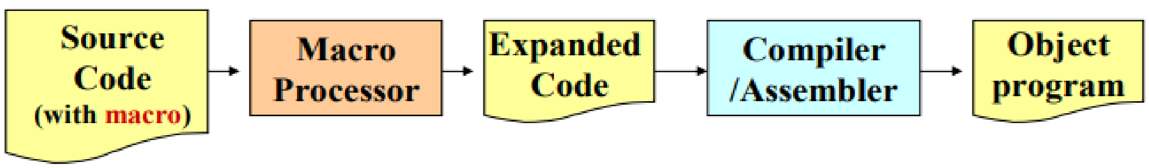
Module 5

Macro Preprocessor

Macro Instruction Definition and Expansion - One pass Macro processor Algorithm and data structures, Machine Independent Macro Processor Features, Macro processor design options

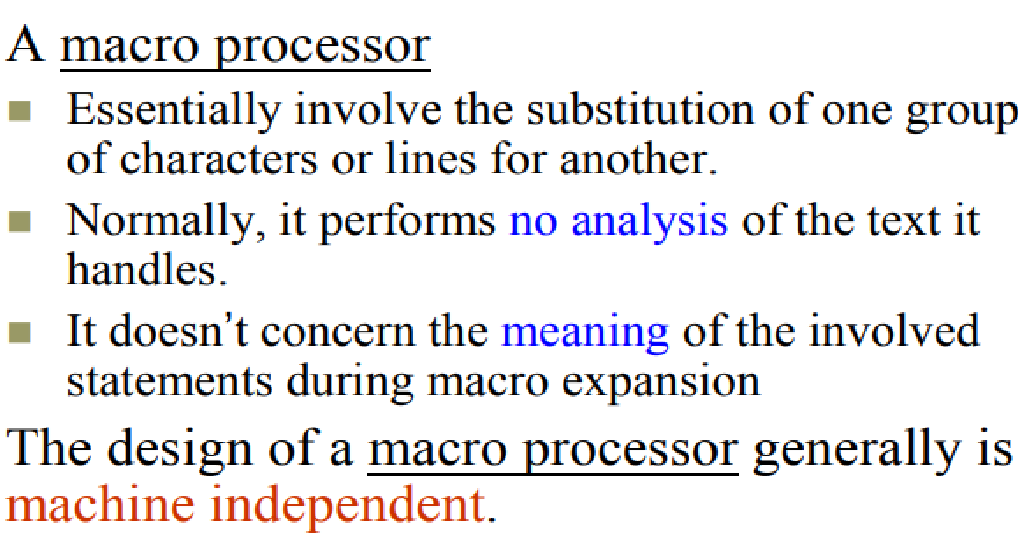
Overview



Macro Instruction Definition and Expansion

A macro instruction (abbreviated to macro) is simply a notational convenience for the programmer. A macro represents a commonly used group of statements in the source programming language.

Expanding a macros – Replace each macro instruction with the corresponding group of source language statements. Macro instruction allows the programmer to write a shorthand version of a program, and leave the mechanical details to be handled by macro processor.



Three examples of actual macro processors:

- A macro processor designed for use by assembler language programmers
- Used with a high-level programming language
- General-purpose macro processor, which is not tied to any particular language

Basic Macro Processor Functions

- Macro Definition
- Macro Invocation
- Macro Expansion

Macro Definition

- 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
- label op operands

```
label    op       operands
name     MACRO    parameters
           :
          body
           :
         MEND
```

- Parameters: the entries in the operand field identify the parameters of the macro instruction
  - We require each parameter begins with '&'
- Body: the statements that will be generated as the expansion of the macro.
- Prototype for the macro:
  - The *macro name* and *parameters* define a pattern or *prototype* for the macro instructions used by the programmer

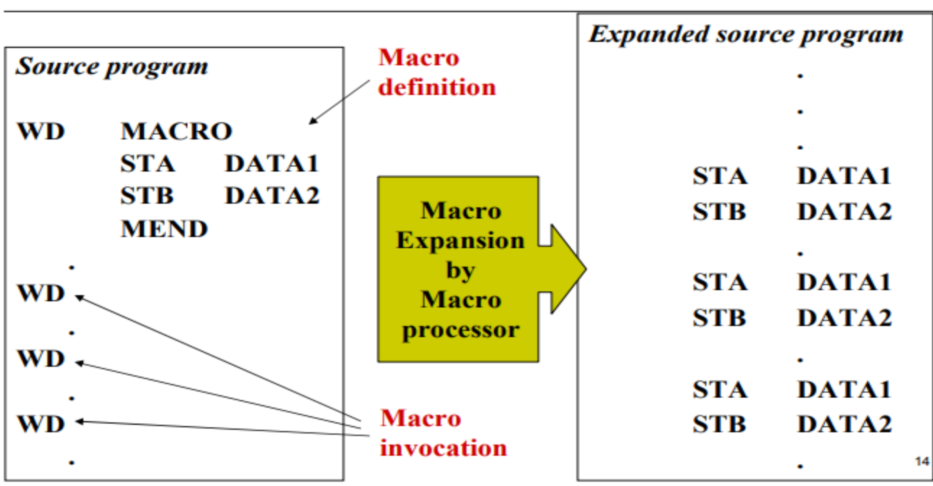
Macro Invocation

- A ***macro invocation statement*** (a ***macro call***) gives the **name** of the macro instruction being invoked and the **arguments** in expanding the macro.
- Macro Invocation vs. Subroutine Call.
  - Statements of the macro body are expanded each time the macro is invoked.
  - Statements of the subroutine appear only one, regardless of how many times the subroutine is called.
  - Macro invocation is more efficient than subroutine call, however, the code size is larger

Macro Expansion

- Each macro invocation statement will be expanded into the statements that form the **body** of the macro.
- Arguments from the macro invocation are substituted for the parameters in the macro prototype.
  - The arguments and parameters are associated with one another according to their **positions.**
    - The first argument in the macro invocation corresponds to the first parameter in the macro prototype, etc.

Macro Expansion



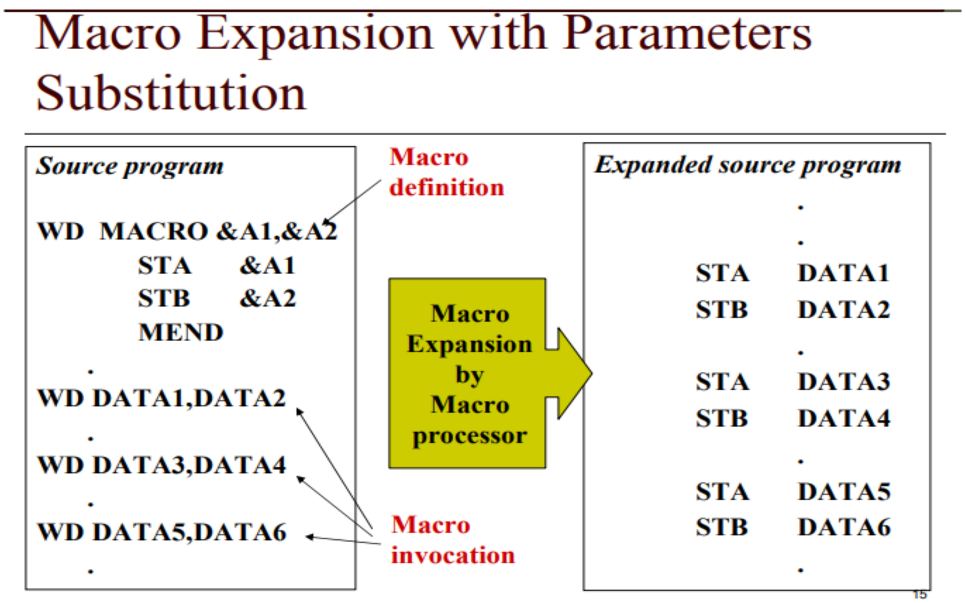


Figure next shows an example of SIC/XE program using macro instructions.

```
COPY     START   0               COPY FILE FROM INPUT TO OUTPUT
RDBUFF   MACRO   &INDEV,&BUFADR,&RECLTH
.
.        MACRO TO READ RECORD INTO BUFFER
.
         CLEAR   X               CLEAR LOOP COUNTER
         CLEAR   A
         CLEAR   S
        +LDT     #4096           SET MAXIMUM RECORD LENGTH
         TD      =X'&INDEV'      TEST INPUT DEVICE
         JEQ     *-3             LOOP UNTIL READY
         RD      =X'&INDEV'      READ CHARACTER INTO REG A
         COMPR   A,S             TEST FOR END OF RECORD
         JEQ     *+11            EXIT LOOP IF EOR
         STCH    &BUFADR,X       STORE CHARACTER IN BUFFER
         TIXR    T               LOOP UNLESS MAXIMUM LENGTH
         JLT     *-19              HAS BEEN REACHED
         STX     &RECLTH         SAVE RECORD LENGTH
         MEND
```

