Module 5

Macro Processors

MACRO PROCESSORS

Ques 1) What is macro processor? What do you mean by expanding macros?

Ans: Macro Processor

A macro processor is a program that copies a stream of text from one place to another, making a systematic set of replacements as it does so. Macro processors are often embedded in other programs, such as assemblers and compilers. Sometimes they are standalone programs that can be used to process any kind of text.

Expanding the Macros

A macro is a unit of specifications for program generation through expansion. Macros are special code fragments that are defined once in the program and are used repetitively by calling them from various places within the program. It is similar to the subprogram in the sense that both can be used to organize the program better by separating-out the frequently used fragment into a different block.

The main program calls this block of code as and when needed. Both of them can have an associated list of parameters. A macro represents a commonly used group of statements in the source programming language. The macro processor replaces each macro instruction with the corresponding group of source language statements. This is called **expanding the macros**.

Ques 2) Write the basic macro processor function.

Ans: Basic Macro Processor Functions

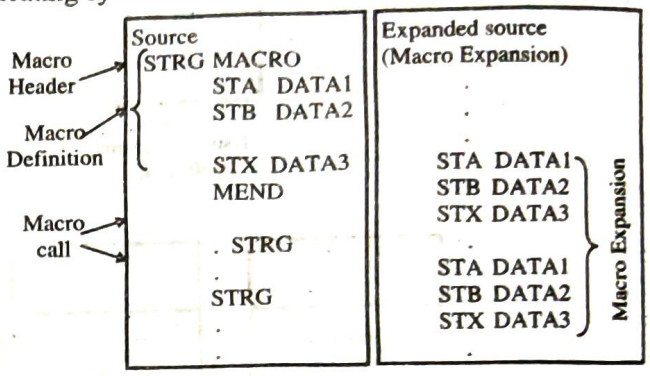
A macro represents a commonly used group of statements in the source programming language.

The macro processors do macro expansion by replacing each macro instruction with the corresponding group of source language statements. Macro instructions allow the programmer to write a shorthand version of a program, and leave the mechanical details to be handled by the macro processor.

For example, use a macro SAVEREGS to save the contents of all registers on SIC/XE machine, instead of a sequence of seven instructions (STA, STB, etc.).

The functions of a macro processor involve the substitution of one group of characters or lines for another. The design and capabilities of a macro processor may be influenced by the form of the programming language statements. The meaning of these statements and their translation into machine languages are of no concern during the macro expansion.

The design of a macro processor is usually machine independent. Macro processors are commonly used in assemblers, high-level programming languages, and operating system command languages.



Ques 3) Explain the macro definition. (2020[02])

Or

Write the syntax of macro definition and define MEND directive.

Or

Explain the concept of macro definition with the help of examples. (2017 [2.5])

Or

Illustrate macro definition using an example. (2019[02])

Ans: Macro Definition

A macro definition consists of three parts:

1) Macro prototype statement.
2) One or more model statements.
3) Macro preprocessor statements.

Note: Each parameter in macro defintion begins with the character '&'(ampersand), this facilitates substitution of parameters douring macro expansion.

A macro is defined using the follwing **syntax**:

```
MACRO
        <Name> [parameter List]
        - - - - - - -
        - - - -
        <assembly language code>
        - - - - - - -
        -
MEND
```

The assembler directive named MEND indicates the terination of MACRO definition. **For example,** consider the following code

```
MACRO                          //macro header
statement
SUM   &x, &y                   //macro prototype
statement
LOAD  &x                       //model statement
ADD   &y                       // model statement
STORE &x                       //model statement
MEND                           //End of defintion unit
```

The macro header statement indicates the existence of a macro definition unit. The next statement in the defintion unit is the prototype for a macro call. It consists of the name of the macto and the list of parameters. In the above example name of the macro is SUM and the list of formal parameters are &x and &y. the prototype id follwed by the so called model statements. These are assembly statements which replace the macro call as a result of macro expansion.

Ques 4) What is macro invocation?

Ans: Macro Invocation

After defining the macro the nest stage is to call the macro. Macro invocation statement gives the name of the macro instruction being invoked and the actual parameters to be used in expanding the macro. The macro invocation statement is often referred to as a macro call. In the above example, the macro is called by its macro name along with actual parameters.

SUM MN → Actual parameters
//macro call

Ques 5) Define macro expansion. Write its algorithm.

Or

What is macro expansion? Discuss the flow of control during expansion.

Or

Explain the concept of expansion with the help of examples. **(2017 [2.5])**

Or

Explain the macro expansion. **(2020[02])**

Or

Illustrate macro definition and expansion using an example. **(2019[01])**

Ans: Macro Expansion

After the calling stage macro expansion is the next stage. The assembly statements (model statements) replace the macro call as a result of macro expansion. The macro expansion for the above example is given as:

```
LOAD    M
ADD     N
STORE   M
```

Two **key notions** concerning macro expansion are as follows:

1) **Expansion Time Control Flow:** This determines the order in which model statements are visited during macro expansion.
2) **Lexical Substitution:** It is used to generate an assembly statement from a model statement.

Flow of Control during Expansion

The default flow of control during macro expansion is sequential. Thus, in the absence of preprocessor statements, the model statements of a macro are visited sequentially starting with the statement following the macro prototype statement and ending with the statement preceding the MEND statement.

A pre-processor statement can alter the flow of control during expansion such that some model statements are either never visited during expansion, or are repeatedly visited during expansion. The former results in conditional expansion and the latter in expansion time loops. The flow of control during macro expansion is implemented using a **Macro Expansion Counter** (MEC).

Algorithm: Macro Expansion

After the calling stage macro expansion is the next stage. The assembly statements (model statements) replace the macro call as a result of macro expansion. The flow of control during macro expansion is implemented using a **Macro Expansion Counter** (MEC).

The algorithm is shown below:

1) (Macro Expansion Counter (MEC):= statement number of first statement following the prototype statement;
2) While statement pointed by MEC is not a MEND statement
   i) If a model statement then
      a) Expand the statement
      b) MEC := MEC + 1;
   ii) Else(i.e., a preprocessor statement)
      MEC: = new value specified in the statement;
3) Exit from macro expansion.

MEC is set to point at the statement following the prototype statement. It is incremented by 1 after expanding a model statement. Execution of a preprocessor statement can set MEC to a new value to implement conditional expansion or expansion time loops.

Ques 6) What is the use of lexical substitution?

Ans: Use of Lexical Substitution

Lexical substitution is used to generate an assembly statement from a model statement. A model statement consists of 3 types of strings:

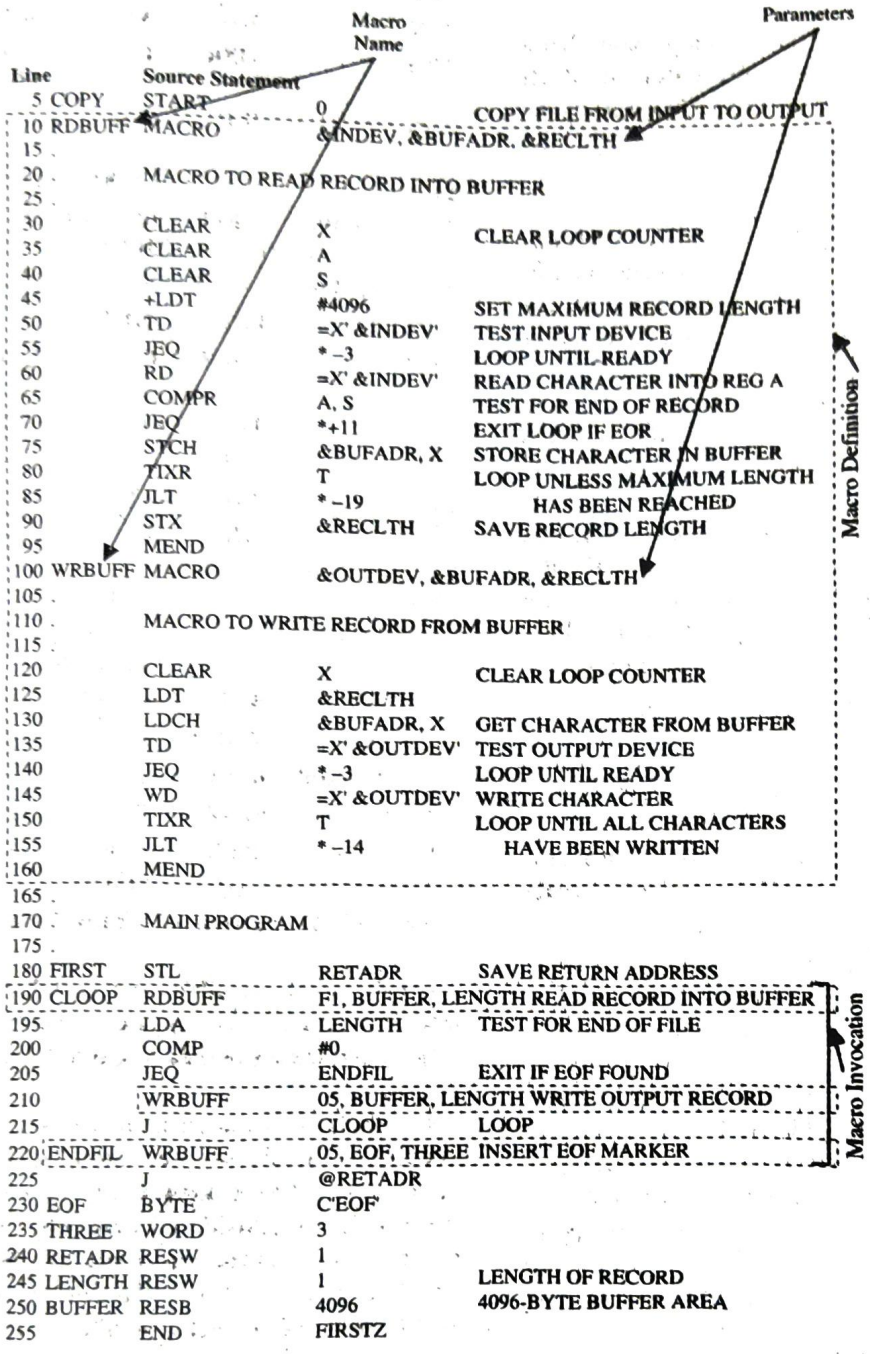
1) An ordinary string, which stands for itself.
2) The name of a formal parameter which is preceded by the character '&'.
3) The name of a preprocessor variable, which is also preceded by the character '&'.

