# MODULE 5

CST 305 SYSTEM SOFTWARE

Prepared by Asha Baby

# Syllabus

- Macro Pre-processor -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

# Macro Instructions

**A macro instruction (macro)**

- It is simply a notational convenience for the programmer to write a shorthand version of a program.
- It represents a **commonly used group of statements** in the source program.
- It is replaced by the macro processor with the corresponding group of source language statements.
- This operation is called **“expanding the macro”**

**For example:**

- Suppose it is necessary to save the contents of all registers before calling a subroutine.
- This requires a sequence of instructions.
- We can define and use a macro, SAVEREGS, to represent this sequence of instructions.

## Macro Processor

- Its functions 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.
- Therefore, the design of a macro processor generally is machine independent.

# Macro processors are used in

- Assembly language .
- High-level programming languages, e.g., C or C++ .
- OS command languages.
- General purpose.

# Format of macro definition

**A macro can be defined as follows**

MACRO - MACRO pseudo-op shows start of macro definition.

Name [List of Parameters] – Macro name with a list of formal parameters.

……. ……. ……. - Sequence of assembly language instructions.

MEND - MEND (MACRO-END) Pseudo shows the end of macro definition

# BASIC MACROPROCESSOR FUNCTIONS

- The fundamental functions common to all macro processors are:

- Macro Definition
- Macro Invocation
- Macro Expansion

# Macro Definition and Expansion

- Two new assembler directives are used in macro definition:
- MACRO: identify the beginning of a macro definition
- MEND: identify the end of a macro definition

Prototype for the macro:

- Each parameter begins with '&'

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

Body: The statements that will be generated as the expansion of the macro.

| 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
```

```
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.

- It shows an example of a **SIC/XE program using macro Instructions.**
- This program defines and uses two **macro instructions, RDBUFF and WRDUFF .**
- The functions and logic of RDBUFF macro are similar to those of the RDBUFF subroutine.
- The WRBUFF macro is similar to WRREC subroutine.
- Two **Assembler directives (MACRO and MEND)** are used in macro definitions.
- The first MACRO statement identifies the beginning of macro definition.

- The Symbol in the label field (RDBUFF) is the name of macro, and entries in the operand field identify the parameters of macro instruction.

- In our macro language, each parameter begins with character &, which facilitates the substitution of parameters during macro expansion.

- The macro name and parameters define **the pattern or prototype** for the macro instruction used by the programmer.

- The **macro instruction definition has been deleted** since they have been no longer needed after macros are expanded.

- Each **macro invocation statement** has been expanded into the statements that form the body of the macro, with the arguments from macro invocation substituted for the parameters in macro prototype.

- The arguments and parameters are associated with one another **according to their positions.**

# 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.
- The processes of macro invocation and subroutine call are quite different.
- 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.

- The macro invocation statements treated as comments and the statements generated from macro expansion will be assembled as though they had been written by the programmer.

# 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.
- Comment lines within the macro body have been deleted, but comments on individual statements have been retained.
- Macro invocation statement itself has been included as a comment line.

| 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.

- In expanding the macro invocation on line 190, the argument F1 is substituted for the parameter and 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 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.
- After macro processing the **expanded file can be used as input to assembler.**
- The macro invocation statement 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.

# Note

- In the above program, macro instructions have been written so that the body of the **macro contains no labels.**
- The line 140 contains the statement JEQ *-3 and the line 155 contains JLT *-14.
- The corresponding statements in the WRREC subroutine are JEQ WLOOP and JLT WLOOP.
- If such a label appeared on the line 135 of the macro body **, it would be generated twice on line 210d and 220d.**

- This would result in an **error (duplicate label definition) when the program is assembled.**
- To avoid duplication of symbols we have eliminated labels from the body of macro definitions.
- But the statements like JLT *-3 is considered to be poor programming practice and also **inconvenient and error prone method.**

# Macro Processor Algorithm and Data Structures

- 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 second pass.

- Such a two pass macro processor **would not allow the body of one macro instruction to contain definitions of other macros.**

```
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

- Defining MACROS or MACROX does not define RDBUFF and the other macro instructions.
- These definitions are processed only when an invocation of MACROS or MACROX is expanded.
- A **one pass macro processor** that can alternate between macro definition and macro expansion is able to handle macros like these.
- There are **3 main data structures involved in our macro processor.**

# Definition table (DEFTAB)

- The macro definition themselves are stored in definition table (DEFTAB), which contains the macro prototype and statements that make up the macro body.

- Comment lines from macro definition are not entered into DEFTAB because they will not be a part of macro expansion.

# Name table (NAMTAB)

- References to macro instruction parameters are converted to a positional entered into NAMTAB, which serves the index to DEFTAB.

- For each macro instruction defined, NAMTAB contains pointers to beginning and end of definition in DEFTAB.

# Argument table (ARGTAB)

- The third Data Structure in an argument table (ARGTAB), which is used during expansion of macro invocations.
- When macro invocation statements are recognized, the arguments are stored in ARGTAB according to their position in argument list.
- As the macro is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body

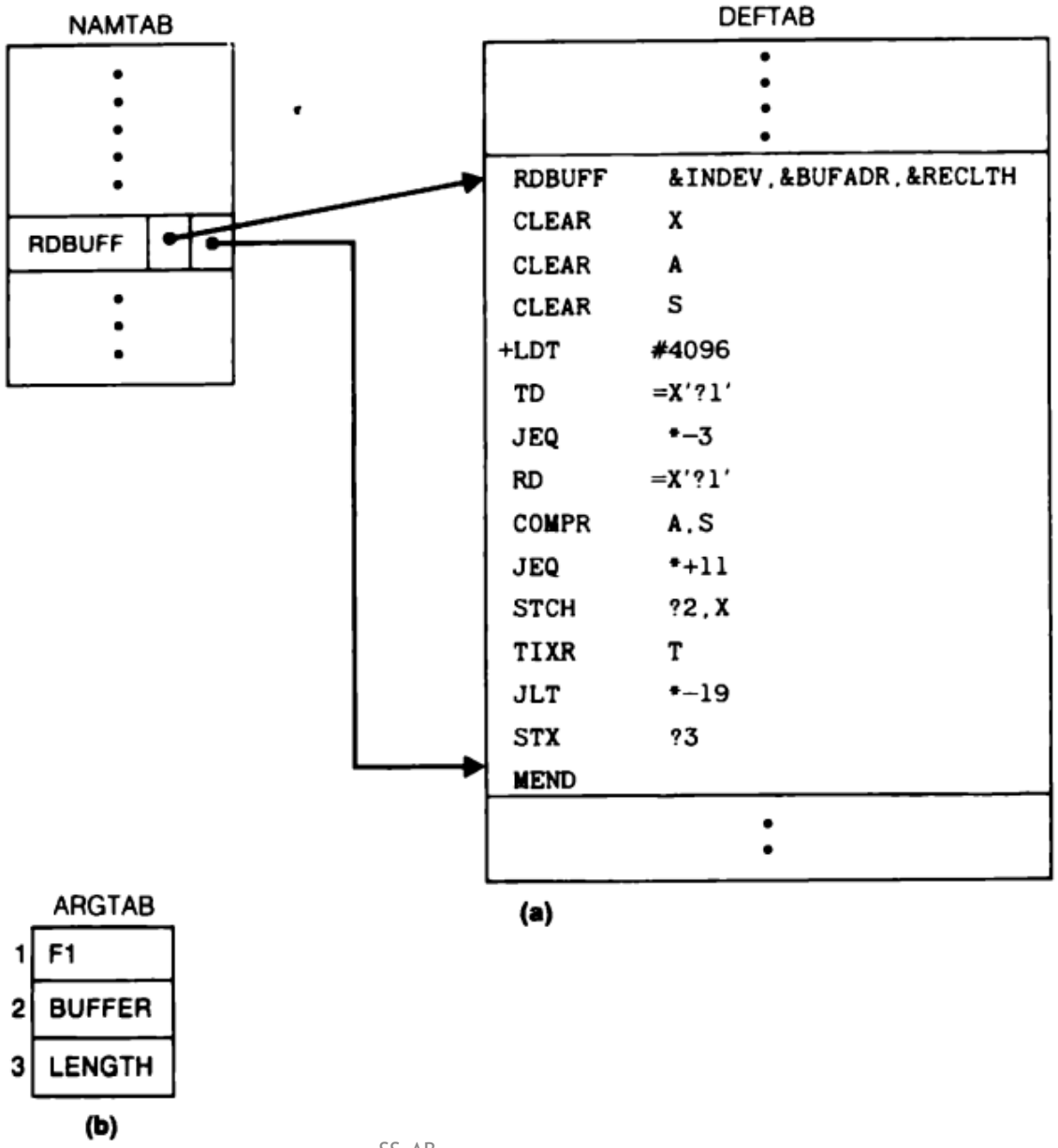



**Figure 4.4** Contents of macro processor tables for the program in Fig. 4.1: (a) entries in NAMTAB and DEFTAB defining macro RDBUFF, (b) entries in ARGTAB for invocation of RDBUFF on line 190

- The **position notation is used for the parameters.**

- The parameter &INDEV has been converted to ?1, &BUFADR has been converted to ?2.

- When the ?n notation is recognized in a line from DEFTAB, a simple indexing operation supplies the property argument from ARGTAB.

| 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
```

```
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.

# Algorithm for a one pass macro processor

