Chapter 4: Macro Processor

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.

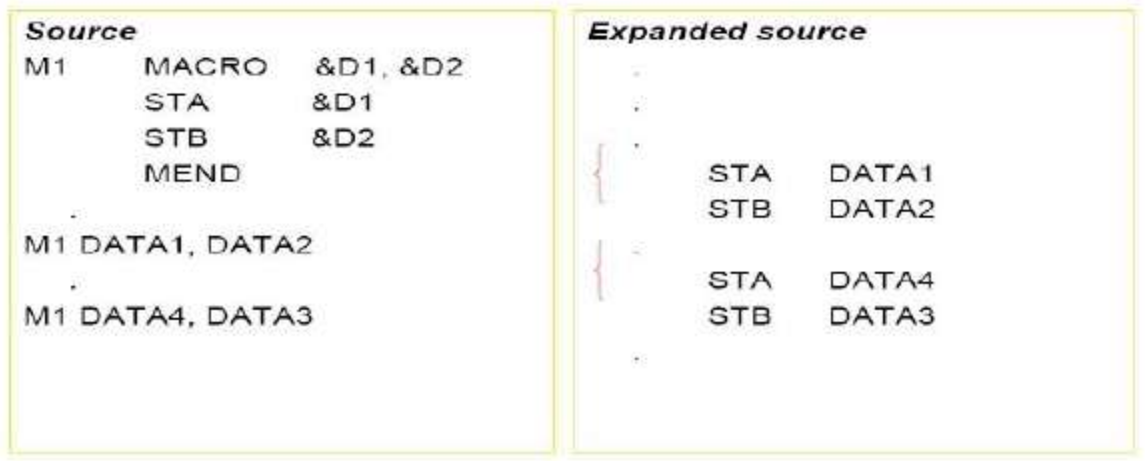
4.1 Basic Macro Processor Functions:

- *Macro Definition and Expansion*
- *Macro Processor Algorithms and Data structures*

4.1.1 Macro Definition and Expansion:

The 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
 .
```

Fig 4.1

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 programmer.

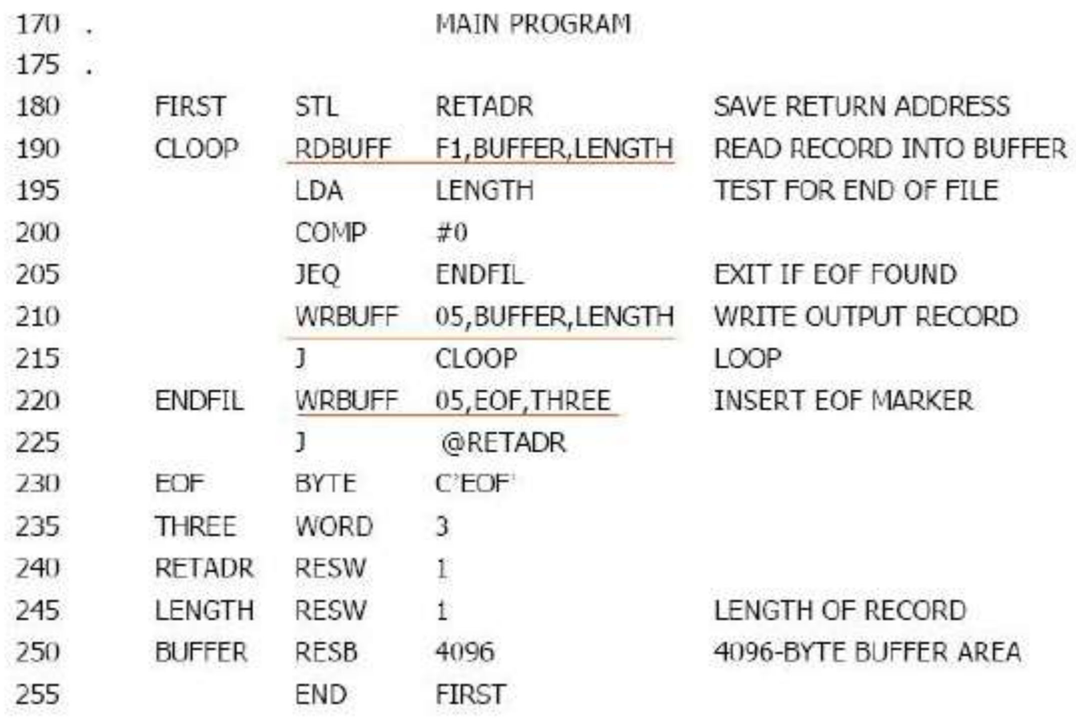
The difference between *Macros* and *Subroutines* is that the statement s from the body of the Macro is expanded the number of times the macro invocation is encountered, whereas the statement of the subroutine appears only once no matter how many times the subroutine is called. Macro instructions will be written so that the body of the macro contains no labels.

- Problem of the label in the body of macro:
  - If the same macro is expanded multiple times at different places in the program …
  - There will be *duplicate labels*, which will be treated as errors by the assembler.
- Solutions:

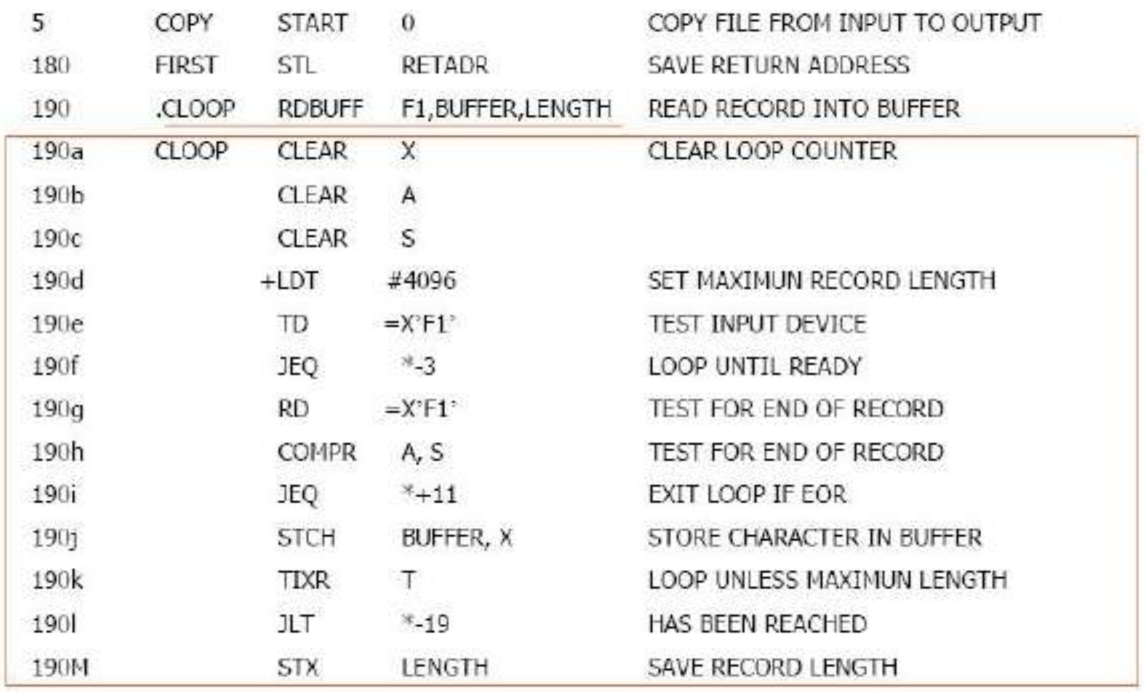
- Do not use labels in the body of macro.
- Explicitly use PC-relative addressing instead.

- Ex, in RDBUFF and WRBUFF macros,
  - JEQ *+11
  - JLT *-14
- It is inconvenient and error-prone.

The following program shows the concept of Macro Invocation and Macro Expansion.



```
.                       MAIN PROGRAM
.
FIRST     STL           RETADR               SAVE RETURN ADDRESS
CLOOP     RDBUFF        F1,BUFFER,LENGTH     READ RECORD INTO BUFFER
          LDA           LENGTH               TEST FOR END OF FILE
          COMP          #0
          JEQ           ENDFIL               EXIT IF EOF FOUND
          WRBUFF        05,BUFFER,LENGTH     WRITE OUTPUT RECORD
          J             CLOOP                LOOP
ENDFIL    WRBUFF        05,EOF,THREE         INSERT EOF MARKER
          J             @RETADR
EOF       BYTE          C'EOF'
THREE     WORD          3
RETADR    RESW          1
LENGTH    RESW          1                    LENGTH OF RECORD
BUFFER    RESB          4096                 4096-BYTE BUFFER AREA
          END           FIRST
```