•Macro definition

Macro body contains no label 9

•Macro definition

```
WRBUFF  MACRO   &OUTDEV,&BUFADR,&RECLTH
.
.       MACRO TO WRITE RECORD FROM BUFFER
.
        CLEAR   X               CLEAR LOOP COUNTER
        LDT     &RECLTH
        LDCH    &BUFADR,X       GET CHARACTER FROM BUFFER
        TD      =X'&OUTDEV'     TEST OUTPUT DEVICE
        JEQ     *-3             LOOP UNTIL READY
        WD      =X'&OUTDEV'     WRITE CHARACTER
        TIXR    T               LOOP UNTIL ALL CHARACTERS
        JLT     *-14              HAVE BEEN WRITTEN
        MEND
```

Macro body contains no label

•Macro invocation

```
.
.       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
```

Figure 4.1 Use of macros in a SIC/XE program.

```
            Source statement

COPY      START   0                    COPY FILE FROM INPUT TO OUTPUT
RDBUFF    MACRO   &INDEV,&BUFADR,&RECLTH
.
.         MACRO TO READ RECORD INTO BUFFER
.
          CLEAR   X                    CLEAR LOOP COUNTER
          CLEAR   A
          CLEAR   S
         +LDT     #4096                SET MAXIMUM RECORD LENGTH
          TD      =X'&INDEV'           TEST INPUT DEVICE
          JEQ     *-3                  LOOP UNTIL READY
          RD      =X'&INDEV'           READ CHARACTER INTO REG A
          COMPR   A,S                  TEST FOR END OF RECORD
          JEQ     *+11                 EXIT LOOP IF EOR
          STCH    &BUFADR,X            STORE CHARACTER IN BUFFER
          TIXR    T                    LOOP UNLESS MAXIMUM LENGTH
          JLT     *-19                   HAS BEEN REACHED
          STX     &RECLTH              SAVE RECORD LENGTH
          MEND
WRBUFF    MACRO   &OUTDEV,&BUFADR,&RECLTH
.
.         MACRO TO WRITE RECORD FROM BUFFER
.
          CLEAR   X                    CLEAR LOOP COUNTER
          LDT     &RECLTH
          LDCH    &BUFADR,X            GET CHARACTER FROM BUFFER
          TD      =X'&OUTDEV'          TEST OUTPUT DEVICE
          JEQ     *-3                  LOOP UNTIL READY
          WD      =X'&OUTDEV'          WRITE CHARACTER
          TIXR    T                    LOOP UNTIL ALL CHARACTERS
          JLT     *-14                   HAVE BEEN WRITTEN
          MEND
.
.         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
```

Figure 4.1 Use of macros in a SIC/XE program.

- This program defines and uses two macro instructions – RDBUFF and WRBUFF. The functions of RDBUFF macro are similar to those of the RDREC subroutine which we have studied earlier. Likewise WRBUFF macro similar to WRREC subroutine. The definitions of these macro instructions appear in the source program following the start statement. Two new assembler directives are used in macro definitions:

  1. MACRO

2. MEND

The first MACRO statement (line 10) identifies the beginning of a macro definition. The symbol in the label field (RDBUFF) is the name of the macro, and the entries in the operand field identify the parameters of the macro instructions. In our macro language, each parameter begins with the character &, which facilitates the substitution of parameters during macro expansion. The macro name and parameters define a pattern or prototype for the macro instructions used by the programmer. Following the MACRO directive, are the statements that make up the body of the macro definition (line 15 through 90). These are the statements that will be generated as the expansion of the macro. The MEND assembler directive (line 95) marks the end of the macro definition. The definition of the WRBUFF macro(line 100 through 160) follows a similar pattern. The main program itself begins at line 180. The statement on line 190 is a macro invocation statement that gives the name of the macro instruction being invoked and the argument5s to be used in expanding the macro. A macro invocation statement is often called referred to as macro call. The macro invocation and subroutine call are different. The above figure gives the input program to a macro processor. The output generated is given in next figure.

•Macro expansion

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 MAXIMUM RECORD LENGTH
190e		TD	=X'F1'	TEST INPUT DEVICE
190f		JEQ	*-3	LOOP UNTIL READY
190g		RD	=X'F1'	READ CHARACTER INTO REG A
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 MAXIMUM LENGTH
190l		JLT	*-19	HAS BEEN REACHED
190m		STX	LENGTH	SAVE RECORD LENGTH

•Macro expansion

```
195    LDA     LENGTH              TEST FOR END OF FILE
200    COMP    #0
205    JEQ     ENDFIL              EXIT IF EOF FOUND
210  . WRBUFF  05,BUFFER,LENGTH    WRITE OUTPUT RECORD
210a   CLEAR   X                   CLEAR LOOP COUNTER
210b   LDT     LENGTH
210c   LDCH    BUFFER,X            GET CHARACTER FROM BUFFER
210d   TD      =X'05'              TEST OUTPUT DEVICE
210e   JEQ     *-3                 LOOP UNTIL READY
210f   WD      =X'05'              WRITE CHARACTER
210g   TIXR    T                   LOOP UNTIL ALL CHARACTERS
210h   JLT     *-14                  HAVE BEEN WRITTEN
```

The macro instruction definitions have been deleted since they are no longer needed after the macros are expanded. Each macro invocation statement has been expanded into the statements that form the body of the macro, with the arguments from the macro invocation substituted for the parameters in the macro prototype. The arguments and parameters are associated with one another according to their positions. The first argument in the macro invocation corresponds to the first parameter in the macro prototype, and so on. In expanding the macro invocation on line 190, for example, the argument F1 is substituted for the parameter &INDEV wherever it occurs in the body of the macro. Similarly, BUFFER is substituted for &BUFADR and LENGTH is substituted for &RECLTH. Lines 190a through 190m show the complete expansion of the macro invocation on line 190. The comment lines within the macro body have been deleted, but comments on individual statements have been retained. Note that the macro invocation statement itself has been included as a comment line. This serves as a documentation of the statement written by the programmer. The label on the macro invocation statement (CLOOP) has been retained as a label on the first statement generated in the macro expansion. This allows the programmer to use a macro instruction in exactly the same way as an assembler language mnemonic. The macro invocations on line 210 and 220 are expanded in the same way. Note that the two invocations of WRBUFF specify different arguments, so they produce different expansions. After macro processing, the expanded file obtained (fig 4.2) servers as the input to assembler. The macro invocation statements will be treated as comments, and the statements generated from the macro expansions will be assembled exactly as though they had been written directly by the programmer.

```
Line    Source statement

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 MAXIMUM RECORD LENGTH
190e              TD       =X'F1'              TEST INPUT DEVICE
190f              JEQ      *-3                 LOOP UNTIL READY
190g              RD       =X'F1'              READ CHARACTER INTO REG A
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 MAXIMUM LENGTH
190l              JLT      *-19                  HAS BEEN REACHED
190m              STX      LENGTH              SAVE RECORD LENGTH
195               LDA      LENGTH              TEST FOR END OF FILE
200               COMP     #0
205               JEQ      ENDFIL              EXIT IF EOF FOUND
210               WRBUFF   05,BUFFER,LENGTH    WRITE OUTPUT RECORD
210a              CLEAR    X                   CLEAR LOOP COUNTER
210b              LDT      LENGTH
210c              LDCH     BUFFER,X            GET CHARACTER FROM BUFFER
210d              TD       =X'05'              TEST OUTPUT DEVICE
210e              JEQ      *-3                 LOOP UNTIL READY
210f              WD       =X'05'              WRITE CHARACTER
210g              TIXR     T                   LOOP UNTIL ALL CHARACTERS
210h              JLT      *-14                  HAVE BEEN WRITTEN
215               J        CLOOP               LOOP
220     .ENDFIL   WRBUFF   05,EOF,THREE        INSERT EOF MARKER
220a    ENDFIL    CLEAR    X                   CLEAR LOOP COUNTER
220b              LDT      THREE
220c              LDCH     EOF,X               GET CHARACTER FROM BUFFER
220d              TD       =X'05'              TEST OUTPUT DEVICE
220e              JEQ      *-3                 LOOP UNTIL READY
220f              WD       =X'05'              WRITE CHARACTER
220g              TIXR     T                   LOOP UNTIL ALL CHARACTERS
220h              JLT      *-14                  HAVE BEEN WRITTEN
225               J        @RETADR
230     EOF       BYTE     C'EOF'
235     THREE     WORD     3
240     RETADR    RESW     1
245     LENGTH    RESW     1                   LENGTH OF RECORD
250     BUFFER    RESB     4096                4096-BYTE BUFFER AREA
255               END      FIRST
```

Figure 4.2 Program from Fig. 4.1 with macros expanded.