```
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}
```

# Machine Independent Macro Processor Features

- Machine independent macro processor features are extended features that are not directly related to architecture of computer for which the macro processor is written.

- Concatenation of Macro Parameter.
- Generation of Unique Labels.
- Conditional Macro Expansion.
- Keyword Macro Parameters.

# Concatenation of Macro Parameter

- 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 by XB1, XB2,XB3,…. Etc.

- If similar processing is to be performed on each series of variables , the programmer 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 etc).
- LDA X&ID1
- Here the parameter &ID is concatenated after the character string X and before the character string 1.

- The beginning of the macro parameter is identified by the starting symbol &.
- However the end of the macro parameter is not marked.
- Most macro processors deal with this problem by providing a special concatenation operator.
- In the SIC macrolanguage **this operator is the →.**

- **LDA X&ID→1**

- Here the end of the parameter &ID is clearly identified.

- Pre-concatenation
  - LDA X&ID1
- Post-concatenation
  - LDA X&ID→1
- Example: Figure 4.6

```
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)

# Generation of Unique Labels

- (a) illustrates one technique for generating unique labels within a macro expansion.
- A definition of the RDBUFF macro is shown.
- Labels used within the macro body begin with **special character $.**
- (b) shows a macro invocation statement and the resulting macro expansion.
- Unique labels are generated within macro expansion.

- Each symbol beginning with $ has been modified by replacing $ with **$AA.**
- 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.
- For succeeding macro expansions, xx will be set to **AB, AC etc.**

- Example
  - JEQ *-3
  - inconvenient, error-prone, difficult to read
- Example Figure 4.7
    - $LOOP  TD=X'&INDEV'
  - **1st call:**
    - $AALOOP  TD=X'F1'
  - **2nd call:**
    - $ABLOOP  TD=X'F1'

```
25    RDBUFF   MACRO   &INDEV,&BUFADR,&RECLTH
30             CLEAR   X              CLEAR LOOP COUNTER
35             CLEAR   A
40             CLEAR   S
45             +LDT    #4096          SET MAXIMUM RECORD LENGTH
50    $LOOP    TD      =X'&INDEV'     TEST INPUT DEVICE
55             JEQ     $LOOP          LOOP UNTIL READY
60             RD      =X'&INDEV'     READ CHARACTER INTO REG A
65             COMPR   A,S            TEST FOR END OF RECORD
70             JEQ     $EXIT          EXIT LOOP IF EOR
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
```

**(a)**

```
.          RDBUFF   F1,BUFFER,LENGTH

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

(b)

**Figure 4.7** Generation of unique labels within macro expansion.

# Conditional Macro Expansion

- In the early example of macro instruction , 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 ,were unchanged.
- Most macro processor can also modify the sequence of statements generated for a macro expansion ,depending on the arguments supplied in the macro invocation.

- The below program shows a definition of a macro RDBUFF , the logic and functions are similar to those previously discussed.

- But it has two additional parameters &EOR, which specifies a hexadecimal character code that marks the end of a record and &MAXLTH which specifies the maximum length record that can be read.

- From line 26 through 28, this SET statement assign a value 1 to &EORCK.
- The symbol &EORCK is a macro time variable , which can be used to store working values during the macro expansion.
- Any symbol that begins with the character & and that is not a macro instruction parameter is assumed to be a macro time variable.
- All such variables are initialized to a value of 0.
- If an argument corresponding to &EOR is not null , the variable &EORCK is set to 1.
- Otherwise it retains its default value of 0.

- The figure b-d shows the expansion of three different macro invocation statements.

- Macro-time conditional structure o IF-ELSE-ENDIF o WHILE-ENDW

# Implementation of Conditional Macro Expansion (IF-ELSE-ENDIF Structure)

- A symbol table is maintained by the macro processor.
- This table contains the values of all macro-time variables used.
- Entries in this table are made or modified when SET statements are processed.
- This table is used to look up the current value of a macro-time variable whenever it is required.

```
25    RDBUFF    MACRO   &INDEV,&BUFADR,&RECLTH,&EOR,&MAXLTH
26              IF      (&EOR NE '')
27    &EORCK    SET     1
28              ENDIF
30              CLEAR   X               CLEAR LOOP COUNTER
35              CLEAR   A
38              IF      (&EORCK EQ 1)
40              LDCH    =X'&EOR'        SET EOR CHARACTER
42              RMO     A,S
43              ENDIF
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
```

**(a)**

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

**(b)**

**Figure 4.8** Use of macro-time conditional statements.

```
                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)**

- The testing of the condition and looping are done while the macro is being expanded.
- When an **IF statement is encountered** during the expansion of a macro, the specified Boolean expression is evaluated.

If value is TRUE

- The macro processor continues to process lines from DEFTAB until it encounters the next ELSE or ENDIF statement.
- If ELSE is encountered, then skips to ENDIF

FALSE

- The macro processor skips ahead in DEFTAB until it finds the next ELSE or ENDLF statement.

- Testing of Boolean expression in IF statement occurs 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 statements and the conditional macro expansion directives have been removed.

# Implementation of Conditional Macro Expansion (WHILE-ENDW Structure)

- When a WHILE statement is encountered during the macro expansion , the specified Boolean expression is evaluated.
- If the value of this expression is FALSE , the macro processor skip ahead in DEFTAB until it finds the next ENDW statement, and the resumes normal macro expansion.
- If the value of the Boolean expression is TRUE , the macro processor continues to process 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.
- Here the programmer can specify the a list of end of records characters.
- In the macro invocation statement there is a list (00,03,04) corresponding to the parameter &EOR.
- Any one of the character is interpreted as marking the end of the record.
- Here the parameter &MAXLTH has been deleted , the maximum length will always be 4096.

- Here macro time variable &EORCT has previously been set to the value %NITEMS(&EOR).
- %NITEMS is a macro processor function that returns as its value the number of members in an argument list.
- For example , if the argument corresponding to is &EOR (00,03,04) then %NITEMS (&EOR) returns the value 3.
- The macro time variable &CTR is used to count the number of times the line following the WHILE statement have been generated.
- The value of &CTR is initialized to 1 and incremented by 1 each time through the loop.

```
 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)**

(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)

**Figure 4.9** Use of macro-time looping statements.

# Keyword Macro Parameters

- We have already discussed positional parameters .
- That is parameters and arguments were associated with each other according to their position in the macro prototype and 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 an null argument to maintain the correct argument positions.

# For example

- Suppose a macro instruction GENER has 10 possible parameters, but in a particular invocation of the macro only the 3rd and 9th parameters are to be specified.
- The macro invocation statement is **GENER , , DIRECT , , , , , , 3.**
- 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.

# Keyword parameters

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

- If the 3rd parameter is named &TYPE and 9th parameter is named &CHANNEL,

macro invocation would be

**GENER TYPE=DIRECT,CHANNEL=3.**

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

# Consider the example

- Here each parameter name is followed by equal sign, which identifies a keyword parameter and a default value is specified for some of the parameters.

```
25    RDBUFF    MACRO    &INDEV=F1,&BUFADR=,&RECLTH=,&EOR=04,&MAXLTH=4096
26              IF       (&EOR NE '')
27    &EORCK    SET      1
28              ENDIF
30              CLEAR    X              CLEAR LOOP COUNTER
35              CLEAR    A
38              IF       (&EORCK EQ 1)
40              LDCH     =X'&EOR'       SET EOR CHARACTER
42              RMO      A,S
43              ENDIF
47              +LDT     #&MAXLTH       SET MAXIMUM RECORD LENGTH
50    $LOOP     TD       =X'&INDEV'     TEST INPUT DEVICE
55              JEQ      $LOOP          LOOP UNTIL READY
60              RD       =X'&INDEV'     READ CHARACTER 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
```

**(a)**

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

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

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

**(c)**

# Macro processor design option

- One pass macro processor algorithm does not work properly if a macro invocation statement appears within the body of macro instruction.

**Different design options are there**

- Recursive macro expansion.
- General purpose macro processor.
- Macro processing within language translators.

# Recursive macro expansion

- We have already dealt with definition of one macro instruction by another.
- We have not dealt with the invocation of one macro by another.
- Figure below shows an example of such macros.

