Macro Processor

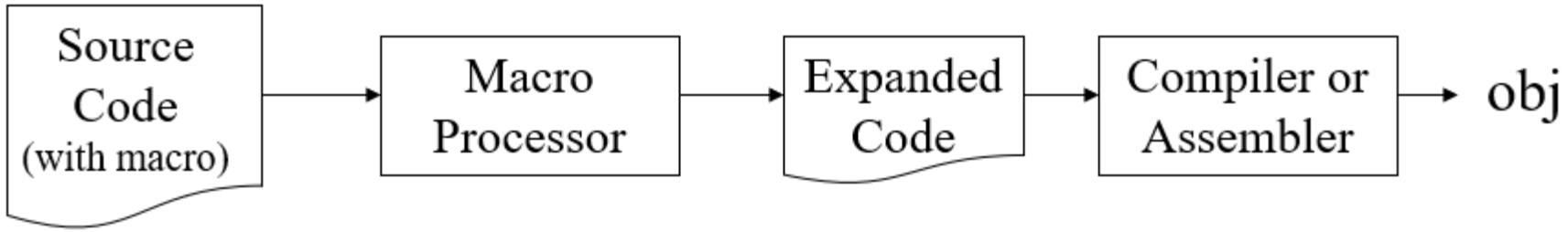
Macro Processor

- Macro instructions or macros are single line abbreviations for a group of instructions
  - A macro instruction is a notational convenience for the programmer
  - Allows the programmer to write short hand versions of programs
- A macro represents a commonly used group of statements in the source programming language.

- ***Expanding the macros***
  - The macro processor replaces each macro instruction with its equivalent block of instructions.

Macro Processor

- Functions of a macro processor
  - Substitution of one group of characters or lines for another.
  - Normally, it performs no analysis of the text it handles.
  - The meaning of the statements and their translation into machine language are of no concern to the macro processor
- The design of a macro processor generally is machine independent (not related to the architecture of the machine on which it is to run)
- General purpose macro processors not tied to any language



Macro Processor

- Macro Definition
  - Definition of the macro instructions appear in the source program following the START statement
- Two new assembler directives are used in macro definition
  - MACRO: identify the beginning of a macro definition
    - The label field specified along with the macro directive specifies the name of the macro
  - MEND: identify the end of a macro definition

- Following the MACRO directive are the statements that make up the body of the macro definition
  - These are the statements that will be generated when the macro is expanded

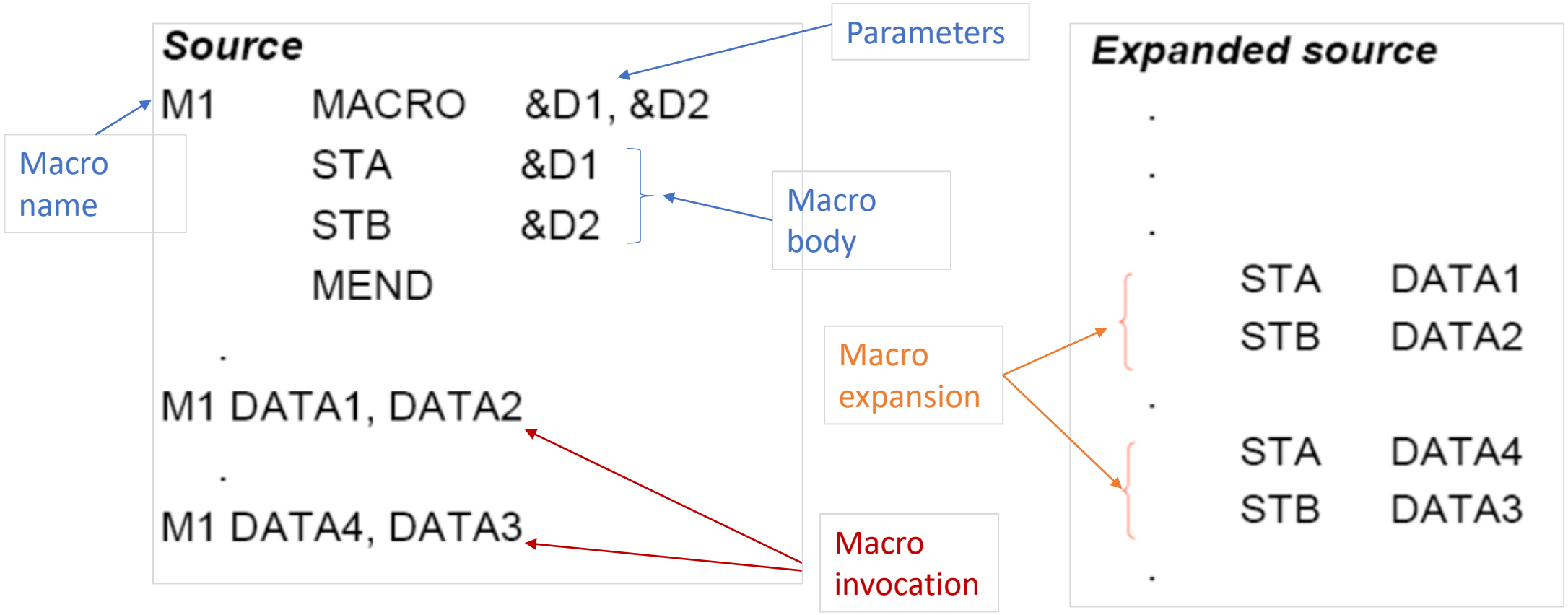
Macro Definition and Expansion

```
name  MACRO parameters
        :
      body
        :
      MEND
```

