ, CONTROL IS TRANSFERRED TO THE STARTING
. ADDRESS (HEX 80).
.
GETC     TD      INPUT    TEST INPUT DEVICE
         JEQ     GETC     LOOP UNTIL READY
         RD      INPUT    READ CHARACTER
         COMP    #4       IF CHARACTER IS HEX 04 (END OF FILE),
         JEQ     80           JUMP TO START OF PROGRAM JUST LOADED
         COMP    #48      COMPARE TO HEX 30 (CHARACTER '0')
         JLT     GETC     SKIP CHARACTERS LESS THAN '0'
         SUB     #48      SUBTRACT HEX 30 FROM ASCII CODE
         COMP    #10      IF RESULT IS LESS THAN 10, CONVERSION IS
         JLT     RETURN      COMPLETE. OTHERWISE, SUBTRACT 7 MORE
         SUB     #7          (FOR HEX DIGITS 'A' THROUGH 'F')
RETURN   RSUB             RETURN TO CALLER
INPUT    BYTE    X'F1'    CODE FOR INPUT DEVICE
         END     LOOP
```

8. Give and explain the algorithm of an 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 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 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. 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	←COPY
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

Figure 3.1 Loading of an absolute program.

9. **Explain in detail, SIC/XE relocation loader algorithm with suitable example.**

   Modification record:

   a. To described each part of the object code that must be changed when the program is relocated.
   b. The extended format instructions are affected by relocation. (absolute addressing)
   c. In this example, all modifications add the value of the symbol COPY, which represents the starting address.
   d. Not well suited for *standard version of SIC*, all the instructions except RSUB must be modified when the program is relocated. (absolute addressing)

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	

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

Figure 3.5 Object program with relocation by Modification records.

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

Figure 3.6 SIC/XE relocation loader algorithm

10. What is a relocating loader? Explain the relocation bit technique for specifying relocation as a part of object program.

There are some address dependent locations in the program , such address constants must be adjusted according to allocated space. The loaders which does this function are refereed as relocating loaders.

Relocation bit mechanism is used for SIC assembled programs.

Relocation bit:

- A *relocation bit* associated with each word of object code.
- The relocation bits are gathered together into a *bit mask* following the length indicator in each Text record.
- If bit=1, the corresponding word of object code is relocated.

```
H^COPY  ^000000^00107A
T^000000^1E^FFC^140033^481039^000036^280030^300015^481061^3C0003^00002A^0C0039^00002D
T^00001E^15^E00^0C0036^481061^080033^4C0000^454F46^000003^000000
T^001039^1E^FFC^040030^000030^E0105D^30103F^D8105D^280030^301057^548039^2C105E^38103F
T^001057^0A^800^100036^4C0000^F1^001000
T^001061^19^FE0^040030^E01079^301064^508039^DC1079^2C0036^381064^4C0000^05
E^000000
```

Figure 3.7 Object program with relocation by bit mask.

In Figure, T^000000^1E^FFC^ (111111111100) specifics that all 10 words of object code are to be modified.

Any value that is to be modified during relocation must coincide with one of these 3-byte segments so that it corresponds to a relocation bit.

The algorithm:

begin
get PROGADDR from operating system
while not end of input **do**
begin
read next record
while record type ≠ 'E' **do**
while record type = 'T'
begin
get length=second data
mask bits (M) as third data
for(i=0,i<length,i++)
if Mi = 1
add PROGADDR at the location
PROGADDR+specified address
else
move object code from record to location
PROGADDR+specified address
read next record

end

end

end

11. Explain the various data structures used for linking loader.

The algorithm for a linking loader is considerably more complicated than the absolute loader program. The modification records are used for relocation so that the linking and relocation functions are performed using the same mechanism. Linking Loader uses two-passes logic.

ESTAB (external symbol table) is the main data structure for a linking loader.

ESTAB is used to store the name and address of each external symbol in the set of control sections being loaded.