```
10    RDBUFF    MACRO    &BUFADR,&RECLTH,&INDEV
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    $LOOP     RDCHAR   &INDEV         READ CHARACTER INTO REG A
65              COMPR    A,S            TEST FOR END OF RECORD
70              JEQ      $EXIT          EXIT LOOP IF EOR
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
```

**(a)**

```
 5    RDCHAR    MACRO    &IN
10    .
15    .         MACRO TO READ CHARACTER INTO REGISTER A
20    .
25              TD       =X'&IN'          TEST INPUT DEVICE
30              JEQ       *-3             LOOP UNTIL READY
35              RD       =X'&IN'          READ CHARACTER
40              MEND
```

**(b)**

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

**(c)**

**Figure 4.11** Example of nested macro invocation.

# Algorithm for a one pass macro processor

```
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}
```

- One pass macro processor is not work when the invocation of one macro by another.
- The above shows an example of such a use of macros.
- (a) is the definition of RDBUFF and RDCHAR macro is used to read one character from a specified device into register A.
- (b) is the definition of macro RDCHAR.
- Unfortunately , the macro processor design we have discussed previously can not handle such invocations of macros within macros.

- Suppose that the one pass macro processor algorithm were applied to the macro invocation statement
- RDBUFF BUFFER,LENGTH,F1
- The procedure **EXPAND** will be called when the macro was recognized.
- The arguments from the macro invocation would be entered into **ARGTAB** as follows.

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

- The Boolean variable **EXPANDING** would be set to **TRUE,** and expansion of the macro invocation statement would begin.
- The processing would proceed normally until 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 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 the middle of expanding macro when it encountered the RDCHAR statement.
- And also the arguments 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.
- These problems are not difficult to solve if the macro processor is being written in a programming language(Pascal or C) that allows recursive calls.
- The compiler would be sure that previous values of any variable declared within a procedure were saved when that procedure was called recursively.

- 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.

- The algorithm and data structures are same for a one macro processor except the EXPAND and GETLINE procedure.

# General purpose macro processor

- The most common use of macro processor is an aid to assembler language programming.
- Macro processor have also been developed for some high level programming languages.

**Goal**

- Macro processors that do not dependent on any particular programming language, but can be used with a variety of different languages.

# Advantages

- Programmers do not need to learn many macro languages.
- So much of the time and expense involved in training are eliminated.
- Although its development costs are somewhat greater than those for a language specific macro processor, this expense does not need to be repeated for each language, thus save substantial overall cost.
- Saving in software maintenance cost.

# Disadvantages

- Large number of details must be dealt with in a real programming language.
- Situations in which normal macro parameter substitution should not occur.
- **Example each language has its own methods for identifying comments.**
- Pascal and C uses special characters to mark the start and end of the comment.

- But Ada uses special characters to mark only the start of a comment.
- The comment is automatically terminated at the end of the source line.
- FORTRAN uses a special symbol to flag entire line as a comment.
- **Facilities for grouping together terms, expressions, or statements are also different for different programming languages.**

- Some languages use keyword such as begin and end for grouping statements.
- Other use special characters such as { and }.
- Many languages use parenthesis for grouping terms and expressions.
- The rules for doing this may vary from one language to another.
- Tokens, e.g., identifiers, constants, operators, keywords different for variety languages.

- In some languages multiple character operators such as ** in FORTRAN and := in Pascal.
- Problems may arise if these are treated by a macro processor as two separate characters rather than a single operator.
- Another potential problem with general purpose macro processors involve the syntax used for macro definition and macro invocation statements.

# Macro processing within language translators

- We have already discussed about **preprocessors.**
- 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.
- But here we discuss an alternative : **combining the macro processor functions with the language translator itself.**

- Line by line macro processor.
- Integrated macro processor.

# Line by line macro processor

- The macro processor reads the source program statements and performs all of its functions as previously described.
- 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 file.

