Module 5 (Macro Preprocessor, Device Driver, Text Editor and Debuggers)

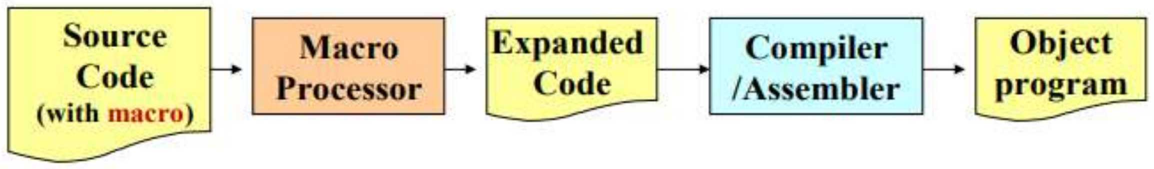
Macro Preprocessor - Macro Instruction Definition and Expansion - One pass Macro processor Algorithm and data structures, Machine Independent Macro Processor Features, Macro processor design options

Device drivers: Anatomy of a device driver, Character and block device drivers, General design of device drivers

Text Editors: Overview of Editing, User Interface, Editor Structure.

Debuggers: Debugging Functions and Capabilities, Relationship with other parts of the system, Debugging Methods- By Induction, Deduction and Backtracking

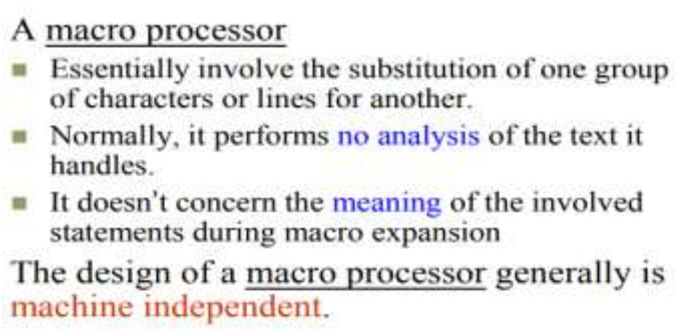
Overview



Macro Instruction Definition and Expansion

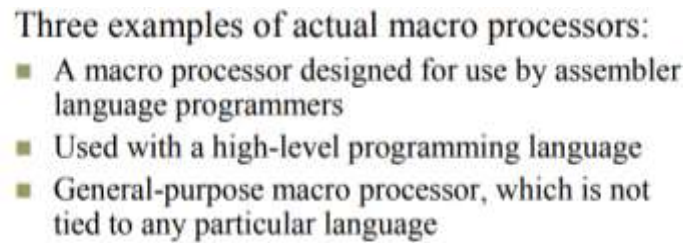
A macro instruction (abbreviated to macro) is simply a notational convenience for the programmer. A macro represents a commonly used group of statements in the source programming language.

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.


A macro processor

- Essentially involve the substitution of one group of characters or lines for another.
- Normally, it performs no analysis of the text it handles.
- It doesn't concern the meaning of the involved statements during macro expansion

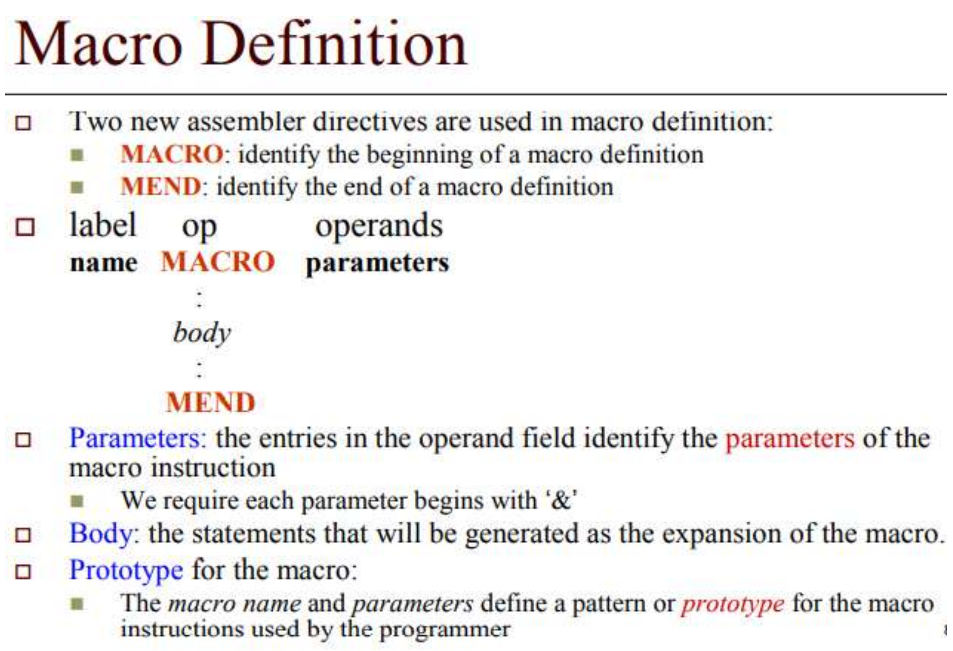
The design of a macro processor generally is machine independent.


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

```
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

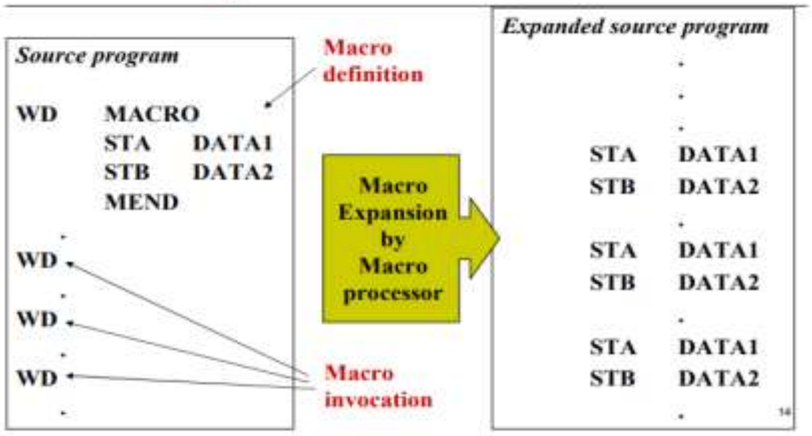
Macro Invocation

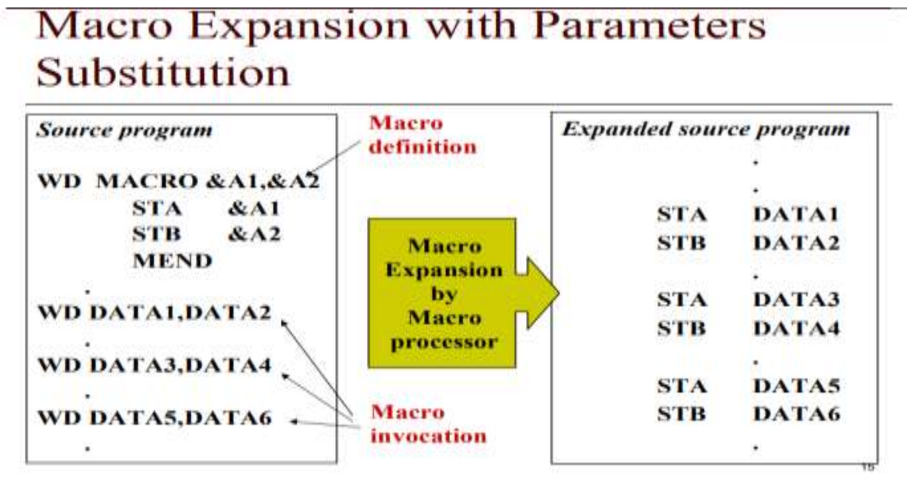
- A ***macro invocation statement*** (a ***macro call***) gives the **name** of the macro instruction being invoked and the **arguments** in expanding the macro.
- Macro Invocation vs. Subroutine Call.
  - Statements of the macro body are expanded each time the macro is invoked.
  - Statements of the 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





Macro Expansion with Parameters Substitution

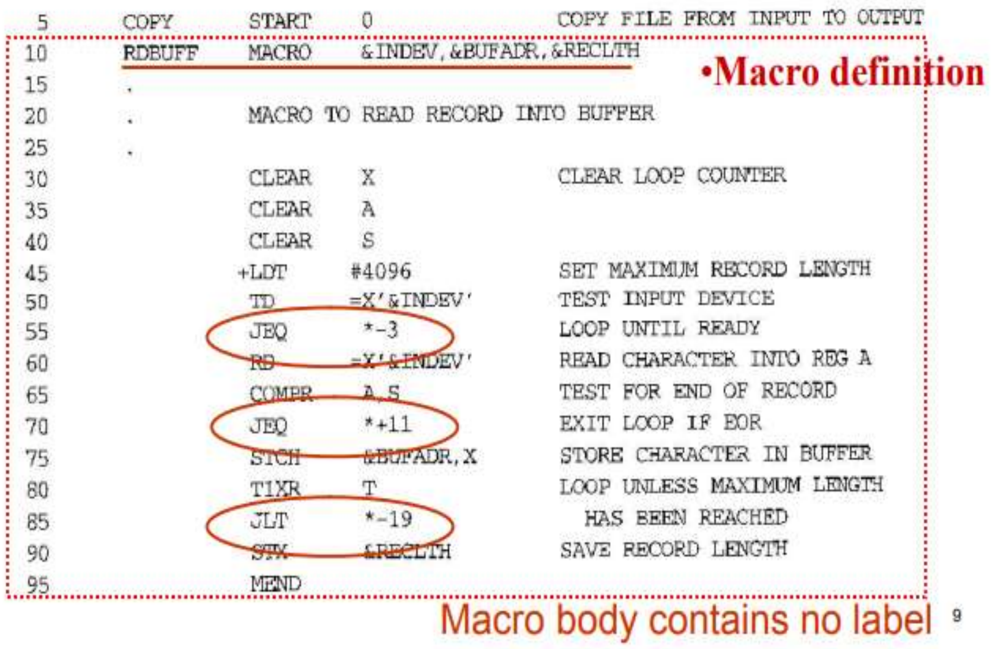
Source program

```
WD  MACRO &A1,&A2
        STA     &A1
        STB     &A2
        MEND
  .
WD DATA1,DATA2
  .
WD DATA3,DATA4
  .
WD DATA5,DATA6
  .
```

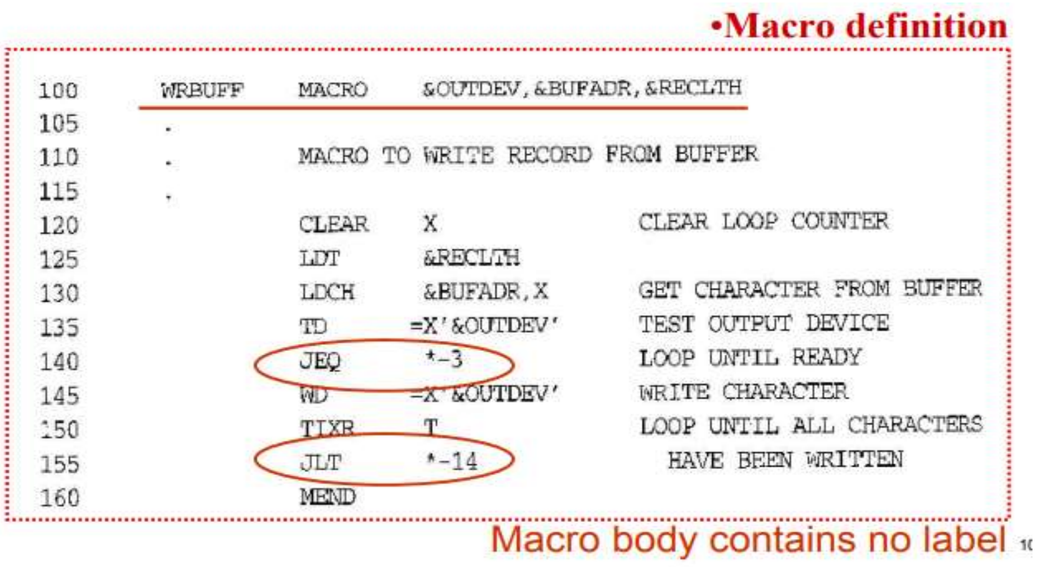
Expanded source program

```
          .
          .
STA     DATA1
STB     DATA2
          .
STA     DATA3
STB     DATA4
          .
STA     DATA5
STB     DATA6
          .
```

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



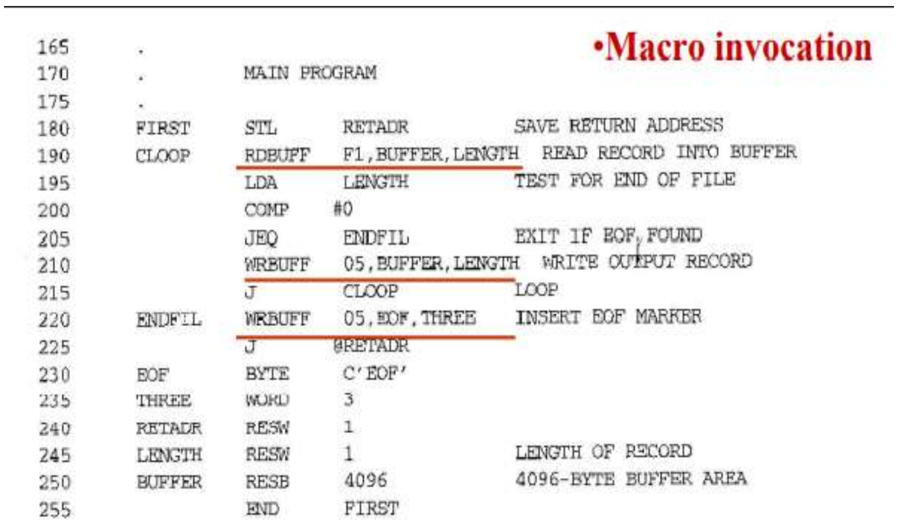
```
COPY      START   0                  COPY FILE FROM INPUT TO OUTPUT
RDBUFF    MACRO   &INDEV,&BUFADR,&RECLTH
.
.         MACRO TO READ RECORD INTO BUFFER
.
          CLEAR   X                  CLEAR LOOP COUNTER
          CLEAR   A
          CLEAR   S
         +LDT     #4096              SET MAXIMUM 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
                                       HAS BEEN REACHED
          JLT     *-19
          STX     &RECLTH            SAVE RECORD LENGTH
          MEND
```



•Macro definition

```
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
```

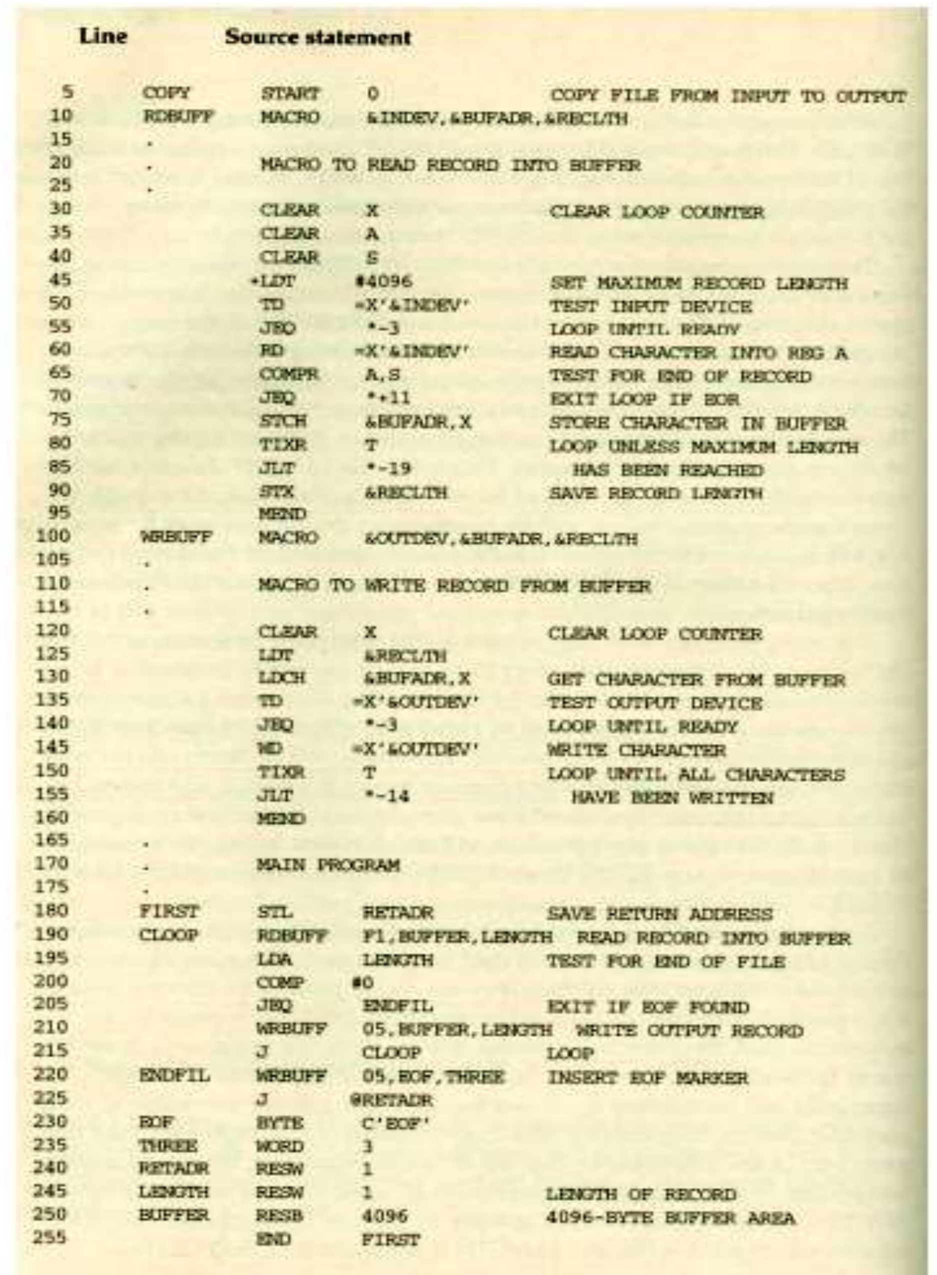
Macro body contains no label



•Macro invocation