Comparing the fig 2.5 and fig 4.2, there is a difference between subroutine call and macro invocation can be clearly understood. In fig 4.2, the statements from the body of the macro WRBUFF are generated twice: line 210a through 210h and lines 220a through 220h. In program 2.5, the corresponding statements appear only once in the subroutine WRREC (line 210 through 240). In general, the statements that form the expansion of a macro are generated (and assembled) each time the macro is invoked. Statements in a subroutine appear only once, regardless of how many times the subroutine is called. **Note: the body of the macro contains no labels**.

In fig 4.1 line 140 contains the statement, JEQ *-3 and line 155 contains JLT*-14. The corresponding statements in the WRREC subroutine are JEQ WLOOP and JLT WLOOP where WLOOP is a label on the TD instruction that tests the output device. If such a label appeared on line 135 of the macro body, it would be generated twice –on lines 210d and 220d of fig 4.2. This would result in an error (a duplicate label definition) when the program is assembled. To avoid duplication of symbols, we have eliminated labels from the body of our macro definitions. The use of statements like JLT *-14 is generally considered to be a poor programming practice. It is somewhat less objectionable within a macro definition; however it is still an in convenient and error prone method. *We will study the solutions to avoid this problem in later class*.

Macro Processor Algorithm and Data Structures

- Two-pass macro processor
- One-pass macro processor

Two-pass macro processor

- Pass1: process all macro definitions
- Pass2: expand all macro invocation statements
- **Problem**
  - Does not allow nested macro definitions
  - **Nested macro definitions** - The body of a macro contains definitions of other macros
  - Because all macros would have to be defined during the first pass before any macro invocations were expanded.

```
1   MACROS    MACRO     {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}
```

(a)

```
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}
```

(b)

Figure 4.3 Example of the definition of macros within a macro body.

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.

- **Solution**
  - One-pass macro processor

One-pass macro processor

- One-pass processor can alternate between macro definition and macro expansion
- In One-pass macro processor, every macro must be defined before it is called
- **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.

Data Structures

There are three data structures involved in one –pass macro processor.

1. DEFTAB
2. NAMTAB
3. ARGTAB

DEFTAB

- A ***definition table*** used to *store macro definition* including
  - macro prototype
  - macro body
- Comment lines are omitted.
- *Positional notation* has been used for the parameters for efficiency in substituting arguments.
  - E.g. the first parameter &INDEV has been converted to ?1 (indicating the first parameter in the prototype)

NAMTAB

- A ***name table*** used to *store the macro names*
- Serves as an index to DEFTAB
  - Pointers to the beginning and the end of the macro definition

ARGTAB

- An argument table is used to store the arguments used in the expansion of macro invocation
- When a macro invocation statement is recognized, the arguments are stored in ARGTAB according to their position in the argument list

- As the macro is expanded, arguments are substituted for the corresponding parameters in the macro body

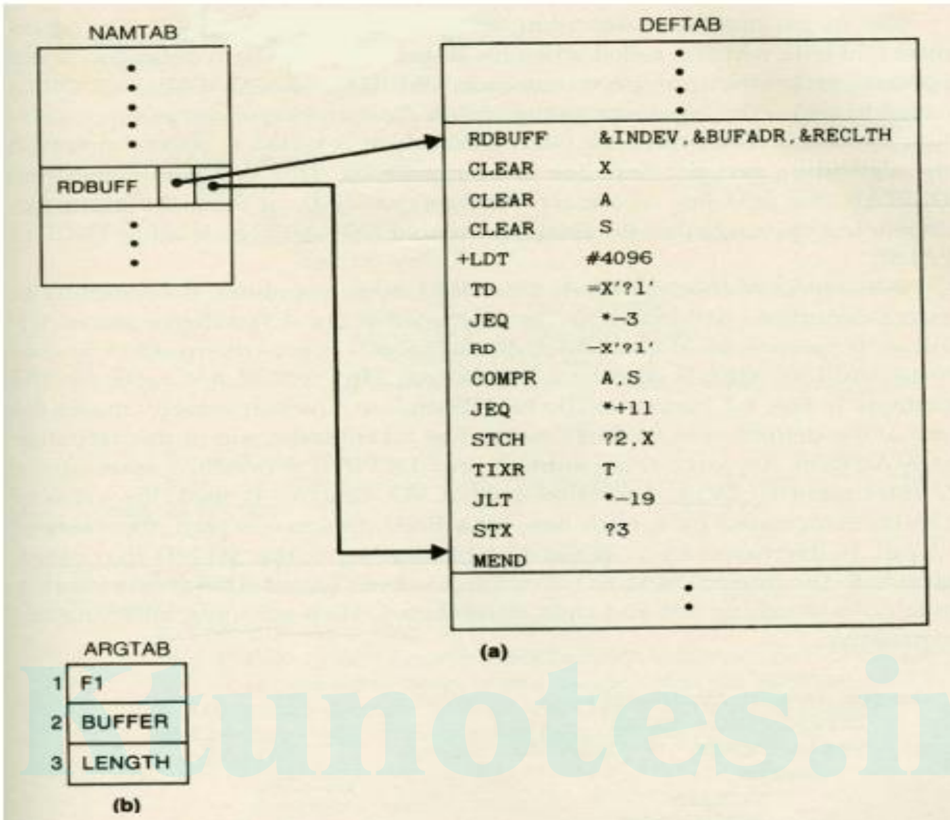


Figure 4.4 Contents of macro processor tables for the program in Fig. 4.1: (a) entries in NAMTAB and DEFTAB defining macro RDBUFF, (b) entries in ARGTAB for invocation of RDBUFF on line 190.

Fig. 4.4 shows positions of the contents of these tables during the processing of the program. Fig. 4.4(a) shows the definition of RDBUFF stored in DEFTAB, with an entry in NAMTAB identifying the beginning and end of the definition. Note the positional notation that has been used for parameters: The parameter &INDEV has been converted to?1(indicating the first parameter in the prototype) &BUFADR has been converted to ?2, and so on. Fig. 4.4(b) shows the ARGTAB as it would appear during expansion of the RDBUFF statement on line 190. For this invocation, the first argument is F1, the second is BUFFER etc. This scheme makes substitution of macro arguments much more efficient. When the ?n notation is recognized in a line from DEFTAB, a simple indexing operation supplies the proper argument from the ARGTAB.

Algorithm

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

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



```
procedure DEFINE
    begin
        enter macro name into NAMTAB
        enter macro prototype into DEFTAB
        LEVEL := 1
        while LEVEL > 0 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
        write 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}
```

- **The procedure PROCESSING**
- **The procedure DEFINE**
  - **Called when the beginning of a macro definition is recognized, makes the appropriate entries in DEFTAB and NAMTAB.**
- **The procedure EXPAND**
  - **Called to set up the argument values in ARGTAB and expand a macro invocation statement.**
- **The procedure GETLINE**
  - **Called at several points in the algorithm, gets the next line to be processed.**
- **EXPANDING is set to TRUE or FALSE.**

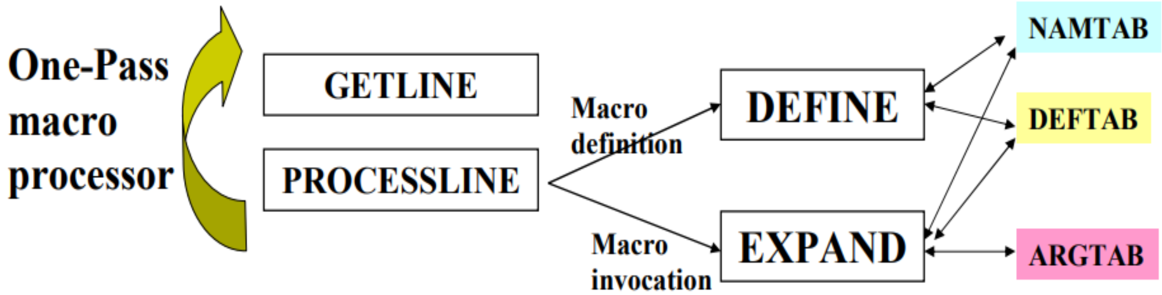
The procedure DEFINE, which is called when the beginning of a macro definition is recognized, makes the appropriate entries in DEFTAB and NAMTAB. EXPAND is called to set up the argument values in ARGTAB and expand a macro invocation statement. The procedure GETLINE, which is called at several points in the algorithm, gets the next line to be processed