- Parameters: the entries in the operand field identify the parameters of the macro instruction (optional)
  - Each parameter begins with '&' which facilitates substitution of the parameters during macro expansion
- Body: the statements that will be generated when the macro is expanded.
- Prototype for the macro: The macro name and parameters define a ***pattern or prototype*** for the macro instructions used by the programmer

Macro Definition and Expansion

A ***macro invocation statement*** (a ***macro call***) gives the **name** of the macro instruction being invoked and the **arguments** in expanding the macro.



```
 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
```

```
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
```

```
  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
```

Macro Definition and Expansion

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

Solutions

- Simply not to use labels in the body of macro.
- Explicitly use PC relative addressing instead.
- For example, in RDBUFF and WRBUFF macros,

```
JEQ     * +11
JLT     *-14
```

- It is inconvenient and error-prone.

Macro Call

- Statements of the macro body are expanded each time the macro is invoked.
- Macro invocation is more efficient than subroutine call
- Code size is larger

Subroutine Call

- Statements of the subroutine appear only once, regardless of how many times the subroutine is called
- Less efficient than macro invocation
- Code size is small

Nested Macro

```
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}
```

Macro Processor Algorithm and Data Structures

- Recognize macro definitions
  - Identified by MACRO and MEND directives
  - All the intervening text including the nested MACROs and MENDs define a single macro instruction
- Save the macro definition
  - Store the macro instruction definition which will be needed for expanding macro calls
- Recognize macro calls
  - Recognize macro calls that appear as operation mnemonics
- Expand macro calls
  - Substitute arguments from the macro call for dummy arguments
  - Substitute the macro definition for a macro call

Macro Processor Algorithm and Data Structures

- **One-pass macro processor**
  - Every macro must be defined before it is called
  - One-pass processor can alternate between macro definition and macro expansion
  - Nested macro definitions are allowed

Macro Processor Algorithm and Data Structures

DEFTAB (Definition Table)