During lexical expansion, strings of type 1 are retained without substitution. Strings of types 2 and 3 are replaced by the 'values' of the formal parameters or preprocessor variables. The value of a formal parameter is the corresponding actual parameter string

Ques 7) Write the SIC/XE program using macro instructions.

Ans: SIC/XE Program Using Macro Instructions

Figure 5.1 shows an example of a SIC/XE program using macro instructions. The definitions of these macro instructions (RDBUFF and WRBUFF) appear in the source program following the START statement.




```
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     FIRSTZ
```

Figure 5.1: Use of Macros in a SIC/XE Program

Two new assembler directives (MACRO and MEND) are used in macro definitions.

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 instruction.

In 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.

The MACRO directive are the statements that make up the body of the macro definition. The MEND assembler directive marks the end of the macro definition. **Figure 5.2** shows the output that would be generated. 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.

For example, in expanding the macro invocation on line 190, 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.

The comment lines within the macro body have been deleted. Note that the macro invocation statement itself has been included as a comment line. This serves as 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.

Note: The two invocations of WRBUFF specify different arguments, so they produce different expansions.

After macro processing, the expanded file (**figure 5.2**) can be used as input to the assembler. 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.

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
1901		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 5.2: Program from Figure 5.1 with Macros Expanded

Ques 8) Is it possible to use labels within the macro body? Explain your answer with the help of examples. Also illustrate a possible solution for the same. (2019[05])

Ans: Body of a macro instruction does not contain labels generally. So the source statement level contains relative addressing. **For example**, the definition of WRBUFF in **Figure 5.3** is considered here. On line 135, if we placed label for TD instruction, label will be defined twice once for each invocation of WRBUFF. This duplicate definition would prevent correct assembly of the resulting expanded program.

Jump instructions on line 140 and 155 were written using the relative operands * – 3 and * – 14. Because of this, it was not possible to place label on line 135. Only for sort jump instruction, it is i.e. "JEQ* –3".

For longer jump instruction, which spans several instructions, this type of natation is very inconvenient, error prone and difficult to read.

By creating special types of labels within macro instruction, many processor avoids these problems.

Figure 5.3 shows the generation of unique labels within a macro expansion.

```
25  RDBUFF  MACRO   &INDEV, &BUFADR, &RECLTH
30          CLEAR   X
35          CLEAR   A
40          CLEAR   S
45          +LDT    #4096
50  $LOOP   TD      =X'&INDEV'
55          JEQ     $LOOP
60          RD      =X'&INDEV'
65          COMPR   A,S
70          JEQ     $EXIT
75          STCH    &BUFADR, X
80          TIXR T
85          JLT     $LOOP
90  $EXIT   STX     &RECLTH
95          MEND
```

(a) RDBUF Macro Definition

RDBUF F1		BUFER, Length
30		CLEAR X
35		CLEAR A
40		CLEAR S
45		+ LDT # 4096
50	$AALOOP	TD = X' F1
55		JEQ $ AALOOP
60		RD = X' F1'
65		COMPR A,S
70		JEQ $EXIT
75		STCH & BUFFER, X
80		TIXR T
85		JLT $ AALOOP
90	$AAEXIT	STX LENGTH

(b) Resulting Macro Expansion

Figure 5.3:

Labels used within the macro body begin with the special character $in **Figure 1(a)**. In **Figure 1(b)**, each symbol beginning with $ has been modified by replacing $ with $AA.

Ques 9) Define the two approaches for macro processor design.

Ans: Approaches for Macro Processor Design

Approach 1: It is easy to design a two-pass macro processor in which all macro definitions are processed during the first pass, and all macro invocation statements are expanded during the second pass.

However, such a two-pass macro processor would not allow the body of one macro instruction to contain definitions of other macros (because all macros would have to be defined during the first pass before any macro invocations were expanded).

Approach 2: A one-pass macro processor that can alternate between macro definition and macro expansion is able to handle macros like those in **figure 5.4**. However, one-pass may be enough because all macros would have to be defined during the first pass before any macro invocations were expanded.

The definition of a macro must appear before any statements that invoke that macro. Moreover, the body of one macro can contain definitions of the other macro.

Consider the example of a Macro defining another Macro. In the example below, the body of the first Macro (MACROS) contains statement that define RDBUFF, WRBUFF and other macro instructions for SIC machine. The body of the second Macro (MACROX) defines the same macros for SIC/XE machine. A proper invocation would make the same program to perform macro invocation to run on either SIC or SIC/XE machine.

MACROS for SIC Machine

```
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}
```

Figure 5.4 (a)

MACROS for SIC/XE Machine

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

Figure 5.4 (b)

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

Ques 10) How could a recursive macro processor be implemented?

Or

Explain recursive macro expansion with example.

(2017 [05])

Or

What do you mean by recursive macro expansion? What are the possible problems associated with it? (05)

Ans: Recursive Macro Processor Implementation

There can be a condition that invocation of one macro is done by another. The purpose of RDCHAR 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.

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.

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. If a programming language that supports recursion is not available, the programmer must take care of handling such items as return addresses and values of local variables.

An **example** is shown in figure 5.5:

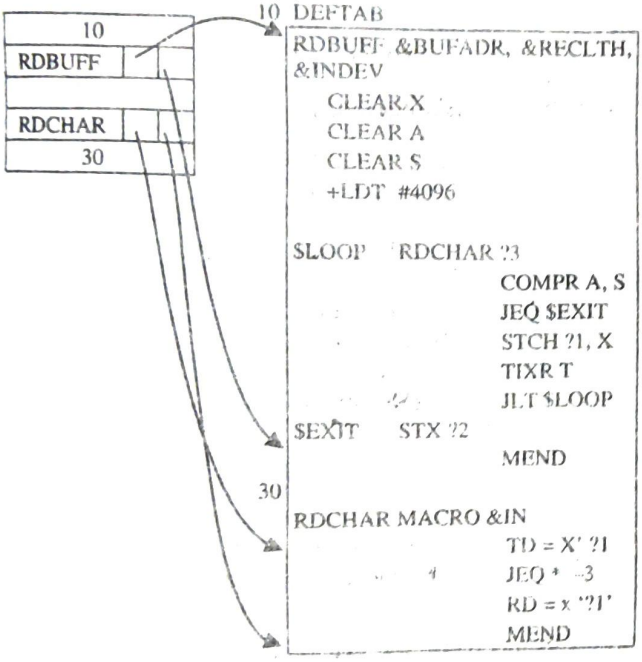


Figure 5.5

Problem Associated with Recursive Macro Expansion
Previous Macro Design Does not take care of recursive macro expansion.

Solutions

1) Write the macro processor in a programming language which automatically takes care of the recursive calls thus retaining the local variables.

2) If written in a language without recursion support, use a stack to take care of pushing and popping the local variables and return addresses thus retaining their values.

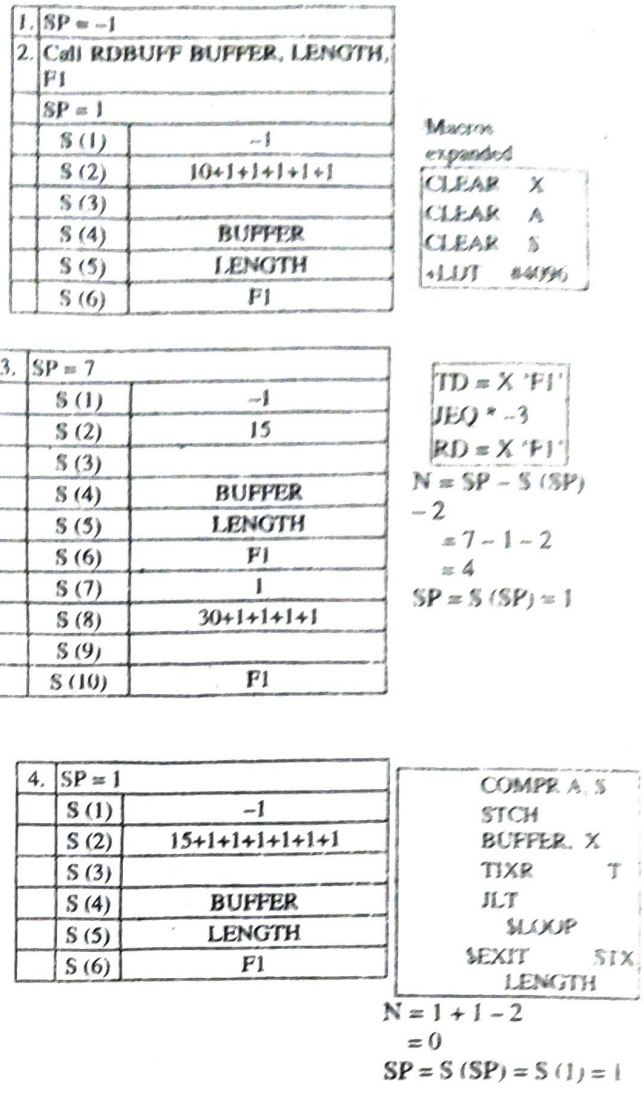


Figure 5.6: Recursive Macro Expansion for the Example in Figure 5.5

Ques 11) Discuss the one pass macro processor. Also write its algorithm and data structure used for this.

Or

Write the one pass macro processor algorithm.

(2020[07])

Or

Write and explain the algorithm for One Pass Macroprocessor. Briefly discuss various Data structures for a design of One Pass Macroprocessor.