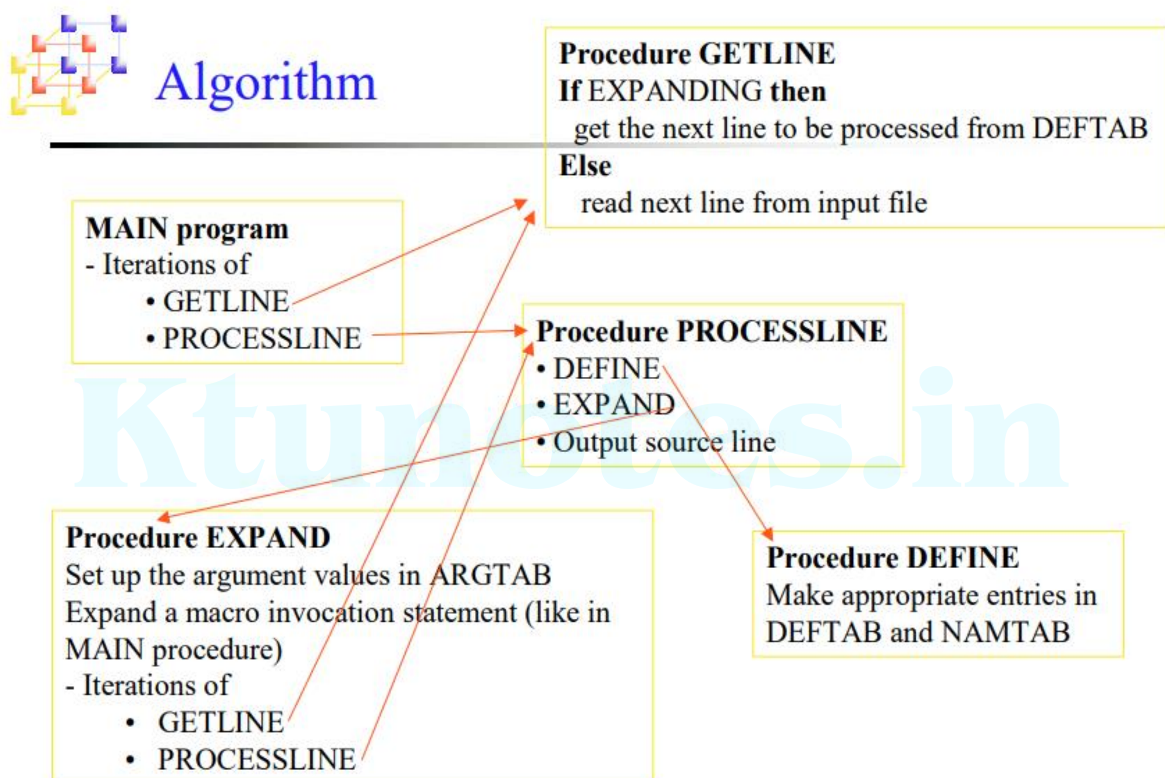
This line may come from DEFTAB (the next line of a macro being expanded), or from the input file, depending upon whether the Boolean variable EXPANDING is set to TRUE or FALSE.

Handling nested macro definition

- In DEFINE procedure
  - When a macro definition is being entered into DEFTAB, the normal approach is to continue until an MEND directive is reached.
  - This would not work for nested macro definition because the first MEND encountered in the inner macro will terminate the whole macro definition process.
  - To solve this problem, a counter LEVEL is used to keep track of the level of macro definitions.
    - Increase LEVEL by 1 each time a MACRO directive is read.
    - Decrease LEVEL by 1 each time a MEND directive is read.
    - A MEND terminates the whole macro definition process when LEVEL reaches 0.
    - This process is very much like matching left and right parentheses when scanning an arithmetic expression.


```
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}
```





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

Machine Independent Macro-Processor Features

1. Concatenation of Macro Parameters
2. Generation of Unique Labels

3. Conditional Macro Expansion
4. Keyword Macro Parameters

Concatenation of Macro Parameters

Most macro processors allow parameters to be concatenated with other character strings. Suppose a program contains one series of variables named by the symbols XA1, XA2, XA3 .... Another series named XB1, XB2,XB3,.. Etc. If similar processing is to be performed on each series of variables, the programmer might want to incorporate this processing into 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). 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&ID
```

in which the parameter &ID is concatenated after the character string X and before the character string 1. There is a problem with such a statement. The beginning of the macro processor is identified by the starting symbol &. However the end of the parameter is not marked. Thus the operand in the foregoing statement could equally well represent the character string X followed by the parameter &ID1. In the particular case, the macro processor could potentially deduce the meaning that was intended. However, if the macro definition contained both &ID and &ID1 as parameters, it would be ambiguous. Most macro processor deal with this problem by providing a special **concatenation operator.**

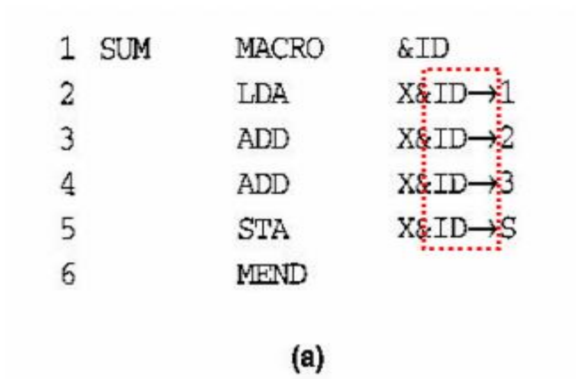
In SIC macro language, this operator is the character →.

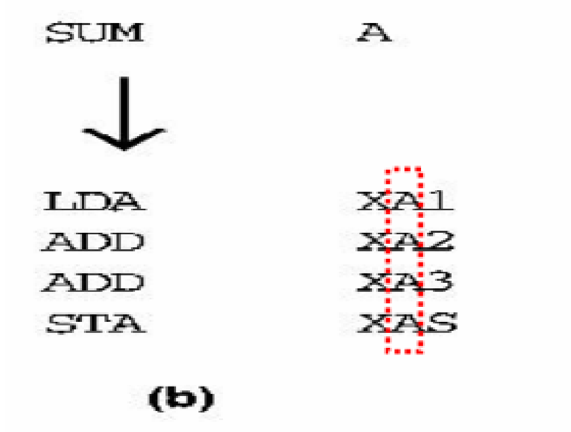
Then the previous statement would be written as:

```
LDA     X&ID→1
```

So that end of the parameter is clearly identified.

The macro processor deletes all occurrences of the concatenation operator immediately after performing parameter substitution, so the character → will not appear in the macro expansion.





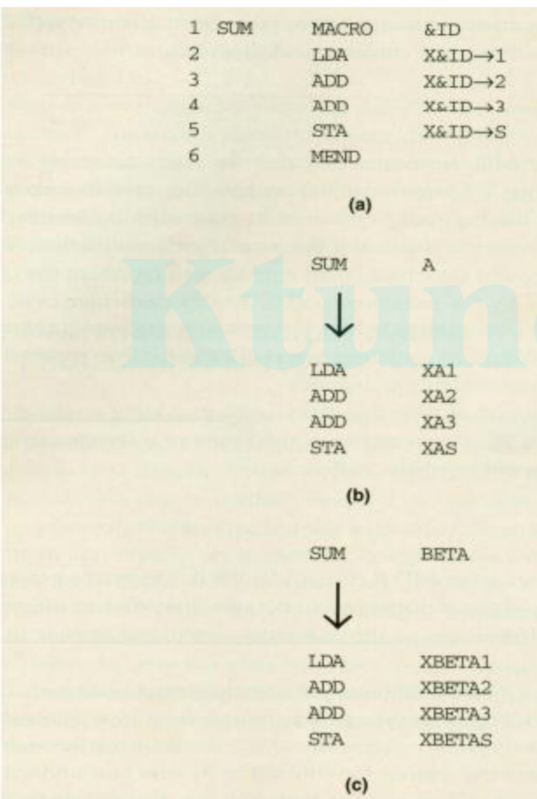
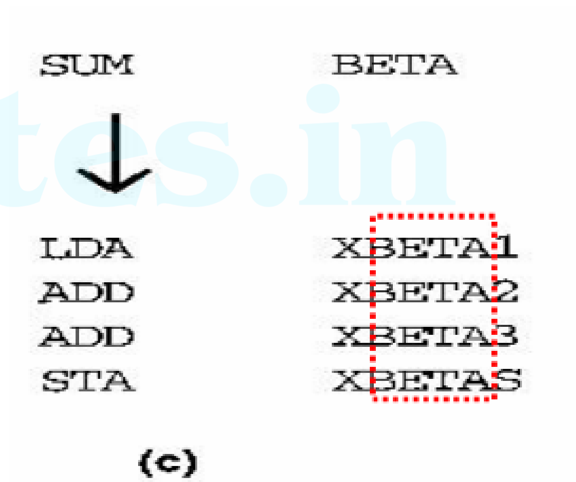
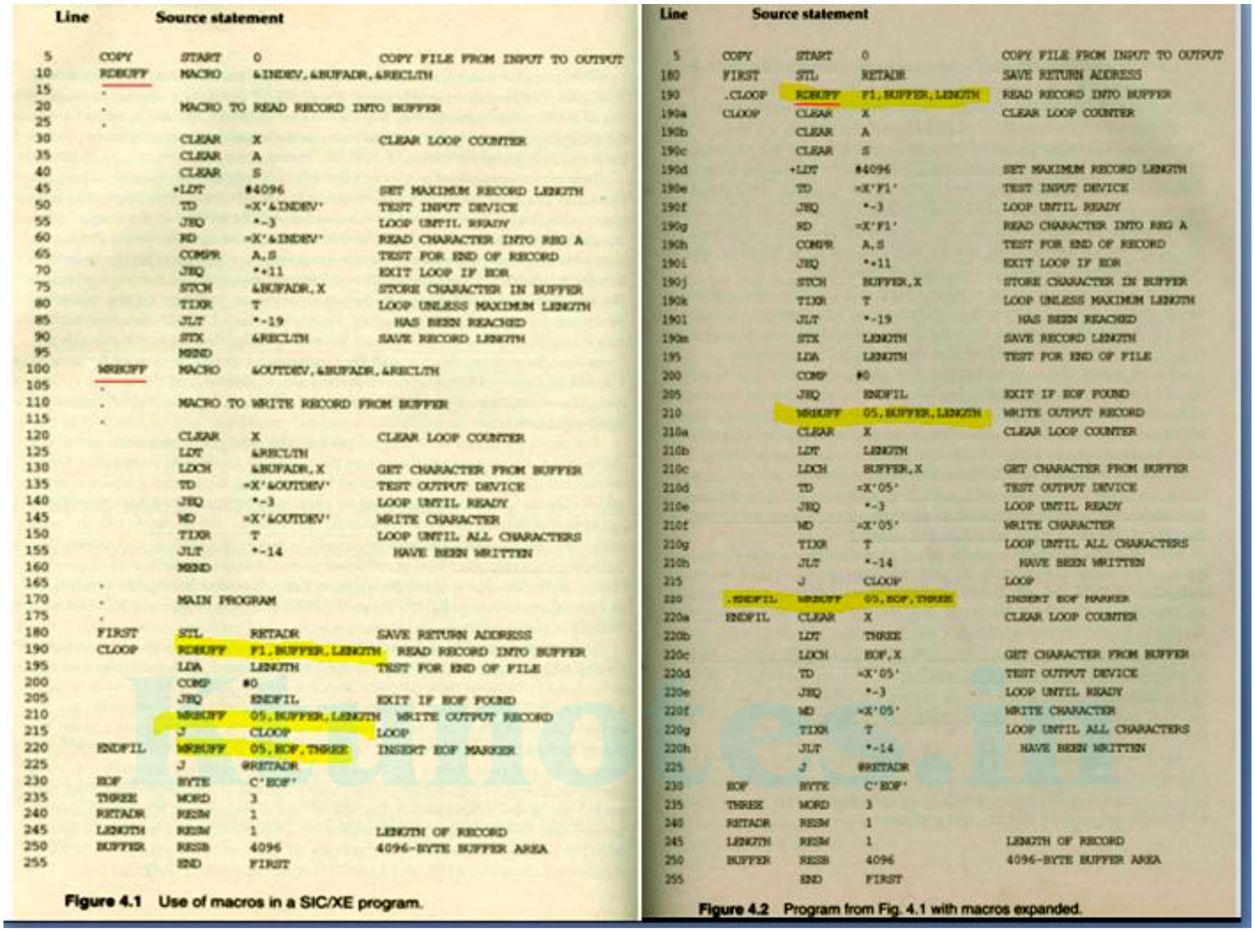


Figure 4.6 Concatenation of macro parameters.



- In fig.(a) shows a macro definition that uses the concatenation operator as previously described

- In fig (b) and(c) shows macro invocation statements and the corresponding macro expansions