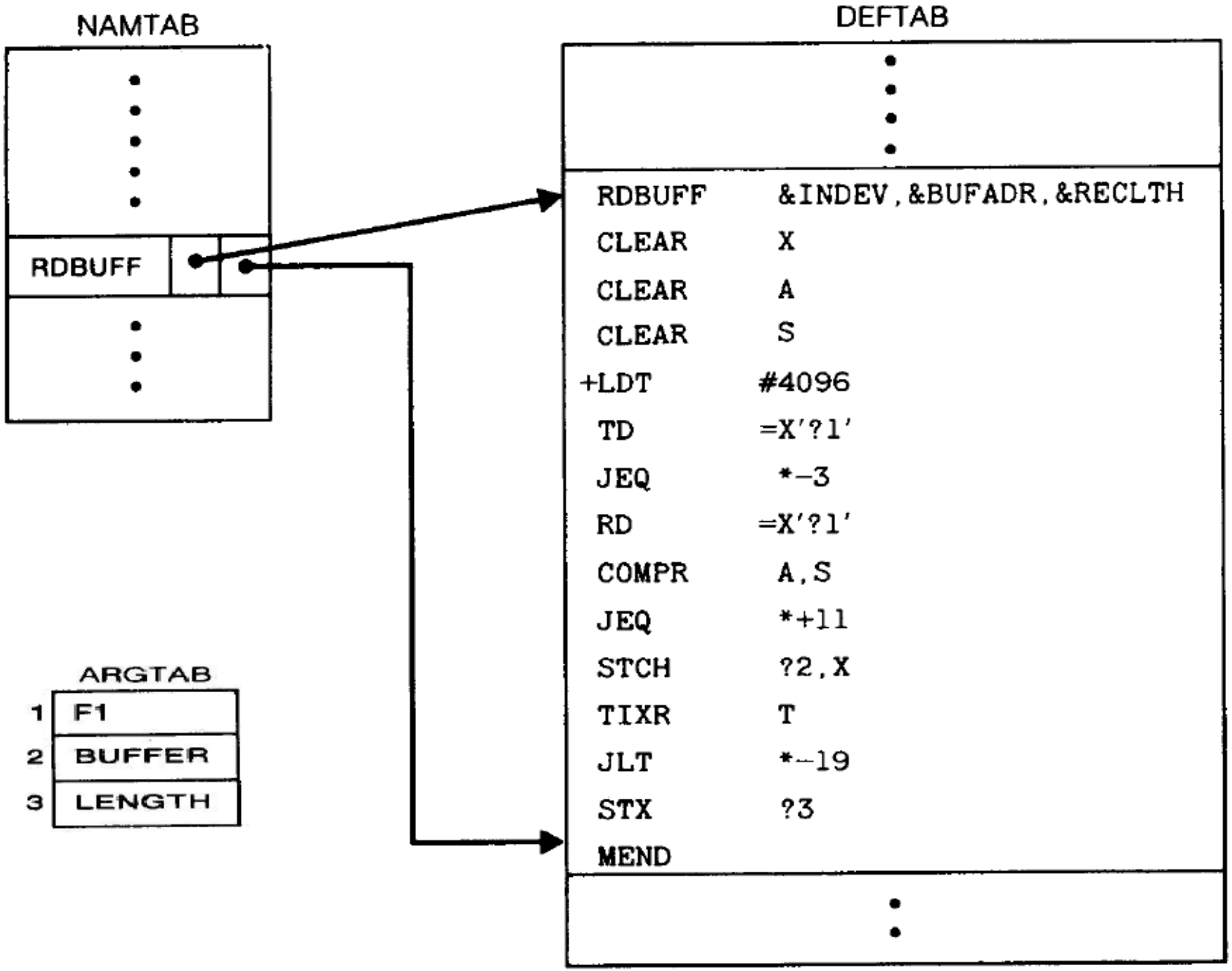
- Used to store macro definition including macro prototype and macro body
- Positional notation has been used for the parameters for efficiency in substituting arguments.
- E.g. the first parameter &INDEV has been converted to ?1 (indicating the first parameter in the prototype)

NAMTAB (Name Table)

- Used to store the macro names
- Serves as an index to DEFTAB
- Pointers to the beginning and end of the macro definition

ARGTAB (Argument Table)

- Used to store the arguments used in the expansion of macro invocation
- As the macro is expanded, arguments are substituted for the corresponding parameters in the macro body.


NAMTAB
RDBUFF
DEFTAB
RDBUFF &INDEV,&BUFADR,&RECLTH
CLEAR X
CLEAR A
CLEAR S
+LDT #4096
TD =X'?1'
JEQ *-3
RD =X'?1'
COMPR A,S
JEQ *+11
STCH ?2,X
TIXR T
JLT *-19
STX ?3
MEND
ARGTAB
1 F1
2 BUFFER
3 LENGTH