Or

Describe the data structures used in a one pass macro processor algorithm. **(2017 [03])**

Or

Give the algorithm for a one pass macro processor.

(2017 [07])

Or

Explain the working of one pass Macro Processor.

(2018[07])

Or

What are the data structures required for a microprocessor algorithm? Explain the format of each.

(2019[04])

Or

Design an iterative algorithm for a one pass macroprocessor. **(2019[06])**

Or

How does a one pass macroprocessor handle recursive macro expansion? Explain with example. **(2020[06])**

Ans: One-Pass Macro Processor

A one-pass macro processor that alternate between macro definition and macro expansion in a recursive way is able to handle recursive macro definition.

Restriction

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

Data Structures

Data structures involved in the design of a macro processor are as follows:

1) **Macro Definition Table (DEFTAB):** It holds the macro definitions present in the input source file. As the macro processor scans through the source lines, it comes across the definition of newer macros.
2) **Macro Name Table (NAMTAB):** This table holds one entry for each macro defined in the input source file. Typical entries in this table include the following:
   i) **Name:** The name of the macro being defined.
   ii) **Num-of-parameters:** The number of parameters that the macro uses.
   iii) **DEFTAB -start-index:** The index of the DEFTAB from which the body of the macro is stored in MDT.
   iv) **DEFTAB -end-index:** The index of the DEFTAB upto which the body of the macro occupies in DEFTAB.
3) **Argument List Table (ARGTAB):** This is a table dynamically created upon encountering a macro call to hold the arguments passed. The content of this table is utilized during the expansion process.

The **figure 5.7** shows the different data structures described and their relationship:

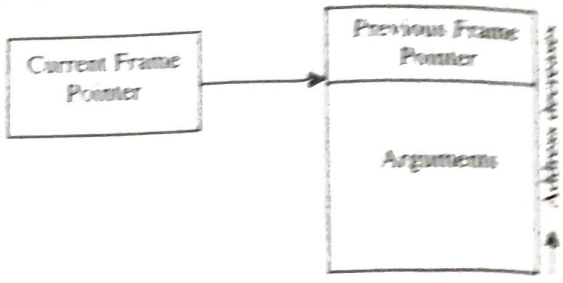
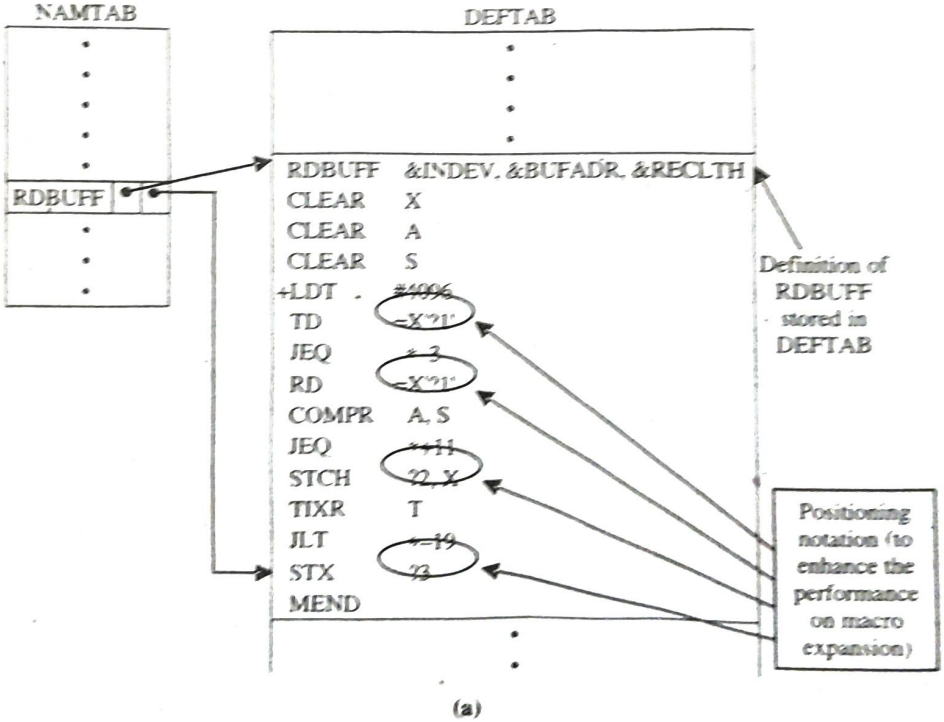


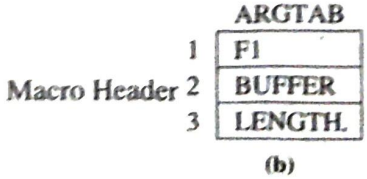
Figure 5.7: Argument List Array Structure

Figure 5.8(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 the parameters – &INDEV → ?1 (indicating the first parameter in the prototype), &BUFADR → ?2, etc.



(a)



(b)

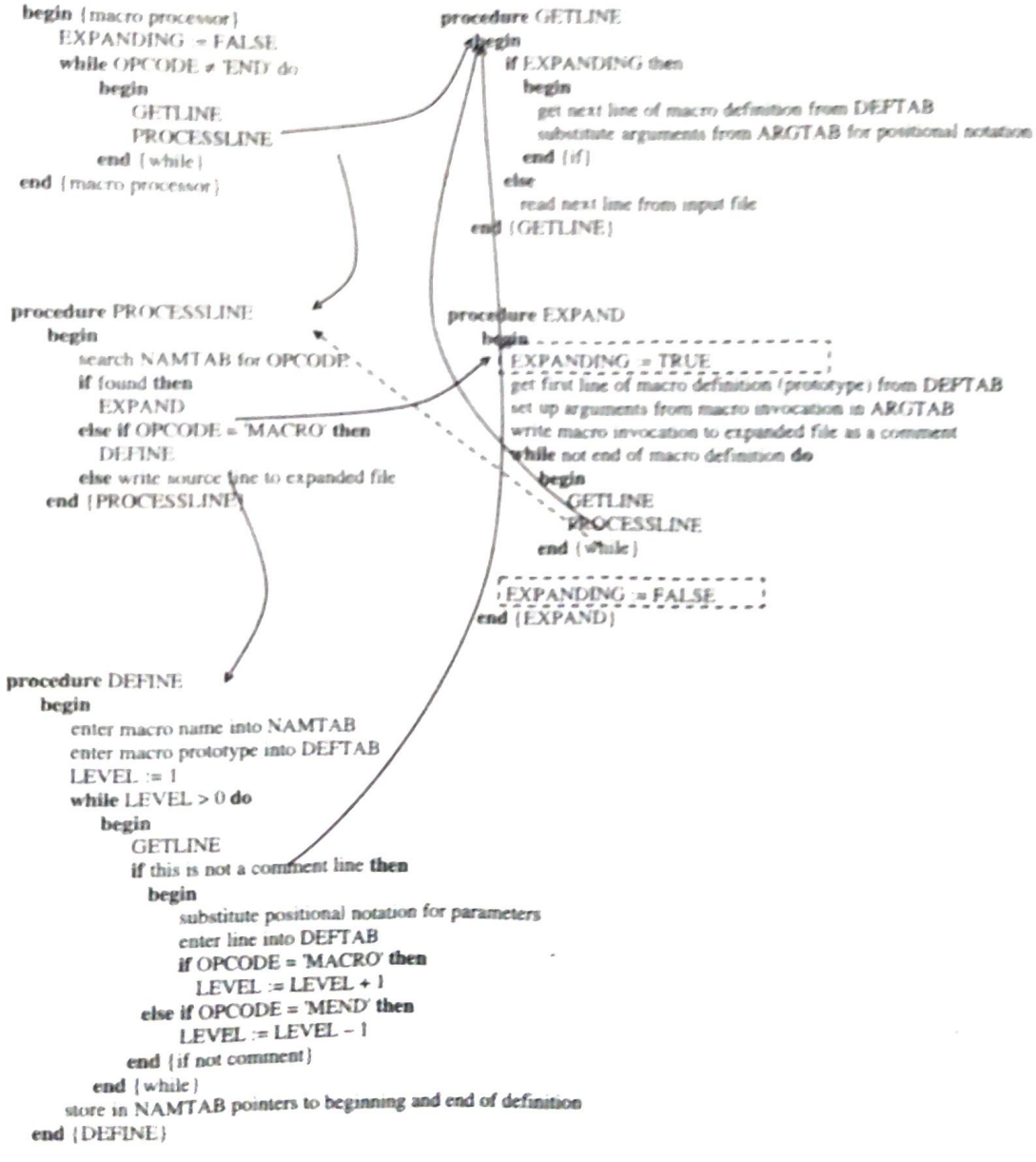
Figure 5.8

Figure 5.8(b) shows ARGTAB as it would appear during expansion of the RDBUFF statement on line 190. In this case (this invocation), the first argument is F1, the second is BUFFER, etc.

One-Pass Macro Algorithm

1) Procedure **DEFINE**:
   i) Being called when the beginning of a macro definition is recognised.
   ii) Make appropriate entries in DEFTAB and NAMTAB.
2) Procedure **EXPAND**:
   i) Being called to set-up the argument values in ARGTAB.
   ii) Expand a macro invocation statement.
3) Procedure **GETLINE**:
   i) Being called at several points to get a line in the algorithm:
   ii) The line may come from:
      a) The input file (EXPANDING = FALSE)
      b) The DEFTAB (EXPANDING = TRUE)
4) Counter **LEVEL** – Count the macro level (similar to match left and right parentheses):
   i) When a MACRO directive is encountered, LEVEL is advanced by 1.
   ii) When a MEND directive is encountered, LEVEL is decreased by 1.

Note: Most macro processors allow the definitions of commonly used macro instructions to appear in a standard system library (to make macro uses convenient).



Ques 12: Discuss different types of parameters used in macro with example.