```
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

     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
```

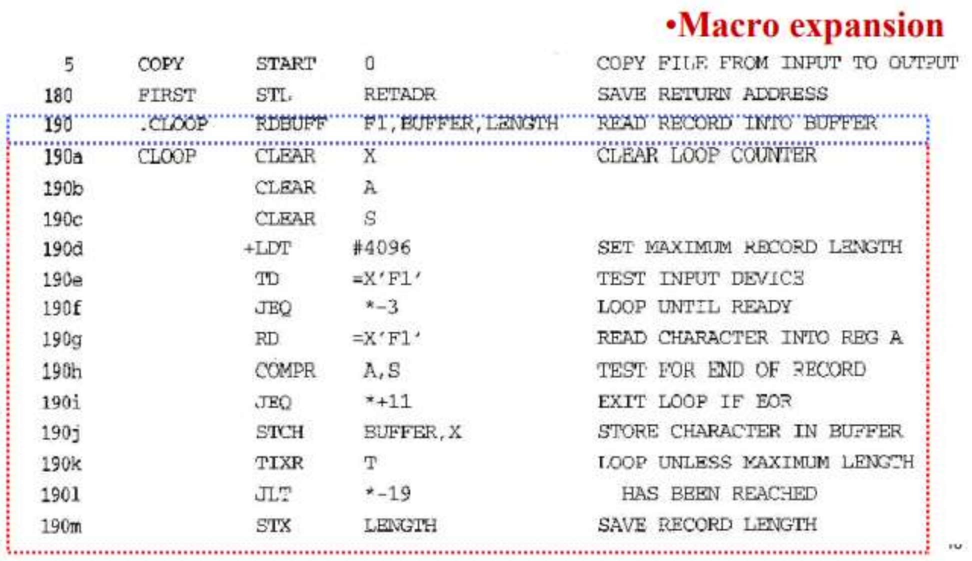
Figure 4.1 Use of macros in a SIC/XE program.

- 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

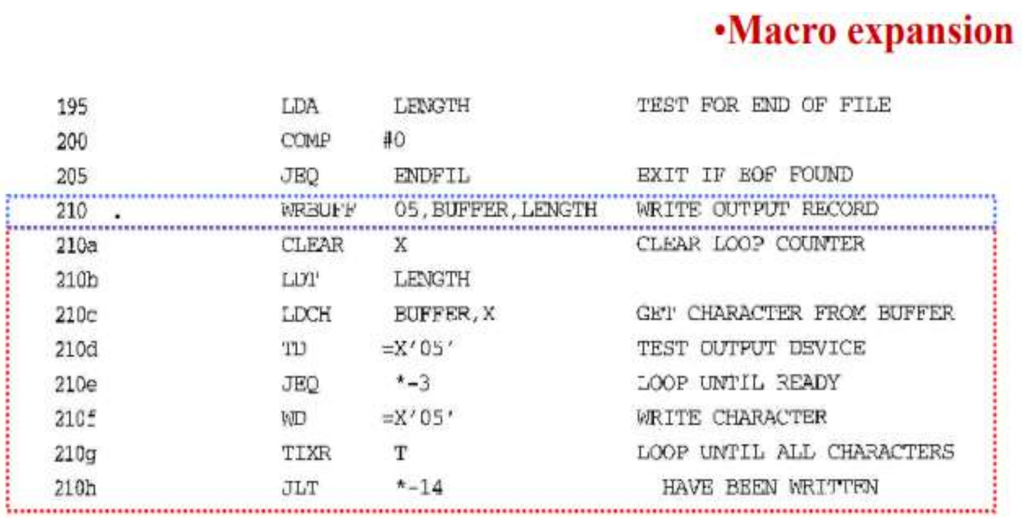
2. MEND

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 arguments 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

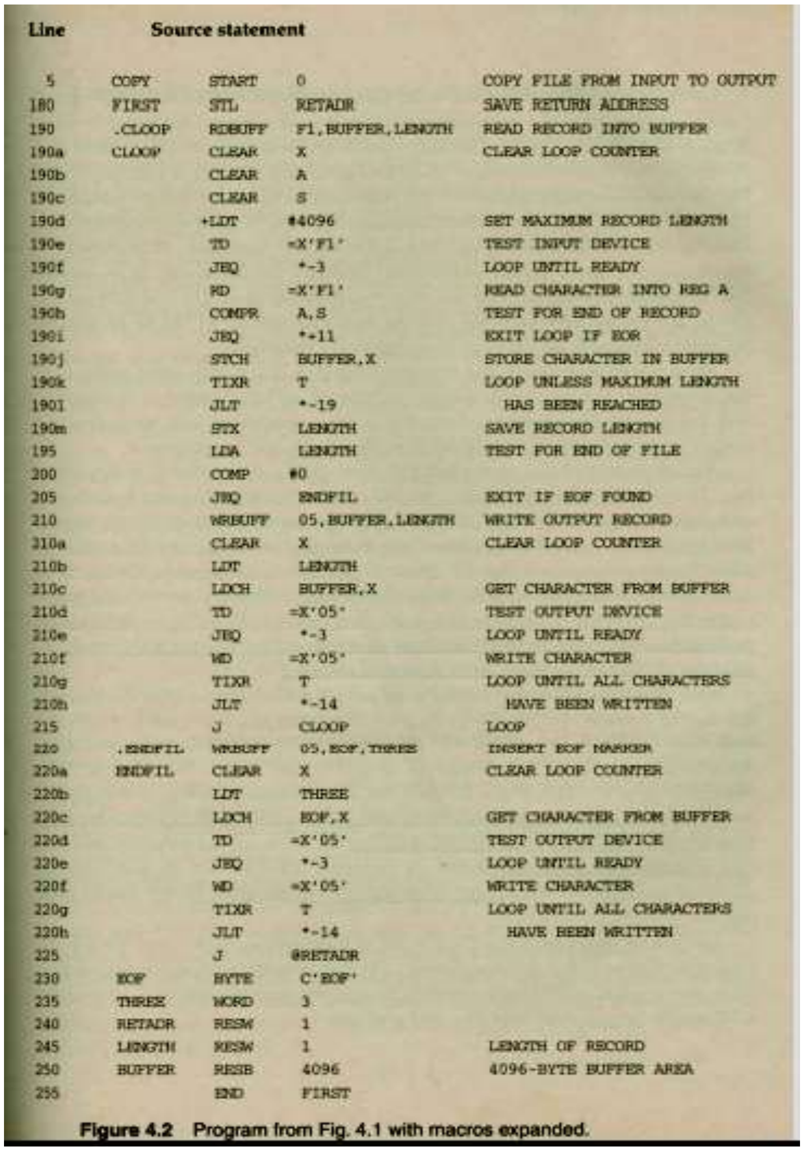
```
  5    COPY     START   0                  COPY FILE FROM INPUT TO OUTPUT
180    FIRST    STL     RETADR             SAVE RETURN ADDRESS
190    .CLOOP   RDBUFF  F1,BUFFER,LENGTH   READ RECORD INTO BUFFER
190a   CLOOP    CLEAR   X                  CLEAR LOOP COUNTER
190b            CLEAR   A
190c            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**.

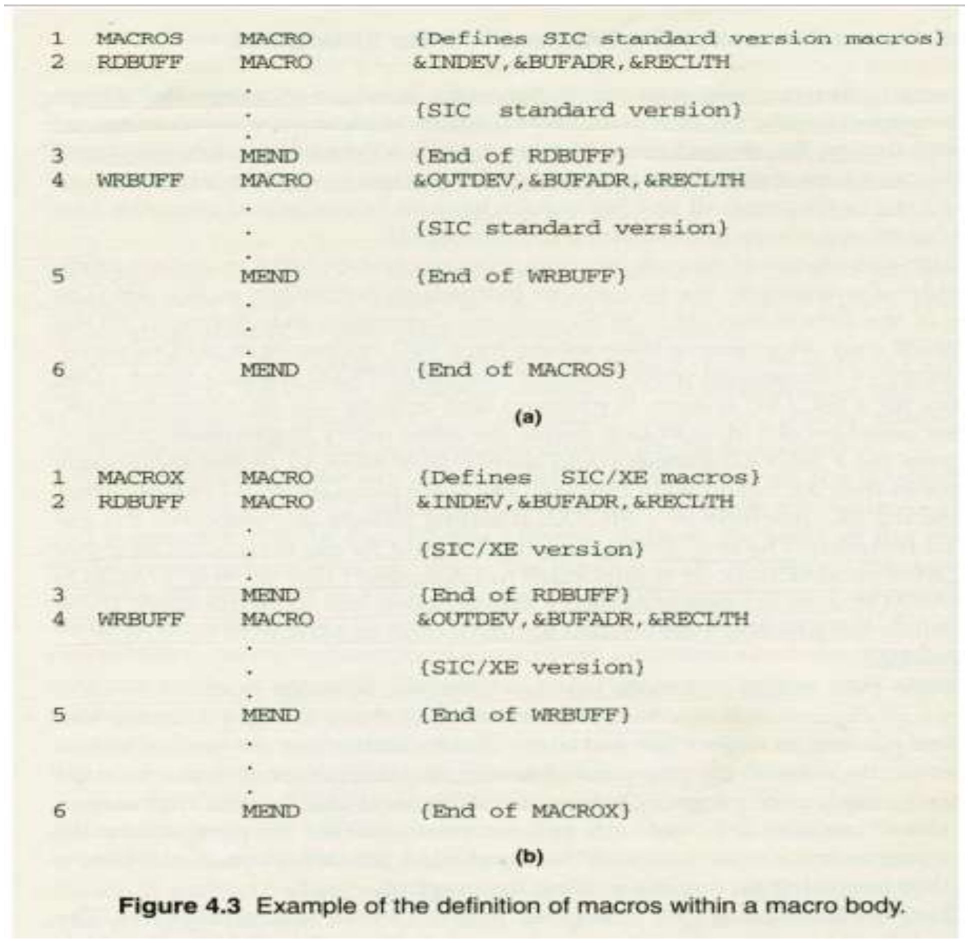
In fig 4.1 line 140 contains the statement, JEQ *-3 and line 155 contains JLT*-14. The corresponding statements in the WRREC subroutine are JEQ WLOOP and JLT WLOOP where WLOOP is a label on the TD instruction that tests the output device. If such a label appeared on line 135 of the macro body, it would 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**

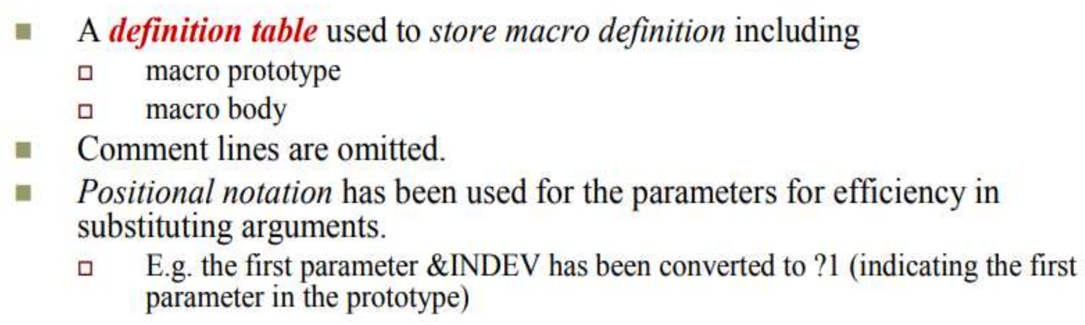
- The definition of a macro must appear in the source program before any statements that invoke that macro
- This restriction does not create any real inconvenience.

Data Structures

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

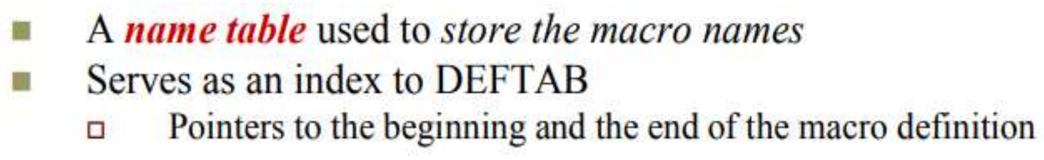
1. DEFTAB
2. NAMTAB
3. ARGTAB

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

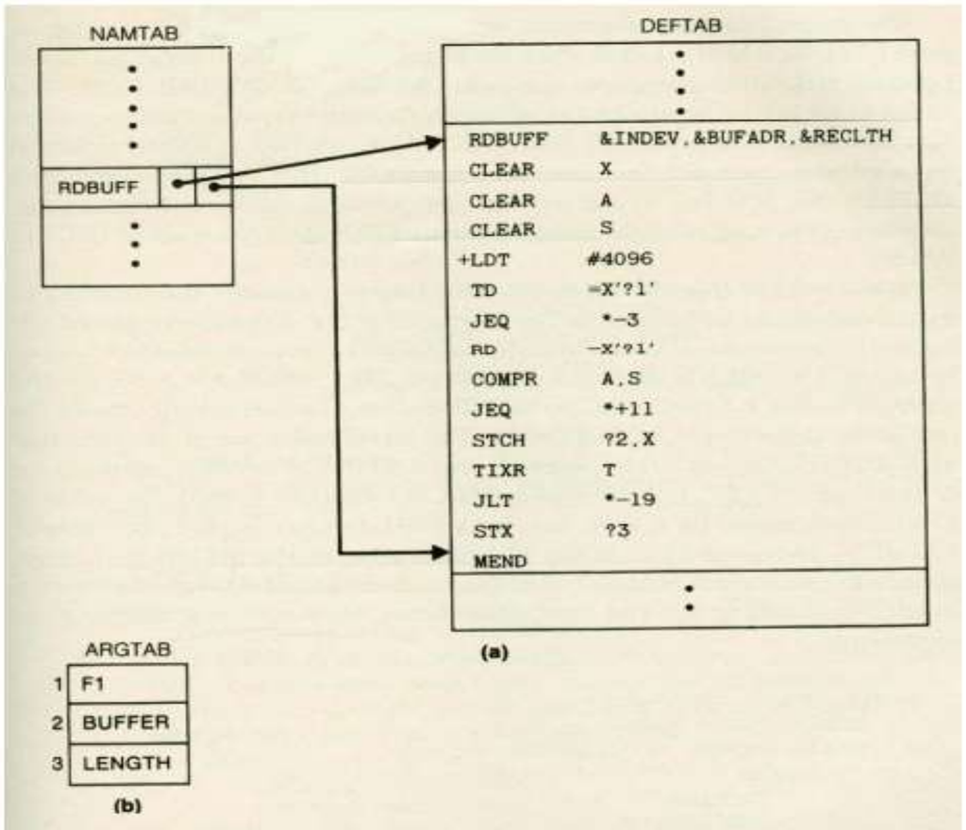
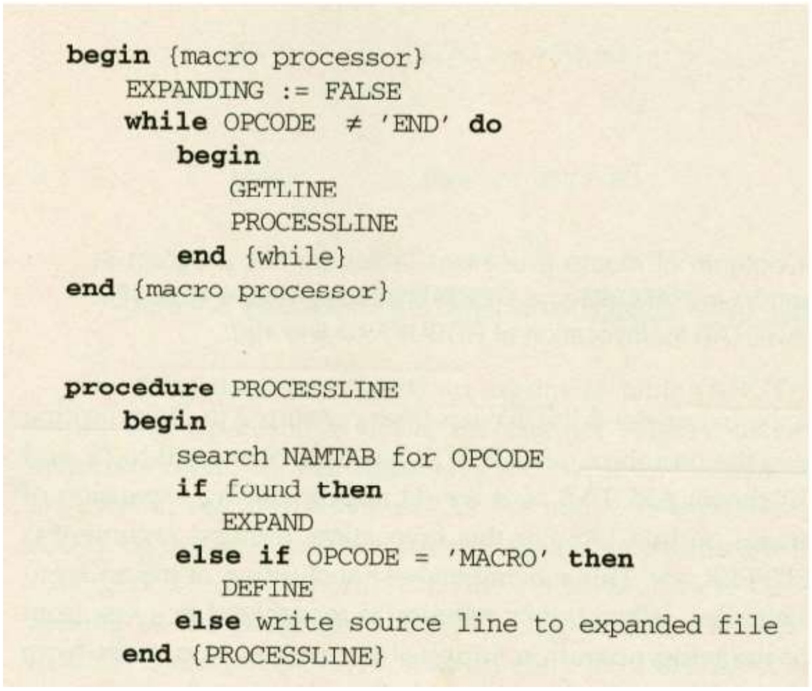


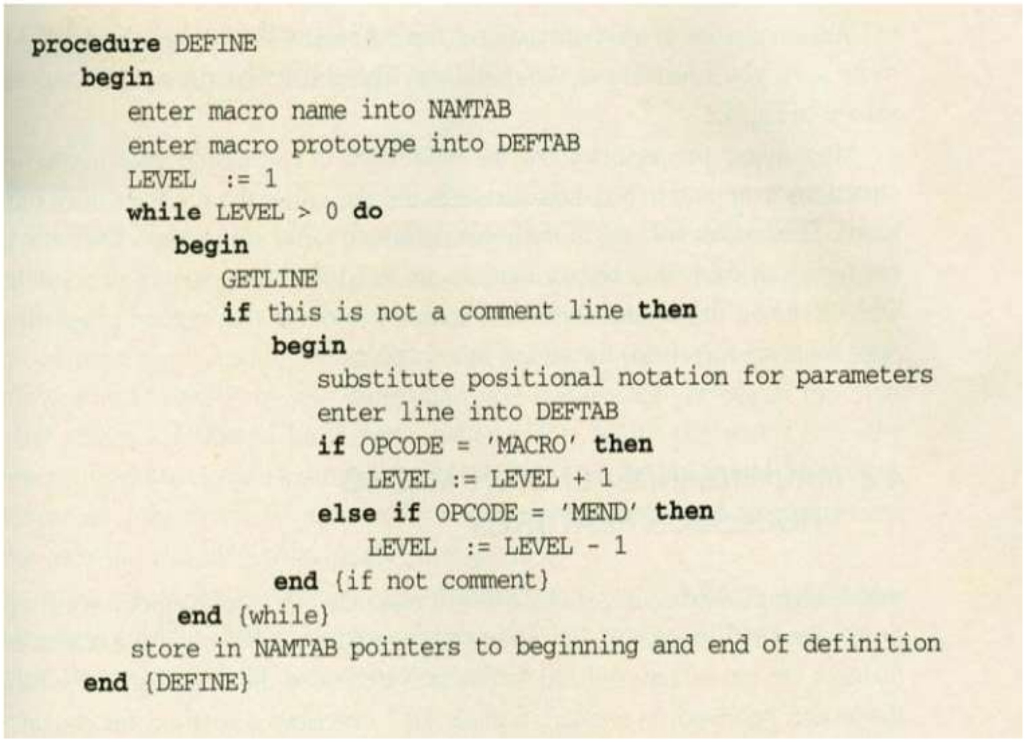
Figure 4.4 Contents of macro processor tables for the program in Fig. 4.1: (a) entries in NAMTAB and DEFTAB defining macro RDBUFF, (b) entries in ARGTAB for invocation of RDBUFF on line 190.