```
5       COPY    START   0                   COPY FILE FROM INPUT TO OUTPUT
180     FIRST   STL     RETADR              SAVE RETURN ADDRESS
190     .CLOOP  RDBUFF  F1,BUFFER,LENGTH    READ RECORD INTO BUFFER
190a    CLOOP   CLEAR   X                   CLEAR LOOP COUNTER
190b            CLEAR   A
190c            CLEAR   S
190d           +LDT     #4096               SET MAXIMUN RECORD LENGTH
190e            TD      =X'F1'              TEST INPUT DEVICE
190f            JEQ     *-3                 LOOP UNTIL READY
190g            RD      =X'F1'              TEST FOR END OF RECORD
190h            COMPR   A, S                TEST FOR END OF RECORD
190i            JEQ     *+11                EXIT LOOP IF EOR
190j            STCH    BUFFER, X           STORE CHARACTER IN BUFFER
190k            TIXR    T                   LOOP UNLESS MAXIMUN LENGTH
190l            JLT     *-19                HAS BEEN REACHED
190M            STX     LENGTH              SAVE RECORD LENGTH
```

Fig 4.2

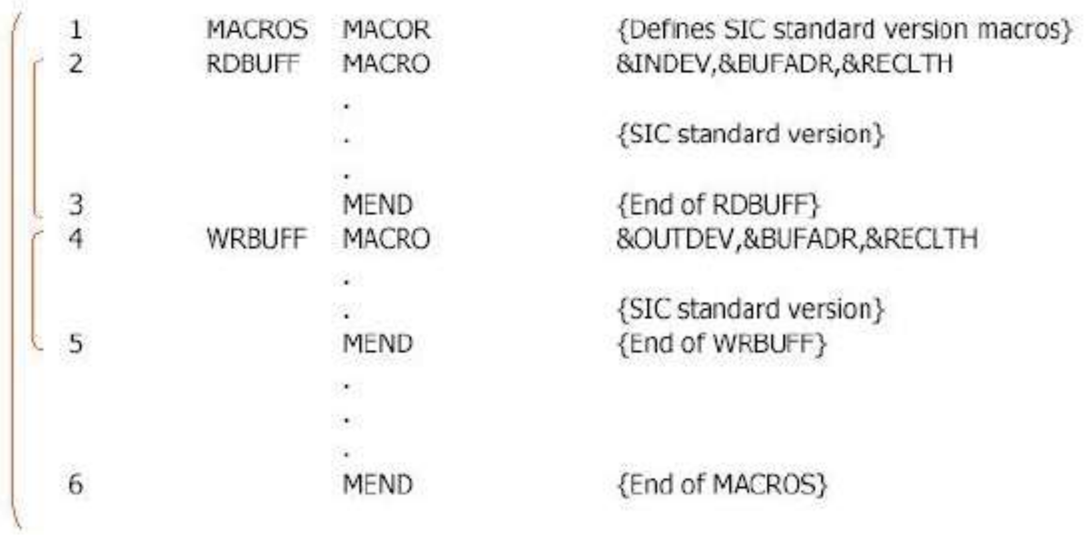
4.1.2 Macro Processor Algorithm and Data Structure:

Design can be done as two-pass or a one-pass macro. In case of two-pass assembler.

Two-pass macro processor

- You may design a two-pass macro processor
    - Pass 1:
        - Process all macro definitions
    - Pass 2:
        - Expand all macro invocation statements
- However, one-pass may be enough
    - Because all macros would have to be defined during the first pass before any macro invocations were expanded.
        - The definition of a macro must appear before any statements that invoke that macro.
- Moreover, the body of one macro can contain definitions of the other macro
- Consider the example of a Macro defining another Macro.
- In the example below, the body of the first Macro (MACROS) contains statement that define RDBUFF, WRBUFF and other macro instructions for SIC machine.
- The body of the second Macro (MACROX) defines the se same macros for SIC/XE machine.
- A proper invocation would make the same program to perform macro invocation to run on either SIC or SIC/XEmachine.

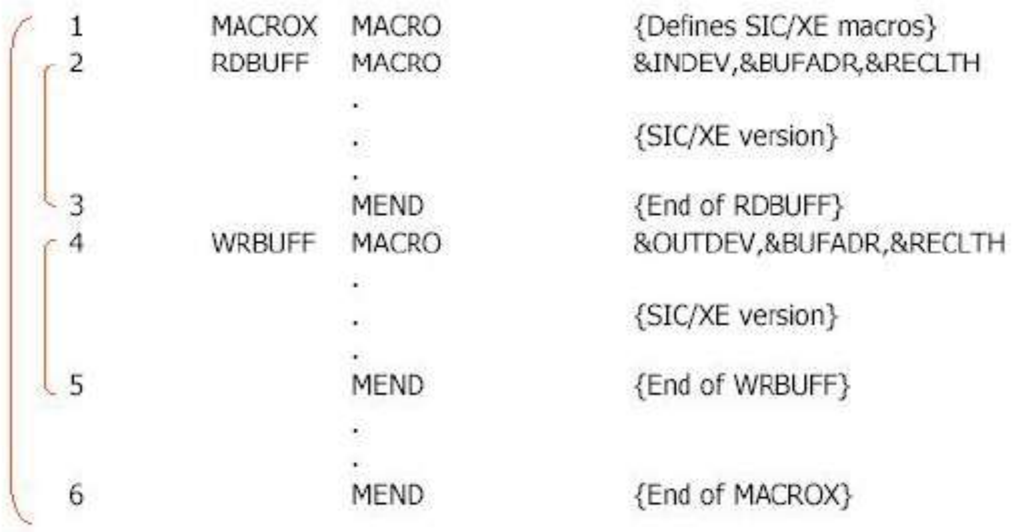
MACROS for SIC machine



```
1   MACROS   MACOR    {Defines SIC standard version macros}
2   RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH
                 .
                 .    {SIC standard version}
                 .
3            MEND     {End of RDBUFF}
4   WRBUFF   MACRO    &OUTDEV,&BUFADR,&RECLTH
                 .
                 .    {SIC standard version}
5            MEND     {End of WRBUFF}
                 .
                 .
                 .
6            MEND     {End of MACROS}
```

Fig 4.3(a)

MACROX for SIC/XE Machine



```
1   MACROX   MACRO    {Defines SIC/XE macros}
2   RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH
                 .
                 .    {SIC/XE version}
                 .
3            MEND     {End of RDBUFF}
4   WRBUFF   MACRO    &OUTDEV,&BUFADR,&RECLTH
                 .
                 .    {SIC/XE version}
                 .
5            MEND     {End of WRBUFF}
                 .
                 .
6            MEND     {End of MACROX}
```

Fig 4.3(b)

- A program that is to be run on SIC system could invoke MACROS whereas a program to be run on SIC/XE can invoke MACROX.
- However, defining MACROS or MACROX does not define RDBUFF and WRBUFF.
- These definitions are processed only when an invocation of MACROS or MACROX is expanded.

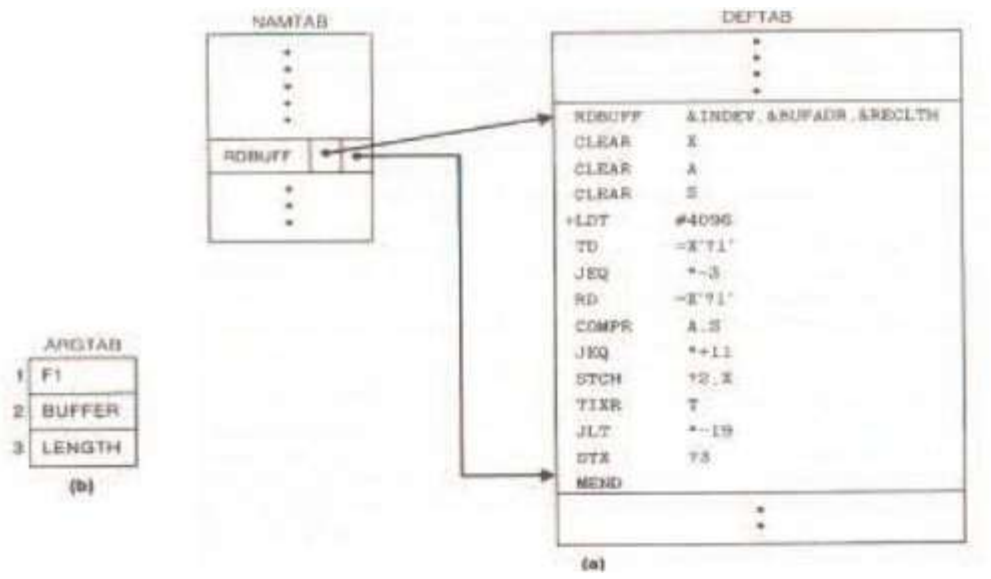
One-Pass Macro Processor:

- A one-pass macro processor that alternate between *macro definition* and *macro expansion* in a recursive way is able to handle recursive macro definition.
- Restriction
    - The definition of a macro must appear in the source program before any statements that invoke that macro.
    - This restriction does not create any real inconvenience.

The design considered is for one-pass assembler. 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.
    - The figure below shows the different data structures described and their relationship.



```
RDBUFF    &INDEV,&BUFADR,&RECLTH
CLEAR     X
CLEAR     A
CLEAR     S
+LDT     #4096
TD       =X'?1'
JEQ       *-3
RD       =X'?1'
COMPR     A,S
JEQ       *+11
STCH      ?2,X
TIXR      T
JLT       *-19
STX       ?3
MEND
```

Fig 4.4

The above figure shows the portion of the contents of the table during the processing of the program in page no. 3. In fig 4.4(a) definition of RDBUFF 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,

```
        TD      =X'?1'
```

The above instruction is to test the availability of the device whose number is given by the parameter &INDEV. In the instruction this is replaced by its positional value? 1.
Figure 4.4(b) shows the ARTAB as it would appear during expansion of the RDBUFF statement as given below:

```
        CLOOP        RDBUFF      F1, BUFFER, LENGTH
```

For the invocation of the macro RDBUFF, the first parameter is F1 (input device code), second is BUFFER (indicating the address where the characters read are stored), and the third is LENGTH (which indicates total length of the record to be read). When the ?n notation is encountered in a line fro DEFTAB, a simple indexing operation supplies the proper argument from ARGTAB.

The algorithm of the Macro processor is given below. This has the procedure DEFINE to make the entry of *macro name* in the NAMTAB, *Macro Prototype* in DEFTAB. EXPAND is called to set up the argument values in ARGTAB and expand a *Macro Invocation* statement. Procedure GETLINE is called to get the next line to be processed either from the DEFTAB or from the file itself.

When a macro definition is encountered it is entered in the DEFTAB. The normal approach is to continue entering till MEND is encountered. If there is a program having a Macro defined within another Macro. While defining in the DEFTAB the very first MEND is taken as the end of the Macro definition. This does not complete the definition as there is another outer Macro which completes the difintion of Macro as a whole. Therefore the DEFINE procedure keeps a counter variable LEVEL. Every time a Macro directive is encountered this counter is incremented by 1. The moment the innermost Macro ends indicated by the directive MEND it starts decreasing the value of the counter variable by one. The last MEND should make the counter value set to zero. So when LEVEL becomes zero, the MEND corresponds to the original MACRO directive.

Most macro processors allow thr definitions of the commonly used instructions to appear in a standard system library, rather than in the source program. This makes the use of macros convenient; definitions are retrieved from the library as they are needed during macro processing.

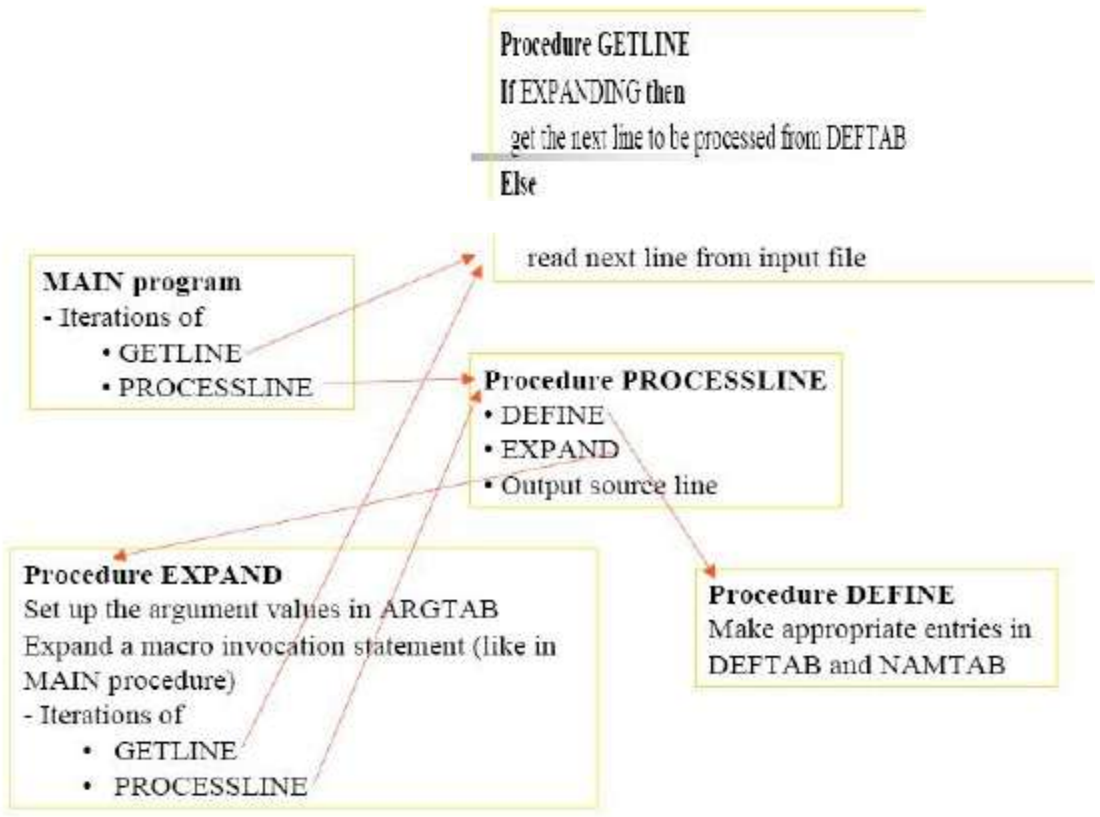
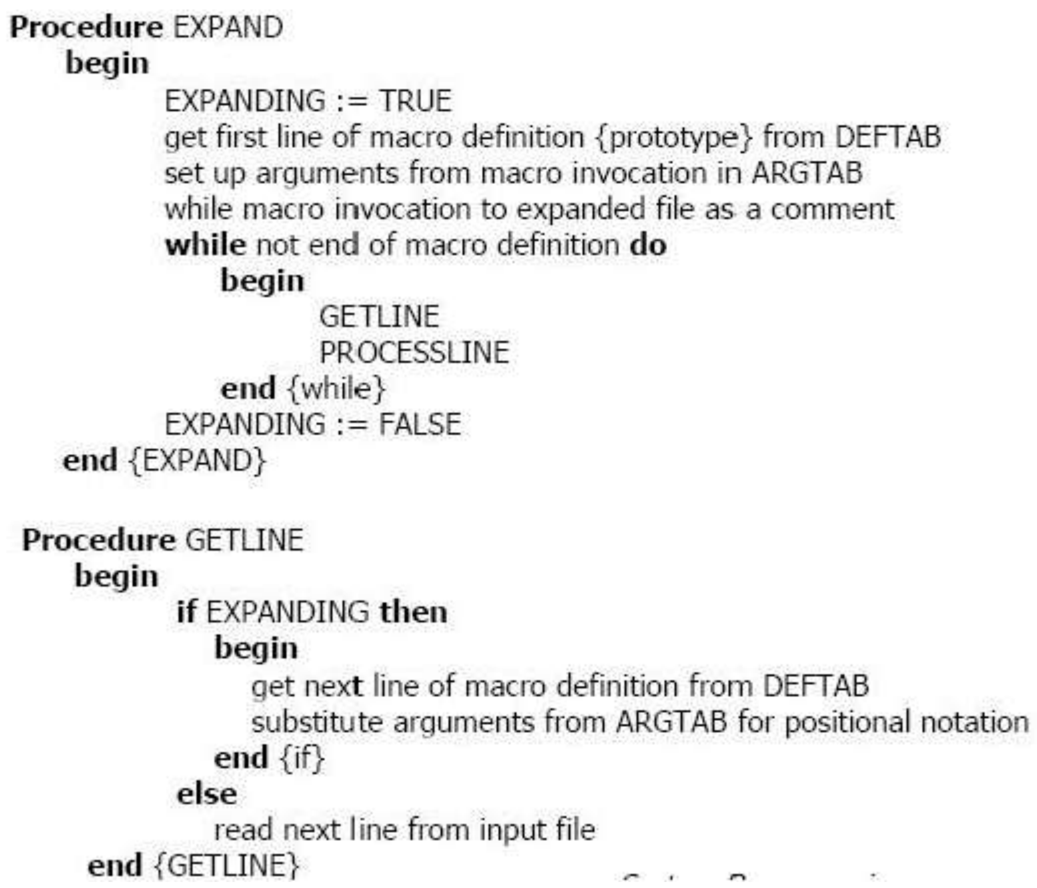

Procedure GETLINE
If EXPANDING then
get the next line to be processed from DEFTAB
Else
read next line from input file
MAIN program
- Iterations of
• GETLINE
• PROCESSLINE
Procedure PROCESSLINE
• DEFINE
• EXPAND
• Output source line
Procedure EXPAND
Set up the argument values in ARGTAB
Expand a macro invocation statement (like in MAIN procedure)
- Iterations of
• GETLINE
• PROCESSLINE
Procedure DEFINE
Make appropriate entries in DEFTAB and NAMTAB

Fig 4.5

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}
```

Fig 4.6

4.1.3 Comparison of Macro Processor Design

- *One-pass algorithm*
    - Every macro must be defined before it is called
    - One-pass processor can alternate between macro definition and macro expansion
    - Nested macro definitions are allowed but nested calls are not allowed.
- *Two-pass algorithm*
    - Pass1: Recognize macro definitions
    - Pass2: Recognize macro calls
    - Nested macro definitions are not allowed

4.1 Machine-independent Macro-Processor Features.

The design of macro processor doesn't depend on the architecture of the machine. We will be studying some extended feature for this macro processor. These features are:

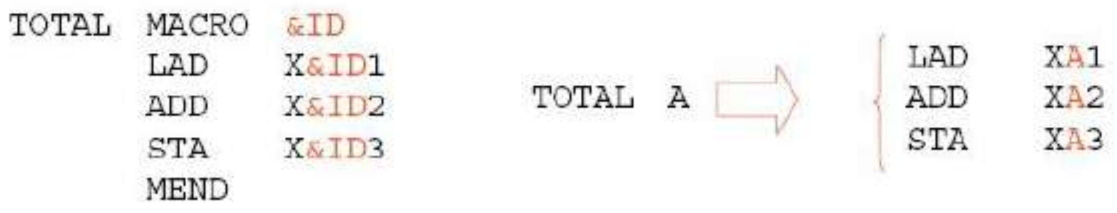
- Concatenation of Macro Parameters
- Generation of unique labels
- Conditional Macro Expansion
- Keyword Macro Parameters

4.2.1 Concatenation of unique labels:

Most macro processor allows parameters to be concatenated with other character strings. Suppose that a program contains a series of variables named by the symbols XA1, XA2, XA3,…, another series of variables named XB1, XB2, XB3,…, etc. If similar processing is to be performed on each series of labels, the programmer might put this as a macro instruction. The parameter to such a macro instruction could specify the series of variables to be operated on (A, B, etc.). The macro processor would use this parameter to construct the symbols required in the macro expansion (XA1, Xb1, etc.).