**Advantages**

- 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 macro preprocessor.

- Some of the data structures required by the macro processor and the language translator can be combined.
- For example OPTAB in an assembler and NAMTAB in the macro processor could be implemented in the same table.
- A line by line macro processor also makes it easier to give diagnostic messages that are related to the source statement containing error.

# Integrated macro processor

- An integrated macro processor can potentially make use of any information about the source program that is extracted by the language translator.

**As an example in FORTRAN DO 100 I = 1,20**

- a DO statement:
- DO: keyword
- 100: statement number
- I: variable name

**DO 100 I = 1 – An assignment statement**

DO100I: variable (blanks are not significant in FORTRAN)

- Thus the proper interpretation of the characters DO, 100 can not be decided until the rest of the statements is examined.
- A FORTRAN compiler must be able to recognize and handle situations such as this.
- But it would be very difficult for an ordinary macro processor (not integrated with a compiler) to do so.
- An integrated macro processor can support macro instructions that depend upon the context in which they occur.

# 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 cost of macro processor development is 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.

# Device drivers

- Anatomy of a device driver.
- Character and block device drivers.
- General design of device drivers.

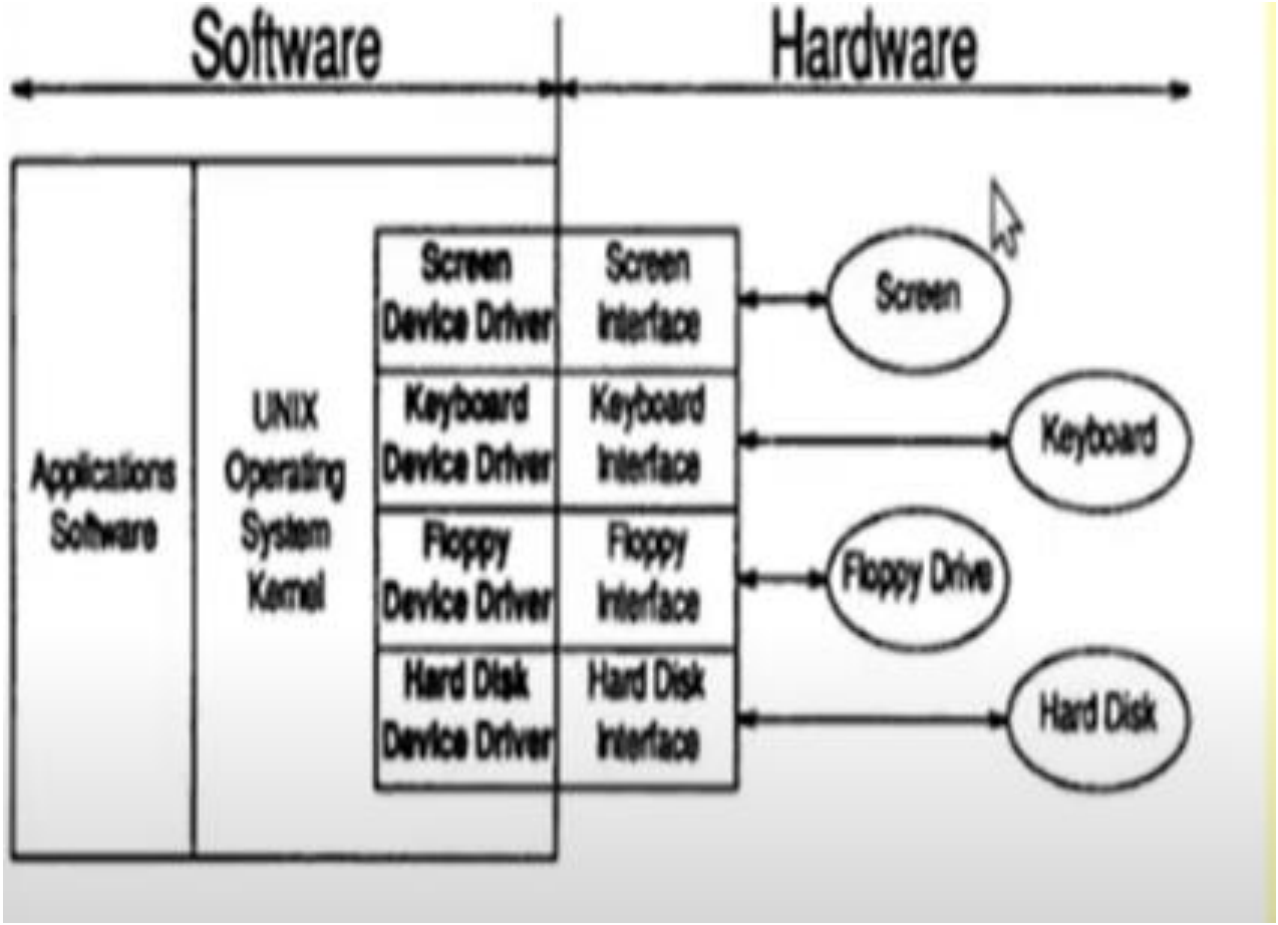
# Functions of device driver

- **Device driver** is a computer program that operates or controls a particular type of device that is attached to a computer.

- A driver provides a software interface to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details about the hardware being used.

- A driver communicates with the device through the computer bus or communications subsystem to which the hardware connects.
- When a calling program invokes a routine in the driver, the driver issues commands to the device (drives it).
- Once the device sends data back to the driver, the driver may invoke routines in the original calling program.
- Drivers are **hardware dependent and operating-system-specific.**

Driver provides software interface to hardware interface.

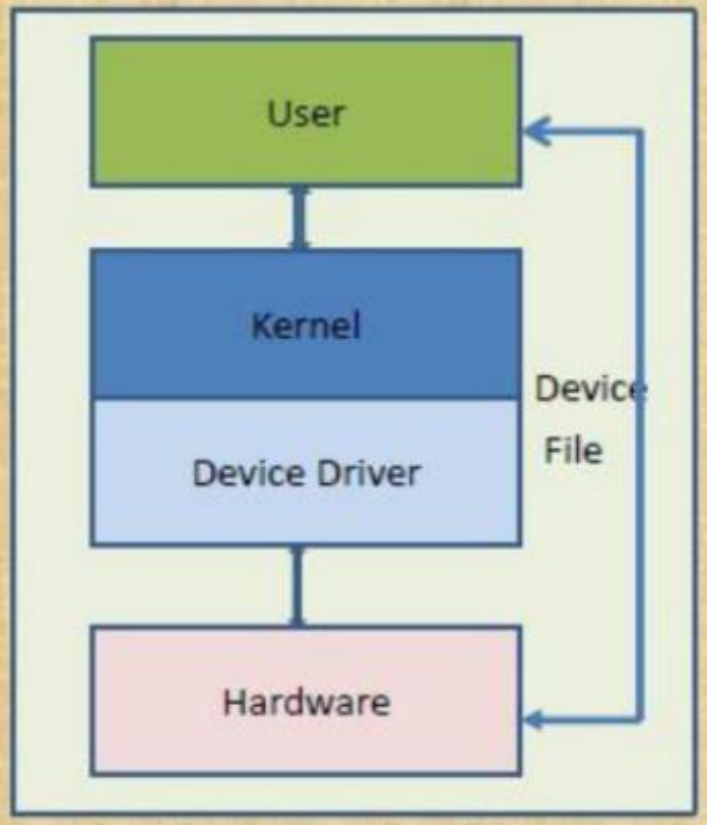

Software
Hardware
Applications Software
UNIX Operating System Kernel
Screen Device Driver
Screen Interface
Screen
Keyboard Device Driver
Keyboard Interface
Keyboard
Floppy Device Driver
Floppy Interface
Floppy Drive
Hard Disk Device Driver
Hard Disk Interface
Hard Disk

## A device driver simplifies programming by

- Acting as the translator between a hardware device and the applications or operating systems that use it.
- Programmers can write the high-level application code independently of whatever specific hardware device.

# Anatomy of device driver

- A device driver has three sides
  - One side talks to the rest of the kernel
  - One talks to the hardware and
  - One talks to the user

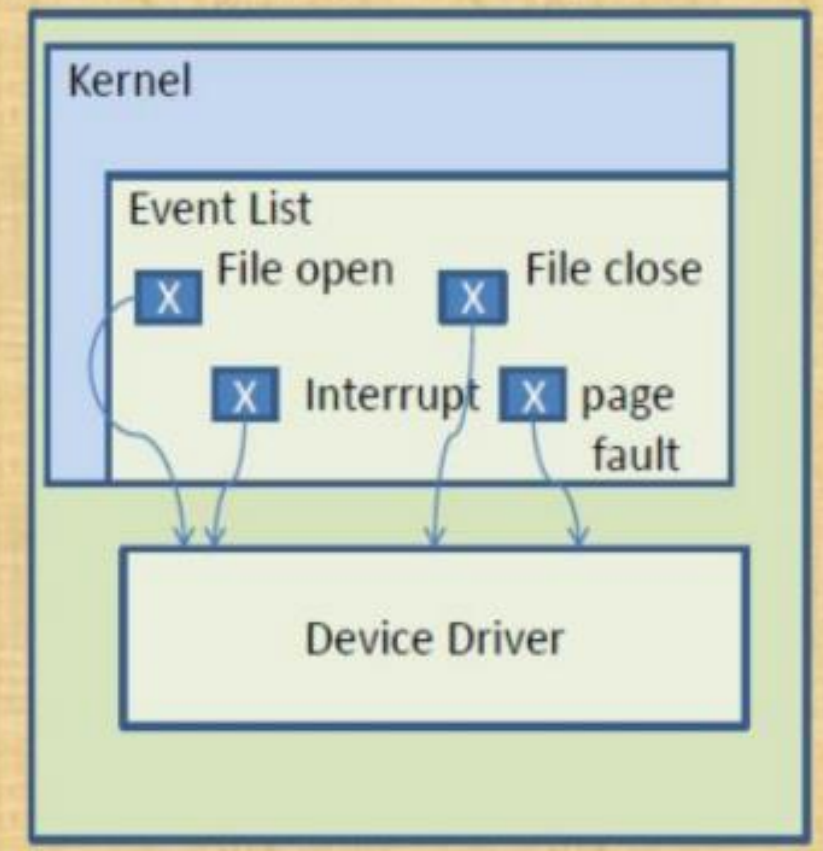


# User interface to the device driver

- A very important Unix design decision was to represent most of the "system objects" as "files"
- It allows applications to manipulate all "system objects" with the normal file API (open, read, write, close, etc.)
- So, devices had to be represented as "files" to the applications
- This is done through a special artefact called a **device file**
- It a special type of file, that associates a file name visible to user space applications to the triplet (type, major, minor) that the kernel understands
  All device files are by convention stored in the /dev directory

# Kernel interface to the device driver

- In order to talk to the kernel, the driver registers with subsystems to respond to events. Such an event might be the opening of a file, closing a file, a page fault, the plugging in of a new USB device, etc.



# Types of device drivers

**There are two main types of devices under all Unix systems:**

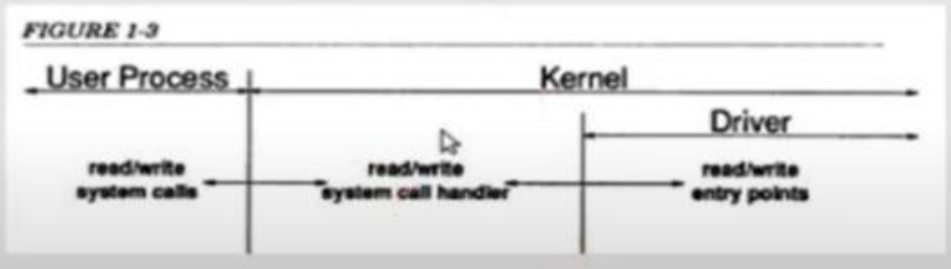
- Character device drivers.
- Block device drivers.

- The **main task** of any device driver is to perform I/O, and many character device drivers do what is called byte-stream or character I/O.
- The driver transfers data to and from the device without using a specific device address.
- This is in **contrast to block device drivers**, where part of the file system request identifies a specific location on the device.

# Character device driver

## Character Drivers

- It can handle I/O requests of arbitary size.
- Mostly used devices, Pritners



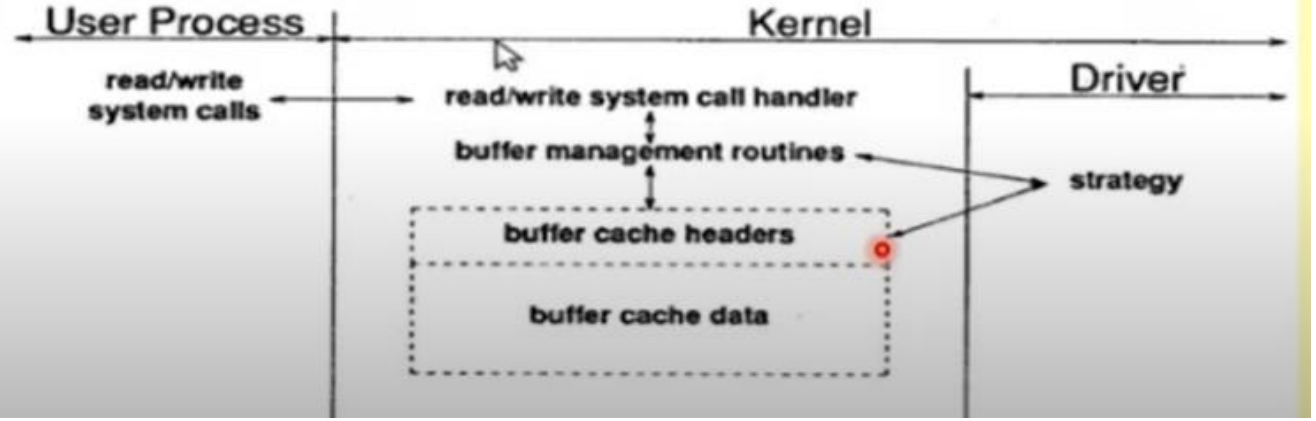
- A character device driver is one that transfers data directly to and from a user process.
- Character device drivers provide procedures for directly reading and writing data from and to the device they drive, block devices do not.
- A **Character Device** is a device whose driver communicates by sending and receiving single characters (bytes).
- Example - serial ports, parallel ports, sound cards, keyboard.
- They can also provide additional interfaces not present in block drivers, such as I/O control commands, memory mapping, and device polling.