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.

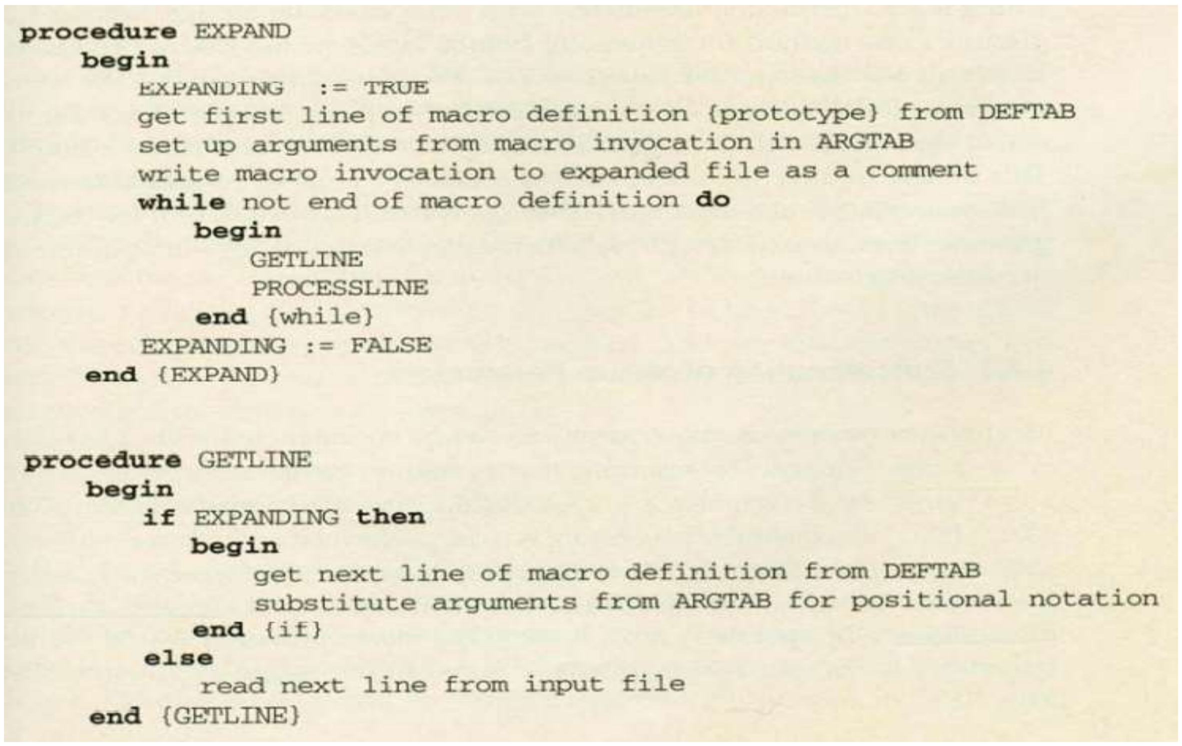
Algorithm



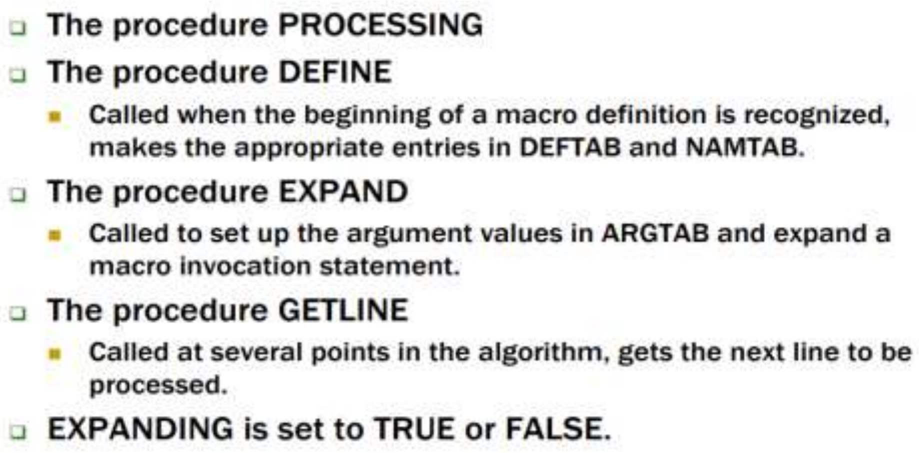
```
begin {macro processor}
    EXPANDING := FALSE
    while OPCODE ≠ 'END' do
        begin
            GETLINE
            PROCESSLINE
        end {while}
end {macro processor}

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.

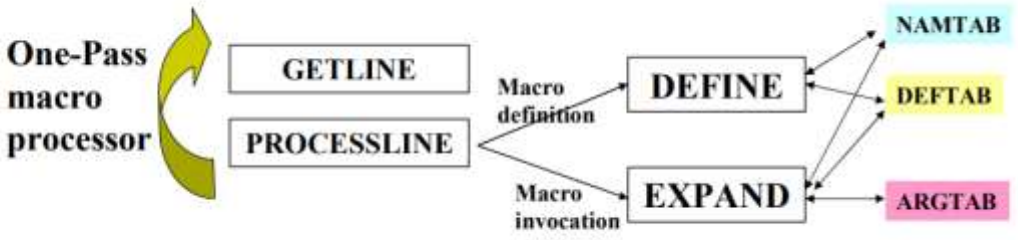
The procedure DEFINE, which is called when the beginning of a macro definition is recognized, makes the appropriate entries in DEFTAB and NAMTAB. EXPAND is called to set up the argument values in ARGTAB and expand a macro invocation statement. The procedure GETLINE, which is called at several points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

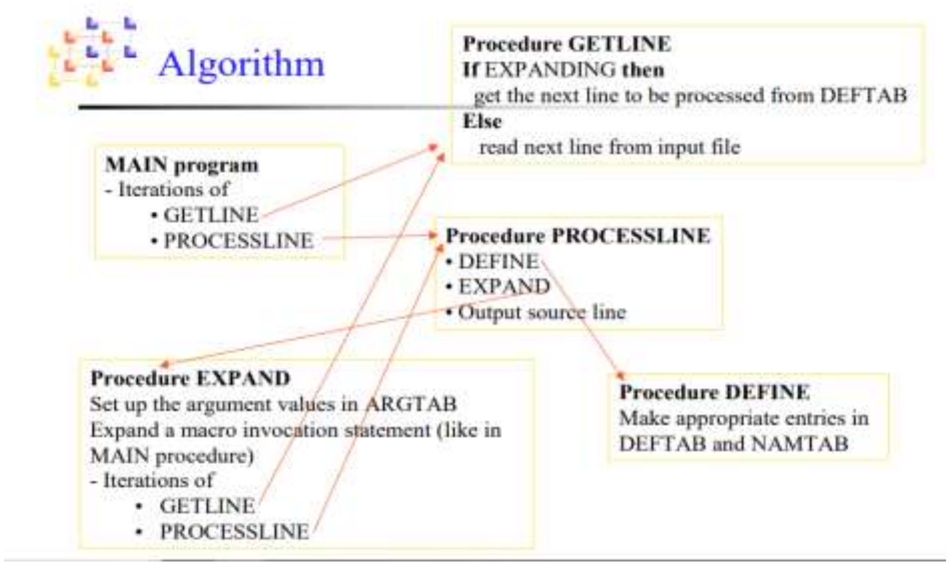
This line may come from DEFTAB (the next line of a macro being expanded), or from the input file, depending upon whether the Boolean variable EXPANDING is set to TRUE or FALSE.

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.**

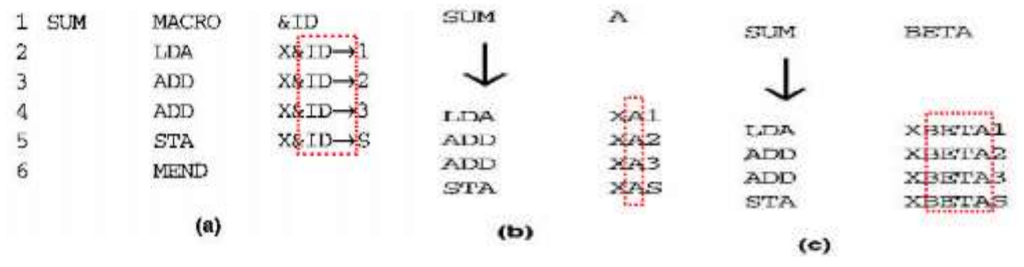
In SIC macro language, this operator is the character **→**.

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.



```
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
```

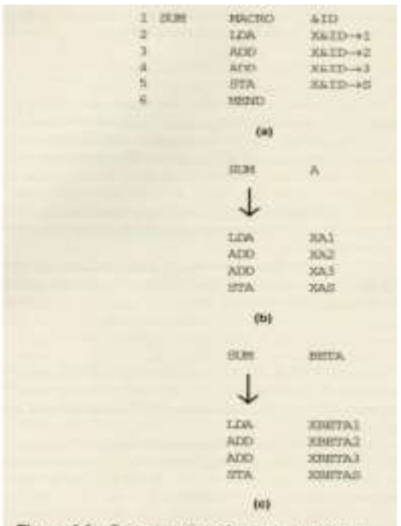
(a)

```
SUM        A
 ↓
LDA        XA1
ADD        XA2
ADD        XA3
STA        XAS
```

(b)

```
SUM        BETA
 ↓
LDA        XBETA1
ADD        XBETA2
ADD        XBETA3
STA        XBETAS
```

(c)



```
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
```

(a)

```
SUM    A
  ↓
LDA    XA1
ADD    XA2
ADD    XA3
STA    XAS
```

(b)

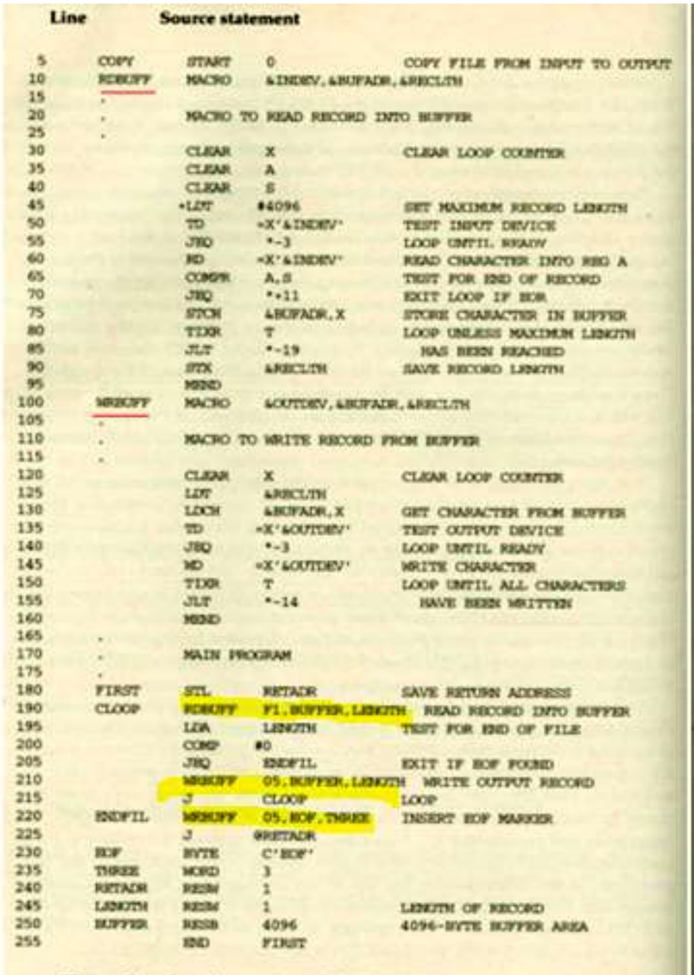
```
SUM    BETA
  ↓
LDA    XBETA1
ADD    XBETA2
ADD    XBETA3
STA    XBETAS
```

(c)

Figure 4.6 Concatenation of macro parameters.

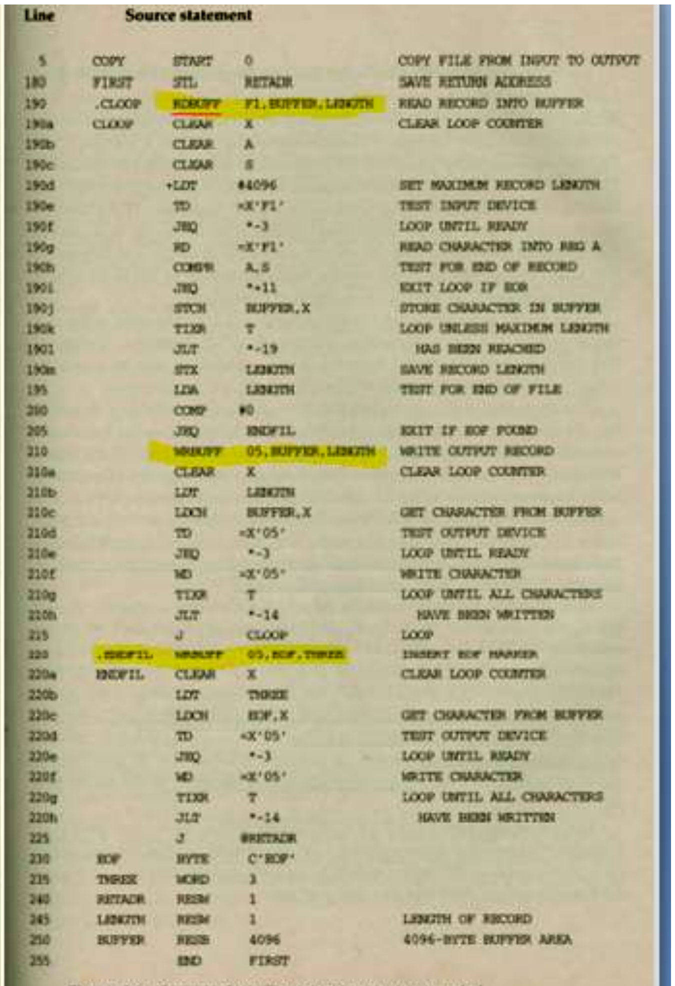
- In fig.(a) shows a macro definition that uses the concatenation operator as previously described
- In fig (b) and(c) shows macro invocation statements and the corresponding macro expansions

- 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



```
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.