Suppose that the parameter to such a macro instruction is named &ID. The body of the macro definition might contain a statement like

```
        LDA         X&ID1
```



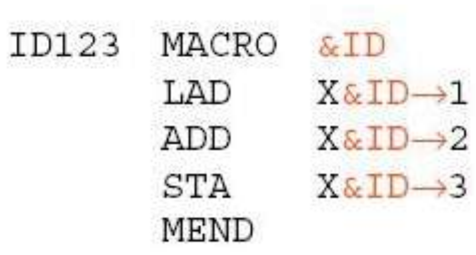
```
TOTAL  MACRO  &ID
       LAD    X&ID1                            LAD    XA1
       ADD    X&ID2        TOTAL  A   =>       ADD    XA2
       STA    X&ID3                            STA    XA3
       MEND
```

Fig 4.7

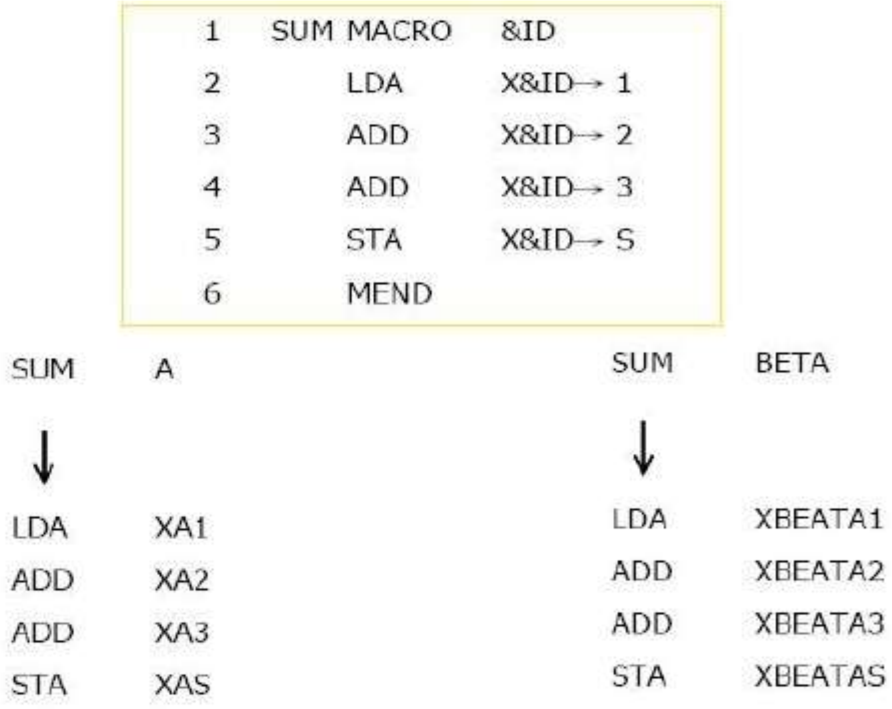
& is the starting character of the macro instruction; but the end of the parameter is not marked. So in the case of &ID1, the macro processor could deduce the meaning that was intended. If the macro definition contains contain &ID and &ID1 as parameters, the situation would be unavoidably ambiguous.

Most of the macro processors deal with this problem by providing a special concatenation operator. In the SIC macro language, this operator is the character →. Thus the statement LDA X&ID1 can be written as

```
        LDA         X&ID→
```



```
ID123  MACRO  &ID
       LAD    X&ID→1
       ADD    X&ID→2
       STA    X&ID→3
       MEND
```



```
1   SUM MACRO  &ID
2       LDA    X&ID→ 1
3       ADD    X&ID→ 2
4       ADD    X&ID→ 3
5       STA    X&ID→ S
6       MEND
```

```
SUM    A                    SUM    BETA
 ↓                           ↓
LDA    XA1                  LDA    XBEATA1
ADD    XA2                  ADD    XBEATA2
ADD    XA3                  ADD    XBEATA3
STA    XAS                  STA    XBEATAS
```

Fig 4.8

The above figure shows a macro definition that uses the concatenation operator as previously described. The statement SUM A and SUM BETA shows the invocation statements and the corresponding macro expansion.

4.2.2 Generation of Unique Labels

As discussed it is not possible to use labels for the instructions in the macro definition, since every expansion of macro would include the label repeatedly which is not allowed by the assembler. This in turn forces us to use relative addressing in the jump instructions. Instead we can use the technique of generating unique labels for every macro invocation and expansion. During macro expansion each $ will be replaced with $XX, where xx is a two-character alphanumeric counter of the number of macro instructions expansion.

For example,

XX = AA, AB, AC…

This allows 1296 macro expansions in a single program.

The following program shows the macro definition with labels to the instruction.

```
RDBUFF   MACRO    &INDEV, &BUFADR, &RECLTH
         CLEAR    X                  CLEAR LOOP COUNTER
         CLEAR    A
         CLEAR    S
        +LDT      #4096              SET MAXIMUM RECORD LENGTH
$LOOP    TD       =X'&INDEV'         TEST INPUT DEVICE
         JEQ      $LOOP              LOOP UNTIL READY
         RD       =X'&INDEV'         READ CHARACTER INTI REG A
         COMPR    A, S               TEST FOR END OF RECORD
         JEQ      $EXIT              EXIT LOOP IF EOR
         STCH     &BUFADR, X         STORE CHARACTER IN BUFFER
         TIXR     $LOOP              HAS BEEN REACHED
$EXIT    STX      &RECLTH            SAVE RECORD LENGTH
         MEND
```

The following figure shows the macro invocation and expansion first time.

```
.        RDBUFF   F1, BUFFER, LENGTH


         CLEAR    X                  CLEAR LOOP COUNTER
         CLEAR    A
         CLEAR    S
        +LDT      #4096              SET MAXIMUM RECORD LENGTH
$AALOOP  TD       =X'F1'             TEST INPUT DEVICE
         JEQ      $AALOOP            LOOP UNTIL READY
         RD       =X'F1'             READ CHARACTER INTI REG A
         COMPR    A, S               TEST FOR END OF RECORD
         JEQ      $AAEXIT            EXIT LOOP IF EOR
         STCH     BUFFER, X          STORE CHARACTER IN BUFFER
         TIXR     T                  LOOP UNLESS MAXIMUM LENGTH
         JLT      $AALOOP            HAS BEEN REACHED
$AAEXIT  STX      LENGTH             SAVE RECORD LENGTH
```

If the macro is invoked second time the labels may be expanded as $ABLOOP $ABEXIT.

4.2.3 Conditional Macro Expansion

There are applications of macro processors that are not related to assemblers or assembler programming.

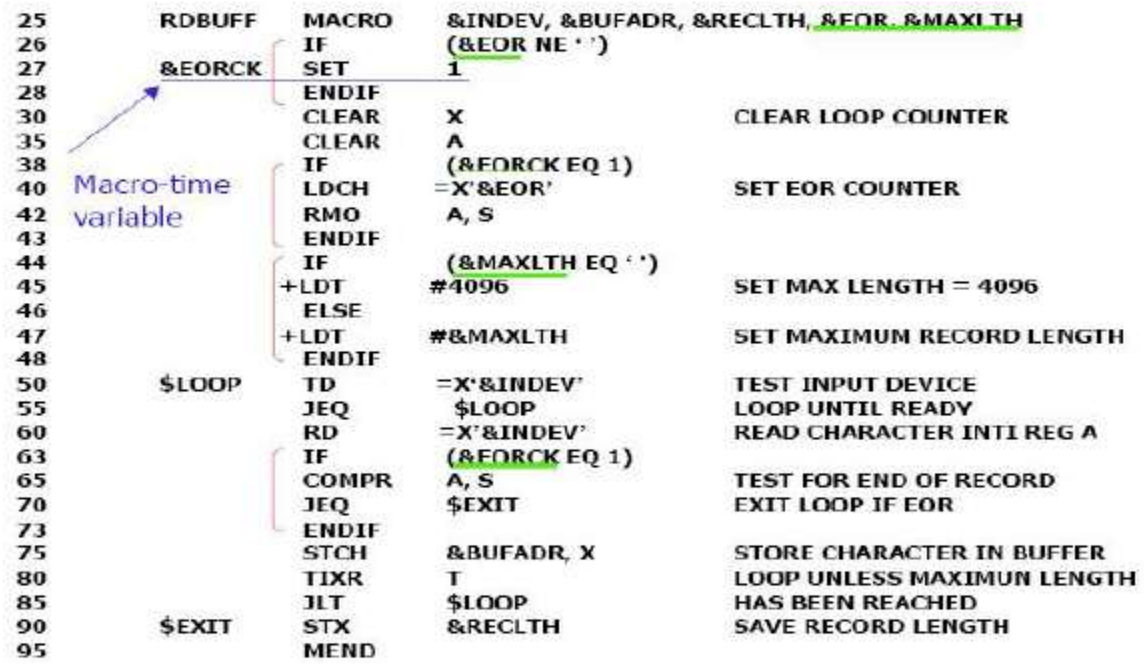
Conditional assembly depends on parameters provides

```
MACRO &COND
……..
  IF (&COND NE ‘’)
        part I
  ELSE
        part II
  ENDIF
………
ENDM
```

Part I is expanded if condition part is true, otherwise part II is expanded. Compare operators: NE, EQ, LE, GT.