Two variables used are: PROGADDR and CSADDR.

PROGADDR is the beginning address in memory where the linked program is to be loaded.

CSADDR contains the starting address assigned to the control section currently being scanned by the loader.

Pass 1: Assign addresses to all external symbols
Pass 2: Perform the actual loading, relocation, and linking
ESTAB - ESTAB for the example (refer three programs PROGA PROGB and PROGC) given is as shown below. The ESTAB has four entries in it; they are name of the control section, the symbol appearing in the control section, its address and length of the control section.

Control section	Symbol	Address	Length
PROGA		4000	63
	LISTA	4040	
	ENDA	4054	
PROGB		4063	7F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	51
	LISTC	4112	
	ENDC	4124	

<u>(NOTE: refer answer of 12th question for the example)</u>

12. Discuss the detailed design of a linking and relocating loader with an example. Hence explain how program linking and relocation is performed by a linking loader when the subprograms use external reference.

The Goal of program linking is to resolve the problems with external references (EXTREF) and external definitions (EXTDEF) from different control sections.

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

ex: EXTDEF BUFFER, BUFFEND, LENGTH
EXTDEF LISTA, ENDA

EXTREF (external reference) - The EXTREF statement names symbols used in this (present) control section and are defined elsewhere.

ex: EXTREF RDREC, WRREC
EXTREF LISTB, ENDB, LISTC, ENDC

How to implement EXTDEF and EXTREF:

The assembler must include information in the object program that will cause the loader to insert proper values where they are required – in the form of Define record (D) and, Refer record(R).

Define record: The format of the Define record (D) along with examples is as shown here.

Col. 1 D
Col. 2-7 Name of external symbol defined in this control section
Col. 8-13 Relative address within this control section (hexadecimal)
Col.14-73 Repeat information in Col. 2-13 for other external symbols
Example records D LISTA 000040 ENDA 000054

D LISTB 000060 ENDB 000070

Refer record The format of the Refer record (R) along with examples is as shown here.

Col. 1 R
Col. 2-7 Name of external symbol referred to in this control section
Col. 8-73 Name of other external reference symbols
Example records R LISTB ENDB LISTC ENDC

R LISTA ENDA LISTC ENDC

R LISTA ENDA LISTB ENDB

Here are the three programs named as PROGA, PROGB and PROGC, which are separately assembled and each of which consists of a single control section. LISTA, ENDA in PROGA, LISTB, ENDB in PROGB and LISTC, ENDC in PROGC are external definitions in each of the control sections. Similarly LISTB, ENDB, LISTC, ENDC in PROGA, LISTA, ENDA, LISTC, ENDC in PROGB, and LISTA, ENDA, LISTB, ENDB in PROGC, are external references.

These sample programs given here are used to illustrate linking and relocation. The following figures give the sample programs and their corresponding object programs. Observe the object programs, which contain D and R records along with other records.