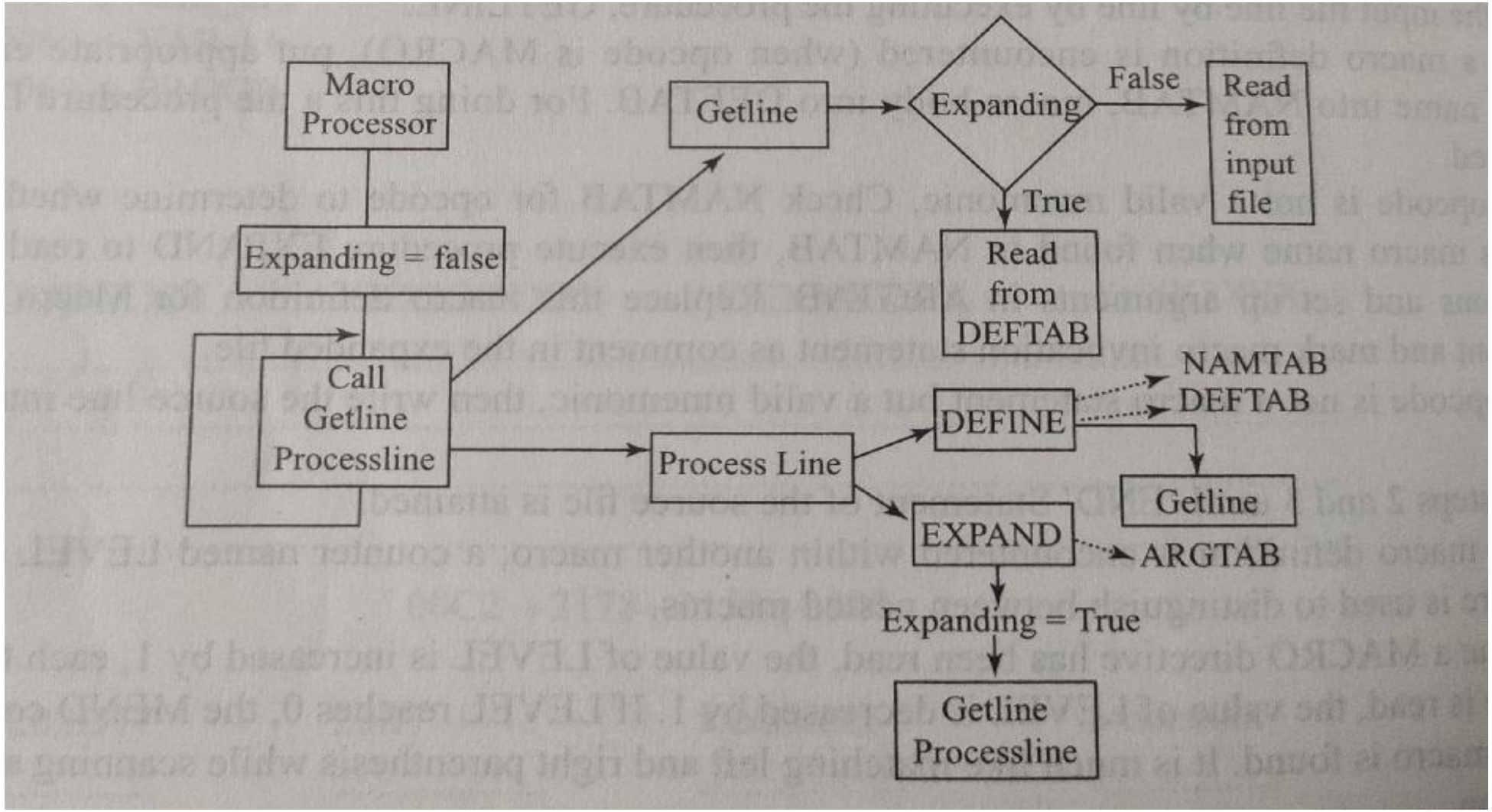
Macro Processor Algorithm and Data Structures

- The procedure DEFINE
  - Called when the beginning of a macro definition is recognized
  - Makes the appropriate entries in DEFTAB and NAMTAB.
- The procedure EXPAND
  - Called to set up the argument values in ARGTAB and expand a macro invocation statement.
- The procedure GETLINE
  - Called at several points in the algorithm, gets the next line to be processed.
  - This line may come from DEFTAB (the next line of a macro begin expanded), or from the input file, depending on whether the Boolean variable EXPANDING is set to TRUE or FALSE.