Macro-Time Variables:

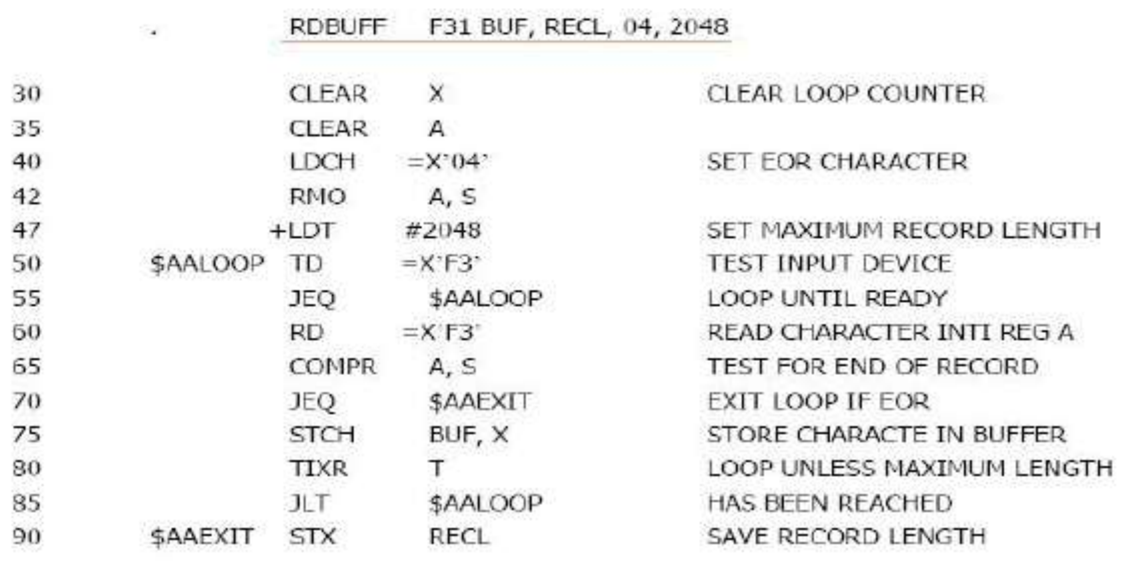
Macro-time variables (often called as SET Symbol) can be used to store working values during the macro expansion. Any symbol that begins with symbol & and not a macro instruction parameter is considered as *macro-time variable.* All such variables are initialized to zero.



```
RDBUFF   MACRO   &INDEV, &BUFADR, &RECLTH, &EOR, &MAXLTH
         IF      (&EOR NE ‘ ’)
&EORCK   SET     1
         ENDIF
         CLEAR   X                     CLEAR LOOP COUNTER
         CLEAR   A
         IF      (&EORCK EQ 1)
         LDCH    =X’&EOR’              SET EOR COUNTER
         RMO     A, S
         ENDIF
         IF      (&MAXLTH EQ ‘ ’)
        +LDT     #4096                 SET MAX LENGTH = 4096
         ELSE
        +LDT     #&MAXLTH              SET MAXIMUM RECORD LENGTH
         ENDIF
$LOOP    TD      =X‘&INDEV’            TEST INPUT DEVICE
         JEQ      $LOOP                LOOP UNTIL READY
         RD      =X’&INDEV’            READ CHARACTER INTI REG A
         IF      (&EORCK EQ 1)
         COMPR   A, S                  TEST FOR END OF RECORD
         JEQ     $EXIT                 EXIT LOOP IF EOR
         ENDIF
         STCH    &BUFADR, X            STORE CHARACTER IN BUFFER
         TIXR    T                     LOOP UNLESS MAXIMUN LENGTH
         JLT     $LOOP                 HAS BEEN REACHED
$EXIT    STX     &RECLTH               SAVE RECORD LENGTH
         MEND
```

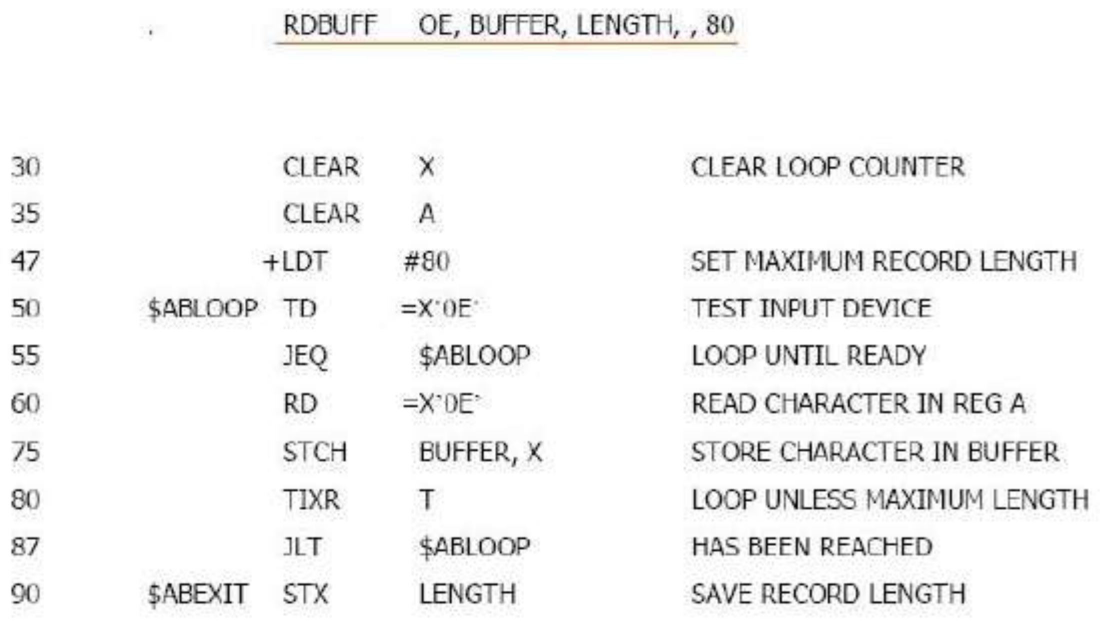
Fig 4.9(a)

Figure 4.5(a) gives the definition of the macro RDBUFF with the parameters &INDEV, &BUFADR, &RECLTH, &EOR, &MAXLTH. According to the above program if &EOR has any value, then &EORCK is set to 1 by using the directive SET, otherwise it retains its default value 0.



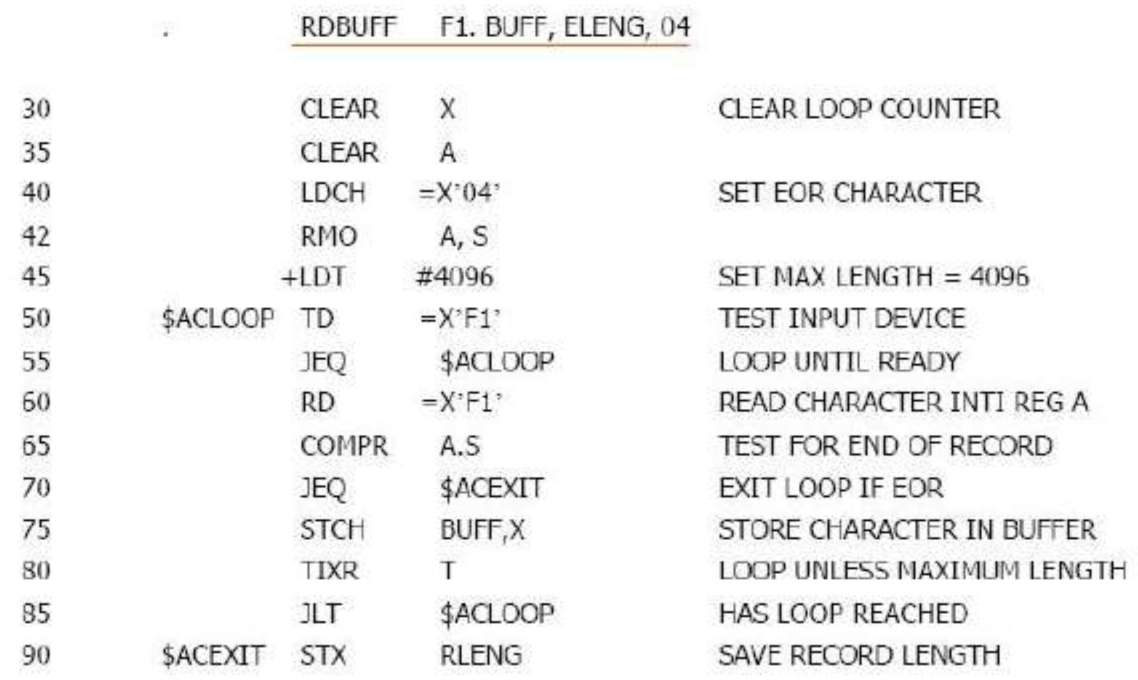
```
 .          RDBUFF    F31 BUF, RECL, 04, 2048

30          CLEAR     X           CLEAR LOOP COUNTER
35          CLEAR     A
40          LDCH      =X'04'      SET EOR CHARACTER
42          RMO       A, S
47         +LDT       #2048       SET MAXIMUM RECORD LENGTH
50  $AALOOP TD        =X'F3'      TEST INPUT DEVICE
55          JEQ       $AALOOP     LOOP UNTIL READY
60          RD        =X'F3'      READ CHARACTER INTI REG A
65          COMPR     A, S        TEST FOR END OF RECORD
70          JEQ       $AAEXIT     EXIT LOOP IF EOR
75          STCH      BUF, X      STORE CHARACTE IN BUFFER
80          TIXR      T           LOOP UNLESS MAXIMUM LENGTH
85          JLT       $AALOOP     HAS BEEN REACHED
90  $AAEXIT STX       RECL        SAVE RECORD LENGTH
```

Fig 4.9(b) Use of Macro-Time Variable with EOF being NOT NULL



```
 .          RDBUFF    OE, BUFFER, LENGTH, , 80


30          CLEAR     X           CLEAR LOOP COUNTER
35          CLEAR     A
47         +LDT       #80         SET MAXIMUM RECORD LENGTH
50  $ABLOOP TD        =X'0E'      TEST INPUT DEVICE
55          JEQ       $ABLOOP     LOOP UNTIL READY
60          RD        =X'0E'      READ CHARACTER IN REG A
75          STCH      BUFFER, X   STORE CHARACTER IN BUFFER
80          TIXR      T           LOOP UNLESS MAXIMUM LENGTH
87          JLT       $ABLOOP     HAS BEEN REACHED
90  $ABEXIT STX       LENGTH      SAVE RECORD LENGTH
```

Fig 4.9(c) Use of Macro-Time conditional statement with EOF being NULL



```
.               RDBUFF   F1. BUFF, ELENG, 04

30              CLEAR    X              CLEAR LOOP COUNTER
35              CLEAR    A
40              LDCH     =X'04'         SET EOR CHARACTER
42              RMO      A, S
45             +LDT      #4096          SET MAX LENGTH = 4096
50    $ACLOOP   TD       =X'F1'         TEST INPUT DEVICE
55              JEQ      $ACLOOP        LOOP UNTIL READY
60              RD       =X'F1'         READ CHARACTER INTI REG A
65              COMPR    A.S            TEST FOR END OF RECORD
70              JEQ      $ACEXIT        EXIT LOOP IF EOR
75              STCH     BUFF,X         STORE CHARACTER IN BUFFER
80              TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85              JLT      $ACLOOP        HAS LOOP REACHED
90    $ACEXIT   STX      RLENG          SAVE RECORD LENGTH
```

Fig 4.9(d) Use of Time-variable with EOF NOT NULL and MAXLENGTH being NULL

The above programs show the expansion of Macro invocation statements with different values for the time variables. In figure 4.9(b) the &EOF value is NULL. When the macro invocation is done, IF statement is executed, if it is true EORCK is set to 1, otherwise normal execution of the other part of the program is continued.