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

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

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

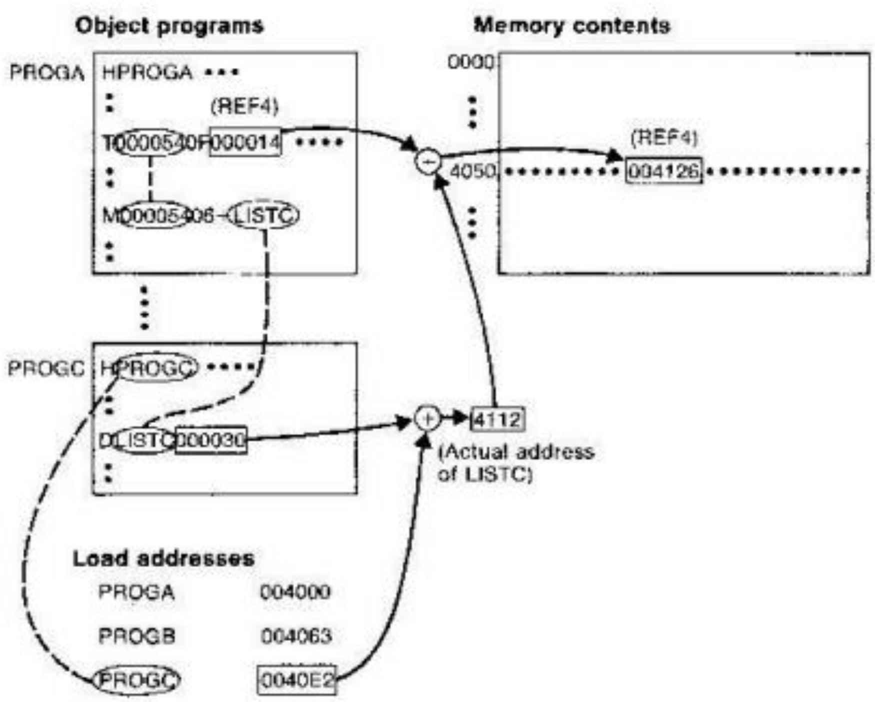
The object programs:

```
H^PROGA ^000000^000063
D^LISTA ^000040^ENDA  ^000054
R^LISTB ^ENDB  ^LISTC ^ENDC
:
T^000020^0A^03201D^77100004^050014
:
T^000054^0F^000014^FFFFF6^00003F^000014^FFFFC0
M^000024^05^+LISTB
M^000054^06^+LISTC
M^000057^06^+ENDC
M^000057^06^-LISTC
M^00005A^06^+ENDC
M^00005A^06^-LISTC
M^00005A^06^+PROGA
M^00005D^06^-ENDB
M^00005D^06^+LISTB
M^000060^06^+LISTB
M^000060^06^-PROGA
E^000020

H^PROGB ^000000^00007F
D^LISTB ^000060^ENDB  ^000070
R^LISTA ^ENDA  ^LISTC ^ENDC
:
T^000036^0B^03100000^772027^05100000
:
T^000070^0F^000000^FFFFF6^FFFFFF^FFFFF0^000060
M^000037^05^+LISTA
M^00003E^05^+ENDA
M^00003E^05^-LISTA
M^000070^06^+ENDA
M^000070^06^-LISTA
M^000070^06^+LISTC
M^000073^06^+ENDC
M^000073^06^-LISTC
M^000076^06^+ENDC
M^000076^06^-LISTC
M^000076^06^+LISTA
M^000079^06^+ENDA
M^000079^06^-LISTA
M^00007C^06^+PROGB
M^00007C^06^-LISTA
E

H^PROGC ^000000^000051
D^LISTC ^000030^ENDC  ^000042
R^LISTA ^ENDA  ^LISTB ^ENDB
:
T^000018^0C^03100000^77100004^05100000
:
T^000042^0F^000030^000008^000011^000000^000000
M^000019^05^+LISTA
M^00001D^05^+LISTB
M^000021^05^+ENDA
M^000021^05^-LISTA
M^000042^06^+ENDA
M^000042^06^-LISTA
M^000042^06^+PROGC
M^000048^06^+LISTA
M^00004B^06^+ENDA
M^00004B^06^-LISTA
M^00004B^06^-ENDB
M^00004B^06^+LISTB
M^00004E^06^+LISTB
M^00004E^06^-LISTA
E
```

The following figure shows these three programs as they might appear in memory after loading and linking. PROGA has been loaded starting at address 4000, with PROGB and PROGC immediately following.

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

For example, the value for REF4 in PROGA is located at address 4054 (the beginning address of PROGA plus 0054, the relative address of REF4 within PROGA). The following figure shows the details of how this value is computed.



The initial value from the Text record

T^ 000054^0F^000014^FFFFF6^00003F^000014^FFFFC0 is 000014.

To this is added the address assigned to LISTC, which is 4112 (the beginning address of PROGC plus 30). The result is 004126.

That is REF4 in PROGA is ENDA-LISTA+LISTC=4054-4040+4112=4126.