# Block device drivers

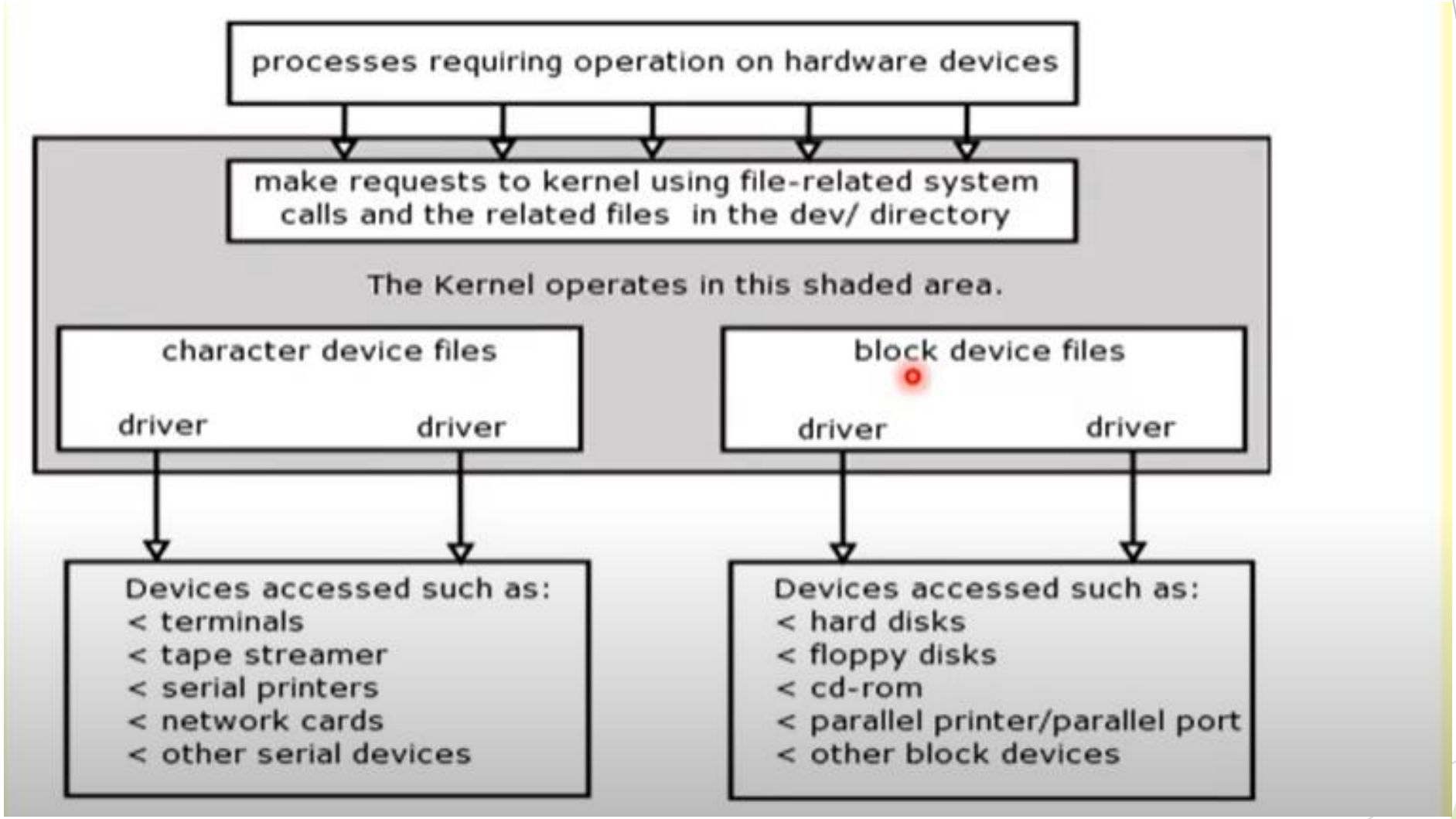
## Block Drivers

- It communicate with OS through a collection of fixed-sized bufferss.
- For example, disks are commonly implemented as block devices.

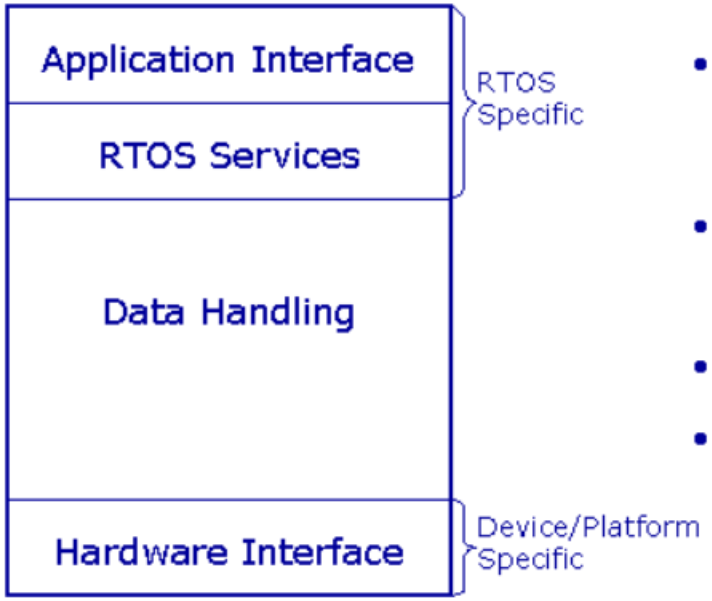
*FIGURE 1-2*



- Block devices gives a single request( ) procedure which is used for both reading and writing.
- A **Block Device** is a device whose driver communicates by sending entire blocks of data. Example - hard disks, USB cameras, Disk-On-Key.
- Block device drivers manage devices with physically addressable storage media, such as disks.
- The only relevant difference between a **character device** and a **regular file** is that you can always move back and forth in the regular file, whereas most character devices are just data channels, which you can only access sequentially.


processes requiring operation on hardware devices
make requests to kernel using file-related system calls and the related files in the dev/ directory
The Kernel operates in this shaded area.
character device files
driver
driver
block device files
driver
driver
Devices accessed such as:
< terminals
< tape streamer
< serial printers
< network cards
< other serial devices
Devices accessed such as:
< hard disks
< floppy disks
< cd-rom
< parallel printer/parallel port
< other block devices

# General design of device drivers




- Application Interface
  - Calling Conventions
  - Parameters, e g
    - Device identification
    - Data
- RTOS Services
  - Semaphores
  - Queues
  - Memory Allocation
- Data Handling
  - Formatting
- Hardware Interface
  - Device setup
  - Read/Write/Control

**Figure 1.**
Typical main parts of a device driver

- A **device driver** contains all the software routines that are needed to be able to use the device.

- Typically a device driver contains a number of main routines like a **initialization routine**, that is used to setup the device, a **reading routine** that is used to be able to read data from the device, and a **write routine** to be able to write data to the device.

- The device driver may be either interrupt driven or just used as a polling routine.

# Design of Device Driver

- 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:**

- Initialization routines for the hardware
- Device drivers
- Interrupt Service Routines

- The device drivers can then be called from the application using RTOS standard calls.
- The RTOS creates during its own initialization 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 device 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 of course 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.

# Synchronous Device Driver

- When a task calls a synchronous device driver it means that the task will wait until the device has some data that it can give to the task, see figure 2.

- In this example the task is blocked on a semaphore until the driver has been able to read any data from the device.

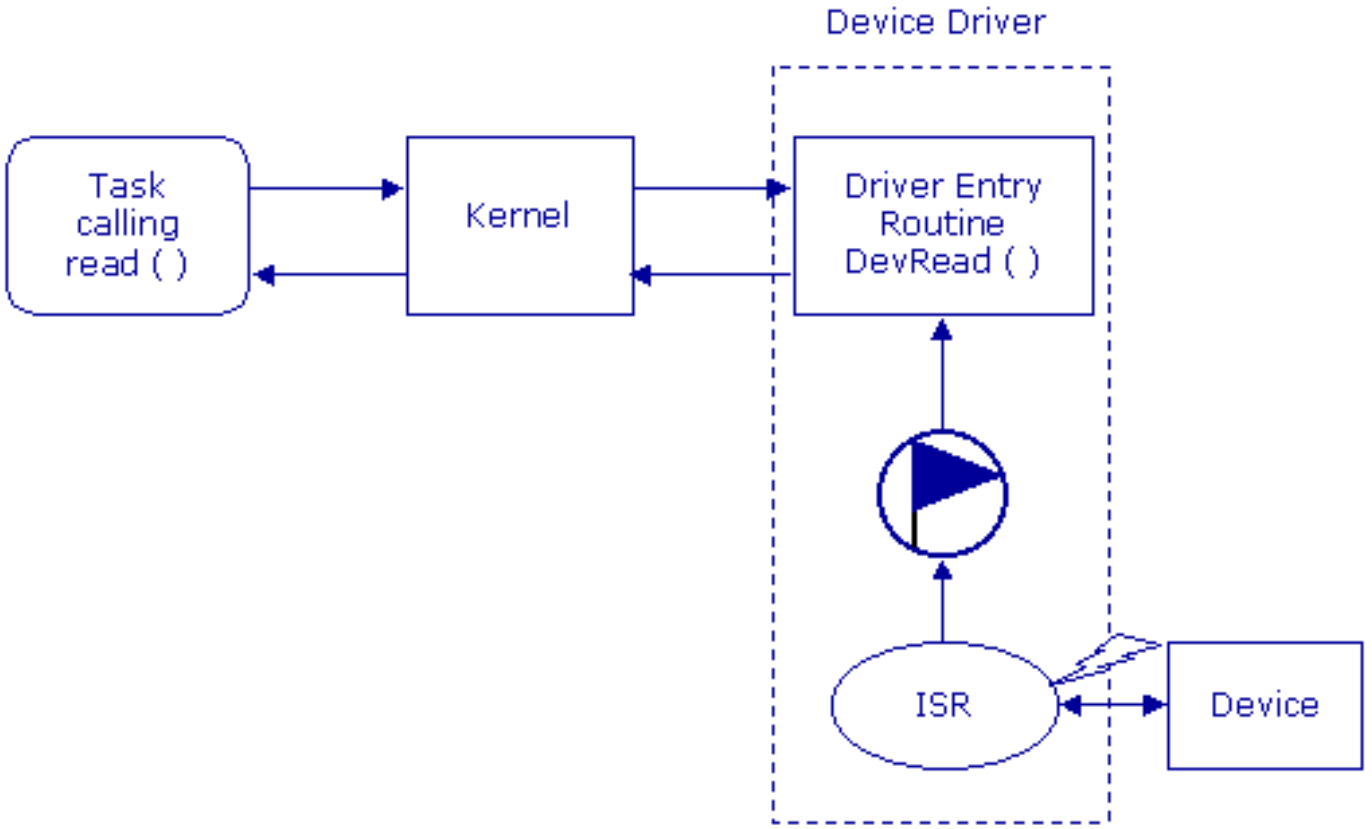



**Figure 2.** Synchronous device driver
The Task calls the device driver via a kernel device call. The Device Entry Routine gets blocked on a semaphore and blocks the task in that way. When an input comes from the device, it generates an interrupt and the ISR releases the semaphore and the Device Entry Routine returns the data to the task and the task continues its execution.

# 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, see figure 3.

- 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.

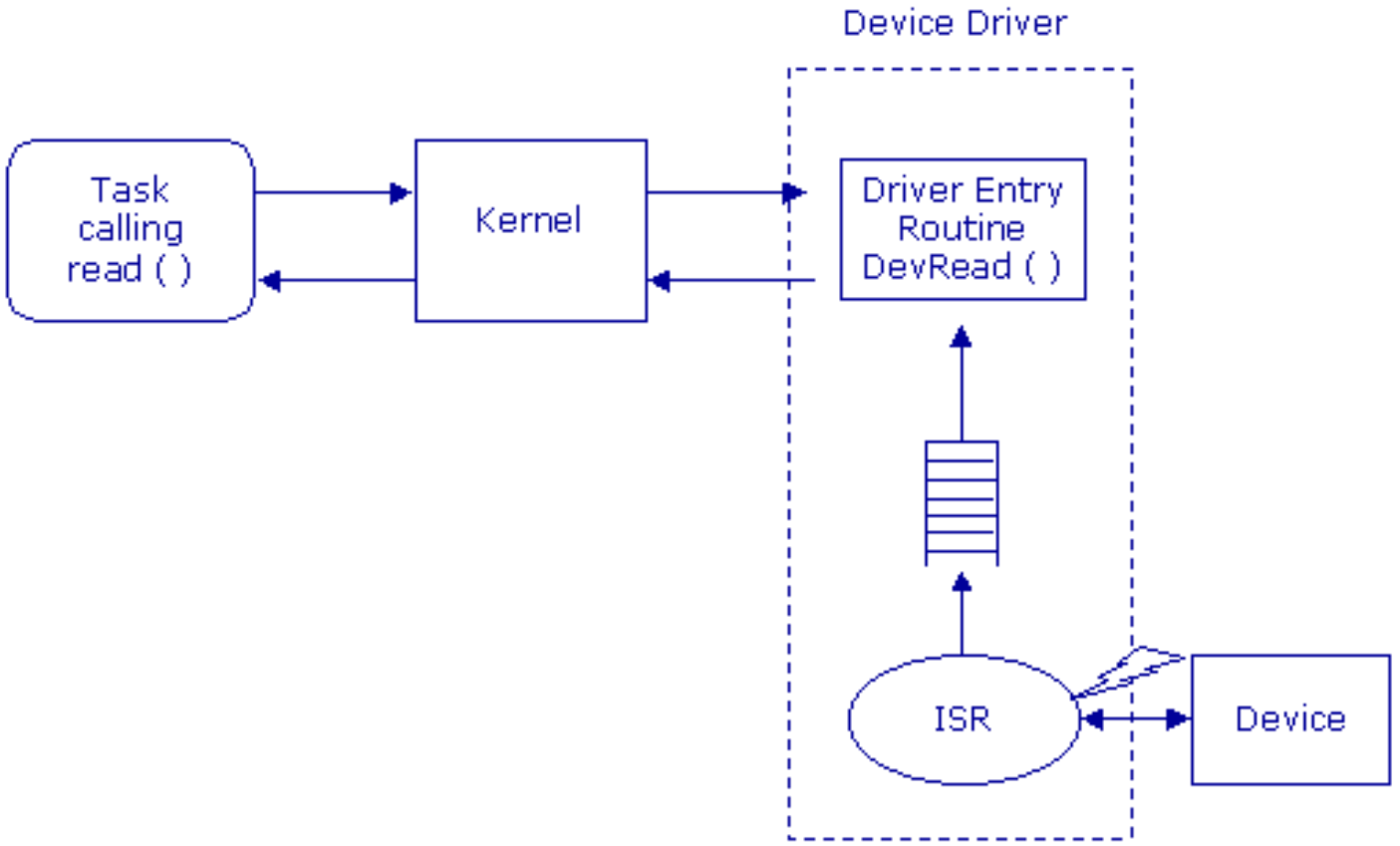



**Figure 3.** Asynchronous device driver
The Task calls the device driver via a kernel device call. The Device Entry Routine receives a message from the queue and returns the data to the task. When new data arrives from the device the ISR puts the data in a message and sends the message to the queue.

# Text Editors

- Overview of Editing.
- User Interface.
- Editor Structure.

# Definition

- **Editors or text editors** are software programs that enable the user to create and edit text files.

- In the field of programming, the term editor usually refers to source code editors that include many special features for writing and editing code.

- Notepad, Wordpad are some of the common editors used on **Windows OS** and vi, emacs, Jed, pico are the editors on **UNIX OS.**

# Types of Text Editors

- Features normally associated with text editors are — moving the cursor, deleting, replacing, pasting, finding, finding and replacing, saving etc.
- **Depending on how editing is performed, and the type of output that can be generated, editors can be broadly classified as**

- Line Editors
- Stream Editors
- Screen Editors
- Word Processors
- Structure Editors

## Line editor:

- In this, you can only edit one line at a time or an integral number of lines.
- You cannot have a free-flowing sequence of characters.
- It will take care of only one line.
- Ex : Teleprinter, edlin, teco

## Stream editors:

- In this type of editors, the file is treated as continuous flow or sequence of characters instead of line numbers, which means here you can type paragraphs.
- Ex : Sed editor in UNIX