The macro processor must maintain a symbol table that contains the value of all macro-time variables used. Entries in this table are modified when SET statements are processed. The table is used to look up the current value of the macro-time variable whenever it is required.

When an IF statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated.

If the value of this expression TRUE,

- The macro processor continues to process lines from the DEFTAB until it encounters the ELSE or ENDIF statement.
- If an ELSE is found, macro processor skips lines in DEFTAB until the next ENDIF.
- Once it reaches ENDIF, it resumes expanding the macro in the usual way.

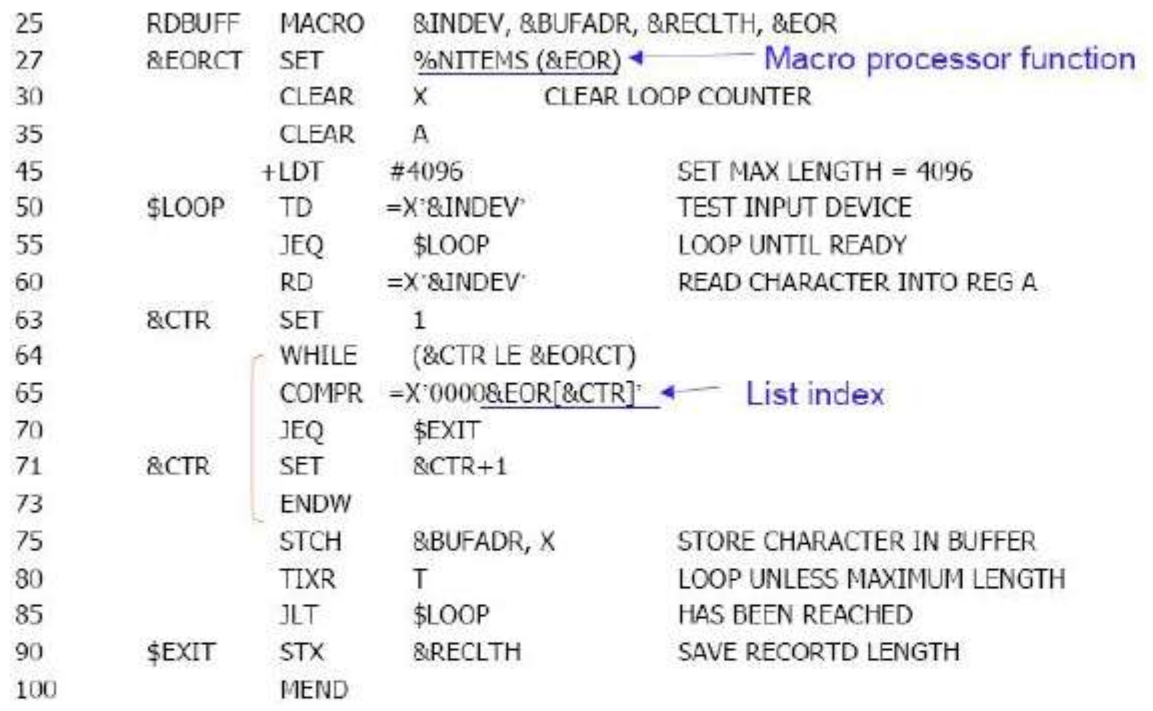
If the value of the expression is FALSE,

- The macro processor skips ahead in DEFTAB until it encounters next ELSE or ENDIF statement.
- The macro processor then resumes normal macro expansion.

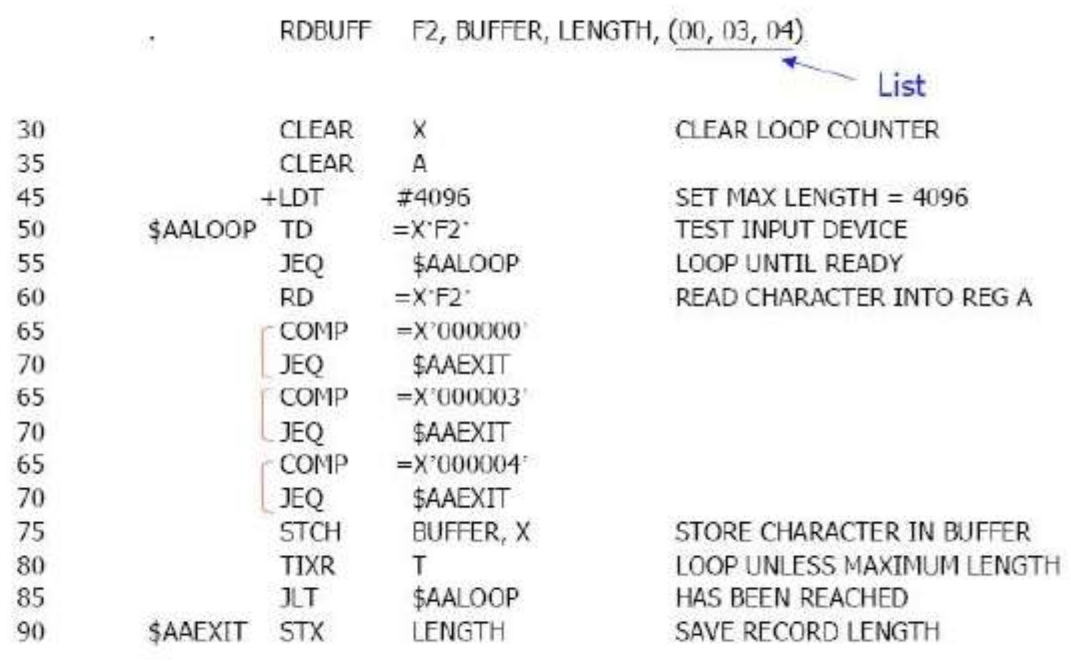
The *macro-time* IF-ELSE-ENDIF structure provides a mechanism for either generating(once) or skipping selected statements in the macro body. There is another construct WHILE statement which specifies that the following line until the next ENDW statement, are to be generated repeatedly as long as a particular condition is true. The testing of this condition, and the looping are done during the macro is under expansion. The example shown below shows the usage of Macro-Time Looping statement.

WHILE-ENDW structure

- When an WHILE statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated.
- TRUE
    - The macro processor continues to process lines from DEFTAB until it encounters the next ENDW statement.
    - When ENDW is encountered, the macro processor returns to the preceding WHILE, re-evaluates the Boolean expression, and takes action based on the new value.
- FALSE
    - The macro processor skips ahead in DEFTAB until it finds the next ENDW statement and then resumes normal macro expansion.



```
RDBUFF   MACRO   &INDEV, &BUFADR, &RECLTH, &EOR
&EORCT   SET     %NITEMS (&EOR)
         CLEAR   X            CLEAR LOOP COUNTER
         CLEAR   A
        +LDT    #4096              SET MAX LENGTH = 4096
$LOOP    TD     =X'&INDEV'         TEST INPUT DEVICE
         JEQ     $LOOP             LOOP UNTIL READY
         RD     =X'&INDEV'         READ CHARACTER INTO REG A
&CTR     SET     1
         WHILE   (&CTR LE &EORCT)
         COMPR  =X'0000&EOR[&CTR]'
         JEQ     $EXIT
&CTR     SET     &CTR+1
         ENDW
         STCH    &BUFADR, X        STORE CHARACTER IN BUFFER
         TIXR    T                 LOOP UNLESS MAXIMUM LENGTH
         JLT     $LOOP             HAS BEEN REACHED
$EXIT    STX     &RECLTH           SAVE RECORTD LENGTH
         MEND
```



```
  .             RDBUFF    F2, BUFFER, LENGTH, (00, 03, 04)

30                CLEAR     X              CLEAR LOOP COUNTER
35                CLEAR     A
45               +LDT       #4096          SET MAX LENGTH = 4096
50     $AALOOP    TD        =X'F2'         TEST INPUT DEVICE
55                JEQ       $AALOOP        LOOP UNTIL READY
60                RD        =X'F2'         READ CHARACTER INTO REG A
65                COMP      =X'000000'
70                JEQ       $AAEXIT
65                COMP      =X'000003'
70                JEQ       $AAEXIT
65                COMP      =X'000004'
70                JEQ       $AAEXIT
75                STCH      BUFFER, X      STORE CHARACTER IN BUFFER
80                TIXR      T              LOOP UNLESS MAXIMUM LENGTH
85                JLT       $AALOOP        HAS BEEN REACHED
90     $AAEXIT    STX       LENGTH         SAVE RECORD LENGTH
```