```
Line   Source statement

  5    COPY     START   0                 COPY FILE FROM INPUT TO OUTPUT
 10    RDBUFF   MACRO   &INDEV,&BUFADR,&RECLTH
 15    .
 20    .        MACRO TO READ RECORD INTO BUFFER
 25    .
 30             CLEAR   X                 CLEAR LOOP COUNTER
 35             CLEAR   A
 40             CLEAR   S
 45            +LDT     #4096             SET MAXIMUM RECORD LENGTH
 50             TD      =X'&INDEV'        TEST INPUT DEVICE
 55             JEQ     *-3               LOOP UNTIL READY
 60             RD      =X'&INDEV'        READ CHARACTER INTO REG A
 65             COMPR   A,S               TEST FOR END OF RECORD
 70             JEQ     *+11              EXIT LOOP IF EOR
 75             STCH    &BUFADR,X         STORE CHARACTER IN BUFFER
 80             TIXR    T                 LOOP UNLESS MAXIMUM LENGTH
 85             JLT     *-19                HAS BEEN REACHED
 90             STX     &RECLTH           SAVE RECORD LENGTH
 95             MEND
100    WRBUFF   MACRO   &OUTDEV,&BUFADR,&RECLTH
105    .
110    .        MACRO TO WRITE RECORD FROM BUFFER
115    .
120             CLEAR   X                 CLEAR LOOP COUNTER
125             LDT     &RECLTH
130             LDCH    &BUFADR,X         GET CHARACTER FROM BUFFER
135             TD      =X'&OUTDEV'       TEST OUTPUT DEVICE
140             JEQ     *-3               LOOP UNTIL READY
145             WD      =X'&OUTDEV'       WRITE CHARACTER
150             TIXR    T                 LOOP UNTIL ALL CHARACTERS
155             JLT     *-14                HAVE BEEN WRITTEN
160             MEND
165    .
170    .        MAIN PROGRAM
175    .
180    FIRST    STL     RETADR            SAVE RETURN ADDRESS
190    CLOOP    RDBUFF  F1,BUFFER,LENGTH  READ RECORD INTO BUFFER
195             LDA     LENGTH            TEST FOR END OF FILE
200             COMP    #0
205             JEQ     ENDFIL            EXIT IF EOF FOUND
210             WRBUFF  05,BUFFER,LENGTH  WRITE OUTPUT RECORD
215             J       CLOOP             LOOP
220    ENDFIL   WRBUFF  05,EOF,THREE      INSERT EOF MARKER
225             J       @RETADR
230    EOF      BYTE    C'EOF'
235    THREE    WORD    3
240    RETADR   RESW    1
245    LENGTH   RESW    1                 LENGTH OF RECORD
250    BUFFER   RESB    4096              4096-BYTE BUFFER AREA
255             END     FIRST
```

Figure 4.1 Use of macros in a SIC/XE program.

```
Line   Source statement

  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 MAXIMUM RECORD LENGTH
190e            TD      =X'F1'            TEST INPUT DEVICE
190f            JEQ     *-3               LOOP UNTIL READY
190g            RD      =X'F1'            READ CHARACTER INTO REG A
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 MAXIMUM LENGTH
190l            JLT     *-19                HAS BEEN REACHED
190m            STX     LENGTH            SAVE RECORD LENGTH
195             LDA     LENGTH            TEST FOR END OF FILE
200             COMP    #0
205             JEQ     ENDFIL            EXIT IF EOF FOUND
210             WRBUFF  05,BUFFER,LENGTH  WRITE OUTPUT RECORD
210a            CLEAR   X                 CLEAR LOOP COUNTER
210b            LDT     LENGTH
210c            LDCH    BUFFER,X          GET CHARACTER FROM BUFFER
210d            TD      =X'05'            TEST OUTPUT DEVICE
210e            JEQ     *-3               LOOP UNTIL READY
210f            WD      =X'05'            WRITE CHARACTER
210g            TIXR    T                 LOOP UNTIL ALL CHARACTERS
210h            JLT     *-14                HAVE BEEN WRITTEN
215             J       CLOOP             LOOP
220    .ENDFIL  WRBUFF  05,EOF,THREE      INSERT EOF MARKER
220a   ENDFIL   CLEAR   X                 CLEAR LOOP COUNTER
220b            LDT     THREE
220c            LDCH    EOF,X             GET CHARACTER FROM BUFFER
220d            TD      =X'05'            TEST OUTPUT DEVICE
220e            JEQ     *-3               LOOP UNTIL READY
220f            WD      =X'05'            WRITE CHARACTER
220g            TIXR    T                 LOOP UNTIL ALL CHARACTERS
220h            JLT     *-14                HAVE BEEN WRITTEN
225             J       @RETADR
230    EOF      BYTE    C'EOF'
235    THREE    WORD    3
240    RETADR   RESW    1
245    LENGTH   RESW    1                 LENGTH OF RECORD
250    BUFFER   RESB    4096              4096-BYTE BUFFER AREA
255             END     FIRST
```

Figure 4.2 Program from Fig. 4.1 with macros expanded.

- Labels in the macro body may have *duplicate labels* problem
  - If the macro is invocated multiple times.
  - Use of relative addressing is very inconvenient, error-prone, and difficult to read.
    - Example
      - JEQ *-3
      - Inconvenient, error-prone, difficult to read

Generation of Unique Labels

The macro body do not contain labels. This leads to the use of relative addressing at the source statement level. Consider for example, the definition of WRBUFF in fig 4.1. If a label were placed on the TD instruction on line 135, this label would be defined twice-once for each invocation of WRBUFF. This duplicate definition would prevent correct assembly of the resulting expanded program. Because it was not possible to place a label on line 135 of this macro definition, the Jump instruction on line 140 and line 155 were written using the relative operands *-3 and *-14. This sort of relative addressing in a source statement may be acceptable for short jumps such as JEQ *-3. However, for longer jumps spanning several instructions, such notation is very in convenient, error prone and difficult to read. Many macro processors avoid these problems by allowing the creation of special types of labels within macro instruction.

•Macro definition