## Screen editors:

- In this type of editors, the user is able to see the cursor on the screen and can make a copy, cut, paste operation easily.
- It is very easy to use mouse pointer. Ex : vi, emacs, Notepad.

## Word Processor:

- Overcoming the limitations of screen editors, it allows one to use some format to insert images, files, videos, use font, size, style features.
- It majorly focuses on Natural language.

## Structure Editor:

- Structure editor focuses on programming languages.
- It provides features to write and edit source code. Ex : Netbeans IDE, gEdit.

# Editing Process

- An **interactive editor** is a computer program that allows a user to create and revise a target document.
- We all by now understand that editors are the program which is **used to create, edit and modify a document.**
- A document may include some images, files, text, equations, and diagrams as well.
- But we will be limited to **text editors** only whose main elements are **character strings**.

**The document editing process mainly compromises of the following four tasks :**

- The part of the document to edited or modifies is selected.
- Determining how to format this lines on view and how to display it.
- Specify and execute the operations that modify the document.
- Update the view properly.

**The above steps include filtering, formatting, and traveling**

- **Formatting :** Visibility representation on display screen.
- **Filtering :** Finding out the main/important subset.
- **Traveling :** Locating the area of interest.
- **Editing phase** involves insert, delete, replace, move, copy, cut, paste etc.

# User Interface of editors

- The user interface of editors typically means the input, output and the interaction language.
- The **input devices** are used to enter text, data into a document or to process commands.
- The **output devices** are used to display the edited form of the document and the results of the operation/commands executed.
- The **interaction language** provides the interaction with the editor.

**Input Devices :**

- Input devices are generally divided as text input, button devices and locator devices.
- Text device is a keyboard. Button devices are special function keys.
- The locator devices include the mouse.
- There are special voice devices as well which writes into text whatever you speak.

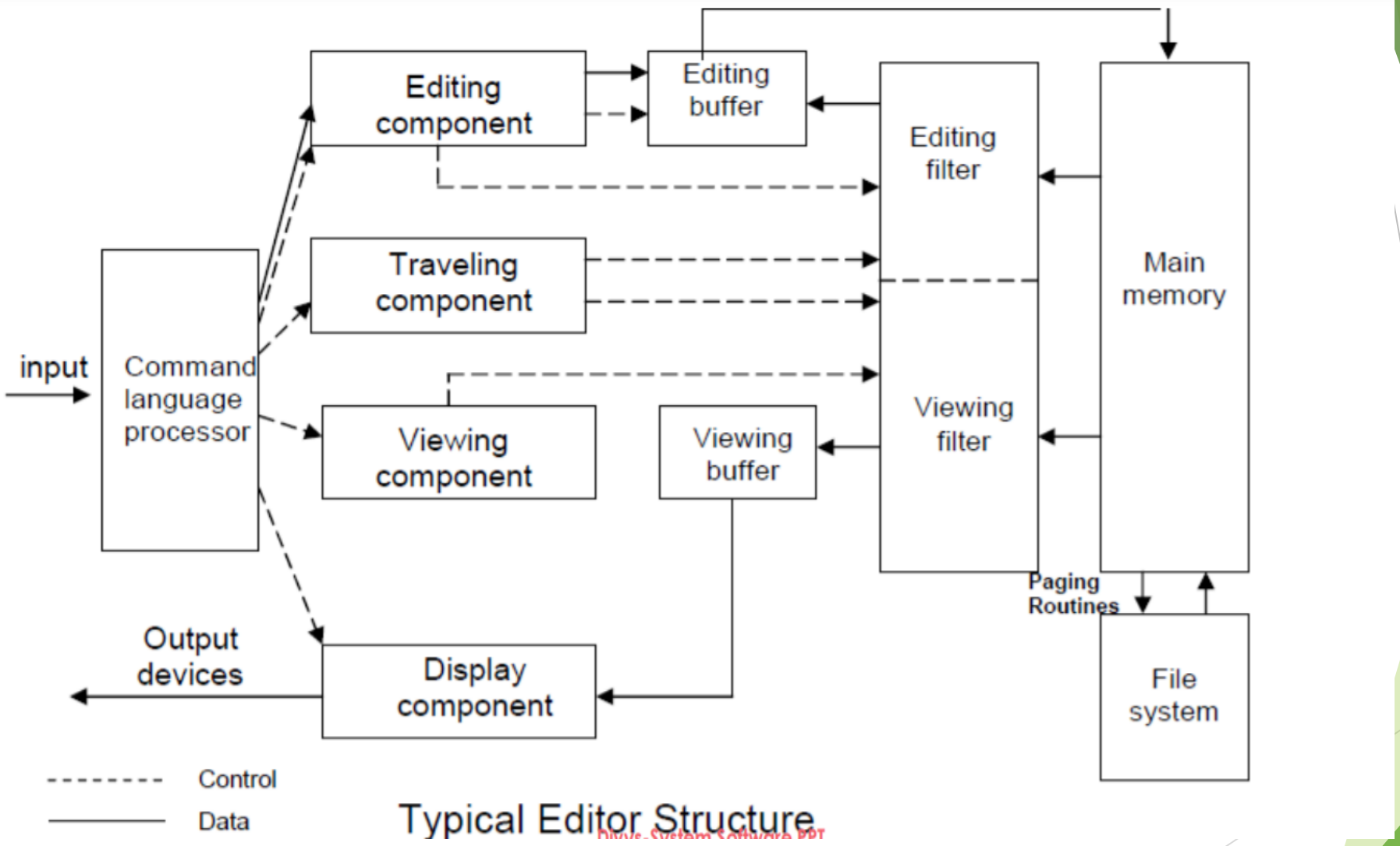
# Output Devices :

- TFT monitors- Thin-film **transistor** technology in an LCD display.
- Printers
- Teletypewriters
- Cathode ray tube technology
- Advanced CRT terminals

# Interaction language

- The interaction language could be, typing oriented or text command-oriented or could be menu oriented user interface as well.
- **Typing or text command-oriented** interaction language is very old used with the oldest editors, in the form of commands, use of functions and control keys etc.
- **Menu oriented** interface has a menu with the set of multiple choice of text strings.
- The display area is limited and the menus can be turned on/off by the user.

## Editor Structure




Typical Editor Structure

- The **command language processor** accepts commands, performs functions such as editing and viewing.
- It involves traveling, editing, viewing and display.
- **Editing operations** are specified by the user and **display operations** are specified by the editor.
- Traveling and viewing components are invoked by the editor or the user itself during the operations.

- **Editing component** is a module dealing with editing tasks.
- The **current editing area** is determined by the **current editing pointer** associated with the editing component.
- When editing command is made, the editing component calls the **editing filter**, generates a new editing buffer.

- **Editing buffer** contains the document to be edited at the current editor pointer location.
- In viewing a document, the start of the area to be viewed is determined by the **current viewing pointer.**
- **Viewing component** is a collection of modules used to see the next view.
- Current viewing can be made to set or reset depending upon the last operation.

- When display needs to be updated, the **viewing component** invokes the **viewing filter,** generates a **new buffer** and it contains the document to be viewed using the current view buffer.

- Then the viewing buffer is pass to the display component which produces the display by buffer mapping.

- The editing and viewing buffers may be identical or completely disjoint.

- The **editing and viewing buffers** can also partially overlap or can be contained one within the another.

- The component of the editor interacts with the document from the user on two levels:

  Main memory and the disk files system.

# Debuggers

- Debugging Functions and Capabilities.
- Relationship with other parts of the system.
- **Debugging Methods**

By induction

By deduction

By backtracking

# Debugging Functions and Capabilities

- Debugging means locating (and then removing) bugs, i.e., faults, in programs.
- An interactive debugging system provides programmers with facilities that aid in testing and debugging of programs.
- One important requirement of any interactive debugging system is the **observation and control of the flow of program execution.**