Or

Write notes on keyword macro parameters, giving suitable example. (2017 (4R))

Or

Differentiate between Keyword and Positional macro parameters. (2018(R))

Ans: Types of Parameters in Macro

1) **Positional Parameter:** The parameters in the macro prototype are separated from the macro call are associated with each other according to their position. A positional parameter is written as &<parameter name>. For example, &SAMPLE where SAMPLE is the name of a parameter. The value of a positional formal parameter XYZ is determined by the rule of positional association as follows:

   i) Find the ordinal position of XYZ in the list of formal parameters in the macro prototype statement.

   ii) Find the actual parameter specification occupying the same ordinal position in the list of actual parameters in the macro call statement.

   Example: Consider the call
   INCR A, B, AREG

   In the following macro definition
```
MACRO
INCR     &MEM_VAL, &INCR_VAL, &REG
MOVER    &REG, &MEM_VAL
ADD      &REG, &INCR_VAL
MOVEM    &REG, &MEM_VAL
MEND
```

Following the rule of positional association, values of the formal parameters are

Formal Parameter	Value
MEM_VAL	A
INCR_VAL	B
REG	AREG

2) **Keyword Parameter:** For keyword parameters, <parameter name> is an ordinary string and <parameter kind> is the string '='. The actual parameter spec is written as
   <formal parameter name> = <ordinary string>

   The value of the formal parameter XYZ is determined by the rule of keyword association as follows:

   i) Find the actual parameter specification which has the form XYZ = <ordinary string>.

   ii) Let <ordinary string> in the specification be the string ABC. Then the value of the formal parameter is ABC.

   Example: The macro definition using keyword parameters:
```
MACRO
INCR_M          &MEM_VAL=,
&INCR_VAL=, &REG=
MOVER           &REG, &MEM_VAL
ADD             &REG, &INCR_VAL
MOVEM           &REG, &MEM_VAL
MEND
```
   Consider the following macro calls:
```
INCR_M    MEM_VAL = A, INCR_VAL = B, REG = AREG

INCR_M    INCR_VAL = B, REG = AREG, MEM_VAL = A
```
   are now equivalent.

Difference between Positional and Keyword Parameters

1) In positional parameters there is one to one correspondence between macro definition and macro call. The calling sequence should be same as defined sequence.

2) In keyword parameters sequence is not important as number of parameters used while calling for macro are does not important.

Ques 13: Define the machine independent features of macro processor.

Or

List the machine independent macro processor features. Also discuss different types of parameters used in macro with example.

Or

Explain the following:
1) **Conditional macro expansion**
2) **Generation of unique labels**

Or

Explain conditional macro expansion with an example. (2017 (R))

Or

Is it possible to include labels in the body of macro definition? Justify your answer. (2018(R))

Or

Write short note on concatenation of macro parameters within a character string. (2018(R))

Or

Explain the different types of conditional macro expansion statements and their implementation with examples. (2019(R))

Or

Write notes on conditional macro expansion. (2019(R))

Or

Explain the following machine independent macro processor features: (2020(R))
1) **Generation of unique labels.**
2) **Keyword macro parameters**

Ans: Machine Independent Macro Processor Features

The design of macro processors doesn't depend on the architecture of the machine. Some extended features of macro processor are:

1) **Concatenation of Macro Parameters:** Most macro processor allows parameters to be concatenated with other character strings. **For example,** consider that the parameter to such a macro instruction is named &ID. The body of the macro definition might contain a statement like

```
LDA     X&ID1
```

&ID is concatenated after the string "X" and before the string "1".

```
LDA     XA1  (&ID = A)
LDA     XB1  (&ID = B)
```

& is the starting character of the macro instruction; but the end of the parameter is not marked. So in the case of &ID1, the macro processor could deduce the meaning that was intended.

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

3) **Conditional Macro Expansion:** If the macro is expanded depends upon some conditions in macro definition (depending on the arguments supplied in the macro expansion) then it is called as conditional macro expansion. Macro definition is designed in such a way that it generates different instruction sequences when different relations hold between the actual parameters of a call.

Macro-Time Variables

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

If the value of this expression **TRUE**:

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

If the value of the expression is **FALSE**:

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

WHILE-ENDW Structure

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

ii) **TRUE:**
   a) The macro processor continues to process lines from DEFTAB until it encounters the next ENDW statement.
   b) When ENDW is encountered, the macro processor returns to the preceding WHILE, re-evaluates the Boolean expression, and takes action based on the new value.

iii) **FALSE:** The macro processor skips ahead in DEFTAB until it finds the next ENDW statement and then resumes normal macro expansion.

4) **Keyword Macro Parameters** The types of macro parameter are classified as:

i) **Positional Parameter:** Parameters and arguments are associated according to their positions in the macro prototype and invocation. If an argument is to be omitted, a null argument (two consecutive commas) should be used to maintain the proper order in macro invocation:

For example, RDBUFF BUFFER, LENGTH, , 80

It is not suitable if a macro has a large number of parameters, and only a few of these are given values in a typical invocation.

ii) **Keyword Parameter:** Each argument value is written with a keyword that names the corresponding parameter. Arguments may appear in any order. Null arguments no longer need to be used.

For example, GENER TYPE=DIRECT, CHANNEL=3

It is easier to read and much less error-prone than the positional method.

Ques 14) Differentiate between a Macro and a Subroutine. **(2019[03])**

Or

What is the difference between macro invocation and subroutine call? **(2020[03])**

Ans: Difference between Macro and Subroutine

Table 2 shows the difference between Macro and Subroutine:

Table 2: Difference between Macro and Subroutine

Macros	Procedure
The corresponding machine code is written every time a macro is called in a program.	The corresponding m/c code is written only once in memory
Program takes up more memory space.	Program takes up comparatively less memory space.
No transfer of program counter	Transferring of program counter is required.

Ques 15) Discuss the various design options for macro processors.

Ans: Macro Processors Design Options

1) **Recursive Macro Expansion:** There can be a condition that invocation of one macro is done by another. The purpose of RDCHAR 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.
An **example** is shown in **figure 5.9**:

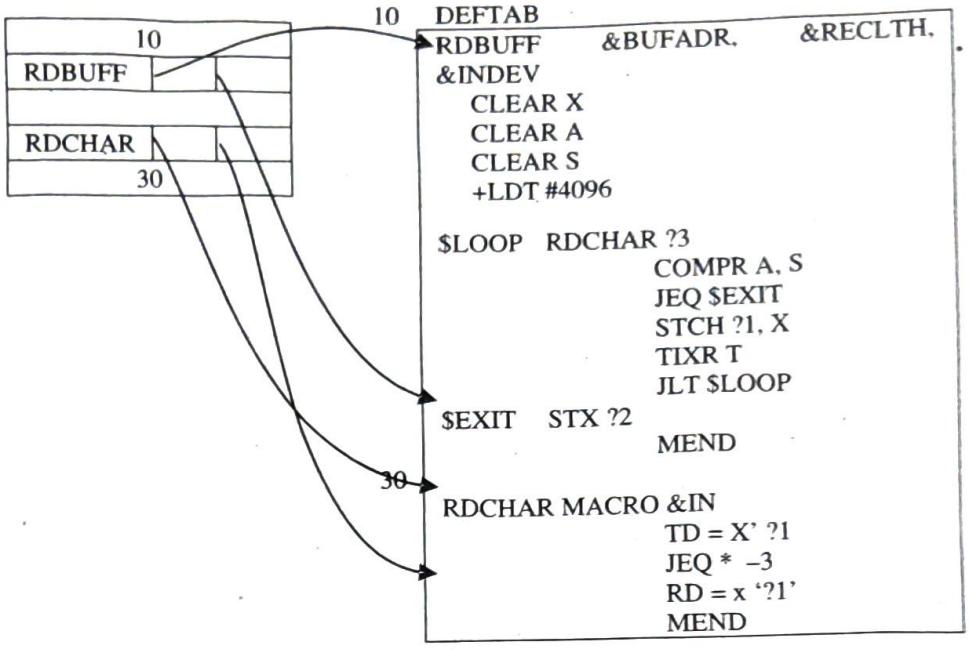


Figure 5.9

1.	SP = –1	
2.	Call RDBUFF BUFFER, LENGTH, F1	
	SP = 1	
	S (1)	–1
	S (2)	10+1+1+1+1+1
	S (3)	
	S (4)	BUFFER
	S (5)	LENGTH
	S (6)	F1

Macros expanded

```
CLEAR  X
CLEAR  A
CLEAR  S
+LDT   #4096
```

3.	SP = 7	
	S (1)	–1
	S (2)	15
	S (3)	
	S (4)	BUFFER
	S (5)	LENGTH
	S (6)	F1
	S (7)	1
	S (8)	30+1+1+1+1
	S (9)	
	S (10)	F1

```
TD = X 'F1'
JEQ * –3
RD = X 'F1'
```

N = SP – S (SP) – 2
= 7 – 1 – 2
= 4
SP = S (SP) = 1

4.	SP = 1	
	S (1)	–1
	S (2)	15+1+1+1+1+1+1
	S (3)	
	S (4)	BUFFER
	S (5)	LENGTH
	S (6)	F1

```
        COMPR A, S
        STCH
          BUFFER, X
        TIXR      T
        JLT       $LOOP
$EXIT             STX
          LENGTH
```

N = 1 + 1 – 2

Figure 5.10: Recursive Macro Expansion for the Example in Figure 5.9

2) **General-Purpose Macro Processors:** The most common use of macroprocessors is as an aid to assembler language programming. Often such macroprocessors are combined with, or closely related to, the assembler. Macroprocessors have also been developed for some high-level programming languages.

3) **Macro Processor within Language Translators:** The macro processors might be called pre-processors. That is, they process macro definitions 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. In this case macro processing functions are combined with the language translator itself. The simplest method of achieving this sort of combination is a line-by-line macro processor. Using this approach, the macro processor reads the source program statements and performs all of its functions. However, the output lines are passed to the language translator as they are generated (one at a time), instead of being written to an expanded source file. Thus, the macro processor operates as a sort of input routine for the assembler or compiler.