```
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 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 MAXIMUM LENGTH
         JLT     $LOOP            HAS BEEN REACHED
$EXIT    STX     &RECLTH        SAVE RECORD LENGTH
         MEND
```


RDBUFF F1, BUFFER, LENGTH

•Macro expansion

```
           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 INTO 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
```

(b)

The figure illustrates one technique for generating unique labels within a macro expansion. A definition of the RDBUFF macro is in fig (a). Labels used within the macro body begin with the special character $. Fig(b) shows a macro invocation statement and the resulting macro expansion. Each symbol beginning with $ has been modified by replacing $ with $AA. More generally, the character $ will be replaced by $xx, where xx is a two character alphanumeric counter of the number of macro instructions expanded. For the first macro expansion in a program, xx will have the value AA.

- $LOOP TD =X'&INDEV'
- 1st call:
  - $AALOOP TD =X'F1'
- 2nd call:
  - $ABLOOP TD =X'F1'

For the succeeding macro expansion, xx will be set to AB, AC etc. *If only alphabetic and numeric characters are allowed in xx, such a two-character counter provides for as many as 1296 macro expansion in a single program.* This results in the generation of unique labels for each expansion of a macro instruction.

Conditional Macro Expansion

*In all our previous examples of macro instructions, each invocation of a particular macro was expanded into the same sequence of statements. These statements could be varied by the substitution of parameters, but the form of the statements, and the order in which they appeared where changed.*Most macro processors can modify the sequence of statements generated for a macro expansion, depending on the arguments supplied in the macro invocation. Such a capability adds greatly to the power and flexibility of a macro language. The term conditional assembly can be used to describe this:

- ***Macro-time conditional statements***
  - Macro processor directives:
    - ***IF-ELSE-ENDIF***
    - ***SET***

```
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 CHARACTER
        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 INTO 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 MAXIMUM LENGTH
        JLT     $LOOP             HAS BEEN REACHED
$EXIT   STX     &RECLTH         SAVE RECORD LENGTH
        MEND
```

(a)

```
.       RDBUFF  F3,BUF,RECL,04,2048

          CLEAR   X             CLEAR LOOP COUNTER
          CLEAR   A
          LDCH    =X'04'        SET EOR CHARACTER
          RMO     A,S
          +LDT    #2048         SET MAXIMUM RECORD LENGTH
$AALOOP   TD      =X'F3'        TEST INPUT DEVICE
          JEQ     $AALOOP       LOOP UNTIL READY
          RD      =X'F3'        READ CHARACTER INTO REG A
          COMPR   A,S           TEST FOR END OF RECORD
          JEQ     $AAEXIT       EXIT LOOP IF EOR
          STCH    BUF,X         STORE CHARACTER IN BUFFER
          TIXR    T             LOOP UNLESS MAXIMUM LENGTH
          JLT     $AALOOP         HAS BEEN REACHED
$AAEXIT   STX     RECL          SAVE RECORD LENGTH
```

(b)

Figure 4.8 Use of macro-time conditional statements.

- ***Macro-time variables*** (also called a *set symbol*)
  - Be used to store working values during the macro expansion
  - Any symbol that begins with the character *&* and is not a macro parameter
  - Be initialized to 0
  - Be changed with their values using SET
    - *&EORCK* SET 1

```
      .         RDBUFF   0E,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 INTO 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
```

(c)

```
      .         RDBUFF   F1,BUFF,RLENG,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 INTO 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 BEEN REACHED
90    $ACEXIT   STX       RLENG              SAVE RECORD LENGTH
```

(d)

Figure 4.8 *(cont'd)*

```
25    RDBUFF    MACRO    &INDEV,&BUFADR,&RECLTH,&EOR,&MAXLTH
26              IF       (&EOR NE '')
27    &EORCK    SET      1
28              ENDIF
30              CLEAR    X                   CLEAR LOOP COUNTER
                CLEAR    A
                IF        (&EORCK EQ 1)
40              LDCH     =X'&EOR'            SET EOR CHARACTER
42              RMO       A,S
43              ENDIF
44              IF        (&MAXLTH EQ '')
45             +LDT      #4096               SET MAX LENGTH = 4096
46              ELSE
47             +LDT      #&MAXLTH            SET MAXIMUM RECORD LENGTH
48              ENDIF
50    $LOOP     TD       =X'&INDEV'          TEST INPUT DEVICE
55              JEQ       $LOOP              LOOP UNTIL READY
60              RD       =X'&INDEV'          READ CHARACTER INTO 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
```

IF-ELSE-ENDIF Structure

Macro-time variable

Boolean expression

•

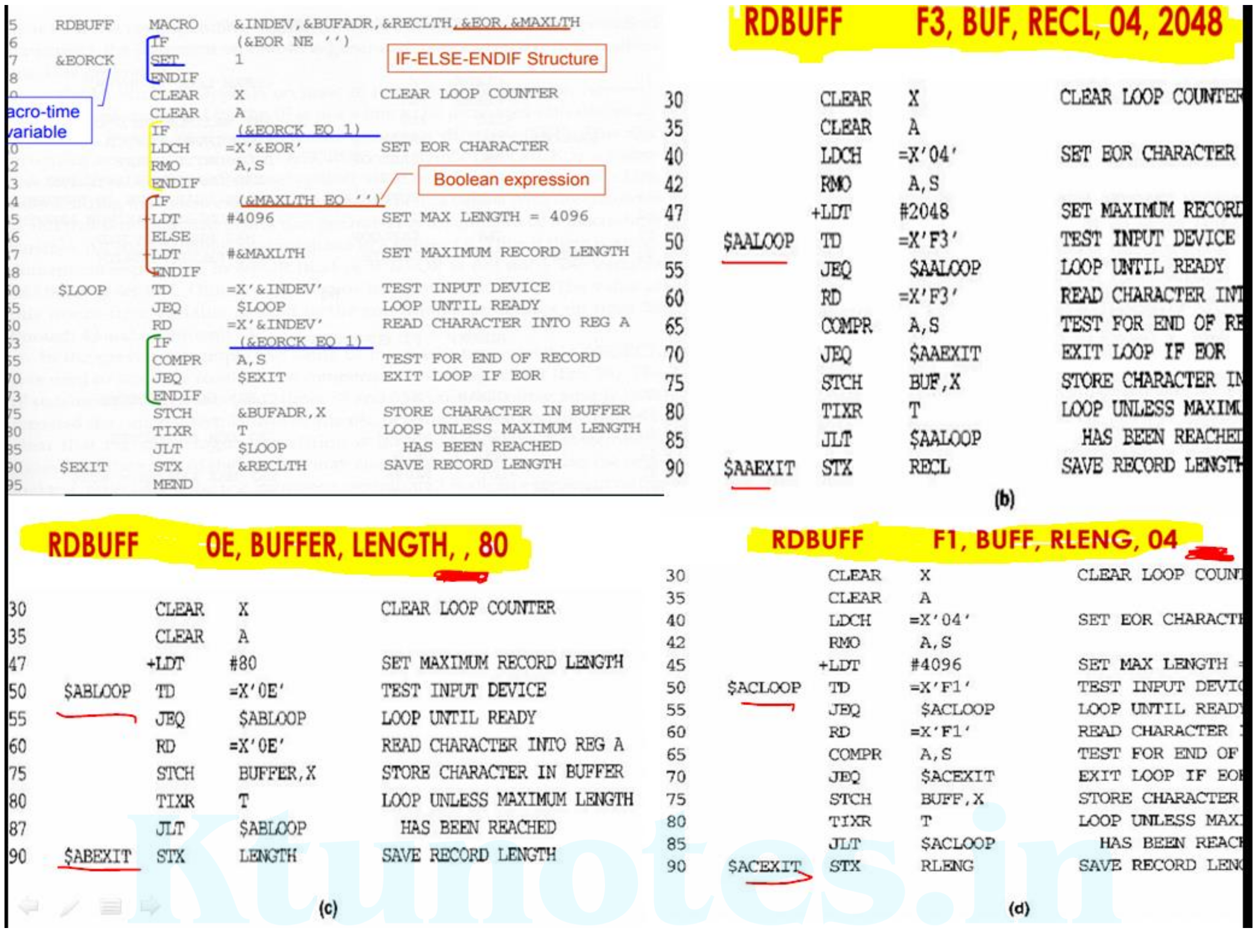


Fig 4.8(a), shows a definition of a macro RDBUFF, the logic and functions of which are similar to those previously explained. However, this definition of RDBUFF has two additional parameters:

- &EOR, which specifies a hexadecimal character code that marks the end of a record
- &MAXLTH, which specifies the maximum length record that can be read

It is possible for either or both of these parameters to be omitted in an invocation of RDBUFF. The statements from line 44 through 48, of the definition illustrate a simple macro-time conditional structure. The IF statement evaluates a Boolean expression, that is its operand. If the value of this expression is TRUE, the statements following the IF are generated until an ELSE is encountered. Otherwise, this statement are skipped, and the statement following the ELSE are generated. The ENDIF statement terminates the conditional expression that was begun by the IF statement (As usual, the ELSE clause can be omitted entirely.). Thus if the parameter &MAXLTH is equal to the null string,(ie., if the corresponding argument was omitted in the macro invocation statement), then statement on line 45 is generated. Otherwise, the statement on line 47 is generated. Similar structure appears on lines 26 through 28. In this case however, the statement controlled by the IF is not a line to be generated into the macro expansion. Instead, it is another macro processor directive (SET). This SET statement assigns the value 1 to &EORCK.

If the value of the specified Boolean expression is FALSE, the macro processor skips ahead in DEFTAB until it finds the next ELSE or ENDIF statement. The macro processor then resumes normal macro expansion. This implementation does not allow nested IF structures. It is extremely important to understand that the testing of Boolean expressions in IF statements occur at the time macros are expanded. By the time the program is assembled, all such decisions have been made. There is only one sequence of source statement (for eg: fig 4.8(C)) and the conditional macro expansion directives have been removed. Thus macro-time IF statements correspond to options that might have been selected by the programmer in writing the source code.

They are fundamentally different from statements such as COMPR (or IF statements in a high level programming language), which test data values during program execution. The same applies to the assignment of values to macro-time variables, and to the other conditional macro-expansion directives as we have discussed earlier. The macro-time IF-ELSE-ENDIF structure provides a mechanism for either generating(once) or skipping selected statements in the macro body. A different type of conditional macro expansion statement fig. 4.9

- ***Macro-time looping statement***
  - Macro processor directives:
    - *WHILE-ENDW*

- Macro processor function
  - %NITEMS: the number of members in an argument list
    - E.g. &EOR=(00,03,04)
      => %NITEMS(&EOR) is 3
    - Specify member in the list: &EOR[1]