Similarly the load address for symbols

LISTA: PROGA+0040=4040,

LISTB: PROGB+0060=40C3

LISTC: PROGC+0030=4112

Keeping these details work through the details of other references and values of these references are the same in each of the three programs.

13. Write pass 1 and pass 2 algorithm of linking loader.

A linking loader usually makes two passes

a. ESTAB is used to store the name and address of each external symbol in the set of control sections being loaded.
b. Two variables: PROGADDR and CSADDR.
c. PROGADDR is the beginning address in memory where the linked program is to be loaded.
d. CSADDR contains the starting address assigned to the control section currently being scanned by the loader.

In Pass 1, concerned only Header and Defined records.

a. CSADDR+CSLTH = the next CSADDR.

b. A load map is generated.

In Pass 2, as each Text record is read, the object code is moved to the specified address (plus the current value of CSADDR).

When a Modification record is encountered, the symbol whose value is to be used for modification is looked up in ESTAB.

This value is then added to or subtracted from the indicated location in memory.

The complete algorithm is:

Pass 1:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR {for first control section}
while not end of input do
   begin
      read next input record {Header record for control section}
      set CSLTH to control section length
      search ESTAB for control section name
      if found then
         set error flag {duplicate external symbol}
      else
         enter control section name into ESTAB with value CSADDR
      while record type ≠ 'E' do
         begin
            read next input record
            if record type = 'D' then
               for each symbol in the record do
                  begin
                     search ESTAB for symbol name
                     if found then
                        set error flag (duplicate external symbol)
                     else
                        enter symbol into ESTAB with value
                           (CSADDR + indicated address)
                  end {for}
         end {while ≠ 'E'}
      add CSLTH to CSADDR {starting address for next control section}
   end {while not EOF}
end {Pass 1}
```

Figure 3.11(a) Algorithm for Pass 1 of a linking loader.

Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
   begin
       read next input record {Header record}
       set CSLTH to control section length
       while record type ≠ 'E' do
          begin
              read next input record
              if record type = 'T' then
                  begin
                      {if object code is in character form, convert
                          into internal representation}
                      move object code from record to location
                          (CSADDR + specified address)
                  end {if 'T'}
              else if record type = 'M' then
                  begin
                      search ESTAB for modifying symbol name
                      if found then
                          add or subtract symbol value at location
                              (CSADDR + specified address)
                      else
                          set error flag (undefined external symbol)
                  end {if 'M'}
          end {while ≠ 'E'}
       if an address is specified {in End record} then
          set EXECADDR to (CSADDR + specified address)
       add CSLTH to CSADDR
   end {while not EOF}
jump to location given by EXECADDR {to start execution of loaded program}
end {Pass 2}
```

Figure 3.11(b) Algorithm for Pass 2 of a linking loader.