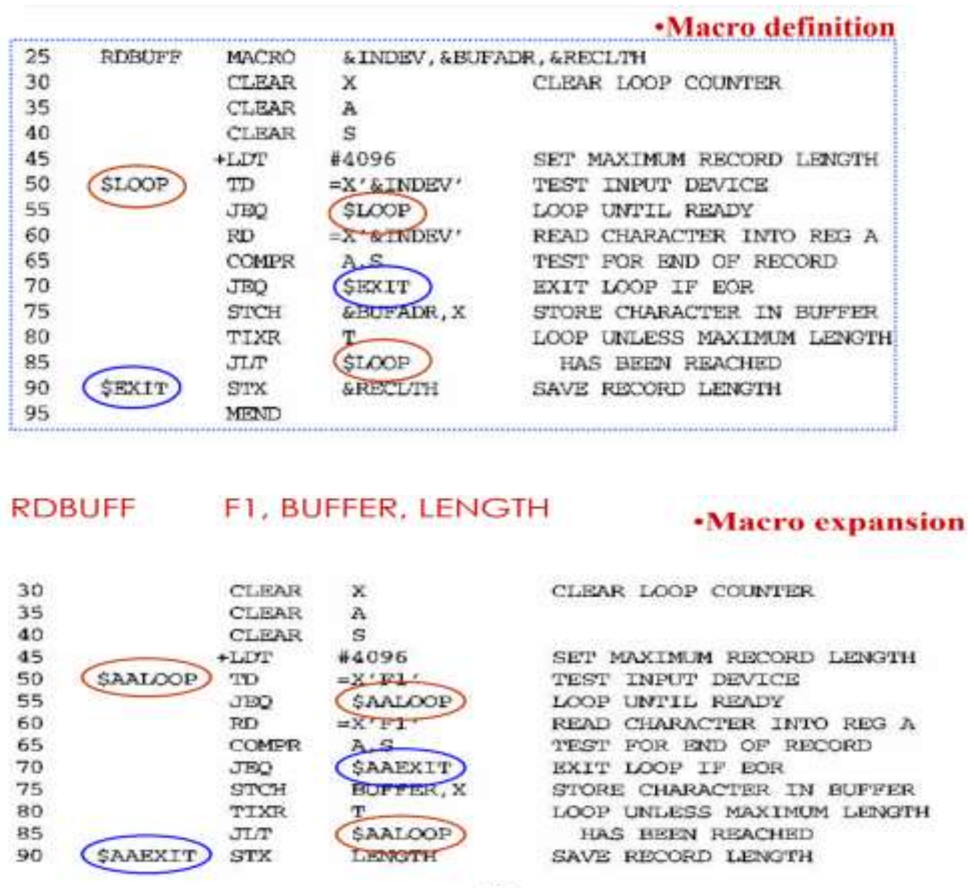


```
Line  Source statement

  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.

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
•Macro expansion

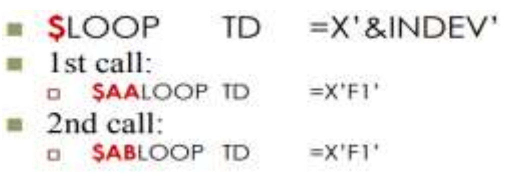
```
RDBUFF   MACRO   &INDEV,&BUFADR,&RECLTH
         CLEAR   X                  CLEAR LOOP COUNTER
         CLEAR   A
         CLEAR   S
        +LDT     #4096              SET MAXIMUM RECORD LENGTH
$LOOP    TD      =X'&INDEV'         TEST INPUT DEVICE
         JEQ     $LOOP              LOOP UNTIL READY
         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
```

```
         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.

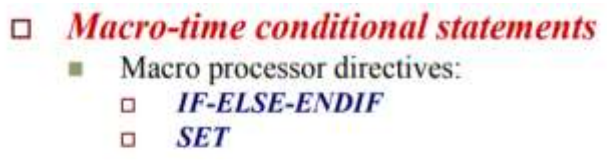


- $LOOP TD =X'&INDEV'
- 1st call:
  - $AALOOP TD =X'F1'
- 2nd call:
  - $ABLOOP TD =X'F1'

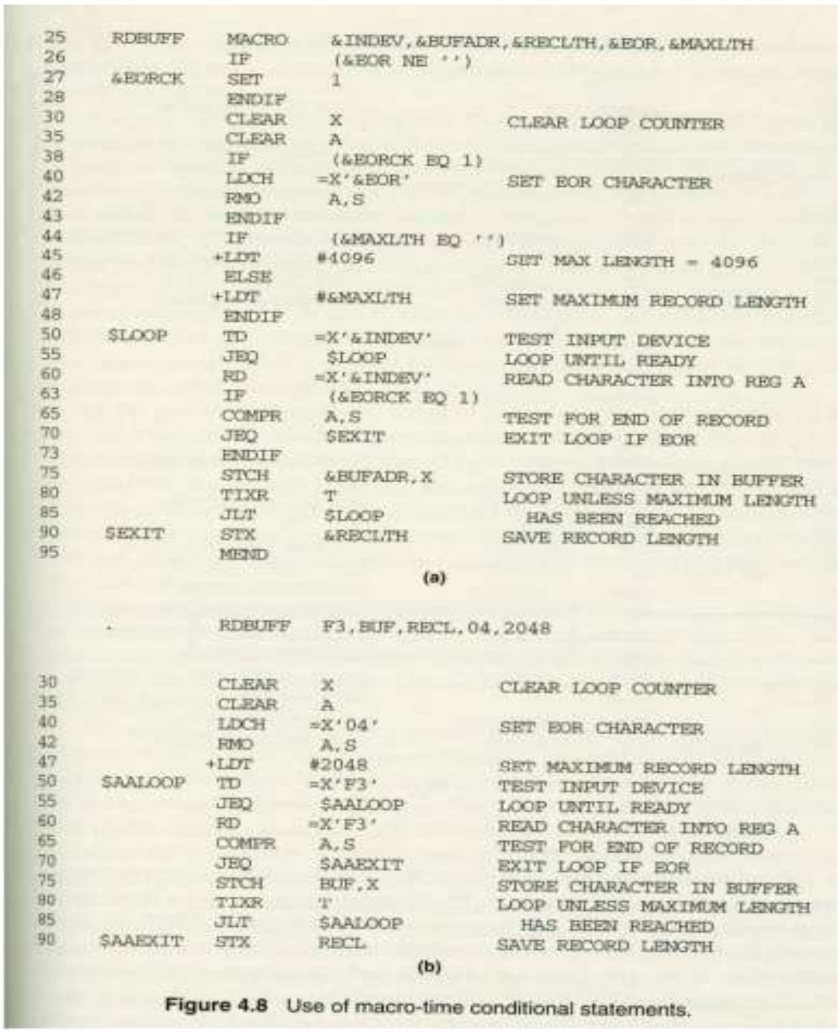
For the succeeding macro expansion, xx will be set to AB, AC etc. *If only alphabetic and numeric characters are allowed in xx, such a two-character counter provides for as many as 1296 macro expansion in a single program.* This results in the generation of unique labels for each expansion of a macro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

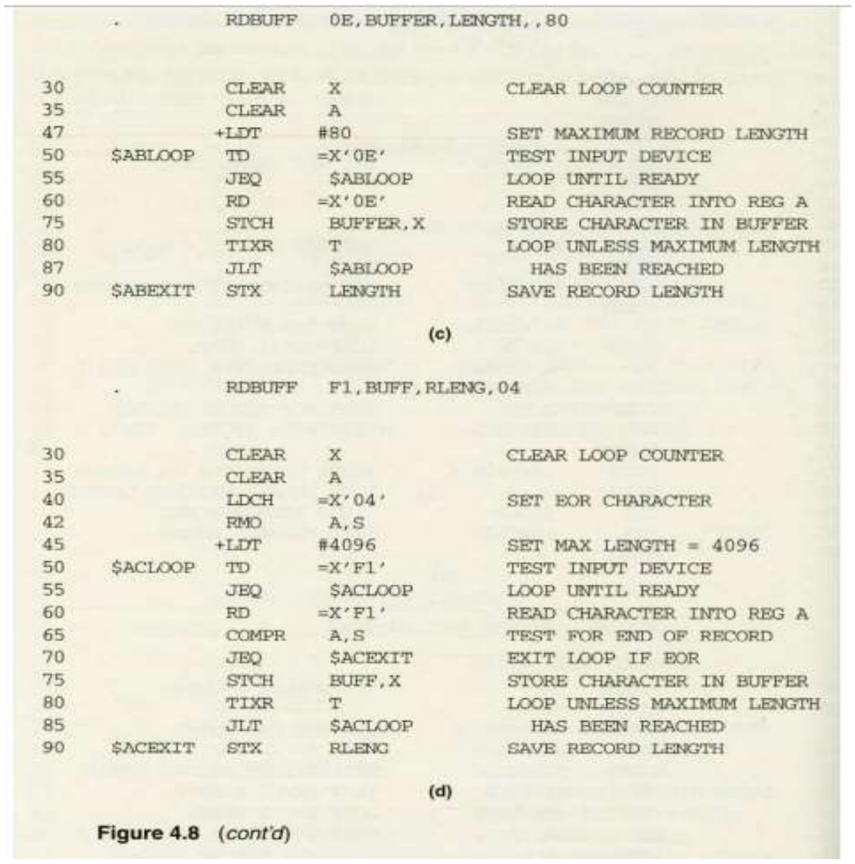
(a)

```
.        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



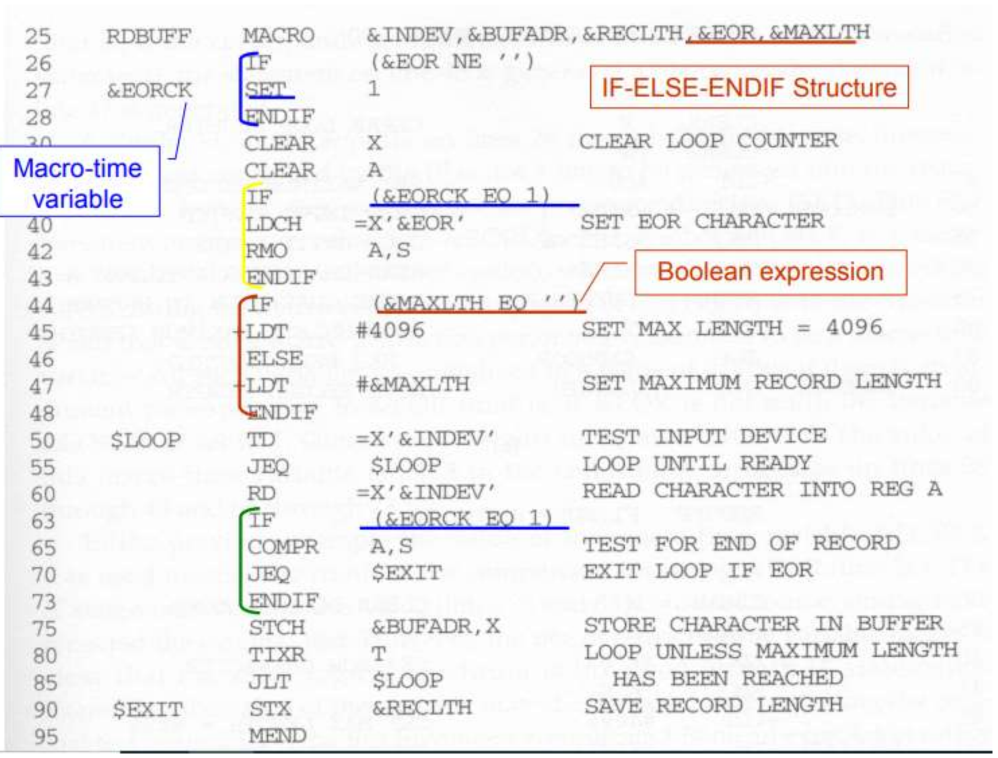
```
   .          RDBUFF    0E,BUFFER,LENGTH,,80

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)

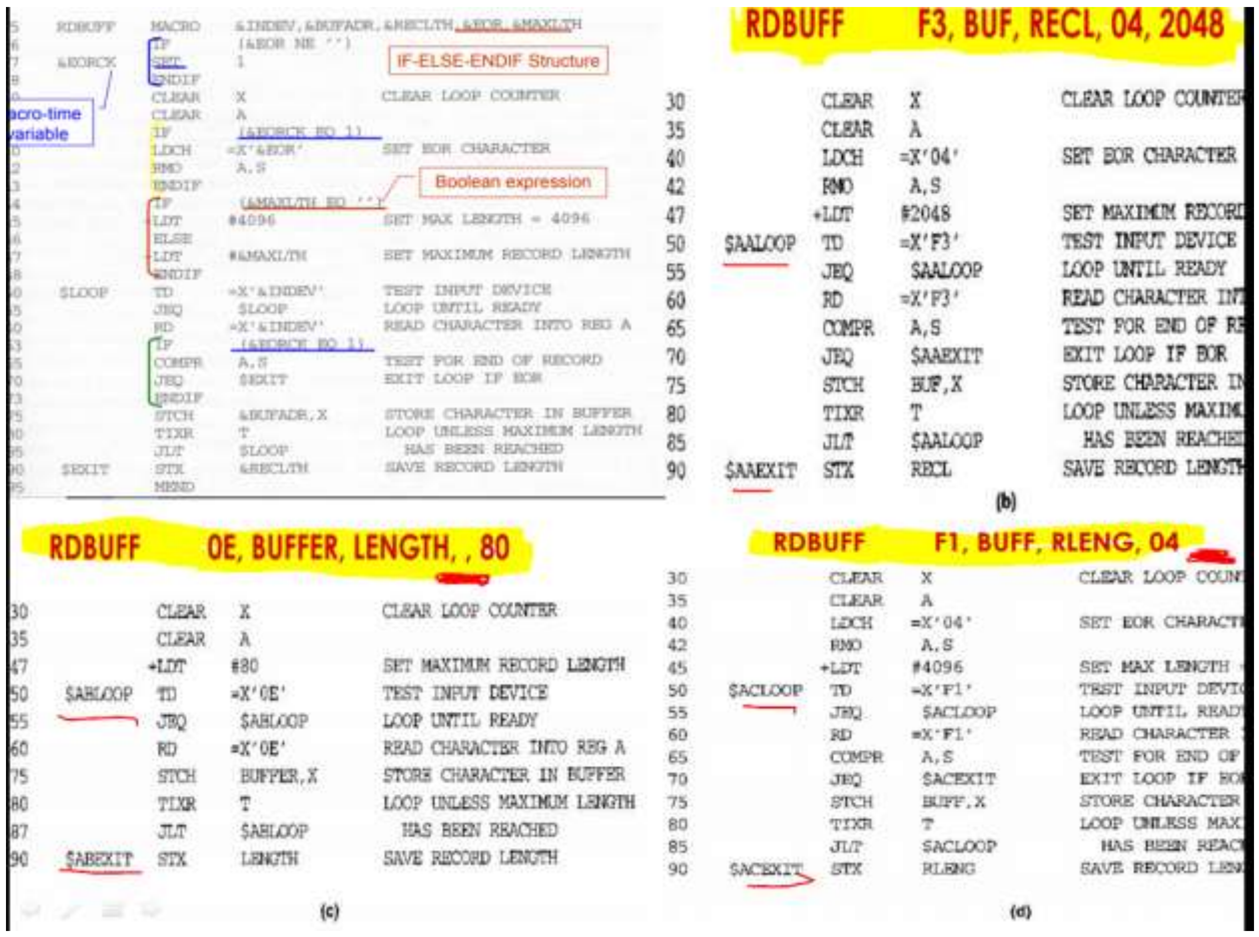
```
   .          RDBUFF    F1,BUFF,RLENG,04

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)*


IF-ELSE-ENDIF Structure
Macro-time variable
Boolean expression

```
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
```



```
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
```

RDBUFF F3, 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
50   $AALOOP   TD      =X'F3'      TEST INPUT DEVICE
55             JEQ     $AALOOP     LOOP UNTIL READY
60             RD      =X'F3'      READ CHARACTER IN
65             COMPR   A,S         TEST FOR END OF RE
70             JEQ     $AAEXIT     EXIT LOOP IF EOR
75             STCH    BUF,X       STORE CHARACTER IN
80             TIXR    T           LOOP UNLESS MAXIM
85             JLT     $AALOOP       HAS BEEN REACHED
90   $AAEXIT   STX     RECL        SAVE RECORD LENGTH
```

(b)