Ques 16) List the advantages and disadvantages of general purpose macro processor and macro processor wwith in language translators.

Ans: Advantages General-Purpose Macro Processors

1) 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.

2) The costs involved in producing a general-purpose macro processor are somewhat greater than those for developing a language-specific processor. However, this expense does not need to be repeated for each language; the result is a substantial overall saving in software development cost.

3) Savings in software maintenance effort should also be realized. Over a period of years, these maintenance costs may be even more significant than the original cost for software development.

Disadvantage General-Purpose Macro Processors

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

Advantages Macro Processor within Language Translators

1) It avoids making an extra pass over the source program (writing and then reading the expanded source file), so it can be more efficient than using a macro pre-processor.

2) Some of the data structures required by the macro processor and the language translator can be combined. Many utility subroutines and functions can be used by both the language translator and the macro processor. These include such operations as scanning input lines, searching tables, and converting numeric values from external to internal representations.

Disadvantages Macro Processor within Language Translators

1) They must be specially designed and written to work with a particular implementation of an assembler or compiler (not just with a particular programming language).

2) The costs of macro processor development must therefore be added to the cost of the language translator, which results in a more expensive piece of software.

DEVICE DRIVERS

Ques 17) What is device driver? Define the anatomy of device driver.

Or

What is a Device Driver? What are the major design issues of a Device Driver? (2018[05])

Or

Explain the anatomy of a device driver with the help of diagrams. (2019[05])

Ans: Device Driver

A device driver is a program routine that links a peripheral device to an operating system of a computer. It is essentially a software program that allows a user to employ a device, such as a printer, monitor, or mouse. It is written by programmers who comprehend the detailed knowledge of the device's command language and characteristics and contains the specific machine language necessary to perform the functions requested by the application. When a new hardware device is added to the computer, such as a CD-ROM drive, a printer, or a sound card, its driver must be installed in order to run it. The operating system "calls" the driver, and the driver "drives" the device.

The basic input/output (I/O) hardware features, such as ports, buses, and device controllers, accommodate a wide variety of I/O devices. To encapsulate the details and unique features of different devices, the kernel of an operating system is set up to use device driver modules. The device drivers present a uniform device-access interface to the I/O subsystem.

Each of the different types of I/O devices is accessed through a standardized set of functions--an interface. The tangible differences are encapsulated in kernel modules (i.e., device drivers) that internally are customized for each device, but that export and utilize one of the standard interfaces.

Design Issues of a Device Driver

To make the design of device drivers easier, we have divided the issues to be considered into three broad categories (Design issues of a device driver):

1) Operating system/Driver communications
2) Driver/Hardware Communications
3) Driver Operations

The first category covers all of the issues related to the exchange of information between the device driver and operating system. It also includes support functions that the kernel provides for the benefits of the device driver. The second covers those issues related to the exchange of the information between the device driver and the device it controls. This also includes issues such as how software talks to hardware – and how the hardware talks back.

The third covers issues related to the actual internal operation of the driver itself. This includes:

1) Interpreting commands received from the operating system
2) Scheduling multiple outstanding requests for service.
3) Managing the transfer of data across both interfaces (operating system and hardware)
4) Accepting and processing hardware interrupts
5) Maintaining the integrity of the driver's and the kernel's data structures.

Anatomy of Device Drivers

Device driver take on a special role in Linux kernel. They are "black boxes" that make a particular piece of hardware respond to a well-defined internal programing interface. The driver translates between the hardware commands understand by the device and the stylized programing interface used by the kernel. User activities are performed by means of a set of standardised calls that are independent of the specific operations that act on a real hardware is then the role of the device driver.A device driver has three sides:

1) One talks to the hardware and
2) One talk to the user.
3) While one side talks to the rest of the kernel.

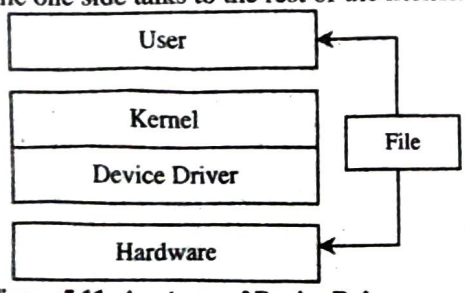


Figure 5.11: Anatomy of Device Drivers

A device driver performs the following jobs:

1) To accept request from the device independent software above to it.
2) Interact with the device controller to take and give I/O and perform required error handling.
3) Making sure that the request is executed successfully.

Ques 18) Define the various types of device drivers. Also differentiate character and block device driver.

Or

Differentiate between characters and block device drives. (2017, 2020 [05])

Or

Distinguish between Character and Block Device Drivers. (2018[05])

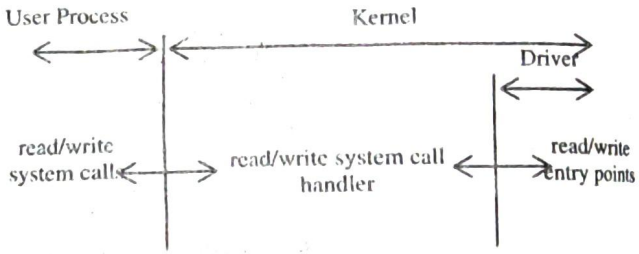
Or

Differentiate between Character and Block Device Drivers. (2019[04])

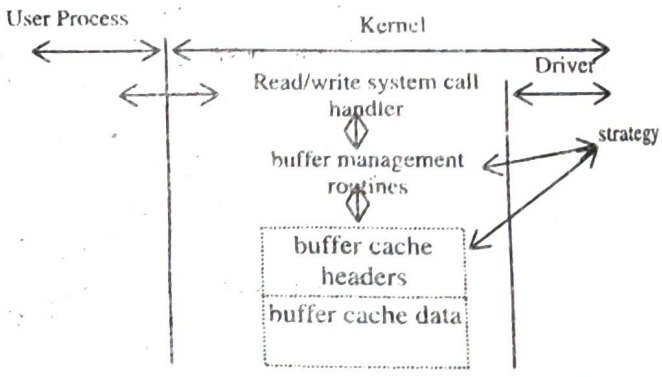
Ans: Types of Device Drivers

Depending upon how data is fetched by the devices, device drivers can be divided into two categories:

1) **Character Device Drivers:** Character device drivers are intended for those devices, which have a simple character based interface such as the keyboard. Such devices do not require any buffering as they send and receive data in single character chunks. Character device drivers normally perform I/O in a byte stream. They can also provide additional interfaces not present in block drivers, such as I/O control (ioctl) commands, memory mapping, and device polling.



2) **Block Device Drivers:** Devices that support a file system are known as block devices. Drivers written for these devices are known as **block** device drivers. Block device drivers are intended for those devices that have a data block based interface such as the hard disk. In order to improve speed and efficiency, these devices transfer the whole data buffer at one time.



Block device drivers take a file system request, in the form of a buf(9S) structure, and issue the I/O operations to the disk to transfer the specified block. The main interface to the file system is the strategy(9E) routine. Block device drivers can also provide a character driver interface that allows utility programs to bypass the file system and access the device directly. This device access is commonly referred to as the raw interface to a block device. **For example,** disks are commonly implemented as block devices.

Difference between Character and Block Device Driver

Character Device Drivers	Block Device Drivers
Perform input and output one character at a time	Transfers data in blocks, rather than one byte at a time
Support single hardware unit	Support more than one hardware unit
Have logical name assigned to them	Do not have file like logical names
Usually control terminal or printer	Usually control floppy disk, fixed or magneto tape drives
Examples: sound cards, parallel cards, serial ports	Examples: USB cameras, hard disk, Disk on key

Ques 19) Give the general design of a device driver. (2017 [05])

Or

Describe the general design of a device driver. (2020[05])

Or

Explain the general design and anatomy of a device driver with the help of diagrams. (2019[05])

Ans: Design of Device Drive

When you design your system it is very good if you can split up the software into two parts, one that is hardware independent and one that is hardware dependent, to make it easier to replace one piece of the hardware without having to change the whole application. In the hardware dependent part you should include:

1) Initialisation routines for the hardware
2) Device drivers
3) Interrupt drivers

The device can then be called from the application using RTOS standard calls. The RTOS creates during its own initialisation tables that contain function pointers to all the device driver's routines. But as device drivers are initialized after the RTOS has been initialized you can in your driver use the functionality of the RTOS. When you design your system, you also have to specify which type of device driver design you need. Should the device driver be interrupt driven, which is most common today, or should the application be polling the device? It depends on the device itself, but also on your deadlines in your system. But you also need to specify if your device driver should called synchronously or asynchronously.

1) **Synchronous Device Driver:** When a task calls synchronous device driver it means that the task will until the device has some data that it can give to the task **(figure 5.12)**. In this example the task is blocked on a semaphore until the driver hat been able to read any data from the device.

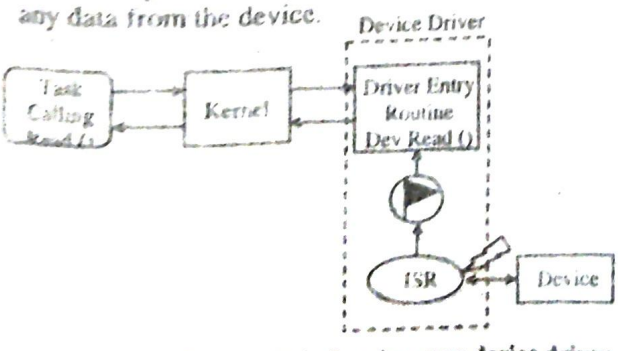


Figure 5.12: Synchronous device driver

The task calls the device driver via a kernel device call. The device entry routine gets blocked on a semaphore and blocks the task in that way. When an input comes from the device, it generates an interrupt and the ISR releases the semaphore and the device Entry routine returns the data to the task and the task continues it execution.

2) **Asynchronous Device Driver:** When a task calls an asynchronous device driver it means that the task will only check if the device has some data that it can give to the task **(figure 5.13)**. In this example the task is just checking if there is a message in the queue. The device driver can independently of the task send data into queue.