4.2.4 Keyword Macro Parameters

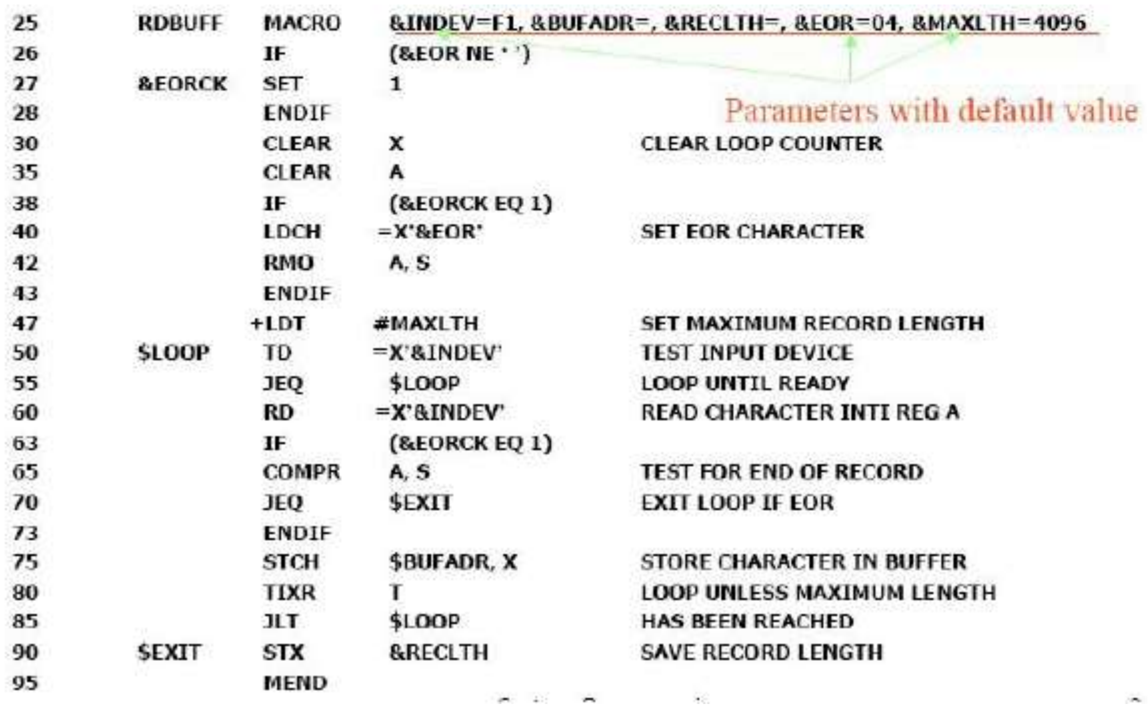
All the macro instruction definitions used positional parameters. Parameters and arguments are matched according to their positions in the macro prototype and the macro invocation statement. The programmer needs to be careful while specifying the arguments. If an argument is to be omitted the macro invocation statement must contain a null argument mentioned with two commas.

Positional parameters are suitable for the macro invocation. But if the macro invocation has large number of parameters, and if only few of the values need to be used in a typical invocation, a different type of parameter specification is required (for example, in many cases most of the parameters may have default values, and the invocation may mention only the changes from the default values).

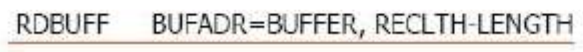
Ex: XXX MACRO &P1, &P2, …., &P20, ….
XXX A1, A2,,,,,,,,,,…,,A20,…..
Null arguments

Keyword parameters

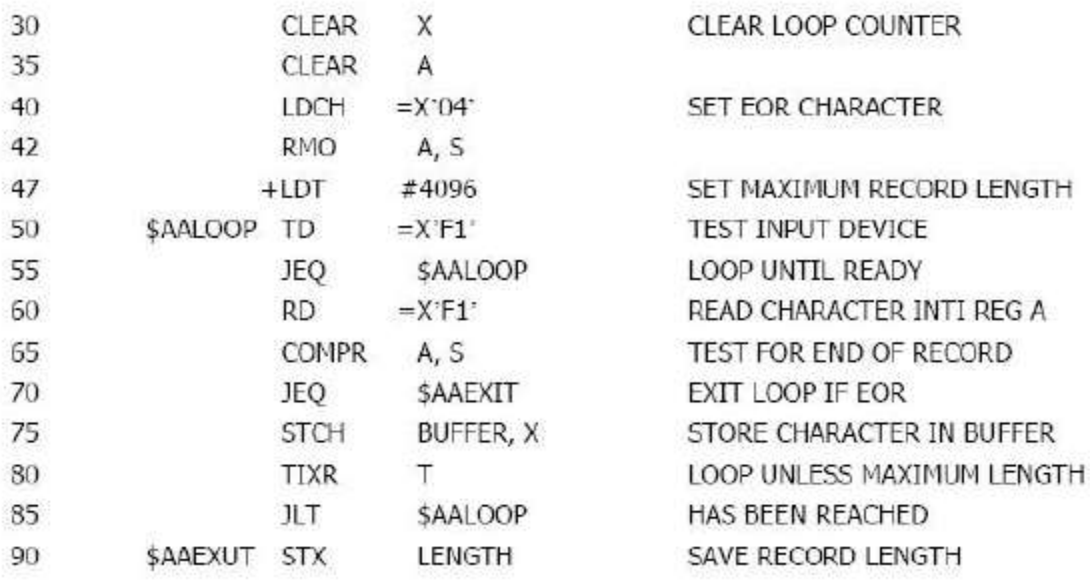
- Each argument value is written with a keyword that names the corresponding parameter.
- Arguments may appear in any order.
- Null arguments no longer need to be used.
- Ex: XXX P1=A1, P2=A2, P20=A20.
- It is easier to read and much less error-prone than the positional method.



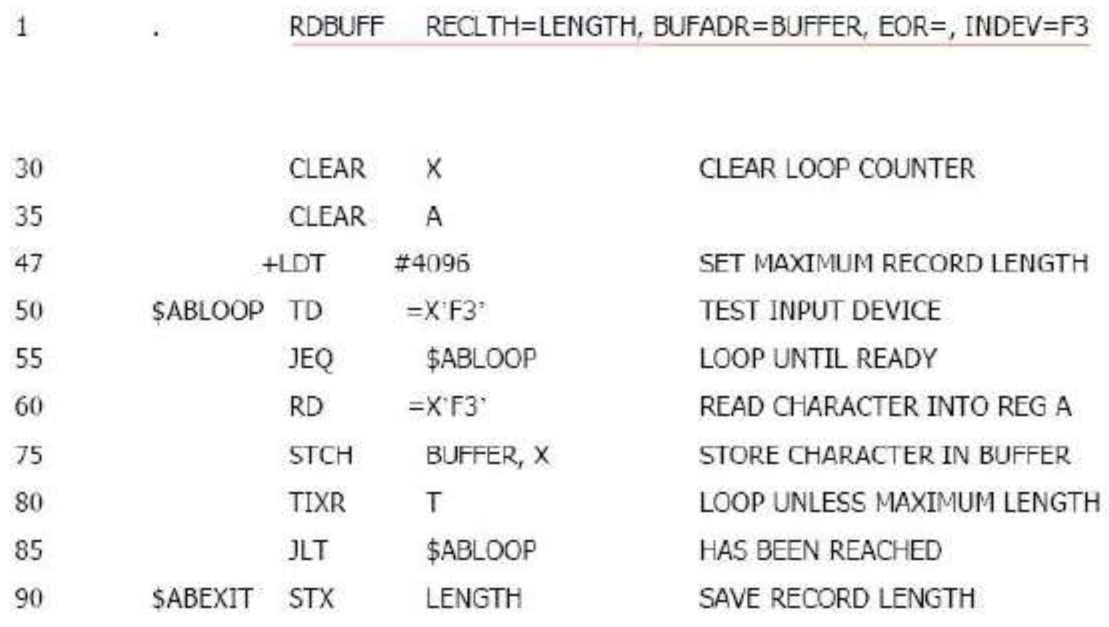
```
25   RDBUFF   MACRO   &INDEV=F1, &BUFADR=, &RECLTH=, &EOR=04, &MAXLTH=4096
26            IF      (&EOR NE ' ')
27   &EORCK   SET     1
28            ENDIF
30            CLEAR   X              CLEAR LOOP COUNTER
35            CLEAR   A
38            IF      (&EORCK EQ 1)
40            LDCH   =X'&EOR'        SET EOR CHARACTER
42            RMO     A, S
43            ENDIF
47           +LDT    #MAXLTH         SET MAXIMUM RECORD LENGTH
50   $LOOP    TD     =X'&INDEV'      TEST INPUT DEVICE
55            JEQ     $LOOP          LOOP UNTIL READY
60            RD     =X'&INDEV'      READ CHARACTER INTI REG A
63            IF      (&EORCK EQ 1)
65            COMPR   A, S           TEST FOR END OF RECORD
70            JEQ     $EXIT          EXIT LOOP IF EOR
73            ENDIF
75            STCH    $BUFADR, X     STORE CHARACTER IN BUFFER
80            TIXR    T              LOOP UNLESS MAXIMUM LENGTH
85            JLT     $LOOP          HAS BEEN REACHED
90   $EXIT    STX     &RECLTH        SAVE RECORD LENGTH
95            MEND
```



```
RDBUFF    BUFADR=BUFFER, RECLTH-LENGTH
```



```
30              CLEAR    X            CLEAR LOOP COUNTER
35              CLEAR    A
40              LDCH    =X'04'        SET EOR CHARACTER
42              RMO      A, S
47             +LDT     #4096         SET MAXIMUM RECORD LENGTH
50   $AALOOP    TD      =X'F1'        TEST INPUT DEVICE
55              JEQ      $AALOOP      LOOP UNTIL READY
60              RD      =X'F1'        READ CHARACTER INTI REG A
65              COMPR    A, S         TEST FOR END OF RECORD
70              JEQ      $AAEXIT      EXIT LOOP IF EOR
75              STCH     BUFFER, X    STORE CHARACTER IN BUFFER
80              TIXR     T            LOOP UNLESS MAXIMUM LENGTH
85              JLT      $AALOOP      HAS BEEN REACHED
90   $AAEXUT    STX      LENGTH       SAVE RECORD LENGTH
```