One Pass Macro Processor Algorithm

- Handling of macro definitions within macros
  - The DEFINE procedure maintains a counter named **LEVEL**.
  - Each time a MACRO directive is read, the value of LEVEL is increased by 1.
  - Each time an MEND directive is read, the value of LEVEL is decreased by 1.
  - When LEVEL reaches 0, the MEND that corresponds to the original MACRO directive has been found.
  - The above process is very much like matching left and right parentheses when scanning an arithmetic expression.

One Pass Macro processor



One Pass Macro processor Algorithm

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


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

One Pass Macro processor Algorithm

```
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

- Concatenation of macro parameters
- Generation of unique labels
- Conditional macro expansion
- Keyword macro parameters

Concatenation of Macro Parameters

- Concatenate parameters with other character strings
- For example, a program contains a set of series of variables
  - XA1, XA2, XA3,
  - XB1, XB2, XB3,
- If similar processing is to be performed on each series of variables, the programmer might want to incorporate this processing into a macro instruction.
- The parameter to such a macro instruction could specify the series of variables to be operated on (A, B, etc.).
- The macro processor construct the symbols by concatenating X, (A, B etc.), and (1,2,3,…) in the macro expansion

Concatenation of Macro Parameters

- Suppose the parameter is named &ID, the macro body may contain a statement: LDA X&ID1
- &ID is concatenated after the string “X” and before the string “1”.
  - LDA XA1 (&ID=A)
  - LDA XB1 (&ID=B)
- **Problem:** Ambiguous situation
  - Eg. X&ID1 may mean “X” + &ID + “1” **OR** “X” + &ID1
- This problem occurs because the end of the parameter is not marked.
- **Solution:**
- Use a special **concatenation operator “→”** to specify the end of the parameter
  - Eg. LDA X&ID → 1
- The macro processor deletes all occurrences of the concatenation operator immediately after performing parameter substitution, so →will not appear in the macro expansion.

Concatenation of Macro Parameters

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

```
SUM     A
```

↓

```
LDA     XA1
ADD     XA2
ADD     XA3
STA     XAS
```

```
SUM     BETA
```

↓

```
LDA     XBETA1
ADD     XBETA2
ADD     XBETA3
STA     XBETAS
```

Generation of Unique Labels

- Labels in the macro body may have duplicate labels problem if the macro is invocated multiple times.
- Use of relative addressing is very inconvenient, error-prone, and difficult to read.
  - Example: JEQ *-3 n
  - Inconvenient, error-prone, difficult to read
- **Solution:**
- It is highly desirable to let the programmer use label in the macro body
- Labels used within the macro body begin with $
- During macro invocation, $ will be replaced by $XX
  - XX is a two-character alphanumeric counter of the number of macro instructions expanded
  - XX=AA, AB, AC etc.

Generation of Unique Labels

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

Generation of Unique Labels

- **Macro definition**

```
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
```

Generation of Unique Labels

RDBUFF F1, BUFFER, LENGTH

•**Macro expansion**

```
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
```

Keyword Macro Parameters

- Positional parameters: 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:
- E.g. GENER ,,DIRECT,,,,,,3
- E.g. RDBUFF 0E, 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.

Keyword Macro 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.
- E.g. GENER TYPE=DIRECT, CHANNEL=3
- It is easier to read and much less error-prone than the positional method.
- In the macro prototype, each parameter name is followed by an equal sign (=), which identifies a keyword parameter.
- After = sign, a default value is specified for some of the parameters.
- The parameter is assumed to have this default value if its name does not appear in the macro invocation statement.
- Default values can simplify the macro definition in many cases.

Keyword Macro 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
```

Default values of parameters

Keyword Macro Parameters

```
.           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
```

Keyword Macro Parameters

```
.           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
```

Conditional Macro Expansion

- Each invocation of a particular macro was expanded into the same sequence of statements
- Most macro processors can also modify the sequence of statements generated for a macro expansion, depending on the arguments supplied in the macro invocation.
- Increases the power and flexibility of a macro language
- The term conditional assembly describes these features and conditional macro expansion for this type of macro invocation
- Two types
  - Macro-time conditional structure: IF-ELSE-ENDIF
  - Macro-time looping statement : WHILE-ENDW

```
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
```

1

2

3

4

Conditional Macro Expansion

```
.           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
```

2

3

4

Figure 4.8 Use of macro-time conditional statements.