```
25   RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH,&EOR
27   &EORCT   SET      %NITEMS(&EOR)
30            CLEAR    X              CLEAR LOOP COUNTER
35            CLEAR    A
45           +LDT      #4096          SET MAX LENGTH = 4096
50   $LOOP    TD       =X'&INDEV'     TEST INPUT DEVICE
55            JEQ      $LOOP          LOOP UNTIL READY
60            RD       =X'&INDEV'     READ CHARACTER INTO REG A
63   &CTR     SET      1
64            WHILE    (&CTR LE &EORCT)
65            COMP     =X'0000&EOR[&CTR]'
70            JEQ      $EXIT
71   &CTR     SET      &CTR+1
73            ENDW
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
100           MEND
```

Macro processor function

Macro-time looping statement

(a)

A list of end-of-record characters

```
              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
```

(b)

Fig.4.9(a) shows another definition of RDBUFF. The purpose and function of the macro are the same as before. With this definition, however the programmer can specify a list of end-of-record characters. In the macro invocation statement in fig. 4.9(b) for example, there is a list (00,03,04) corresponding to the parameter &EOR. Any one of these characters is to be interpreted as marking the end of a record. To simplify the macro definition, the parameter &MAXLTH has been deleted. The maximum record length will always be 4096. The definition in Fig. 4.9(a) uses a macro-time looping statement WHILE.

The WHILE statement specifies that the following lines, until the next ENDW statement are to be generated repeatedly, as long as a particular condition is true. As before, the testing of this condition, and the looping, are done while the macro is being expanded. The conditions to be tested involve macro—time variables and arguments, not run-time data values.

Implementation

When a WHILE statement is encountered during macro expansion, the specified Boolean expression is evaluated. If the value of this expression is FALSE, the macro processor skips ahead in DEFTAB until it finds the next ENDW statement, and then resumes normal macro expansion. If the value of the Boolean expression is TRUE, the macro processor continues to process the lines from DEFTAB in the usual way until the next ENDW statement. When the ENDW is encountered, the macro processor returns to the preceding WHILE, re-evaluates the Boolean expression, and takes action based on the new value of this expression as previously described. This method of implementation does not allow for nested WHILE structures.

Keyword Macro Parameters

In macro definitions we have used positional parameters. The parameters and arguments were associated with each other according to their positions in the macro prototype and the macro invocation statement. With positional parameters, the programmer must be careful to specify the arguments in the proper order. If an argument is to be omitted, the macro invocation statement must contain a null argument(two consecutive commas) to maintain the correct argument positions. (check fig.4.8(c)). Positional parameter are quite suitable for most macro instructions.

However, if a macro has a large number of parameters, and only few of these are given values in a typical invocation, a different form of parameter specification is more useful. Such a macro may occur in a situation in which a large and complex sequence of statements- perhaps even an entire operating system is to be generated from a macro invocation. In such a case most of the parameters may have acceptable default values. The macro invocation specifies only the changes from the default set of values. Suppose that a macro instruction GENER has 10 possible parameters, but in a particular invocation of the macro, only the third and ninth parameters are to be specified. If positional parameters were used, the macro invocation statement might look like:

```
GENER MACRO &1, &2, &type, …, &channel, &10

GENER       , , DIRECT, , , , , , 3
```

Using a different form of parameter specification, called keyword parameters, each argument value is written with a keyword that names the corresponding parameter. Arguments may appear in any order. If the third parameter in the previous example, is named &TYPE and the ninth parameter is named &CHANNEL, the macro invocation statement would be:

```
GENER       , , DIRECT, , , , , , 3
GENER       TYPE=DIRECT, CHANNEL=3
GENER       CHANNEL=3, TYPE=DIRECT
            parameter=argument
```


This statement is obviously much easier to read, and much less error prone, than positional version. Fig4.10(a) shows a version of the RDBUFF macro definition using keyword parameters.

The parameter is assumed to have this default value if its name does not appear in the macro invocation statement. Thus the default value for the parameter &INDEV is F1. There is no default value for the parameter &BUFADR. Default values can simply the macro definition in many cases: For example, the macro definitions in Fig.4.10(a) and 4.8(a) both provide for setting the maximum record length to 4096 unless a different value is specified by the user. The default value is established in fig.4.10(a) takes care of this automatically.

```
RDBUFF   MACRO    &INDEV=F1,&BUFADR=,&RECLTH=,&EOR=04,&MAXLTH=4096
         IF       (&EOR NE '')
&EORCK   SET      1
         ENDIF
         CLEAR    X               CLEAR LOOP COUNTER
         CLEAR    A
         IF       (&EORCK EQ 1)
         LDCH     =X'&EOR'        SET EOR CHARACTER
         RMO      A,S
         ENDIF
        +LDT      #&MAXLTH        SET MAXIMUM RECORD LENGTH
$LOOP    TD       =X'&INDEV'      TEST INPUT DEVICE
         JEQ      $LOOP           LOOP UNTIL READY
         RD       =X'&INDEV'      READ CHARACTER INTO 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 MAXIMUM LENGTH
         JLT      $LOOP             HAS BEEN REACHED
$EXIT    STX      &RECLTH         SAVE RECORD LENGTH
         MEND
```

(a)

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

         CLEAR    X               CLEAR LOOP COUNTER
         CLEAR    A
         LDCH     =X'04'          SET EOR CHARACTER
         RMO      A,S
        +LDT      #4096           SET MAXIMUM RECORD LENGTH
$AALOOP  TD       =X'F1'          TEST INPUT DEVICE
         JEQ      $AALOOP         LOOP UNTIL READY
         RD       =X'F1'          READ CHARACTER INTO 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
```

(b)

Figure 4.10 Use of keyword parameters in macro instructions.

```
.        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
```

(c)

Figure 4.10 (*cont'd*)

In fig4.8(a), an IF-ELSE-ENDIF structure is required to accomplish the same thing. Other part of fig 4.10 contains examples of the expansion of keyword macro invocation statements. In fig 4.10(b), all default values are accepted .In fig 4.10(c), the value of &INDEV is specified as F3, and the value of &EOR is specified as null.These values override the corresponding defaults. Note that, the arguments may appear in any order in the macro invocation statement.

Macro Processor Design Options

1. Recursive Macro Expansion
2. General Purpose Macro Processors
3. Macro Processing within language Translators

Recursive Macro Expansion

Fig 4.11(a) example of invocation of one macro by the other- nested macro invocation.


```
RDBUFF   MACRO    &BUFADR,&RECLTH,&INDEV
.
.        MACRO TO READ RECORD INTO BUFFER
.
         CLEAR    X                  CLEAR LOOP COUNTER
         CLEAR    A
         CLEAR    S
        +LDT      #4096              SET MAXIMUM 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 MAXIMUM LENGTH
         JLT      $LOOP                HAS BEEN REACHED
$EXIT    STX      &RECLTH            SAVE RECORD LENGTH
         MEND
```

RDCHAR is also a macro

26

(a)

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

(b)

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

(c)

Figure 4.11 Example of nested macro invocation.