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 COUN
35             CLEAR   A
40             LDCH    =X'04'      SET EOR CHARACT
42             RMO     A,S
45            +LDT     #4096       SET MAX LENGTH
50   $ACLOOP   TD      =X'F1'      TEST INPUT DEVIC
55             JEQ     $ACLOOP     LOOP UNTIL READY
60             RD      =X'F1'      READ CHARACTER
65             COMPR   A,S         TEST FOR END OF
70             JEQ     $ACEXIT     EXIT LOOP IF EO
75             STCH    BUFF,X      STORE CHARACTER
80             TIXR    T           LOOP UNLESS MAX
85             JLT     $ACLOOP       HAS BEEN REACH
90   $ACEXIT   STX     RLENG       SAVE RECORD LEN
```

(d)

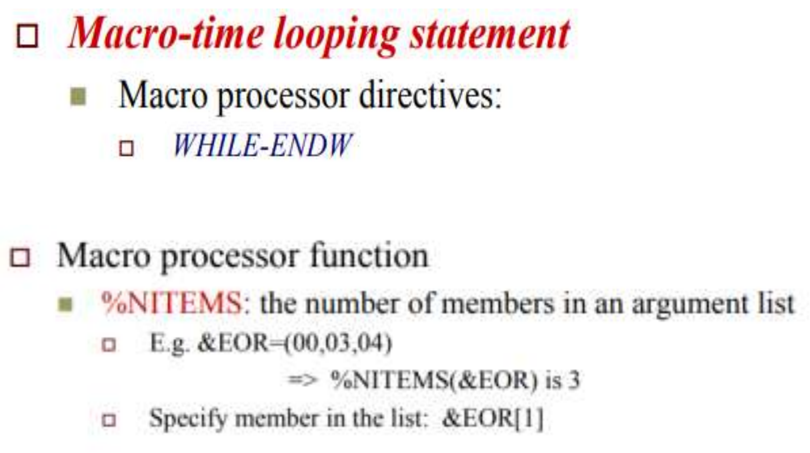
Fig 4.8(a), shows a definition of a macro RDBUFF, the logic and functions of which are similar to those previously explained. However, this definition of RDBUFF has two additional parameters:

- &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.

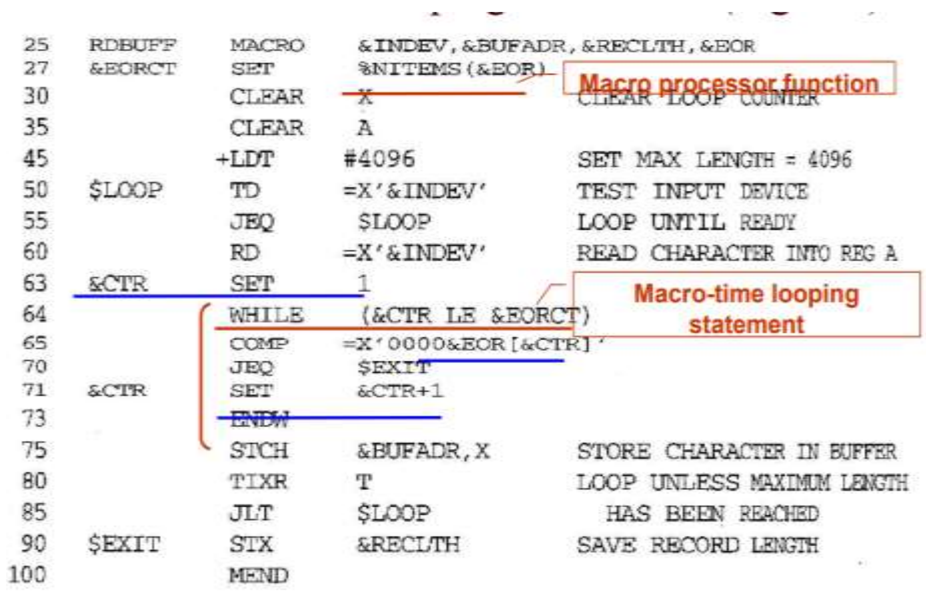
If the value of the specified Boolean expression is FALSE, the macro processor skips ahead in DEFTAB until it finds the next ELSE or ENDIF statement. The macro processor then resumes normal macro expansion. This implementation does not allow nested IF structures. It is extremely important to understand 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



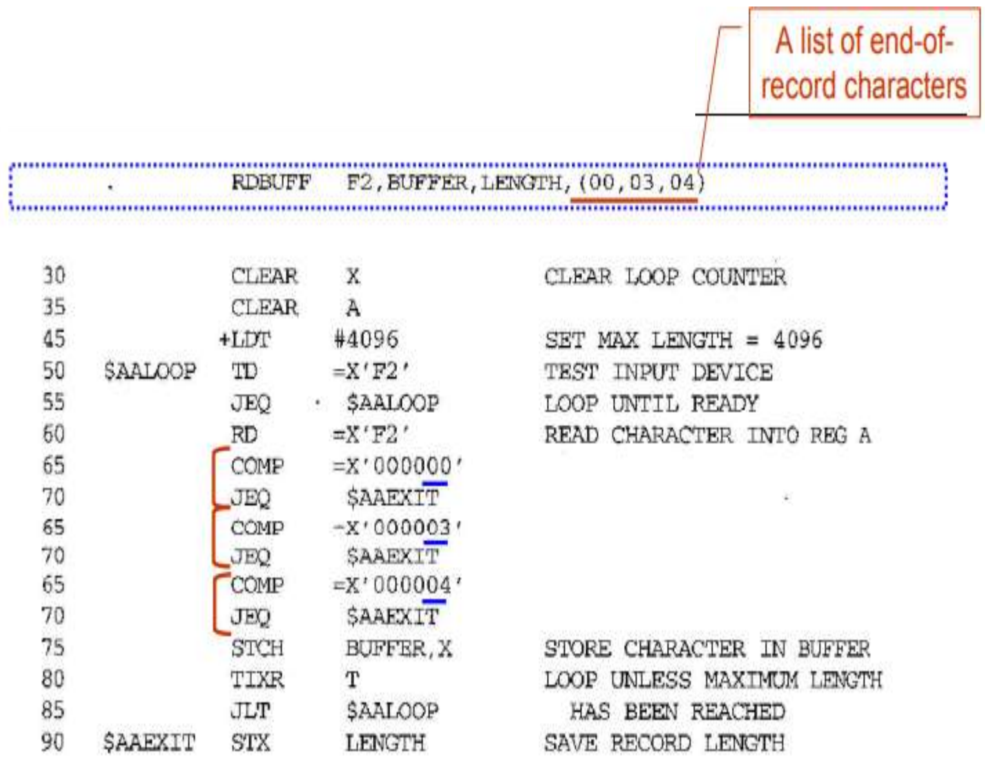
- ***Macro-time looping statement***
  - Macro processor directives:
    - *WHILE-ENDW*

- 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
```

(a)



```
               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.

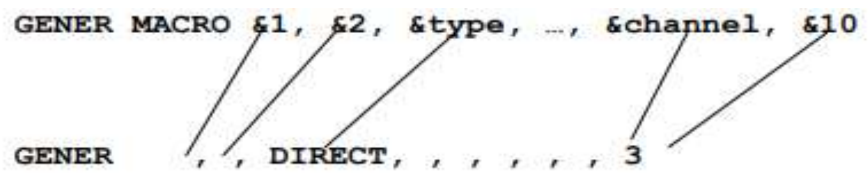
Implementation

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

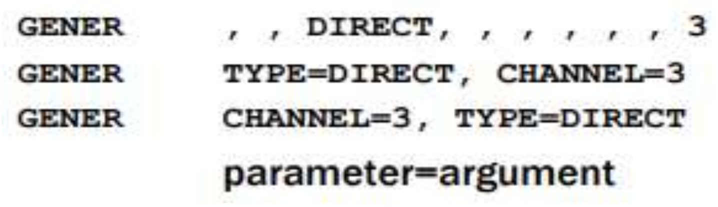
In macro definitions we have used positional parameters. The parameters and arguments were associated with each other according to their positions in the macro prototype and the macro invocation statement. With positional parameters, the programmer must be careful to specify the arguments in the proper 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
```

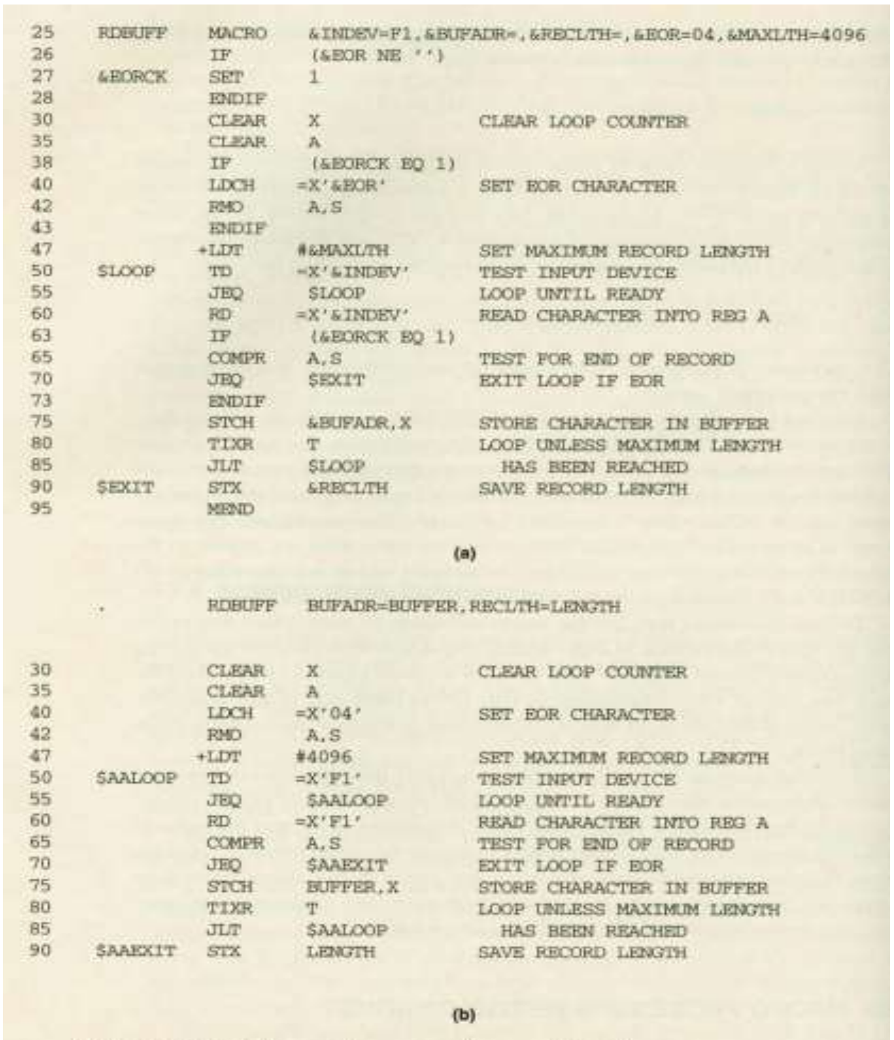
Using a different form of parameter specification, called keyword parameters, each argument value is written with a keyword that names the corresponding parameter. Arguments may appear in any order. If the third parameter in the previous example, is named &TYPE and the ninth parameter is named &CHANNEL,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



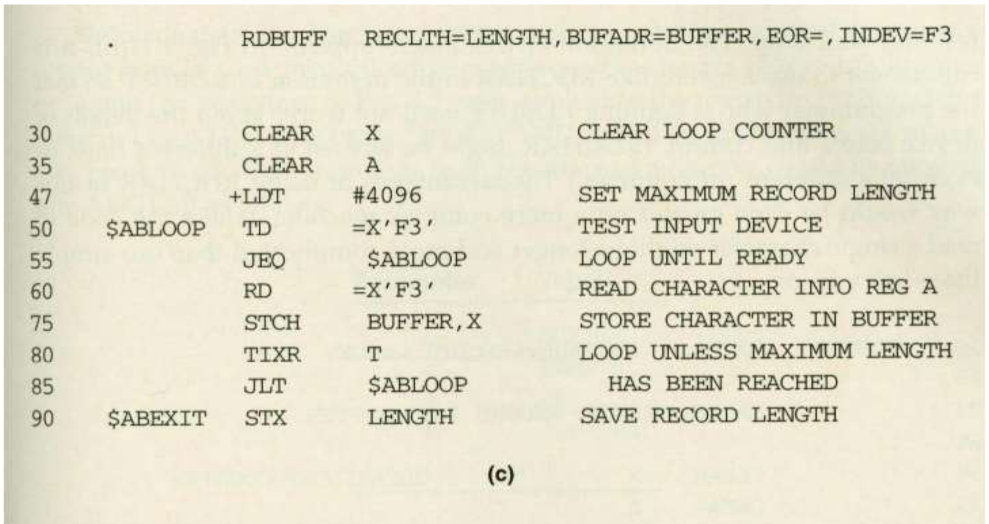
```
RDBUFF   MACRO    &INDEV=F1,&BUFADR=,&RECLTH=,&EOR=04,&MAXLTH=4096
         IF       (&EOR NE '')
&EORCK   SET      1
         ENDIF
         CLEAR    X                 CLEAR LOOP COUNTER
         CLEAR    A
         IF       (&EORCK EQ 1)
         LDCH     =X'&EOR'          SET EOR CHARACTER
         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)*

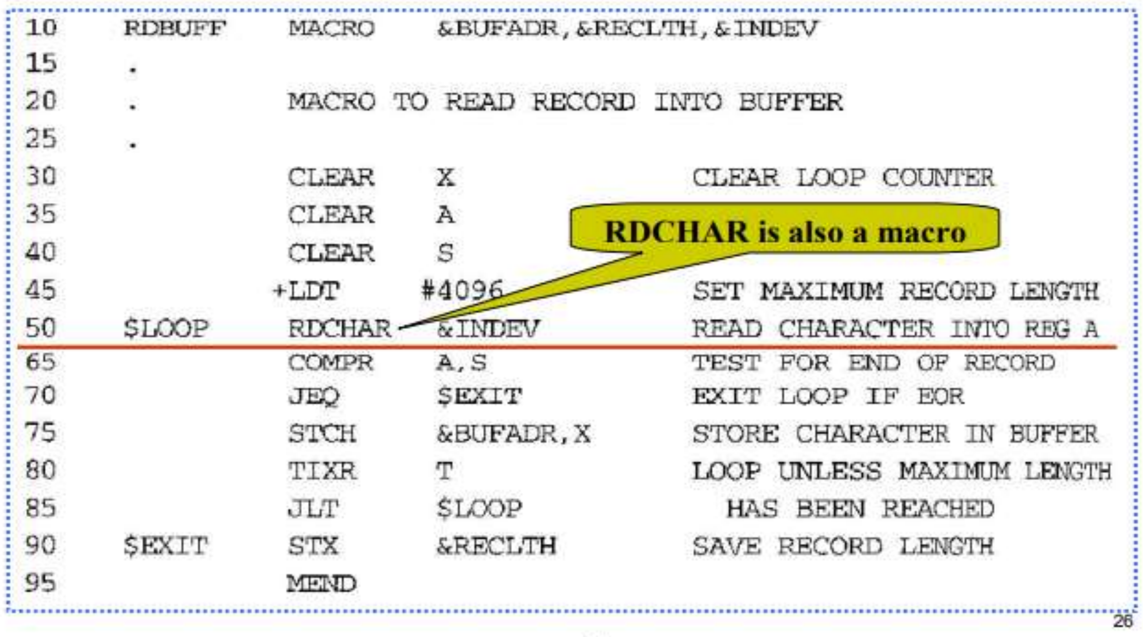
In fig4.8(a), an IF-ELSE-ENDIF structure is required to accomplish the same thing. Other part of fig 4.10 contains examples of the expansion of keyword macro invocation statements. In fig 4.10(b), all default values are accepted .In fig 4.10(c), the value of &INDEV is specified as F3, and the value of &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

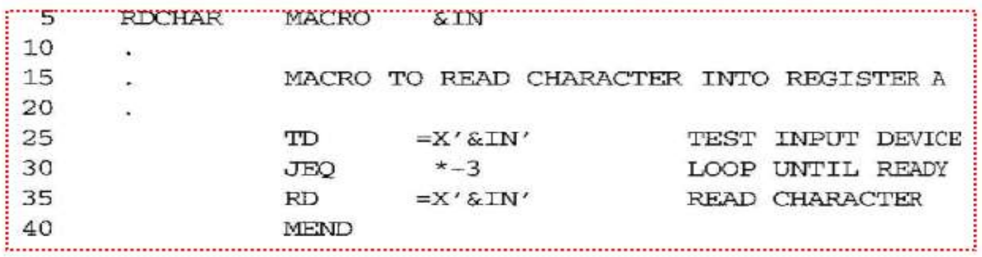
Recursive Macro Expansion

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
```

(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)

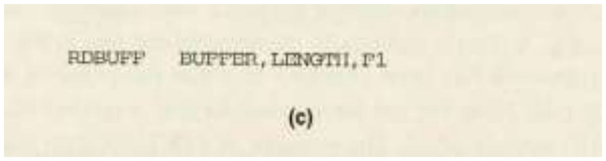
```
RDBUFF   BUFFER,LENGTH,F1
```

(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.

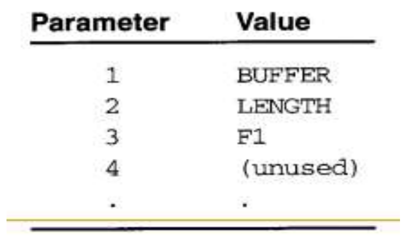
Suppose if the algorithm applied to the macro invocation statement in fig.4.11(c)



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

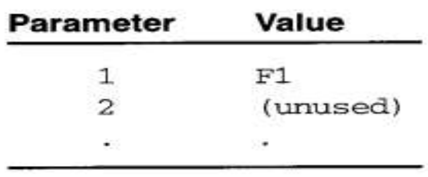
(c)

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.

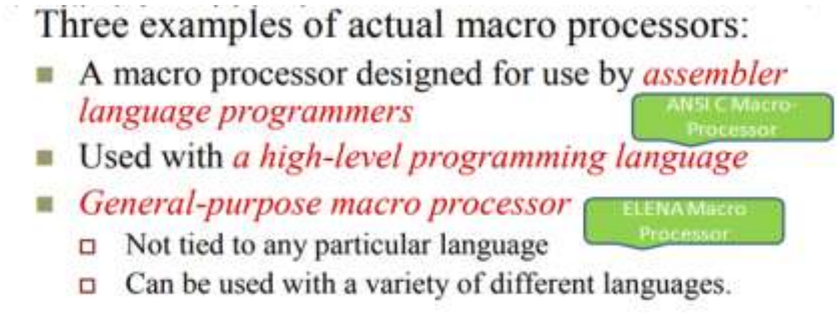
The cause of these difficulties is the recursive call of the procedure EXPAND. When the RDBUFF macro invocation is encountered, EXPAND is called. Later it calls PROCESSLINE for line 50, which results in another call to EXPAND before a return is made from the original call. A similar problem would occur 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


ANSI C Macro-Processor
ELENA Macro Processor

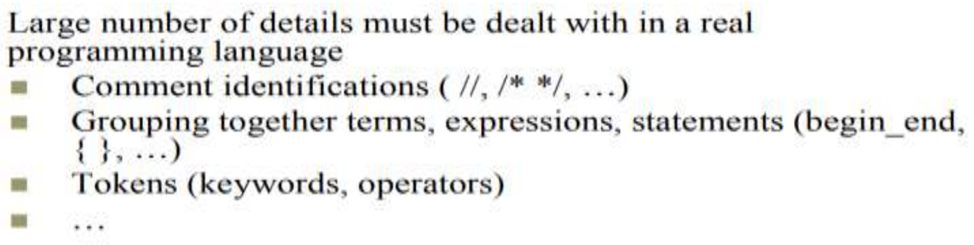
Three examples of actual macro processors:

- A macro processor designed for use by *assembler language programmers*
- Used with *a high-level programming language*
- *General-purpose macro processor*
  - Not tied to any particular language
  - Can be used with a variety of different languages.