- 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 occurs 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.

- 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**

**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.

- Execution is suspended - use debugging commands to analyze the progress of the program
- Resume execution of the program.

**Setting some conditional expressions**

- They are evaluated during the debugging session
- Program execution is suspended, when conditions are met, analysis is made, later execution is resumed.

## Tracing

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

## Traceback

- 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.

# Program display capabilities

- A **debugger** should have good program-display capabilities.
- Program being debugged should be displayed completely with statement numbers.
- **Keep track** of any changes made to the programs during the debugging session.
- Support for symbolically displaying or modifying the contents of any of the variables and constants in the program.
- **Resume execution** after these changes.

# Language dependent and independent

- To provide these functions, a debugger should consider the language in which the program being debugged is written.
- A **single debugger** for many programming languages means it should be **language independent.**
- The debugger for a specific programming language would be **language dependent.**
- The debugger must be sensitive to the specific language being debugged.

- The context being used has many different effects on the debugging interaction.
- The statements are different depending on the language.

- **Assignment statements in different languages**

- Cobol : MOVE 6.5 TO X
- Fortran : X = 6.5
- C : X = 6.5

- Similarly, the condition that X be unequal to Z may be expressed as

- Cobol : IF X NOT EQUAL TO Z
- Fortran : IF ( X.NE.Z)
- C : if ( X != Z)

- Similar differences exist with respect to the form of statement labels, keywords and so on.

# Part of language translators

- The notation used to specify certain debugging functions varies according to the language of the program being debugged.
- Sometimes the **language translator itself has debugger interface modules** that can respond to the request for debugging by the user.
- The source code may be displayed by the debugger in the standard form or as specified by the user or translator.

# Optimized code

- It is also important that a debugging system be able to deal with optimized code.

**Many optimizations like**

- Invariant expressions can be removed from loops
- Separate loops can be combined into a single loop
- Redundant expression may be eliminated
- Elimination of unnecessary branch instructions leads to rearrangement of segments of code in the program.

- All these optimizations create problems for the debugger, and should be handled carefully

# Difference between testing & debugging

| | |
|---|---|
| Testing is the process to find bugs and errors. | Debugging is the process to correct the bugs found during testing. |
| It is the process to identify the failure of implemented code. | It is the process to give the a solution to code failure. |
| Testing is the display of errors. | Debugging is a deductive process. |

| | |
|---|---|
| Testing is done by the tester. | Debugging is done by either programmer or developer. |
| There is no need of design knowledge in the testing process. | Debugging can’t be done without proper design knowledge. |
| Testing can be done by insider as well as outsider. | Debugging is done only by insider. Outsider can’t do debugging. |
| Testing can be manual or automated. | Debugging is always manual. Debugging can’t be automated. |

| | |
|---|---|
| It is based on different testing levels i.e. unit testing, integration testing, system testing etc. | Debugging is based on different types of bugs. |
| Testing is a stage of software development life cycle (SDLC). | Debugging is not an aspect of software development life cycle, it occurs as a consequence of testing. |
| Testing is composed of validation and verification of software. | While debugging process seeks to match symptom with cause, by that it leads to the error correction. |
| Testing is initiated after the code is written. | Debugging commences with the execution of a test case. |

# Relationship with other part of the system

- The important requirement for an interactive debugger is that it **always be available.**
- Must appear **as part of the run-time environment** and an integral part of the system.
- When an error is discovered, immediate debugging must be possible.
- The debugger must **communicate and cooperate with other operating system components** such as interactive subsystems.

- 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.
- The debugger must also exist in a way that is consistent with **the security and integrity components of the system.**
- The debugger must coordinate its activities with those of **existing and future language compilers and interpreters.**

# User interface criteria

- Debugging systems should be simple in its organization and familiar in its language and must closely reflect common user tasks.
- The simple organization contribute greatly to ease of training and ease of use.