The definition of RDBUFF in fig 4.11(a) is same as in fig 4.1. The order of the parameters has been changed. In this case, we assume that a related macro instruction (RDCHAR) already exists. The purpose of RDCHAR Fig. 4.11(b) is to read one character from a specified device into register A, taking care of the necessary test-and-wait loop. It is convenient to use a macro like RDCHAR in the definition of RDBUFF so that the programmer who is defining RDBUFF need not worry about the details of device access and control. (RDCHAR might be written at different time, or even by a different programmer). The advantages of using RDCHAR in this way would be even greater on a more complex machine, where the code to read a single character might be longer and more complicated than our simple three-line version. The algorithm we studied does not work properly if a macro invocation statement appears within the body of a macro instruction.

Suppose if the algorithm applied to the macro invocation statement in fig.4.11(c)

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

(c)

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 be begin. The processing would proceed normally until line 50, which contains a statement invoking RDCHAR. At that point, PROCESSLINE would call EXPAND again. „ This time, ARGTAB would look like:

Parameter	Value
1	F1
2	(unused)
.	.

The expansion of RDCHAR would also proceed normally.At the end of this expansion, however, a problem would appear. When the end of the definition of RDCHAR was recognized, EXPANDING would be set to FALSE. Thus the macro processor would "forget" that it had been in middle of expanding a macro when it encountered the RDCHAR statement. In addition, the argument from the original macro invocation (RDBUFF) would be lost because the values in ARGTAB were overwritten with the arguments from the invocation of RDCHAR.

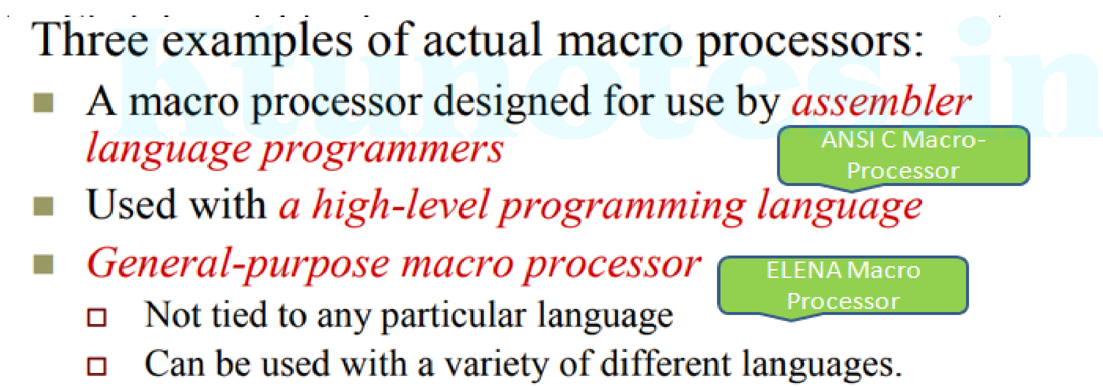
The cause of these difficulties is the recursive call of the procedure EXPAND. When the RDBUFF macro invocation is encountered, EXPAND is called. Later it calls PROCESSLINE for line 50, which results in another call to EXPAND before a return is made from the original call. A similar problem would occur with PROCESSLINE, since this procedure too would be called recursively. For ex: there might be confusion about whether the return from PROCESSLINE should be made to the main (outermost) loop of the macro processor logic or to the loop within EXPAND.

These problems are not so difficult to solve, if the programming language (such as Pascal or C) that allows the recursive calls. Then the compiler would be sure of the previous values of any variables declared within a procedure were saved when that procedure was called recursively. It would take care of other details involving return from the procedure. If the programming language supports recursion is not available, the programmer must take care of handling such items as return addresses and values of local variables. In such case, the PROCESSLINE and EXPAND would probably not be procedures at all. Instead the same logic would be incorporated into a looping structure, with data values being saved on the stack.

Solution

- Use a Stack to save ARGTAB.
- Use a counter to identify the expansion.

General Purpose Macro-Processors



The advantages of such a general-purpose approach to macro processing are:

- The programmer does not need to learn about a different macro facility for each compiler or assembler language, so much of the time and expense involved in training are eliminated
- The cost involved in producing a general purpose macro processor is greater than those for developing a language-specific processor. However this expense does not need to be repeated for each language. Overall saving in software development cost and software maintenance effort

In spite of the advantages noted, there are still relatively few general-purpose macro processors. Why?

Large number of details must be dealt with in a real programming language

- Comment identifications (//, /* */, …)
- Grouping together terms, expressions, statements (begin_end, { }, …)
- Tokens (keywords, operators)
- …

1. Large number of details must be dealt with in a real programming language

- In a special purpose macro processor, these details can be built into its logic and structure
- In the general –purpose, the user must define the specific set of rules to be followed

2. In a typical programming language, there are several situations in which normal macro parameter substitution should not occur

   Eg:. comments should usually be ignored by a macro processor

3. Another difference between programming languages is related to their facilities for grouping together terms, expressions, or statements –

   – E.g. Some languages use keywords such as begin and end for grouping statements. Others use special characters such as { and }.

4. A more general problem involves the tokens of the programming language

– E.g. identifiers, constants, operators, and keywords

– E.g. blanks

5. Another potential problem with general purpose macro processors involves the syntax used for macro definitions and invocation statements. With most special-purpose macro processors, macro invocations are very similar in form to statements in the source programming language.

Macro Processing within Language Translators

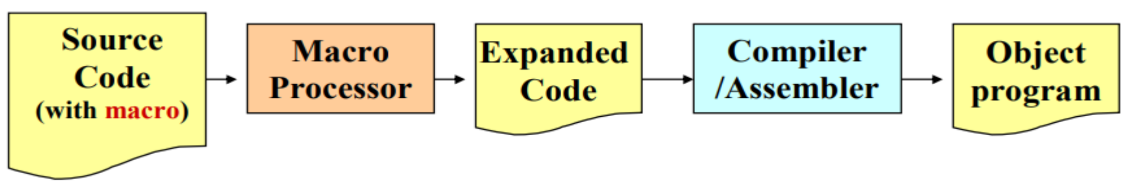
- Preprocessors
  - They process macro definition and expand macro invocations, producing an expanded version of the source program
  - This expanded program is then used as input to an assembler or compiler
- Combining the macro processing function with the language translator itself
- Achieved using Line –by –line macro processor
  - The macro processor reads the source program statements
  - Process the statement
  - The output lines are passed to the language translator as they are generated, instead of being written to an expanded source file
  - Thus macro processor operates as a sort of input routine for the assembler or compiler
- Advantages of line-by line macro processor:

- It avoids 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.
  - i.e., the source statement error can be quickly identified without need to backtrack the source

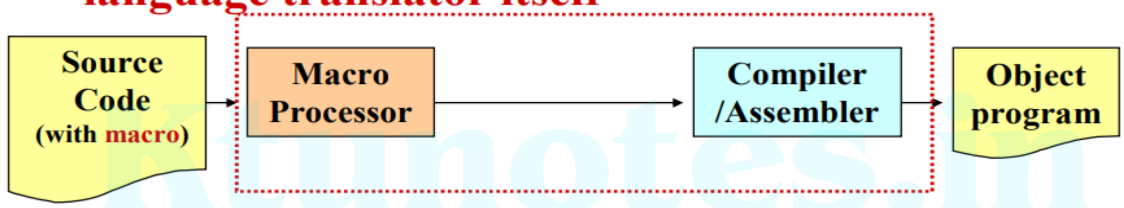
- Line –by-line macro processor may use some of the same utility routines as the language translator, the functions of macro processing and program translation are relatively independent

- The main form of communication between the two functions is the passing of source statements from one to the other
- It is possible to have even closer cooperation between the macro processor and the assembler or compiler
- Such a scheme can be thought of as a language translator with an integrated macro processor

- **Preprocessors**



- **Combining macro processing functions with the language translator itself**



Integrated Macro Processor

- Integrate a macro processor with a language translator (e.g., compiler)
- Advantages:

- An integrated macro processor can potentially make use of any information about the source program that is extracted by the language translator.
- An integrated macro processor can support macro instructions that depend upon the context in which they occur.
- Since the Macro Processor may recognize the meaning of source language

Drawbacks of Line-by-line or Integrated Macro Processor

- They must be specially designed and written to work with a particular implementation of an assembler or compiler.
- The costs of macro processor development are added to the costs of the language translator, which results in a more expensive software.
- The assembler or compiler will be considerably larger and more complex than it would be if a macro preprocessor were used
- The size may be a problem if the translator is to run on a computer with limited memory
- In any case, the additional complexity will add to the overhead of language translation
- Decision about what type of macro processor to use should be based on considerations such as the frequency and complexity of macro processing that is anticipated, and other characteristics of computing environment