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

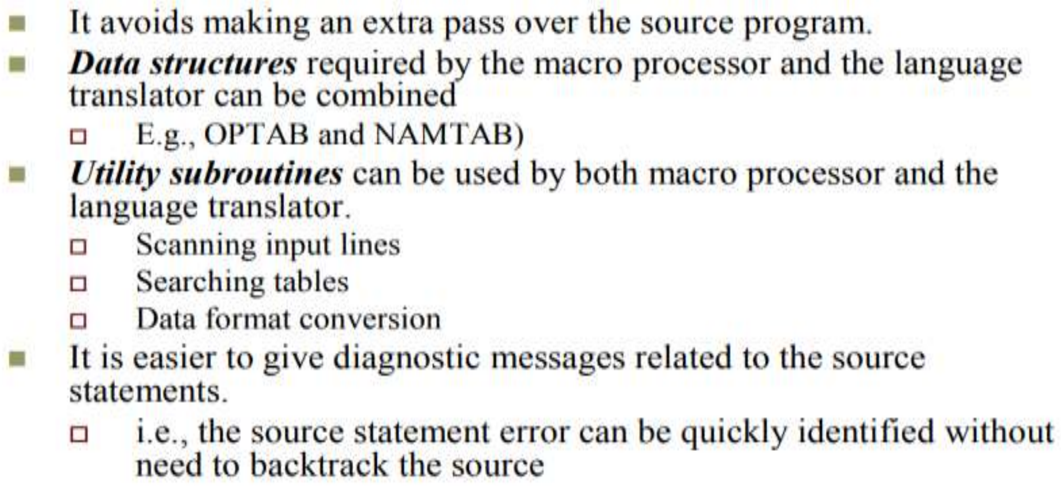
5. Another potential problem with general purpose macro processors involves the syntax used for macro definitions and invocation statements. With most special-purpose macro processors, macroinvocations are very similar in form to statements in the source programming language**.**

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

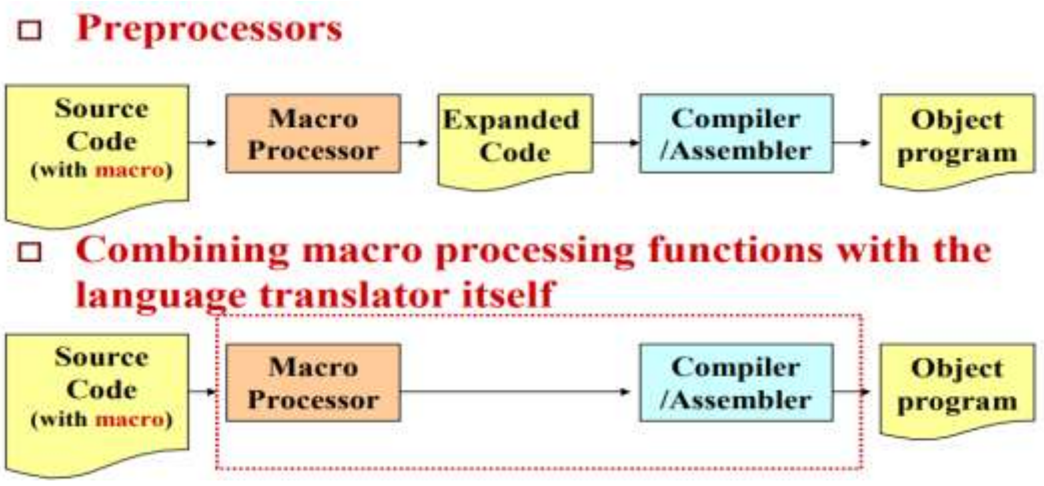
- Combining the macro processing function with the language translator itself
- Achieved using Line –by –line macro processor
  - The macro processor reads the source program statements
  - Process the statement
  - The output lines are passed to the language translator as they are generated, instead 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



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.

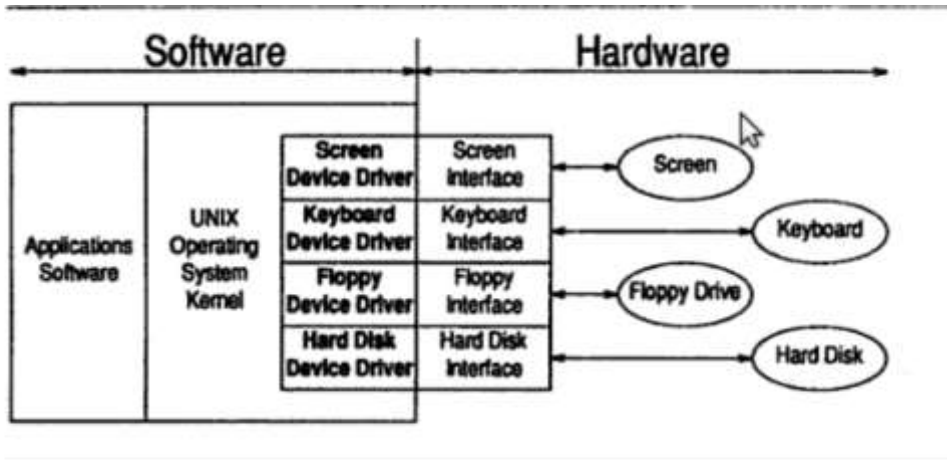
- The assembler or compiler will be considerably larger and more complex than it would be if a macro preprocessor were used
- The size may be a problem if the translator is to run on a computer with limited memory
- In any case, the additional complexity will add to the overhead of language translation
- 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

Device drivers

General design of device drivers

A device driver is glue between OS and its I/O devices. Device Drivers act as translators which converts generic requests received from the operating system into commands that specific peripheral controllers can understand.

The application software makes system calls to the operating system requesting services. The operating system analyses these requests and when, necessary, issues requests to the appropriate device driver. The device driver in turn analyses the request from the operating system and when necessary, issues commands to the hardware interface to perform the operations needed to service the request.



Although the process may seem unnecessarily complex, the existence of device drivers actually simplifies the operating system considerably. Without device drivers operating system would be responsible for talking directly to the hardware. This would require the operating system's designer to include support for all of the devices that users might want to connect to their computer. It would also mean that adding support for a new device would mean modifying the operating system itself.

By separating the device driver's functions from the operating system itself, the designer of the operating system can concern himself with issues that relate to the operation of the system as a whole. Furthermore, details related to individual hardware devices can be ignored and generic requests for I/O operations can be issued by the operating system to the device driver.

The device driver writer, on the other hand, does not have to worry about the many issues related to general I/O management, as these are handled by the operating system. All the device driver writer has to do is to take detailed device-independent requests from the operating system and to manipulate the hardware in order to fulfill the request.

Finally, the operating system can be written without any knowledge of the specific devices that will be connected. And the device driver writer can connect any I/O device to the system without having to modify the operating system. The operating system views all hard disks through the same interface, and the device driver writer can connect almost any type of disk to the system by providing a device driver so that the operating system is happy.

The result is a clean separation of responsibilities and the ability to add device drivers for new devices without changing the operating system. Device drivers provide the operating system with a standard interface to non-standard I/O devices.

Major design Issues

3 board categories:

1. Operating System/ Driver communications
2. Driver/Hardware Communications
3. Driver Operations

Operating System/ Driver communications

All issues related to exchange of information (commands and data) between the device driver and the operating system. It also includes support functions that the kernel provides for the benefit of the device driver.

Driver/Hardware Communications

Issues related to the exchange of information (commands and data) between the device driver and the device it controls (ie., hardware). Includes issues such as how software talks to hardware – and how the hardware talks back.

Driver Operations

Issues related to actual operation of the driver itself. It includes:

- Interpreting commands received from the operating system
- Scheduling multiple outstanding requests for service
- Managing the transfer of data across both interfaces (os and h/w)
- Accepting and processing hardware interrupts
- Maintaining the integrity of the driver's and the kernel's data structures

UNIX Device Driver

Is a collection of functions, usually written in C. Called by the UNIX Operating system, using the standard C function-calling mechanism. These routines are called entry points. The compiled code for the device driver is linked with the code for the operating system itself and the result is a new file containing the bootable operating system with all the device drivers.

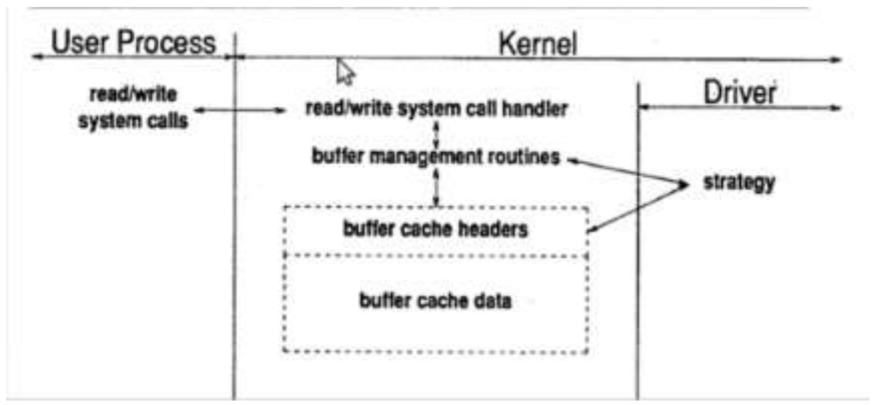
Based on the difference in the way they communicate with the UNIX operating system there are 4 different types:

1. Block Drivers
2. Character Drivers
3. Terminal Drivers

4. STREAMS

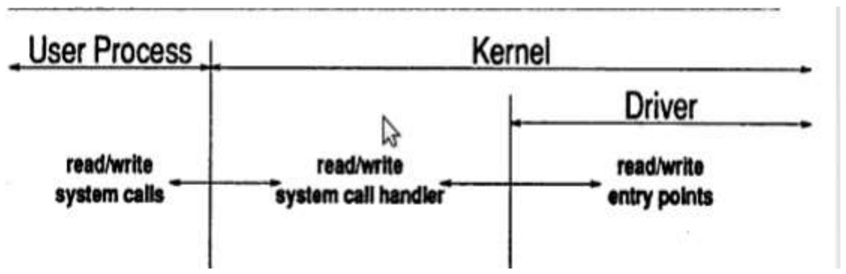
The kernel data structures that are accessed and the entry points that the driver can provide vary between the various types of drivers. These differences affect the type of devices that can be supported with each interface.

Block Drivers



Block Drivers communicate with the operating system through a collection of fixed-sized buffers. The OS manages a cache of these buffers and attempts to satisfy user requests for data by accessing buffers in the cache. The driver is invoked only when the requested data is not in the cache, or when buffers in the cache have been changed and must be written out. Because of this buffer cache the driver is insulated from many of the details of the user's requests and need only handle requests from the operating system to fill or empty fixed-sized buffers. Block drivers are used primarily to support devices that can contain file systems such as hard disk.

Character Drivers

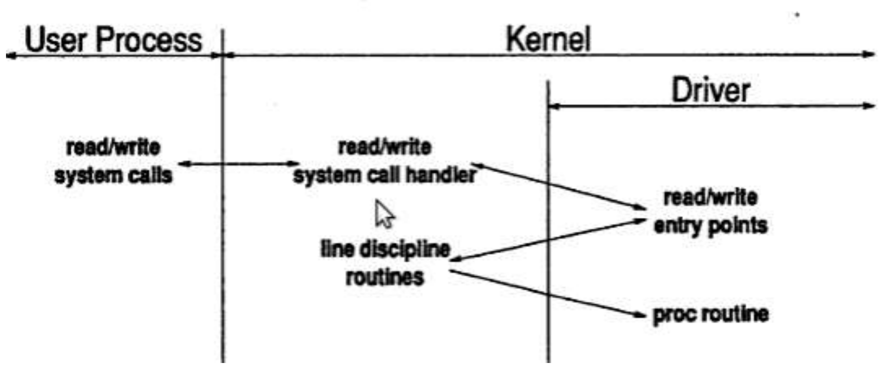


Character Drivers can handle I/O requests of arbitrary size and can be used to support almost any type of device. Usually, character drivers are used for devices that either deal with data a byte at a time (such as line printers) or work best with data in chunks smaller or larger than the standard fixed size buffers used by block drivers (such as analog-to-digital converters or tape drivers).

Major difference between character driver and block drivers is that:

- The user processes interact with block drivers only indirectly through the buffer cache, their relationship with character drivers is very direct.
- The I/O request is passed essentially unchanged to the driver to process and the character driver is responsible for transferring the data directly to and from the user process's memory.

Terminal Drivers

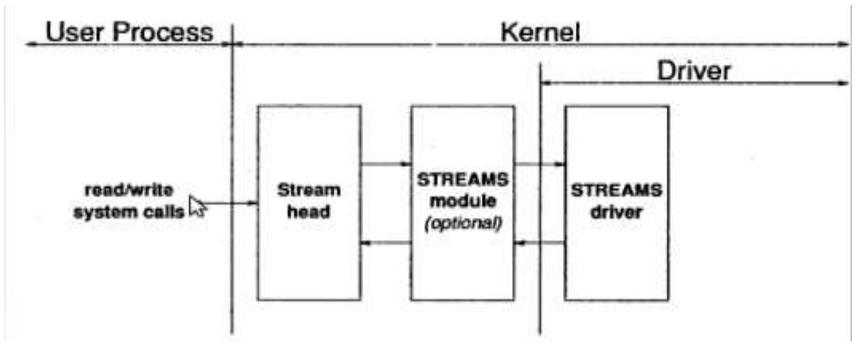


Terminal drivers are just character drivers specialised to deal with communication terminals that connect users to the central UNIX computer system. Terminal drivers are responsible not only for shipping data to and from users terminals, but also for handling line editing, tab expansion, and the many other terminal functions that are part of the standard UNIX terminal. Because of this additional processing that terminal drivers must perform, and the additional kernel routines and data structures that are provided to handle this. It is useful to consider terminal drivers as a separate type of driver altogether.

STREAMS Driver

STREAM Drivers are used to handle high-speed communication devices such as networking adapters that deal with unusual-sized chunks of data and that need to handle protocols. In System V Release 4, STREAMS drivers are also used to interface terminals. Versions of UNIX prior to

System V Release 3 supported network devices using character drivers. This was unsatisfactory because the character model assumes that a single driver sits between the user process and the device. With Character drivers the user process's request is handed directly to the driver with little intervention or processing by the kernel.



Networking devices usually support a number of layered protocols. The character model essentially required that each layer of the protocol be implemented within the single driver. This lack of modularity and reusability reduced the efficiency of the system. As a result, Dennis Ritchie, of Bell Laboratories developed extension to the character driver model called STREAMS. This new type of driver was introduced by AT&T in UNIX System V Release 3 and makes it possible to stack protocol processing modules between the user process and the driver.

Anatomy of a Device Driver

A driver is a set of entry points (routines) that can be called by the operating system. A driver can also contain:

- Data structure private to the driver
- References to kernel data structures external to the driver
- Routines private to the driver(ie., not entry points)

Most device drivers are written as a single source file. The initial part of the driver is sometimes called the **prologue.**

The prologue is everything before the first routine (like in c programming) contains:

- **#include:** directives referencing header files which define various kernel data types and structures

- **#define:** directives that provide mnemonic names for various constants used in the driver (in particular constants related to the location and definition of the hardware registers)
- Declarations of variables and data structures

The remaining parts of the driver are the entry points (c functions referenced by the operating system) and routines (c functions private to the driver).

Text Editors

A text editor is a piece of computer software for editing plain text. Text editors are often provided with the operating system or software development packages, and are used to change configuration files and programming language source codes. Text editors are considered as the primary interface to the computer for all types of "knowledge workers" as they compose, organize, study and manipulate computer-based information.

Text editors come in the following forms:

- **Line Editors**
  The scope of edit operations in a line editor is limited to a line of text. The line is designated *positionally*, eg: by specifying its serial number in the text, or *contextually*, eg: by specifying a context which uniquely identifies it. The primary advantage of line editors is their simplicity.
- **Stream Editors**
  Stream editors view the entire text as a stream of characters. This permit edits operations to cross line boundaries. Stream editors typically support character, line and context oriented commands based on the current editing context indicated by the position of a text pointer. The pointer can be manipulated using positioning and search commands
  - Line and Stream editors typically maintain multiple representations of text. One representation (the display form) shows the text as a sequence of lines. The editor maintains an internal form which is used to perform the edit operations. This form contains end-of-line characters and other edit characters. The editor ensures that these representations are compatible at every moment.
- **Screen Editors**

A line or stream editors does not display the text in the manner it would appear if printed. A screen editor uses the what-you-see-is-what-you-get principle in editor design. The editor displays a screen full of text at a time. The user can move the cursor over the screen, position it at the point where he desires to perform some editing and proceed with the editing directly. Thus it is possible to see the effect of an edit operation on the screen. This is very useful while formatting the text to produce printed documents.