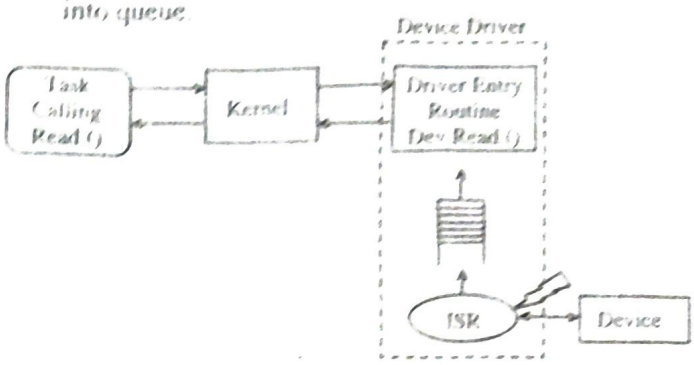


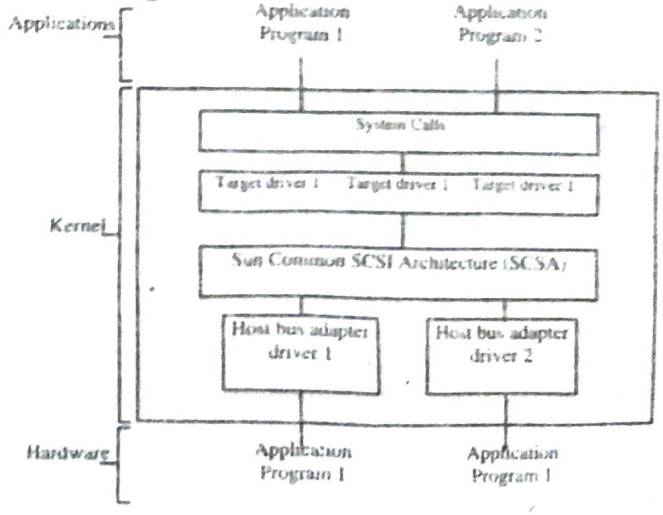
Figure 5.13: Asynchronous device driver

The task calls the device driver via a kernel device call. The device entry routine receives a message from the queue and returns the data to the task. When new data arrives from the device the ISR puts the data in a message and sends the message to the queue.

Ques 20) Describe the working of device driver.

Ans: Working of Device Driver

When we get a peripheral device such as printer, scanner, keyboard or modem, the device comes together with a driver CD which needs to be installed before the device starts working.



As soon we install the driver software into the computer, it detects and identifies the peripheral device and we become able to control the device with the computer. A device driver is a piece of software that allows your computer's

operating system to communicate with a hardware device, the driver is written for. Generally a driver communicates with the device through the computer bus which is used to connect the device with the computer. Device drivers works within the kernel layer of the operating system. Kernel is the part of the operating system that directly interacts with the physical structure of the system. Instead of accessing a device directly, an operating system loads the device drivers and calls the specific functions in the driver software in order to execute specific tasks on the device. Each driver contains the device specific codes required to carry out the actions on the device.

Consider an **example** of a printer, when it is connected to the computer and the specific device driver is installed, a device object is created on the computer which is designed to control the device. This device object represents the printer and its physical structure modes that allow your computer's operating system to control its functions.

When we choose an operation (like Control + P to print a document) on the printer then this command goes to the device driver through the kernel of the operating system. Resultantly a calling program invokes a routine in the device driver and the driver issues corresponding commands to the microcontrollers within the printer. Further these microcontrollers control the components of the printer like motors etc. to start printing the document.

Ques 21) What is a Device Driver Entry Point?

Ans: Device Driver Entry Point
An entry point is a function within a device driver that can be called by an external entity to get access to some driver functionality or to operate a device. Each device driver provides a standard set of functions as entry points.The illumos kernel uses entry points for these general task areas:

1) Loading and unloading the driver
2) Autoconfiguring the device – Autoconfiguration is the process of loading a device driver's code and static data into memory so that the driver is registered with the system.
3) Providing I/O services for the driver

Drivers for different types of devices have different sets of entry points according to the kinds of operations the devices perform. A driver for a memory-mapped character-oriented device, while a block driver does not support this entry.

TEXT EDITORS

Ques 22) What is the meaning of editing? Define various types of editors.

Or

What is editing? List the steps involved in document-editing process in an interactive user-computer dialogue.

Or

List out the main four tasks associated with the Document Editing Process. **(2018[02])**

Ans: Editing
An interactive editor is a computer program that allows a user to create and revise a target document. Document includes objects such as computer diagrams, text, equations tables, diagrams, line art, and photographs. In text editors, character strings are the primary elements of the target text.

Steps Involved in Document-Editing Process
Document-editing process in an interactive user-computer dialogue has 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.

The above **task** involves:

1) **Travelling:** Selection of the part of the document to be viewed and edited. It involves first travelling through the document to locate the area of interest such as "next screenful", "bottom", and "find pattern".
2) **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.
3) **Formatting:** Formatting then determines how the result of filtering will be seen as a visible representation (the view) on a display screen or other device.
4) **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. Manuscript-oriented editors operate on elements such as single characters, words, lines, sentences, and paragraphs; Program-oriented editors operate on elements such as identifiers, key words and statements.

Editing phase involves:

a. Insert b. Delete
c. Replace d. Move
e. Copy f. Cut
g. Paste, etc.

Types of Editors
The editors may be classified into two types as given below:

1) **Text Editors:** A text editor is a program, which allows the user to create the source program in the form of text into the main memory. Generally the user prepares HLL program or any source program. The creation, editing modification, deletion, updating of document or files can be done with the help of the text editor. The scope of the editing is limited to the text only.

Index	Element
1	22
2	33
3	66
4	05
5	44
6	89

05

Figure 5.14(a)

Index	Element
1	22
2	33
3	66
4	05
5	44
6	89

05

Figure 5.14(b)

2) **Graphical Editors:** Graphical editors are GUI (Graphical User Interface) based editors. These are much user-friendly. The Window-based editors are best example for the graphical editors. Notepad is an example for a graphical editor.

Ques 23) Define text editor. Also define the various types of text editors.

Ans: Text Editors

A text editor is a tool that allows a user to create and revise documents in a computer. Though this task can be carried out in other modes, the word text editor commonly refers to the tool that does this interactively. A popular text editor in IBM's large or mainframe computers is called **XEDIT**. In UNIX systems, the two most commonly used text editors are **Emacs** and **vi**. In personal computer systems, word processors are more common than text editors. However, there are variations of mainframe and UNIX text editors that are provided for use on personal computers. An **example** is KEDIT, which is basically XEDIT for Windows.

Types of Text Editors

1) **Line Editors:** During original creation lines of text are recognised and delimited by end-of-line markers, and during subsequent revision, the line must be explicitly specified by line number or by some pattern context. **For example,** edlin editor in early MS-DOS systems.

2) **Stream Editors:** The idea here is similar to line editor, but the entire text is treated as a single stream of characters. Hence the location for revision cannot be specified using line numbers. Locations for revision are either specified by explicit positioning or by using pattern context. **For example,** sed in Unix/Linux. Line editors and stream editors are suitable for text-only documents.

3) **Screen Editors:** These allow the document to be viewed and operated upon as a two dimensional plane, of which a portion may be displayed at a time. Any portion may be specified for display and location for revision can be specified anywhere within the displayed portion. **For example,** vi, emacs, etc.

4) **Word Processors:** Provides additional features to basic screen editors. Usually support non-textual contents and choice of fonts, style, etc.

Ques 24) Describe the salient aspects of text editor.

Ans: Salient Aspects of Text Editor

A text editor has to cover the following main aspects related to document creation, storage and revision:

1) **Interactive User Interface:** An important consideration of a document is its layout to a human reader. For this it is essential to present a document in a rectangular form, on the monitor screen (terminal) or via a printer (as an output of the interactive process). Further since a document can potentially be very large and since in a computer terminal can display only a limited size of a document at a time, hence it is necessary for a text editor to display only a portion (page) of a large document that cannot be displayed in entirety at once.

The process of creating and revising a document in a computer is called **editing**. For interactive editing a text editor displays a document on the terminal and accepts various types of input from the user through different types of input devices, viz., keyboard, mouse, data-tablet (flat, rectangular, electromagnetically sensitive panel over which a ball-point-pen-like stylus is moved and the coordinates of the stylus is sent to the system). Initially the text editor may display the starting portion of the document. During editing the user may give inputs to bring other portions to the display. Besides the displayed page, there is the notion of a point within the page, indicated by an easily distinguishable symbol called the cursor, where the next insertion, deletion or modification will be effective.

Many text editors allow pictures and other non-text information to be included in a document. Such editors provide mechanisms to either create/modify such information or embed such objects inside the document. The output of an editing process, i.e., the document, can be either saved as file(s) in the secondary storage or sent to some destination in the network, such as a printer or some remote host.

2) **Appropriate Format for Storing the Document in File in Secondary Storage:** The most natural way to visualise a document is as a planar layout of the information, within some bounded space. Thus, one possible format to save (store) the document in secondary storage is to save the attributes of each visible point (pixel) by traversing the layout in some order, say, left-to-right lines from top to bottom. However, this is not the done for obvious reasons. For textual information, there are widely used code-sets (e.g., ASCII, EBCDIC, ISCII, Unicode, etc.) that are used to encode the information. These code sets also contain codes for white spaces and new-line, and hence most textual information can be conveniently represented preserving the desired layout. Many text editors allow the appearance (script, size, colour and style) of text to be specified by the user. From the point of view of storing such variations of appearance

of text, this is usually achieved by storing extra information describing the appearance along with the actual text. Thus in secondary storage, the file corresponding to a document contains the textual information as well as attributes of the text, sometimes called **metadata** that controls the appearance of the text in output medium. Similarly, if the document has to contain non-textual information, that too is embedded using suitable metadata.