14. What is dynamic binding? Explain the process of loading and calling of subroutine using dynamic linking loader.

Dynamic linking (dynamic loading, load on call)

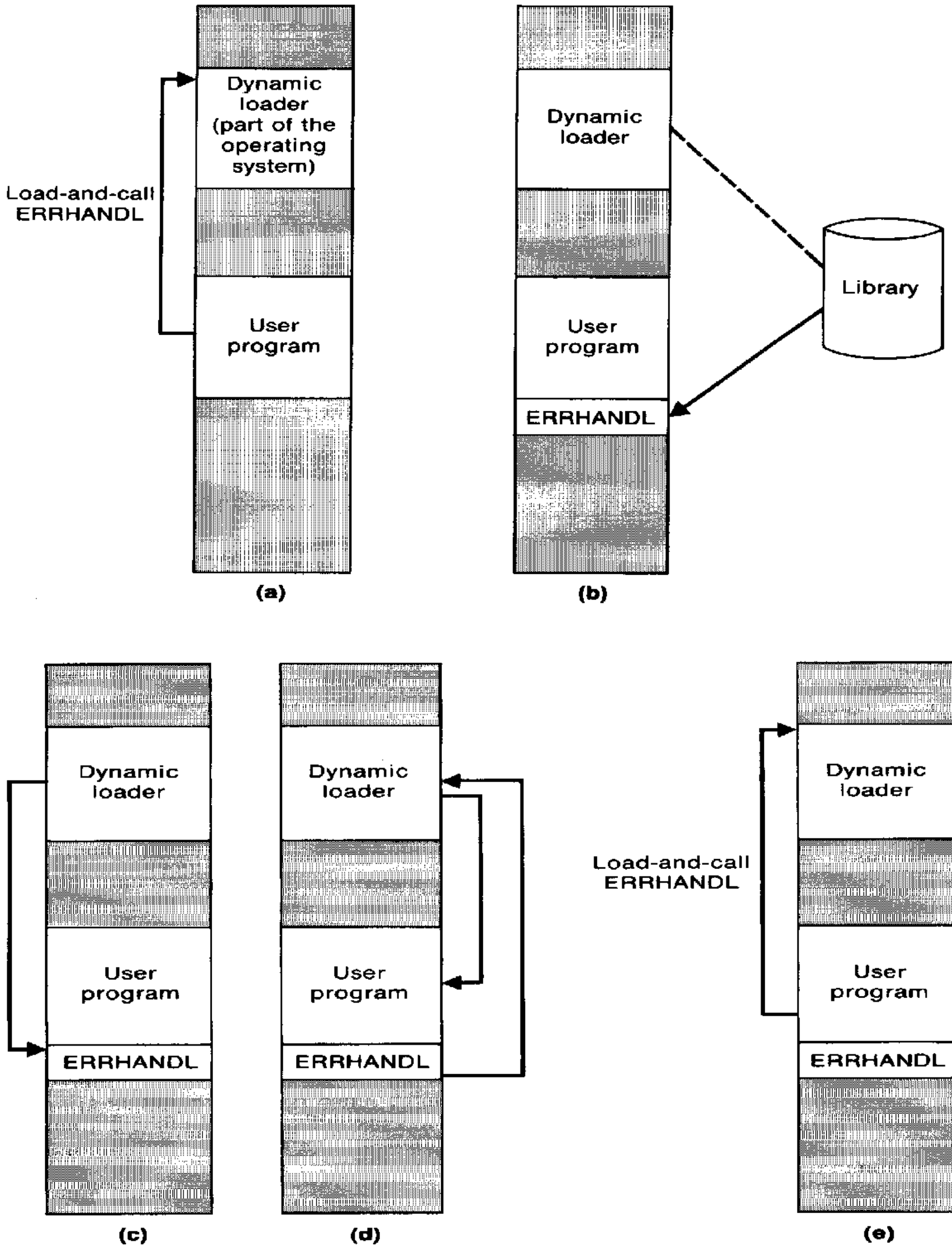
a. Postpones the linking function until execution time.
b. A subroutine is loaded and linked to the rest the program when is first loaded.
c. Dynamic linking is often used to allow several executing program to share one copy of a subroutine or library.

Dynamic linking provides the ability to load the routines only when (and if) they are needed.

a. For example, that a program contains subroutines that correct or clearly diagnose error in the input data during execution.
b. If such error are rare, the correction and diagnostic routines may not be used at all during most execution of the program.

c. However, if the program were completely linked before execution, these subroutines need to be loaded and linked every time.

Dynamic linking avoids the necessity of loading the entire library for each execution. The following Fig illustrates a method in which routines that are to be dynamically loaded must be called via an operating system (OS) service request.



- The program makes a load-on-call service request to OS. The parameter of this request is the symbolic name of the routine to be loaded.
- OS examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded form the specified user or system libraries.
- Control id then passed form OS to the routine being called.
- When the called subroutine completes its processing, OS then returns control to the program that issued the request.
- If a subroutine is still in memory, a second call to it may not require another load operation.