- **Word Processor**

  Word Processors are basically document editors with additional features to produce well formatted hard copy output. Essential features of word processors are commands for moving sections of text from one place to another, merging of text, and searching and replacement of words. Many word processors support a spell-check option. With the advent of personal computer, word processors have seen widespread use amongst authors, office personnel and computer professionals. WordStar is a popular editor of this class.

- **Structure Editors**

  A structure editor incorporates an awareness of the structure of a document. This is useful in browsing through a document, eg: if a programmer wishes to edit a specific function in a program file. The structure is specified by the user while creating or modifying the document. Editing requirements are specified using the structure. A special class of structure editors, called syntax-directed editors, is used in programming environments.

  - Contemporary editors support a combination of line, string and screen editing functions. The Vi editors of Unix and the editors in desktop publishing systems are typical examples.

Overview of an Editing Process

An interactive editor is a computer program that allows a user to create and revise a target document. The term document includes objects such as computer programs, texts, equations, tables, diagrams, line art and photographs-anything that one might find on a printed page. Text editor is one in which the primary elements being edited are character strings of the target text.

The editor operates in two different modes:

1. **Command Mode**: The editor accepts certain user commands and performs the editing function accordingly
2. **Data Mode:** The editor accepts the user-entered text values and adds it to the file**.**

The document editing process is an interactive user computer dialogue designed to accomplish four tasks:

1) Select the part of the target document to be viewed and manipulated

2) Determine how to format this view on-line and how to display it.

3) Specify and execute operations that modify the target document.

4) Update the view appropriately.

Traveling – Selection of the part of the document to be viewed and edited. It involves first traveling through the document to locate the area of interest such as "next screenful", "bottom", and "find pattern". Traveling specifies where the area of interest is. This may be done explicitly by the user (eg: the line number command of a line editor) or may be implied in a user command (eg: the search command of a stream editor).

Filtering - The selection of what is to be viewed and manipulated is controlled by filtering. Filtering extracts the relevant subset of the target document at the point of interest such as next screenful of text or next statement.

Formatting/viewing - Formatting determines how the result of filtering will be seen as a visible representation (the view) on a display screen or other device. This is an abstract view, independent of the physical characteristics of an I/O device. The display component maps this view into the physical characteristics of the display device being used. This determines where a particular view may appear on the user's screen. The separation of viewing and display functions give rise to interesting possibilities like multiple windows on the same screen, concurrent edit operations using the same display terminal, etc. A simple text editor may choose to combine the viewing and display functions.

Editing - In the actual editing phase, the target document is created or altered with a set of operations such as insert, delete, replace, move or copy. The editing functions are often specialized to operate on elements meaningful to the type of editors. For example, Manuscript oriented editors operate on elements such as single characters, words, lines, sentences and paragraphs and program-oriented editors operates on elements such as identifiers, keywords and statements.

The user-interface of an Editor

The user of an interactive editor is presented with a conceptual model of the editing system. This model is an abstract framework on which the editor and the world on which the operations are based. The conceptual model, in essence provides an easily understood abstraction of the target document and its elements, with a set of guidelines describing the effects of operations on these elements. The line editors simulated the world of the keypunch they allowed operations on numbered sequence of 80-character card image lines, either within a single line or on an integral number of lines.

Some of the modern Screen-editors define a world in which a document is represented as a quarter-plane of text lines, unbounded both down and to the right. The user sees, through a cutout, only a rectangular subset of this plane on a multi line display terminal. The cutout can be moved left or right, and up or down, to display other portions of the document.

Besides the conceptual model, the user interface is also concerned with the **input devices, the output devices, and the interaction language of the system.**

1. **Input Devices**: The input devices are used to enter elements of text being edited, to enter commands, and to designate editable elements. Input devices used with the editors are categorized as: 1) Text devices 2) Button devices 3) Locator devices

   Test Devices/ string devices are typically typewriter like keyboards on which user presses and release keys, sending unique code for each key. Virtually all computer key boards are of the QWERTY type.

   Button or Choice devices: generate an interrupt or set a system flag, usually causing an invocation of an associated application program. Also special function keys are also available on the key board. Alternatively, buttons can be simulated in software by displaying text strings or symbols on the screen. The user chooses a string or symbol instead of pressing a button.

Locator devices: They are two-dimensional analog-to-digital converters that position a cursor symbol on the screen by observing the user's movement of the device. The most common such devices are the mouse and the data tablet. The Data Tablet is a flat, rectangular, electromagnetically sensitive panel. Either the ballpoint pen like stylus or a puck, a small device similar to a mouse is moved over the surface. The tablet returns to a system program the co-ordinates of the position on the data tablet at which the stylus or puck is currently located. The program can then map these data-tablet coordinates to screen coordinates and move the cursor to the corresponding screen position. Text devices with arrow (Cursor) keys can be used to simulate locator devices. Each of these keys shows an arrow that point up, down, left or right. Pressing an arrow key typically generates an appropriate character sequence; the program interprets this sequence and moves the cursor in the direction of the arrow on the key pressed.

Voice-input devices: which translate spoken words to their textual equivalents, may prove to be the text input devices of the future. Voice recognizers are currently available for command input on some systems.

2. **Output devices:** The output devices let the user view the elements being edited and the result of the editing operations. Earlier the output devices are limited in range, but now it become diverse.

   - The first output devices were teletypewriters and other character-printing terminals that generated output on paper.
   - Next "glass teletypes" based on Cathode Ray Tube (CRT) technology which uses CRT screen essentially to simulate the hard-copy teletypewriter.(although a few operations such as backspacing, were performed more elegantly)
   - Advanced CRT terminals use hardware assistance for such features as moving the cursor, inserting and deleting characters and lines, and scrolling lines and pages.
   - The modern professional workstations are based on personal computers with high resolution displays; support multiple proportionally spaced character fonts to produce realistic facsimiles of hard copy documents.

Thus the user can see the document portrayed essentially as it will look when printed on paper.

3. **Interaction Language:** The interaction language of the text can be:

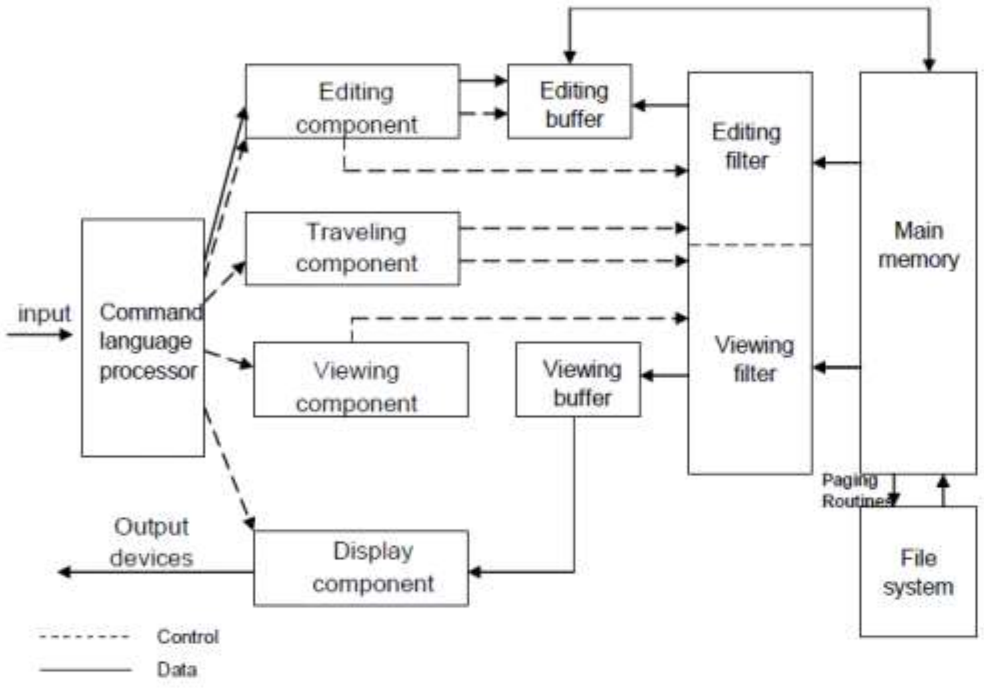
   The typing oriented or text command-oriented method: It is the oldest of the major editing interfaces. The user communicates with the editor by typing text strings both for command names and for operands. These strings are sent to the editor and are usually echoed to the output device. Typed specification often requires the user to remember the exact form of all commands, or at least their abbreviations. If the command language is complex, the user must continually refer to a manual or an on-line Help function. The typing required can be time consuming for in-experienced users.

   Function key interfaces: The function-key interface addresses these deficiencies. Here each command has associated with it a marked key on the user's keyboard. For example, the insert character command might have associated with it a key marked IC. Function-key command specification is typically coupled with cursor-key movement for specifying operands, which eliminates much typing. For the common commands in a function-key editor, usually only a single key is pressed. For less frequently invoked commands or options, an alternative textual syntax may be used. More commonly, however special keys are used to shift the standard function-key interpretations, just as SHIFT key on a typewriter shifts from the lowercase to uppercase. As an alternative to shifting function keys, the standard alphanumeric keyboard is often overloaded to simulate function keys. For example, the user may press a control key simultaneously with a normal alphanumeric key to generate a new character that is interpreted like a function key. The function key-oriented systems often have either too few keys, requiring multiple key stroke commands, or have too many unique keys, results in an un-widely keyboard. In either case, the function-key systems demand even more agility of the user than a standard keyboard does.

   Menu oriented interface: A menu is a multiple choice set of text strings or icons which are graphical symbols that represent objects or operations. The user can perform actions by selecting items from the menu. The editor prompts the user with a menu of only those actions that can be taken at the current state of the system. One problem with menu oriented system can arise when there are many possible actions and several choices are required to complete an action. The display area of the menu is rather limited, therefore the user might be presented with several consecutive menus in a hierarchy before the appropriate command and its options appear.

Since this can be annoying and detrimental to the performance of an experienced user, some menu-oriented systems allow the user to turn off menu control and return to a typing or function-key interface. Other systems have the most-used functions on a main command menu and have secondary menus to handle the less frequently used functions. Still other systems display the menu only when the user specifically asks for it. For example, the user might press a button on a mouse to display a menu with the full choice of applicable commands (temporarily over laying some existing information on the screen). The mouse could be used to select the appropriate command. The system would then execute the command and delete the menu. Interfaces like this, in which prompting and menu information are given to the user at little added cost and little degradation in response time, are becoming increasingly popular.

Editor Structure



The structure of editor is shown is the figure. Most text editors have a structure similar as in the figure, regardless of the particular features they offer and the computers on which they are implemented.

The command Language Processor: accepts input from the user's input devices, and analyzes the tokens and syntactic structure of the commands. It functions much like the compiler, the lexical and syntactic phases of a compiler. The command language processor may also invoke the semantic routines directly. In a text editor, these semantic routines perform functions such as editing and viewing. Alternatively, the command language processor may produce an intermediate representation is then decoded by an interpreter that invokes the appropriate semantic routines. The use of an intermediate representation allows the editor to provide a variety of user-interaction languages with a single-set of semantic routines that are driven from a common intermediate representation. The semantic routines involve traveling, editing, viewing and display functions. Editing operations are always specified by the user and display operations are specified implicitly by the other three categories of operations. However, the traveling and viewing operations may be invoked either explicitly by the user or implicitly by the editing operations. In particular, there need not be a simple one-to-one relationship between what is currently displayed on the screen and what can be edited.

The components of editor structures are:

1. **Editing Component:** In editing a document, the start of the area to be edited is determined by the current editing pointer maintained by the editing component, which is the collection of modules dealing with editing tasks. The current editing pointer can be set or reset explicitly by the user using travelling commands, such as next paragraph and next screen, or implicitly as a side effect of the previous editing operation such as delete paragraph.
2. **Traveling Component:** The traveling component of the editor actually performs the setting of the current editing and viewing pointers, and thus determines the point at which the viewing and /or editing filtering begins.
3. **Editing Filter**: When the user issues an editing command, the editing component invokes the editing filter. This component filters the document to generate a new **editing buffer** based on the current editing pointer as well as on the editing filter parameters. These

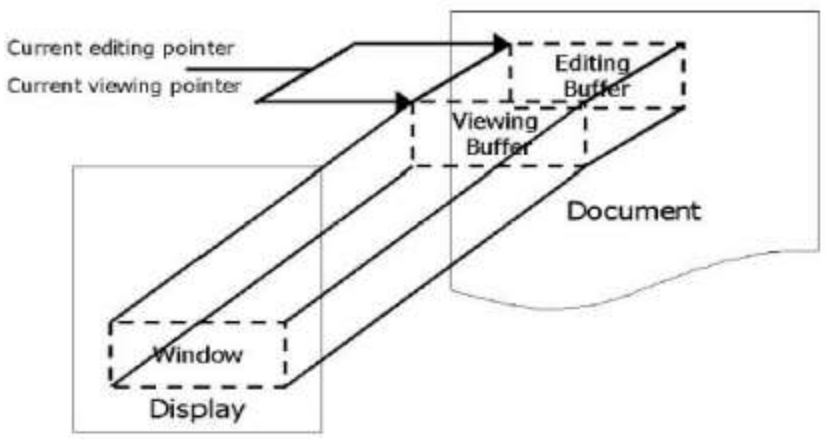
parameters, which are specified both by the user and the system, provide information such as range of text that, can be affected by an operation. Filtering consists of the selection of contiguous characters beginning at the current point. The filtering may depend on more complex user specifications pertaining to the content and structure of the document. Such filtering might result in the gathering of portions of the document that are not necessarily contiguous. The semantic routines of the editing component then operate on the dieting buffer, which is essentially a filtered subset of the document data structure.

4. **Viewing Component:** In viewing a document, the start of the area to be viewed is determined by the current viewing pointer. This pointer is maintained by the viewing component of the editor, which is a collection of modules responsible for determining the next view. The current viewing pointer can be set or reset explicitly by the user or implicitly by system as a result of previous editing operation.
5. **Viewing Filter:** When the display needs to be updated, the viewing component invokes the viewing filter. This component filters the document to generate a new **viewing buffer** based on the current viewing pointer as well as on the viewing filter parameters. These parameters which are specified both by the user and by the system, provide information such as number of characters needed to fill the display, and how to select them from the document. In Line editors, the viewing buffer may contain the current line; in screen editors, this buffer may contain rectangular cut out of the quarter-plane of text. This viewing buffer is then passed to the display component of the editor
6. **Display Component**: It takes the idealized view from the viewing component and maps it to a physical output device in the most efficient manner. The display component produces a display by mapping the buffer to a rectangular subset of the screen, usually a window.

The editing and viewing buffers, while independent, can be related in many ways. In simplest case, they are identical: the user edits the material directly on the screen as in the figure shown below. On the other hand, the editing and viewing buffers may be completely disjoint. E.g. The user of a certain editor might travel to line 75,and after viewing it, decide to change all occurrences of “ugly duckling” to “swan” in lines 1 through 50 of the file by using a change command such as

[1,50] c/ugly duckling/swan/

As a part of the editing command there is implicit travel to the first line of the file. Lines 1 through 50 are then filtered from the document to become the editing buffer. Successive substitutions take place in this editing buffer without corresponding updates of the view. If the pattern is found, the current editing and viewing pointers are moved to the last line on which it is found, and that line becomes the default contents of both the editing and viewing buffers. If the pattern is not found, line 75 remains in the editing and viewing buffers. The editing and viewing buffers can also partially overlap, or one may be completely contained in the other. For example, the user might specify a search to the end of the document, starting at a character position in the middle of the screen. In this case the editing filter creates an editing buffer that contains the document from the selected character to the end of the document. The viewing buffer contains the part of the document that is visible on the screen, only the last part of which is in the editing buffer.



Simple relationship between editing and viewing buffers

Windows typically cover the entire screen or rectangular portion of it. Mapping viewing buffers to windows that cover only part of the screen is especially useful for editors on modern graphics based workstations. Such systems can support multiple windows, simultaneously showing different portions of the same file or portions of different file. This approach allows the user to perform inter-file editing operations much more effectively than with a system only a single window. The mapping of the viewing buffer to a window is accomplished by two components of the system:

(i) The viewing component formulates an ideal view often expressed in a device independent intermediate representation. This view may be a very simple one consisting of a windows worth of text arranged so that lines are not broken in the middle of words. At the other extreme, the idealized view may be a facsimile of a page of fully formatted and typeset text with equations, tables and figures.

(ii) The display component takes these idealized views from the viewing component and maps it to a physical output device the most efficient manner possible.

Updating of a full-screen display connected over low-speed lines is slow if every modification requires a full rewrite of the display surface. Greatly improved performance can be obtained by using optimal screen-updating algorithms. These algorithms are based on comparing the current version of the screen with following screen. They make use of innate capabilities of the terminal, such as insert-character and delete-character functions, transmitting only those characters needed to generate a correct display.

Device-independent output, like device-independent input, helps to provide portability of the interaction language. Decoupling editing and viewing operations from the display functions for the output avoids the need to have a different version of the editor for each output device. Many editors make use of a terminal-control database. Instead of having explicit terminal-control sequences in the display routines, these editors simply call terminal-independent library routines such as scroll down or read cursor position. These library routines use the terminal-control sequences for a particular terminal. Consequently, adding a new terminal merely entails adding a database description of that terminal.