3) **Efficient Transfer of Information between the User Interface and the File in Secondary Storage:** The user interface of a text editor has to give emphasis to the visual appearance of the portion of the document that is displayed at any time, and convenience of the user to perform editing operations and viewing the document. On the other hand the format in which the document is stored in the secondary storage basically provides an unambiguous, static, and non-visual model of the document. Thus the text editor has to convert the user input into the file format, and file format into the display format. This conversion is one of the basic requirements for the text editor design.

Ques 25) Discuss about the criteria for the user interface design.

Or

Write notes on the user interface of a text editor.

(2017 [05])

Or

Describe the user interfaces used in a text editor.

(2019[05])

Ans: User Interface

Conceptual model of the editing system provides an easily understood abstraction of the target document and its elements. **For example,** Line editors – simulated the world of the key punch – 80 characters, single line or an integral number of lines, Screen editors – Document is represented as a quarter-plane of text lines, unbounded both down and to the right.

The user interface (UI) is concerned with, the input devices, the output devices and, the interaction language. The input devices are used to enter elements of text being edited, to enter commands. The output devices, lets the user view the elements being edited and the results of the editing operations and, the interaction language provides communication with the editor. A user interface plays important role in simplifying the interaction of a user with an application. UI functionalities have two important aspects. User interface is concerned with the input devices, the output devices, and the interaction language of the system, which are as follows:

1) **Input Devices:** These are used to enter elements of the text being edited, to enter commands, and to designate editable elements. These devices, as used with editors, can be divided into there categories, which are as follows:
   i) **Text or String Devices:** These are typically typewriter-like keyboards on which a user presses and releases keys, sending a unique code for each key. Text devices with arrow (cursor) keys can be used to simulate locator devices. 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.
   ii) **Button or Choice Devices:** These generate an interrupt or set a system flag, usually causing invocation of an associated application-program action. Such devices typically include a set of special function keys on an alphanumeric keyboard or on the display itself.
   iii) **Locator Devices:** These 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 for editing applications are the mouse and the data tablet. Locator devices usually incorporate one or more buttons that can be used to specify editing operations.

2) **Output Devices:** Formerly limited in range, output devices for editing are becoming more diverse. The output device lets the user view the elements being edited and the results of the editing operation. The first output devices were teletypewriters and other character-printing terminals that generated output on paper.

3) **Interaction Language:** The interaction language of a text editor is generally one of several common types. 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 for a description of less frequently used commands. In addition, the typing required can be time consuming, especially for inexperienced users. The function-key interface addresses these deficiencies. Here each command has associated with it a marked key on the user's keyboard.

Ques 26) Discuss the design of an Editor.

Ans: Design of an Editor

The structure of a text editor depends largely on the types of editing features and displaying capabilities that are to be supported. To implement the displaying capabilities, the semantics of the metadata that may be present in the document file needs to be implemented as display actions. **For example,** if the metadata implies a particular colour to be used for a segment of text, editor should invoke methods to affect that colour for the particular segment of text. Since at a time only a finite portion of the document can be displayed, such actions are to be taken for a portion of the information in the file. However, the user may specify some other portion to be displayed (through page

up, page-down, pattern search, etc.), in which case the display actions must be performed for that portion. Thus the editor program should keep track of the size of the display window, and the boundaries of the current displayed portion in terms of offsets from some fixed point in the document (such as line number of the first and the last displayed lines, etc.). The choice of data structure for the memory image (buffers) is important, since it has to support efficient insertion and deletion, while allowing the size of the document to vary from small to very large.

A simple 2-dimensional array with each row containing a line of text, may not suit for obvious reasons (what are the reasons?). A linked list may facilitate easy insertion and deletion, but having each letter in a single node in the linked list may be wasteful of memory. Also, user commands such as page-up, page-down, etc., may become inefficient.

Thus some kind of combination of array and linked list may have to be used. For example, the entries of an array may point to individual buffers for each line of the document. The buffer for a line may either be arrays or linked lists (with, say, a word in each node). For very large documents it may not be a good idea to hold the entire document in such buffers since only a small portion is displayed at a time and editing operations for a reasonable duration are likely to be in the neighbourhood of the displayed portion.

In such situations, a text editor may load only the required portion of the document into memory buffers, but be able to load any other portion as and when required. It needs to be remembered that in the memory image too it is essential to represent the metadata corresponding to the different portions of the document.

Ques 27) Explain the structure of typical editor with the help of diagram.

Or

Explain the structure of a text editor with the help of a diagram. **(2017 [05])**

Or

Draw the structure of a Typical Text Editor and describe the functions of each block. **(2018[08])**

Or

With the help of a diagram describe the structure of a text editor. **(2021[10])**

Or

Explain the structure of text editor with the help of diagram. **(2019[06])**

Ans: Structure of Text Editor

Most text editors have a structure similar to that shown in **figure 5.16**, regardless of the particular features they offer and the computers on which they are implemented.

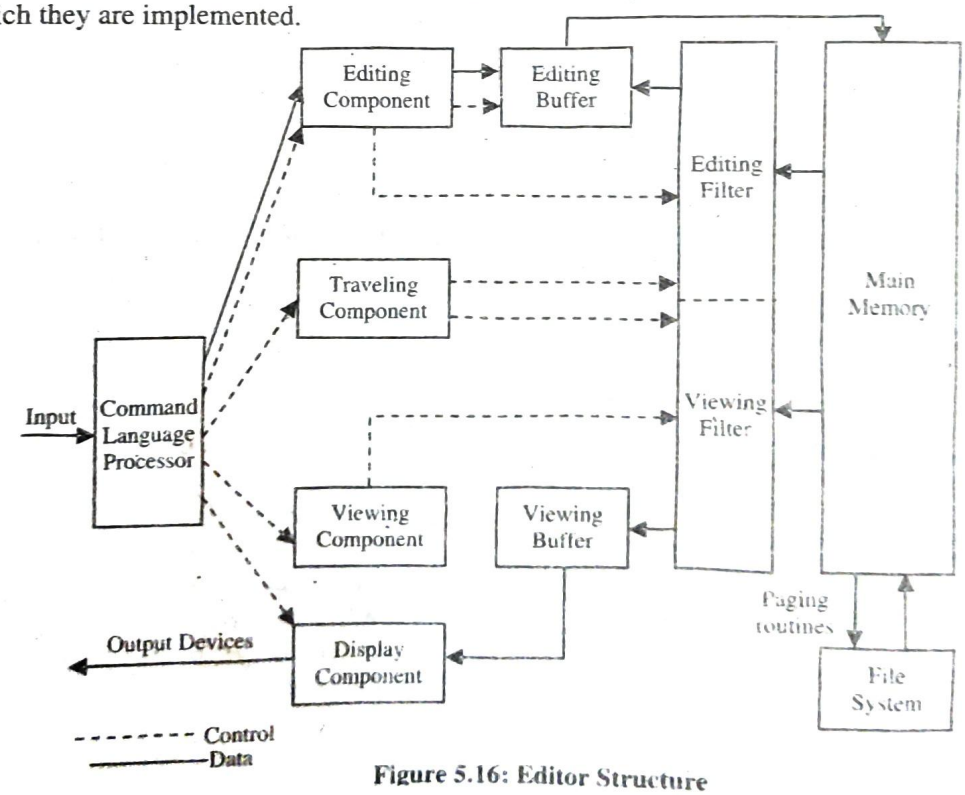


Figure 5.16: Editor Structure

The major components of editors are:

1) **Command Language Processor:** The command language processor accepts input from the user's input devices, and analyzes the tokens and syntactic structure of the commands.

2) **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.

3) **Travelling Component:** The travelling 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.

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.

5) **Display Component:** This produces a display by mapping the buffer to a rectangular subset of the screen, usually called a **window**.

6) **Editing and Viewing Buffers:** The editing and viewing buffers, while independent, can be related in many ways. In the simplest case, they are identical – the user edits the material directly on the screen. On the other hand, the editing and viewing buffers may be completely disjoint.

7) **Viewing and Editing 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. This viewing buffer is then passed to the display component of the editor.

DEBUGGERS

Ques 28) What do you mean by debugging? Explain.

Ans: Debugging

Debugging means locating (and then removing) bugs (faults in programs). In the entire process of program development errors may occur at various stages and efforts to detect and remove them may also be made at various stages. However, the word debugging is usually in context of errors that manifest while running the program during testing or during actual use. The most common steps taken in debugging are to examine the flow of control during execution of the program, examine values of variables at different points in the program, examine the values of parameters passed to functions and values returned by the functions, examine the function call sequence, etc. To overcome several such drawbacks of debugging by inserting extra statements in the program, there is various kind of tool called **debugger** that helps in debugging programs by giving the programmer some control over the execution of the program and some means of examining and modifying different program variables during runtime.

Ques 29) Discuss Interactive Debugging System.

Ans: Interactive Debugging Systems

Debugging is a methodical process of finding ad reducing the number of debugs, or defects in a computer program. An interactive debugging system provides programmers with facilities that aid in the testing and debugging of programs. The approaches to debugging are:

1) Print statements
2) Printing to log files
3) Sprinkling the code with assertions
4) Using post-mortem dumps
5) Having programs that provide function call stacks on termination
6) Profiling
7) Heap checking
8) System call tracing tools
9) Interactive source-level debugging
10) Reverse execution etc.

The basic principles of debugger are:

1) **Heisenberg Principle:** This principle says that the debugger must intrude on the debuggee in a minimal way.

2) **Truthful Debugging:** The debugger must be truthful so the programmer always trusts it.

3) **Providing Context Information:** The debugger should provide the program context information during debugging. Program context information are
   i) Source code, ii) Stack back-track, iii) Variable values, iv) Thread information

Ques 30) Discuss the basic operations supported by a debugger.

Ans: Basic Operations Supported by a Debugger

A debugger provides an interactive interface to the programmer to control the execution of the program and observe the proceedings. The program (executable file) to be debugged is provided as an input to the debugger. The basic operations supported by a debugger are:

1) **Breakpoints:** Setting breakpoints at various positions in the program. The breakpoints are points in the program at which the programmer wishes to suspend normal execution of the program and perform other tasks.