15. Differentiate the processing of an object program by linking loader and linkage editor with necessary diagrams.

Linking loader	Linkage editor
The linking loader performs all the linking and relocation operations including automatic library search then loads the program directly into the memory for execution.	The linkage editor produces a linked version of the program. Such a linked version is also called as load module or executable image. This load module is generally written in a file or library for later execution.
There is no need of relocating loader.	The relocating loader loads the load module into the memory.
The linking loader searches the libraries and resolves the external references every time the program is executed.	If the program is executed many times without being reassembled then linkage editor is the best choice.
When program is in development stage then at that time the linking loader can be used.	When program development is finished or when the library is built then linkage editor can be used.
The loading may require two passes.	The loading can be accomplished in one pass.

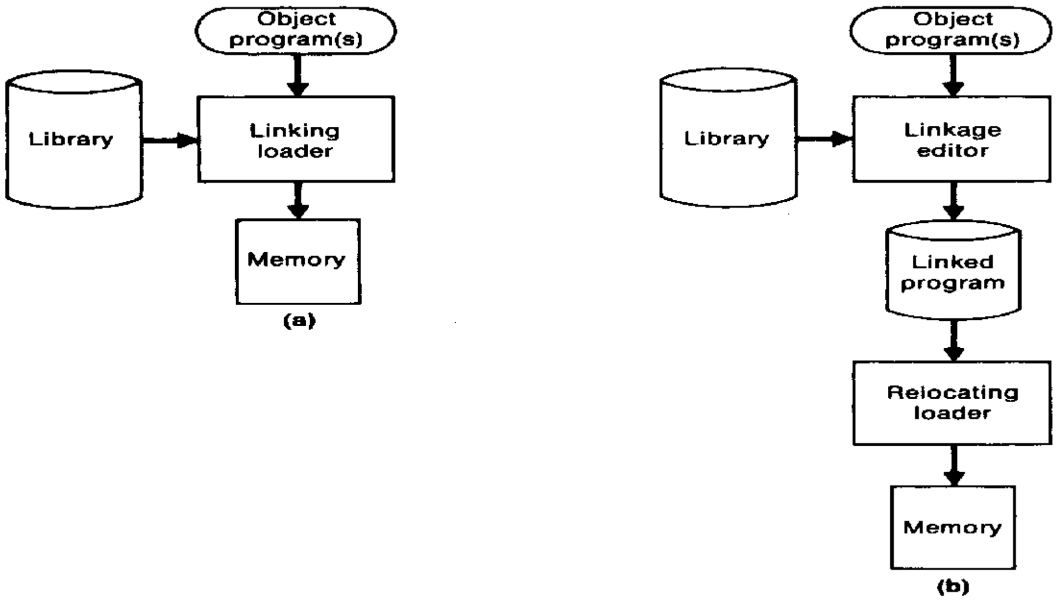


Figure 3.13 Processing of an object program using (a) linking loader and (b) linkage editor.

16. Enlist any 5 loader option commands.

INCLUDE program-name (library-name) - read the designated object program from a library

DELETE csect-name – delete the named control section from the set pf programs being loaded

CHANGE name1, name2 - external symbol name1 to be changed to name2 wherever it appears in the object programs

LIBRARY MYLIB – search MYLIB library before standard libraries

NOCALL STDDEV, PLOT, CORREL – no loading and linking of unneeded routines

Here is one more example giving, how commands can be specified as a part of object file, and the respective changes are carried out by the loader.
LIBRARY UTLIB
INCLUDE READ (UTLIB)
INCLUDE WRITE (UTLIB)
DELETE RDREC, WRREC
CHANGE RDREC, READ
CHANGE WRREC, WRITE
NOCALL SQRT, PLOT
The commands are, use UTLIB (say utility library), include READ and WRITE control sections from the library, delete the control sections RDREC and WRREC from the load, the change command causes all external references to the symbol RDREC to be changed to the symbol READ, similarly references to WRREC is changed to WRITE, finally, no call to the functions SQRT, PLOT, if they are used in the program.