The components of the editor deal with a user document on two levels:

(i) In main memory

(ii) In the disk file system.

Loading an entire document into main memory may be infeasible. However if only part of a document is loaded and if many user specified operations require a disk read by the editor to locate the affected portions, editing might be unacceptably slow. In some systems this problem is solved by the mapping the entire file into virtual memory and letting the operating system perform efficient demand paging. An alternative is to provide is the editor paging routines which read one or more logical portions of a document into memory as needed. Such portions are often termed pages, although there is usually no relationship between these pages and the hard copy document pages or virtual memory pages. These pages remain resident in main memory until a user operation requires that another portion of the document be loaded.

Documents are often represented internally not as sequential strings of characters, but in an editor data structure that allows addition, deletion and modification with a minimum of I/O and character movement. When stored on disk, the document may be represented in terms of this data structure or in an editor-independent general-purpose format, which might consist of character strings with embedded control characters such as linefeed and tab.

Editor's function in three basic types of computing environment, and each type of environment imposes some constraints on the design of an editor.

(i) Time-sharing environment: The time-sharing editor must function swiftly within the context of the load on the computer's processor, central memory and I/O devices. *Some time-sharing editing systems take advantage of hardware capabilities to perform editing tasks. Many workstations and intelligent terminals have their own micro-processors and local buffer memories in which editing manipulations can be done. Small actions are not controlled by the CPU of the host processor, but are handled by the local workstation. For example, the editor might send a full screen of material from the host processor to the workstation. The user would then be free to add and delete characters and lines, using the local buffer and workstation-based control commands. After the buffer has been edited in this way, its updated contents would be transmitted back to the host processor.*

The advantage of this scheme is that the host needs to be concerned with each minor change or keystroke. However, this is also the major disadvantage. With a non intelligent terminal, the CPU sees every character as it is typed and can react immediately to perform error checking, to prompt, to update the data structure etc. With an intelligent workstation, the lack of constant CPU intervention often means that the functionality provided to the user is more limited. Also local editing operations on the workstation may be lost in the event of the system crash. On the other hand, systems that allow each character to interrupt the CPU may not use the full hardware editing capabilities of the workstation because the CPU needs to see every keystroke and provide character-by-character feedback.

(ii) Stand-alone environment: The editor on a stand-alone system must have access to the functions that the time sharing editors obtain from its host operating system. This may be provided in part by a small local operating system or they may be built into the editor itself if the stand alone system is dedicated to editing.

(iii) Distributed environment. The editor operating in a distributed resource sharing local network must, like a standalone editor, run independently on each user's machine and must, like a time sharing editor, content for shared resources such as files.

Debuggers

Debugging is the process of isolating and correcting the causes of known errors in a program. An interactive debugging system provides programmers with facilities that aid in testing and debugging of programs.

Debugging Functions and Capabilities

1. The most obvious requirement is for a set of **unit test functions** that can be specified by the programmer. One important group of such unit test functions deals with **execution sequencing.** Execution Sequencing is the observation and control of the flow of program execution. For example, the program may be halted after a fixed number of instructions are executed. Similarly, the programmer may define:

- Breakpoints – The programmer may define break points which cause execution to be suspended, when a specified point in the program is reached. After execution is suspended, the debugging command is used to analyze the progress of the program and to diagnose errors detected. Execution of the program can then be resumed.
- Conditional Expressions – Programmers can define some conditional expressions, evaluated during the debugging session, program execution is suspended, when conditions are met, analysis is made, later execution is resumed
- Gaits- Given a good graphical representation of program progress may even be useful in running the program in various speeds.

2. A debugging system should also provide functions such as **tracing and trace-back**.

- Tracing can be used to track the flow of execution logic and data modifications based on the conditional expressions. The control flow can be traced at different levels of detail – procedure, branch, individual instruction, and so on.

- Trace-back can show the path by which the current statement in the program was reached. It can also show which statements have modified a given variable or parameter. This kind of information should be displayed symbolically, and should be related to the source program – for example, as statement numbers are displayed rather than as hexadecimal displacements.

3. Debugging system should have good **program display capabilities**.

- It must be possible to display the program being debugged, complete with statement numbers. The user should be able to control the level at which the display occurs. For eg; the program may be displayed as it was originally written, after macro expansion and so on. It is also useful to be able to modify and incrementally recompile the program during the debugging session. The system should save all the debugging specifications (break-points definitions, display modes, etc.) across such a recompilation, so the programmer does not need to reissue all of these debugging commands. It should be possible to symbolically display or modify the contents of any of the variables and constants in the program, and then resume the execution. The intent is to give the appearance of an interpreter, regardless of the underlying mechanism that is actually used.

Besides providing these functions and capabilities, the debugging system should consider the language in which the program being debugged is written. Most user environments and many applications systems involve the use of different programming languages.

4. **A single debugging tool should be available to multilingual situations.**

- Debugger commands that initiate actions and collect data about program's execution should be common across languages. However, a debugging system must be sensitive to the specific language being debugged so that the procedural, arithmetic and conditional logic can be coded in the syntax of that language.

These requirements have a number of consequences for the debugger and for other software. When the debugger receives control, the execution of the program being debugged is temporarily suspended.

5. The debugger must then be able to determine the language in which the program is written and **set its context accordingly**.

- The debugger should be able to switch its context when a program written in one language calls a program written in a different language.
- To avoid confusion, the debugger should inform the user of such changes in context.
- The context being used has many different effects on the debugging interaction. For example, assignment statements that change the values of variables during debugging should be processed according to the syntax and semantics of the source programming language.
- In COBOL, the user might enter the debugging command as MOVE 3.5 TO A, whereas in FORTRAN, user might enter A = 3.5. Likewise, conditional expressions should use the notation of the source language. In COBOL, the condition that A be unequal to B might be expressed as IF A NOT EQUAL TO B for debugging, while in FORTRAN, IF (A .NE. B). Similar differences exist with respect to the form of statement labels, keywords and so on.

The notation used to specify certain debugging functions varies according to the language of the program being debugged. However, the functions themselves are accomplished in essentially the same manner regardless of the source programming language. To perform these operations,

6. The debugger must have access to information gathered by the language translator.
   - The internal symbol dictionary formats vary between different language translators.
   - One approach for the language translators to produce the needed information in a standard external form for the debugger regardless of the internal form used in the translator.
   - Another possibility is for the language translator to provide debugger interface modules that can respond to requests for information in a standard way regardless of the language being debugged.
7. The display of source code during the debugging session.
   - One option is that the language translator may provide the source code or source listing tagged in some standard way so that the debugger has a uniform method of navigating about it.
   - Another option, the translator may supply an interface module that does the navigation and display in response to requests from the debugger.

8. The debugging system should be able to deal with optimized code.
   - Application code used in production environments is usually optimized. The requirement to handle optimized code may create many problems for the debugger.
   - Many optimizations involve the rearrangement of segments of code in the program. For eg. - invariant expressions can be removed from loops, separate loops can be combined into a single loop, or a loop may be partially unrolled into straight-line code, redundant expression may be eliminated, block of codes may be rearranged to eliminate of unnecessary branch instructions.
   - The user of a debugging system deals with the source program in its original form, before optimizations are performed. However, code rearrangement alters the execution sequence and may affect tracing, breakpoints and even statement counts if entire statements are involved. If an error occurs, it may be difficult to relate the error to the appropriate location in the original source program.
9. Storage of variables
   - When a program is translated, the compiler normally assigns a home location in main memory or in an activation record to each variable.
   - The variable value may be temporarily held in registers at various times to improve speed of access.
   - Statement referring to these variables use the value stored in the register, instead of taking the variable value from its home location. These optimizations present no problem for displaying the values of such variables.
   - However, if the user changes the value of the variable in its home location while debugging, the modified value might not be used by the program as intended when execution is resumed.
   - In a similar type of global optimization, a variable may be permanently assigned to a register. In this case, there may be no home location at all.
10. The debugging of optimized code requires a substantial amount of cooperation from the optimizing compiler.
   - The compiler must retain information about any transformations that it performs on the program. The debugger should use this information to modify the debugging request

made by the user and thereby perform the intended operation. For eg: it may be possible to stimulate the effect of a break point that was set on an eliminated statement.

- A modified variable value can be stored in the appropriate register as well as at the home location for that variable.

However, some more complex optimizations cannot be handled as easily. In such cases, the debugger should merely inform the user that a particular function is unavailable at this level of optimization, instead of attempting some incomplete imitation of the function.

Relationship with other parts of the System

An interactive debugger must be related to other parts of the system in many different ways:

- **Availability**: Interactive debugger must appear to be a part of the run-time environment and an integral part of the system. When an error is discovered, immediate debugging must be possible because it may be difficult or impossible to reproduce the program failure in some other environment or at some other time. Thus the **debugger must communicate and cooperate with other operating system components such as interactive subsystems.**
- **Debugging is important at production time than at the application development time.** The users need to be able to debug in a production environment. When an application fails during a production run, work dependent on that application stops. Since the production environment is different from the test environment, many program failures cannot be repeated outside the production environment.
- **Consistency with security and integrity components of the system**: Use of debugger must be subjected to the normal authorization mechanism and must leave the usual audit trails. Someone (unauthorized user) must not access any data or code. It must not be possible to use the debuggers to interface with any aspect of system integrity. Debugger compares with the storage dump which includes information that happens to have been left in storage and presents information only for the contents of specific named objects.
- **Coordination with existing and future language translators:** The debugger must coordinate its activities with those of existing and future language compilers and interpreters. It is assumed that debugging facilities in existing language will continue to

exist and be maintained. The requirement of cross-language debugger assumes that such a facility would be installed as an alternative to the individual language debuggers.

User- Interface Criteria

The interactive debugging system should be user friendly. The facilities of debugging system should be organized into few basic categories of functions which should closely reflect common user tasks.

- **Full – screen displays and windowing systems**: The user interaction should make use of full-screen display and windowing systems. The primary advantage of such interface is that the information can be displayed and changed easily and quickly. With menus and full screen editors, the user has far less information to enter and remember.
- **Menus:** If the tasks a user needs to perform are reflected in the organization of menus, then the system become friendly to the user. Menus should have tiles that identify the task they perform. The directions should precede any choices available to the user. Techniques such as indentation should be used to help separate portions of the menu.

  Menu systems lack direct routing. It should be possible to go directly to the menu that the user wants to select without having to retrace an entire hierarchy of menus.

  When a full-screen terminal device is not available, user should have an equivalent action in a linear debugging language by providing commands.
- **Command language**: The command language should have a clear, logical, simple syntax. Commands should be simple rather than compound and should require as few parameters as possible. There should be a consistent use of parameter names across the set of commands. Parameters should automatically be checked for errors for type and range values. Defaults should be provided for parameters, and the user should be able to determine when such defaults have been applied. Command formats should be flexible as possible. Command language should minimize punctuations such as parenthesis, slashes, and other special characters. Where possible, information should be invoked through prompting techniques.
- **On Line HELP facility:** Good interactive system should have an on-line HELP facility. Even a list of the available commands can provide valuable assistance for the inexperienced or occasional user. For advanced users, the HELP function can be multi-level and quite specific. One powerful use of HELP with menus is to provide explanatory text for all options present on the screen. These can be selectable by option number or name, or by filling the

choice slot with a question mark. HELP should be accessible from any state of the debugging session.

Debugging Methods

Debugging by Induction

Many errors can be found by using a disciplined thought process without ever going near the computer. One such thought process is induction, where one proceeds from the particulars to the whole. By starting with the symptoms of the error, possibly in the result of one or more test cases, and looking for relationships among the symptoms, the error is often uncovered.

The induction process is illustrated in Figure 1:

- **Locate the pertinent data.** A major mistake made when debugging a program is failing to take account of all available data or symptoms about the problems. The first step is the enumeration of all that is known about what the program did correctly, and what it did incorrectly (i.e., the symptoms that led one to believe that an error exists). Additional valuable clues are provided by similar, but different, test cases that do not cause the symptoms to appear.
- **Organize the data.** Remembering that induction implies that one is progressing from the particulars to the general, the second step is the structuring of the pertinent data to allow one to observe patterns, of particular importance is the search for contradictions (i.e., "the errors occurs only when the pilot perform a left turn while climbing"). A particularly useful organizational technique that can be used to structure the available data is shown in the following table. The "What" boxes list the general symptoms, the "Where" boxes describe where the symptoms were observed, the "When" boxes list anything that is known about the times that the symptoms occur, and the "To What Extent" boxes describes the scope and magnitude of the symptoms. Notice the "Is" and "Is Not" columns. They describe the contradictions that may eventually lead to a hypothesis about the error.

?	Is	Is Not
What		
Where		
When		
To What Extent		

- **Devise a hypothesis.** The next steps are to study the relationships among the clues and devise, using the patterns that might be visible in the structure of the clues, one or more hypotheses about the cause of the error. If one cannot devise a theory, more data are necessary, possibly obtained by devising and executing additional test cases. If multiple theories seem possible, the most probable one is selected first.

- **Prove the hypothesis.** A major mistake at this point, given the pressures under which debugging is usually performed, is skipping this step by jumping to conclusions and attempting to fix the problem. However, it is vital to prove the reasonableness of the hypothesis before proceeding. A failure to do this often results in the fixing of only a symptom of the problem, or only a portion of the problem. The hypothesis is proved by comparing it to the original clues or data, making sure that this hypothesis completely explains the existence of the clues. If it does not, the hypothesis is invalid, the hypothesis is incomplete, or multiple errors are present.

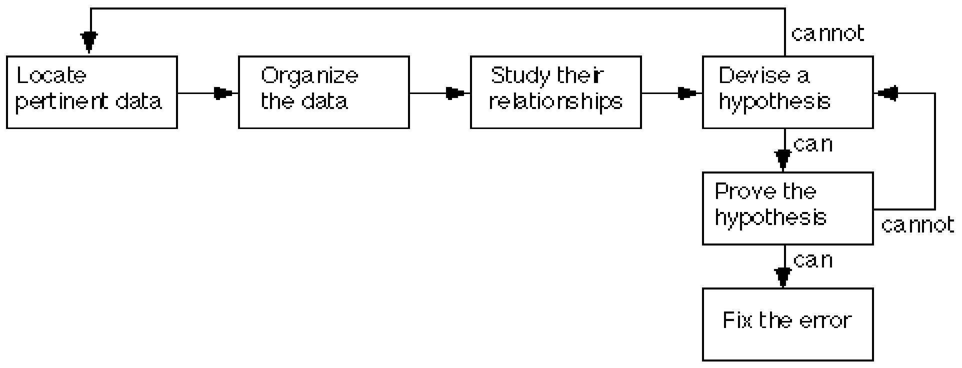


Figure 1. Inductive Debugging Process

Debugging By Deduction

An alternate thought process, that of deduction, is a process of proceeding from some general theories or premises, using the processes of elimination and refinement, to arrive at a conclusion. This process is illustrated in Figure 2:

- **Enumerate the possible causes or hypotheses.** The first step is to develop a list of all conceivable causes of the error. They need not be complete explanations; they are merely theories through which one can structure and analyze the available data.
- **Use the data to eliminate possible causes.** By a careful analysis of the data, particularly by looking for contradictions (the previous table could be used here), one attempts to eliminate all but one of the possible causes. If all are eliminated, additional data are needed (e.g., by devising additional test cases) to devise new theories. If more than one possible cause remains, the most probable cause (the prime hypothesis) is selected first.
- **Refine the remaining hypothesis.** The possible cause at this point might be correct, but it is unlikely to he specific enough to pinpoint the error. Hence, the next step is to use the available clues to refine the theory to something more specific.
- **Prove the remaining hypothesis.** This vital step is identical to the fourth step in the induction method.

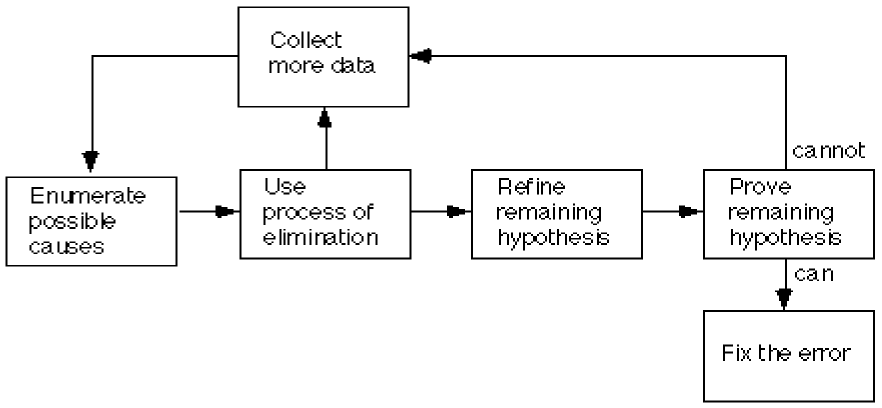

Collect more data
Enumerate possible causes
Use process of elimination
Refine remaining hypothesis
Prove remaining hypothesis
cannot
can
Fix the error

Figure 2. Deductive Debugging Process

Debugging by Backtracking

For small programs, the method of backtracking is often used effectively in locating errors. To use this method, start at the place in the program where an incorrect result was produced and go backwards in the program one step at a time, mentally executing the program in reverse order, to derive the state (or values of all variables) of the program at the previous step. Continuing in this fashion, the error is localized between the point where the state of the program was what was expected and the first point where the state was not what was expected.