2) **Examining Values of Different Memory Locations:** When the execution of a program is suspended, the contents of specified memory locations can be examined. This includes local variables (usually on the stack), function parameters, and global (extern) variables.

3) **Examining the Contents of the program Stack:** The contents of the program stack reveals information related to the function call sequence that is active at that moment.

4) **Depositing Values in Different Memory Locations:** While the execution of the debugged program is not underway (yet to start or suspended at a breakpoint), the programmer can deposit any value in the memory locations corresponding to the program variables, parameters to subroutines, and processor registers.

Ques 31) Describe the important functions and capabilities of an interactive debugging system.

Or

Explain the debugging functions and debugging capabilities.

Or

Write notes on the debugging functions and capabilities of an interactive debugging system. (2017 [05])

Or

Describe the functions and capabilities of an Interactive Debugging System. (2018[06])

Ans: Debugging Functions and Debugging Capabilities

Following points highlight the debugging functions and capabilities:

1) Debugging system should provide functions such as tracing and traceback.
2) Tracing can be used to track the flow of execution logic and data modifications. It can also be based on conditional expressions.
3) Traceback can show the path by which the current statement was reached. It can also show which statements have modified a given variable or parameter.
4) Debugging system have a good program display capabilities. It must be possible to display the program being debugged, complete with statement numbers.
5) The system should save all the debugging specifications across such a recompilation, so the user does not need to reissue all of these debugging commands.
6) Debugging system must be sensitive to the specific language being debugged so that procedural, arithmetic and conditional logic can be coded in the syntax of that language.
7) The debugger should be able to switch its context when a program written in one language calls a program written in a different language.
8) A debugging system must be able to deal with optimised code. Application code used in production environments is usually optimised.

Ques 32) What are the different components of a debugging system? Explain.

Ans: Components of Debugging System

The major components of debugging system are:

1) **Debugging Database System:** Debugging database system contains a database with persistent debugging information (a . . . n) for a variety of executions of programs. The term persistent is used here in to mean that the debugging information remains available in the database after execution of the program has ceased. Database is managed by debug database server, which responds to queries from the other components of debugging system by writing to or reading from debug information as required by the query. Conceptually, debug data base server has two sets of clients:
   i) **Debugger client** which provides update queries containing debug information to database, and
   ii) **One or more user interface clients** which provide queries that read selected debug information from database.

   Debug data base system may be specially implemented for system or it may be one of the many commercial database systems. One of the functions of debug database server is to coordinate reads and writes of database. In some debugger systems, it will be enough simply to ensure that a read query never attempts to read data which is not yet available in database; in others, it may be necessary to ensure that what is read is the last complete write done by debugger client. One method of coordinating reads and writes is to treat each read and write as a transaction and to use standard database transaction processing techniques; where consistency requirements are less stringent, the overhead of transaction processing can be avoided and sufficient coordination of reads and writes may be achieved by ordering the writes such that information which a given record in the data base depends on is written to the database before the record itself is written, thereby ensuring that all the information needed to respond to a query for which the given record is a result is in the database by the time the record itself can be queried.

2) **Debugger Client:** Debugger client is a debugger which is executing a program being debugged in debugger process. For purposes of the present discussion, a debugger client may be any entity which executes a computer program in such a fashion that information about the execution which would not normally be available to a user of the program becomes available. Thus, debugger client may be implemented expressly for system or it may be any kind of existing interactive debugger. Debugger client may have access to both the source code and object code for program. Debugger client may interpret source code, but more generally, it will execute object code and use source code to make debug info more understandable to the user of the debugger.

   Debug information obtained from source code and by means of execution of object code is written to debug database by means of update queries to debug database server. In some embodiments there may be more than one debugger client; **for example,** a programmer may want to watch the behavior of two closely-cooperating programs, or there may be debugger clients specialized for different programming languages or for different programming problems.

3) **User Interface Client:** Each user interface client receives inputs from a user, responds to some inputs by making a read query (Rquery) for debug data base server, and responds to the results of Rquery by formatting the results and providing the formatted results to the user as formatted output. The forms of the inputs received and the outputs provided by a given interface client depends on the kind of interactive interface employed by the user. Again, the interactive interface may be any presently existing or future interactive interface. Debugger client and the user interface clients further communicate with each other by means of control channel, which may be any arrangement which permits transfer of messages between debugger client and a user interface client. User interface client uses control channel to transfer debugger commands to debugger client, while

debugger client uses control channel to transfer debugger event messages to user interface client. **For example,** a first debugger command may instruct the debugger concerning the kinds of information it is to output, while another may instruct the debugger to stop execution of program at a predetermined point.

Ques 33) List out the criteria that should be met by the user interface of an Efficient Debugging System. (2018[04])

Ans: Criteria Required to Meet by User Interface of an Efficient Debugging System

1) User friendly.
2) Use full screen display and windowing display.
3) Use menu bars:
   i) Menus should have the heading of the task.
   ii) Possible to go directly to menus and it should not entire hierarchy.
4) Command Language:
   i) It must be clear, logical, simple syntax.
   ii) Should minimise punctuations, eg : { }, / , etc ....
5) Available Help Facility.

Ques 34) Explain the relationship of debugger with other parts of the system.

Ans: Relationship with Other Parts of the System

The important requirement for an interactive debugger is that it always be available.

1) Debugger must appear as part of the run-time environment and an integral part of the system.
2) When an error is discovered, immediate debugging must be possible.
3) The debugger must communicate and cooperate with other operating system components such as interactive subsystems.
4) Debugging is more important at production time than it is at application- development time. When an application fails during a production run, work dependent on that application stops.
5) The debugger must also exist in a way that is consistent with the security and integrity components of the system.
6) The debugger must coordinate its activities with those of existing and future language compilers and interpreters.

Ques 35) Discuss the various methods of debugging.

Or

How induction and deduction methods are used for debugging?

Or

How backtracking is used for debugging?

Or

Describe any two commonly used debugging methods. (2017 [05])

Or

Explain the following methods of debugging: (2020[10])

1) Induction

2) Deduction

3) Backtracking

Or

List and explain the different debugging techniques. (2019[05])

Ans: Methods/Techniques Used for Debugging

Following are the method used for debugging:

1) **Debugging by Brute Force:** The most common and least effective method of program debugging is by "brute force". It requires little thought and is the least mentally taxing of all the methods. The brute-force methods are characterized by either debugging with a memory dump; scattering print statements throughout the program, or debugging with automated debugging tools. Using a memory dump to try to find errors suffers from the following drawbacks:
   i) Establishing the correspondence between storage locations and the variables in the source program is difficult.
   ii) Massive amounts of data, most of which is irrelevant, must be dealt with.
   iii) A dump shows only the static state of the program at only one instant in time. The dynamics of the program (i.e., state changes over time) are needed to find most errors.
   iv) The dump is rarely produced at the exact time of the error. Hence the dump does not show the program's state at the time of the error.
   v) No formal procedure exists for finding the cause of an error analyzing a storage dump.

2) **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 5.17** and described by Myers as follows:
   i) **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.
   ii) **Organise 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		

iii) **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.

iv) **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.

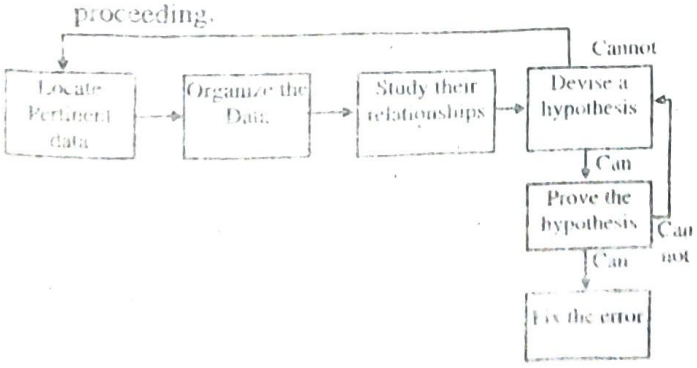


Figure 5.17: Inductive Debugging Process

3) **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 5.18** and also described by Myers as follows:

i) **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.

ii) **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 attempt 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.

iii) **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.

iv) **Prove the Remaining Hypothesis:** This vital step is identical to the fourth step in the induction method.

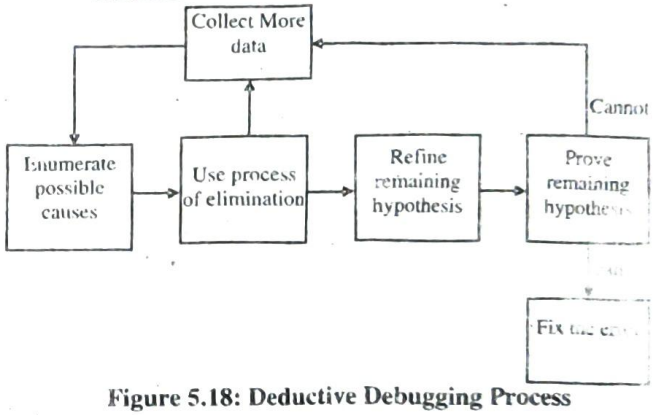


Figure 5.18: Deductive Debugging Process

4) **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.

5) **Debugging by Testing:** The use of additional test cases is another very powerful debugging method which is often used in conjunction with the induction method to obtain information needed to generate a hypothesis and/or to prove a hypothesis and with the deduction method to eliminate suspected causes, refine the remaining hypothesis, and/or prove a hypothesis. The test cases for debugging differ from those used for integration and testing in that they are more specific and are designed to explore a particular input domain or internal state of the program.

Test cases for integration and testing tend to cover many conditions in one test, whereas test cases for debugging tend to cover only one or a very few conditions. The former are designed to detect the error in the most efficient manner whereas the latter are designed to isolate the error most efficiently