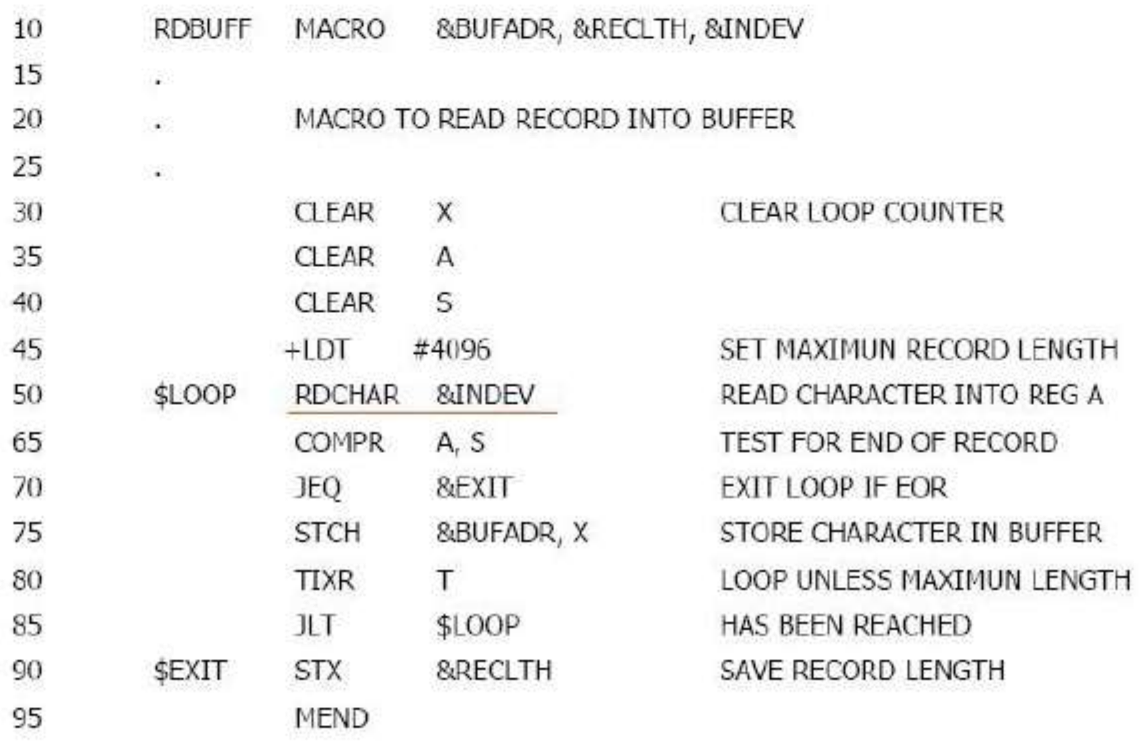
```
.        RDBUFF   RECLTH=LENGTH, BUFADR=BUFFER, EOR=, INDEV=F3


                  CLEAR    X                 CLEAR LOOP COUNTER
                  CLEAR    A
                 +LDT     #4096              SET MAXIMUM RECORD LENGTH
         $ABLOOP  TD       =X'F3'            TEST INPUT DEVICE
                  JEQ       $ABLOOP          LOOP UNTIL READY
                  RD       =X'F3'            READ CHARACTER INTO REG A
                  STCH      BUFFER, X        STORE CHARACTER IN BUFFER
                  TIXR      T                LOOP UNLESS MAXIMUM LENGTH
                  JLT       $ABLOOP          HAS BEEN REACHED
         $ABEXIT  STX       LENGTH           SAVE RECORD LENGTH
```

Fig 4.10 Example showing the usage of Keyword Parameter

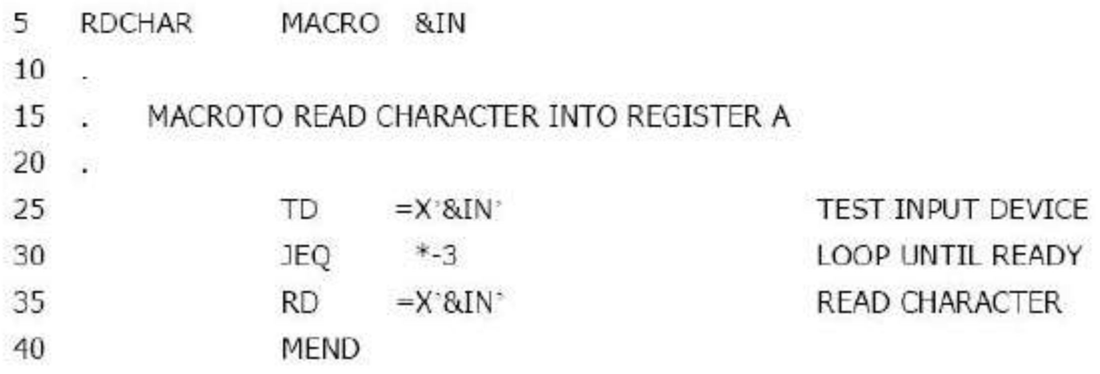
4.3 **Macro Processor Design Options**

4.3.1 Recursive Macro Expansion

We have seen an example of the *definition* of one macro instruction by another. But we have not dealt with the *invocation* of one macro by another. The following example shows the invocation of one macro by another macro.



```
RDBUFF   MACRO    &BUFADR, &RECLTH, &INDEV
.
.        MACRO TO READ RECORD INTO BUFFER
.
         CLEAR    X                   CLEAR LOOP COUNTER
         CLEAR    A
         CLEAR    S
        +LDT      #4096               SET MAXIMUN RECORD LENGTH
$LOOP    RDCHAR   &INDEV              READ CHARACTER INTO REG A
         COMPR    A, S                TEST FOR END OF RECORD
         JEQ      &EXIT               EXIT LOOP IF EOR
         STCH     &BUFADR, X          STORE CHARACTER IN BUFFER
         TIXR     T                   LOOP UNLESS MAXIMUN LENGTH
         JLT      $LOOP               HAS BEEN REACHED
$EXIT    STX      &RECLTH             SAVE RECORD LENGTH
         MEND
```



```
RDCHAR   MACRO   &IN
.
.   MACROTO READ CHARACTER INTO REGISTER A
.
         TD      =X'&IN'              TEST INPUT DEVICE
         JEQ      *-3                 LOOP UNTIL READY
         RD      =X'&IN'              READ CHARACTER
         MEND
```

Problem of Recursive Expansion

- Previous macro processor design cannot handle such kind of recursive macro invocation and expansion
  - The procedure EXPAND would be called recursively, thus the invocation arguments in the ARGTAB will be overwritten. (P.201)
  - The Boolean variable EXPANDING would be set to FALSE when the "inner" macro expansion is finished, *i.e.*, the macro process would forget that it had been in the middle of expanding an "outer" macro.
- Solutions
  - Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained.
  - If you are writing in a language without recursion support, use a stack to take care of pushing and popping local variables and return addresses.

The procedure EXPAND would be called when the macro was recognized. The arguments from the macro invocation would be entered into ARGTAB as follows:

Parameter	Value
1	BUFFER
2	LENGTH
3	F1
4	(unused)
-	-

The Boolean variable EXPANDING would be set to TRUE, and expansion of the macro invocation statement would begin. The processing would proceed normally until statement invoking RDCHAR is processed. This time, ARGTAB would look like

Parameter	Value
1	F1
2	(Unused)
--	--

At the expansion, when the end of RDCHAR is recognized, EXPANDING would be set to FALSE. Thus the macro processor would 'forget' that it had been in the middle of expanding a macro when it encountered the RDCHAR statement. In addition, the arguments from the original macro invocation (RDBUFF) would be lost because the value in ARGTAB was overwritten with the arguments from the invocation of RDCHAR.

4.3.2 General-Purpose Macro Processors

- Macro processors that do not dependent on any particular programming language, but can be used with a variety of different languages
- **Pros**
    - Programmers do not need to learn many macro languages.
    - Although its development costs are somewhat greater than those for a language specific macro processor, this expense does not need to be repeated for each language, thus save substantial overall cost.
- **Cons**
    - Large number of details must be dealt with in a real programming language
        - Situations in which normal macro parameter substitution should not occur, e.g., comments.
        - Facilities for grouping together terms, expressions, or statements
        - Tokens, e.g., identifiers, constants, operators, keywords
        - Syntax had better be consistent with the source programming language

4.3.3 Macro Processing within Language Translators

- The macro processors we discussed are called “Preprocessors”.
  - Process macro definitions
  - Expand macro invocations
  - Produce an expanded version of the source program, which is then used as input to an assembler or compiler
- You may also combine the macro processing functions with the language translator:
  - Line-by-line macro processor
  - Integrated macro processor

4.3.4 Line-by-Line Macro Processor

- Used as a sort of input routine for the assembler or compiler
  - Read source program
  - Process macro definitions and expand macro invocations
  - Pass output lines to the assembler or compiler
- Benefits
  - Avoid making an extra pass over the source program.
  - Data structures required by the macro processor and the language translator can be combined (e.g., OPTAB and NAMTAB)
  - Utility subroutines can be used by both macro processor and the language translator.
    - Scanning input lines
    - Searching tables
    - Data format conversion
  - It is easier to give diagnostic messages related to the source statements

4.3.5 Integrated Macro Processor

- An integrated macro processor can potentially make use of any information about the source program that is extracted by the language translator.
  - Ex (blanks are not significant in FORTRAN)
    - DO 100 I = 1,20
      - a DO statement
    - DO 100 I = 1
      - An assignment statement
      - DO100I: variable (blanks are not significant in FORTRAN)
- An integrated macro processor can support macro instructions that depend upon the context in which they occur.