- The user interaction should make use of full-screen displays and windowing-systems as much as possible.
- With menus and full-screen editors, the user has far less information to enter and remember.

- **There should be complete functional equivalence between commands and menus**

If a user is unable to use full- screen interactive debugging system may use commands.

- The command language should have a clear, logical and simple syntax.
- Command formats should be as flexible as possible.
- Any good interactive debugging system should have an on-line HELP facility.
- HELP should be accessible from any state of the debugging session.

# Debugging methods

- Debugging by induction
- Debugging by deduction
- Debugging by backtracking

- During the **testing process**, the programmer observes the input-output relationships i.e., the output that the program produces for each input test cases.

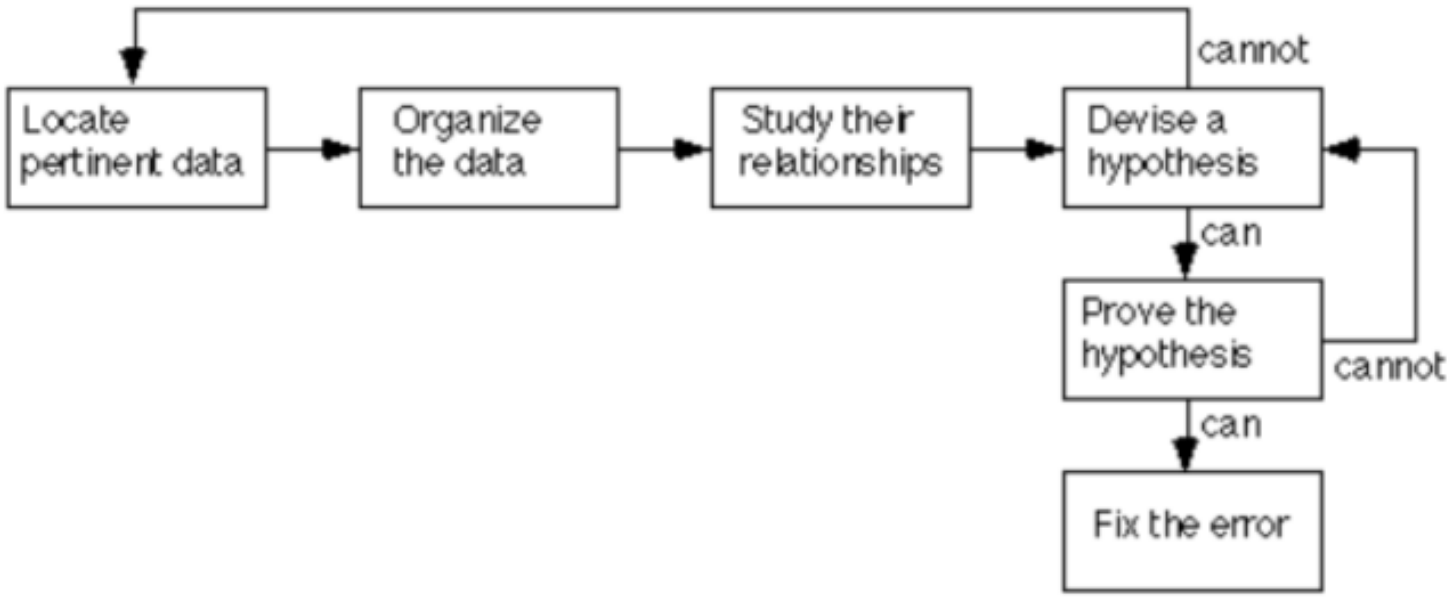
- If the program produces the expected output and obeys the specification for each test case, then the program is successfully tested.

- But, if the **output for one of the test cases is not correct**, then the **program is incorrect** i.e., it contains **errors, defects or bugs.**

- In such a situation**, testing only reveals the presence of errors,** but doesn't tell us what the errors are or how the code needs to be fixed.

- Programmer must go through the **debugging process** to identify the cause and fix the errors.

# Induction


Locate pertinent data
Organize the data
Study their relationships
Devise a hypothesis
cannot
can
Prove the hypothesis
cannot
can
Fix the error

# 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.

# 2. Organize the data

- Structure the pertinent data to let you observe the patterns.

- 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.
- "What" boxes list the general symptoms
- "Where" boxes describe where the symptoms were observed
- "When" boxes list anything that is known about the times that the symptoms occur
- "To What Extent" boxes describes the scope and magnitude of the symptoms.
- "Is" and "Is Not" columns. describe the contradictions that may eventually lead to a hypothesis about the error.

| ? | Is | Is Not |
|---|---|---|
| What | | |
| Where | | |
| When | | |
| To What Extent | | |

# 3. Devise a hypothesis

- The next steps is 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.

# 4. 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.

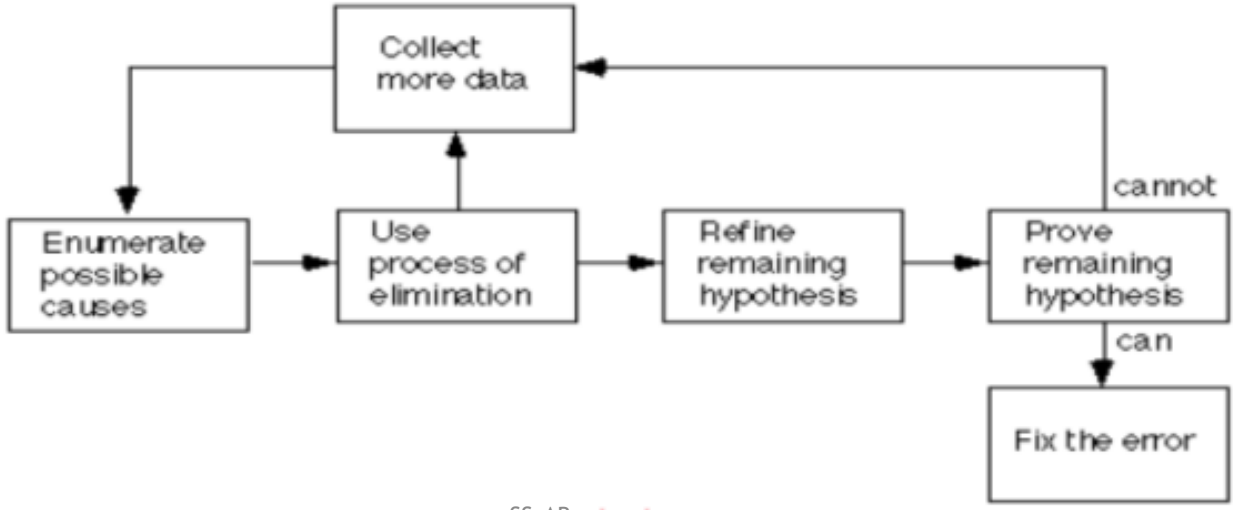
- If it does not, either the hypothesis is invalid, the hypothesis is incomplete, or multiple errors are present.

# 5. Fix the error

- If hypothesis is proved successfully then go on to fix the error else repeat step 3 & 4.

# By deduction

- It is a process of proceeding from some general theories or premises, using the processes of elimination and refinement, to arrive at a conclusion.



# 1. Enumerate the possible causes or hypotheses

- **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.**

# 2. Use process of elimination

- Use the data to eliminate possible causes.
- By a careful analysis of the data, particularly by looking for contradiction, 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.**

# 3. Refine the remaining hypothesis

- The possible cause at this point might be correct, but it is unlikely to be specific enough to pinpoint the error.

- Hence, the next step is to use the available clues to **refine the theory to something more specific.**

# 4. Prove the remaining hypothesis.

- This vital step is identical to the fourth step in the induction method.

# 5. Fix the errors

- If hypothesis is proved successfully then go on to fix the error.

# 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.

# University questions…

- Differentiate between keyword and positional macro parameters.
- Is it possible to include labels in the body of macro definition? Justify your answer.
- List out the criteria that should be met by the user interface of an efficient debugging system.
- Write a short note on concatenation of macro parameters within a character string.
- What is a Device Driver? What are the major design issues of a Device Driver?
- Distinguish between Character and Block Device drivers.
- Explain recursive macro expansion

- What is a Debugger?
- How are unique labels generated in a Macro Expansion?
- Explain any one of the debugging methods in detail.
- List out the main four tasks associated with the Document Editing Process
- List out and explain different data structures used in macro processor algorithms.
- Explain the working of One pass macro processor algorithm.
- Draw the structure of a typical text editor and describe the functions of each block.
- Describe the functions and capabilities of an Interactive debugging system.

- Explain the different types of Text Editors and User Interface and also explain editor structure in detail with neat figures.
- Explain general purpose macro processor in detail.
- Explain macro processor design options in detail.
- Explain the general design of device drivers.
- Explain the following in detail.

1. Generation of unique labels.
2. Conditional macro expansion.
3. Keyword macro parameters.