Conditional Macro Expansion

- Testing of Boolean expressions in IF statements occurs at the time macros are expanded.
- By the time the program is assembled, all such decisions must have been made and conditional macro expansion directives will be removed in the expanded program.
- The same applies to the assignment of values to macro-time variables and to the other conditional macro expansion directives.

Conditional Macro Expansion

- RDBUF has two additional parameters
  - &EOR: hex character code that marks the end of a record
  - &MAXLTH: maximum length record that can be read.
- **Lines: 44-48**
- The IF statement evaluates a Boolean expression that is its operand (In this case, it is &MAXLTH EQ ‘ ‘)
- If TRUE, the statements following the IF are generated until an ELSE is encountered (Line 45 is generated)
- If FALSE, these statements are skipped, and the statements following the ELSE are generated (Line 47 is generated)
- The ENDIF statement terminates the conditional expression that was begun by the IF statement.

Conditional Macro Expansion

- **Lines 26-28**
- SET macro processor directive
- Does not generate the line into macro expansion
- This SET directive assigns the value 1 to &EORCK.
- The symbol &EORCK is a **macro time variable** or a **SET symbol** that 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.
- The value of this conditional variable are used on lines 38-43 and 63-73
- The values of macro time variables can be changed using SET directive

```
&EORCK   SET   1
```

Conditional Macro Expansion

- **Implementation**
- Macro processor maintains a symbol table that contains the values of all macro-time variables used.
- Entries in this table are made or modified when SET statements are processed.
- The table is used to look up the current value of a macro-time variable whenever it is required.

  IF (Boolean expression) (statements) ELSE (statements) ENDIF
- When an IF statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated.
  - If TRUE, the macro processor continues to process lines from DEFTAB until it encounters the next ELSE or ENDIF statement.
  - If an ELSE is found, the macro processor then skips lines in DEFTAB until the next ENDIF.
  - Upon reaching the ENDIF, it resumes expanding the macro in the usual way.
  - If FALSE, the macro processor skips ahead in DEFTAB until it finds the next ELSE or ENDIF statement. The macro processor then resumes normal macro expansion.
- The implementation outlined above does not allow for nested IF structures.
- The macro-time IF-ELSE-ENDIF structure provides a mechanism for either generating once or skipping selected statements in the macro body

Conditional Macro Expansion

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

```
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
```

```
.           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
```

Conditional Macro Expansion

- **Lines 63 through 73**
- %NITEMS is a macro processor function that returns as it value the number of members in the argument list
- If the argument corresponding to &EOR is (00,03,04), then %NITEMS(&EOR)=3.
- The macro-time variables &EORCT is set (line 27) to the value %NITEMS(&EOR).
- The macro-time variable &CTR is used to count the number of times the lines following the WHILE statement have been generated.
- The value of &CTR is initialized to 1 (line 63), and incremented by 1 each time through the loop (line 71).

Conditional Macro Expansion

- WHILE statement specifies that the macro-time loop will continue to be executed as long as the value of &CTR is less than or equal to the value of &EORCT
- i.e. the statements on lines 65 and 70 will be generated once for each member of the list corresponding to the parameter &EOR
- Value of &CTR is used as a subscript to select the proper member of the list for each iteration of the loop
- Thus, on the first iteration, the expression &EOR[&CTR] on line 65 has the value 00, on the second iteration, it has the value 03 and so on.

Conditional Macro Expansion

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

Macro Processor Design Options

- Recursive Macro Expansion
  - Implementing macro calls within macros
  - If a macro call is encountered during the expansion of a macro, the macro processor will have to expand the included macro call and then finish expanding the enclosed macro

Recursive Macro Expansion

```
 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
```

Recursive Macro Expansion

```
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
```

Recursive Macro Expansion

- Consider the macro call

  RDBUFF BUFFER, LENGTH, F1

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

Parameter	Value
1	BUFFER
2	LENGTH
3	F1
4	Unused
.	..

Recursive Macro Expansion

- 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
.	..

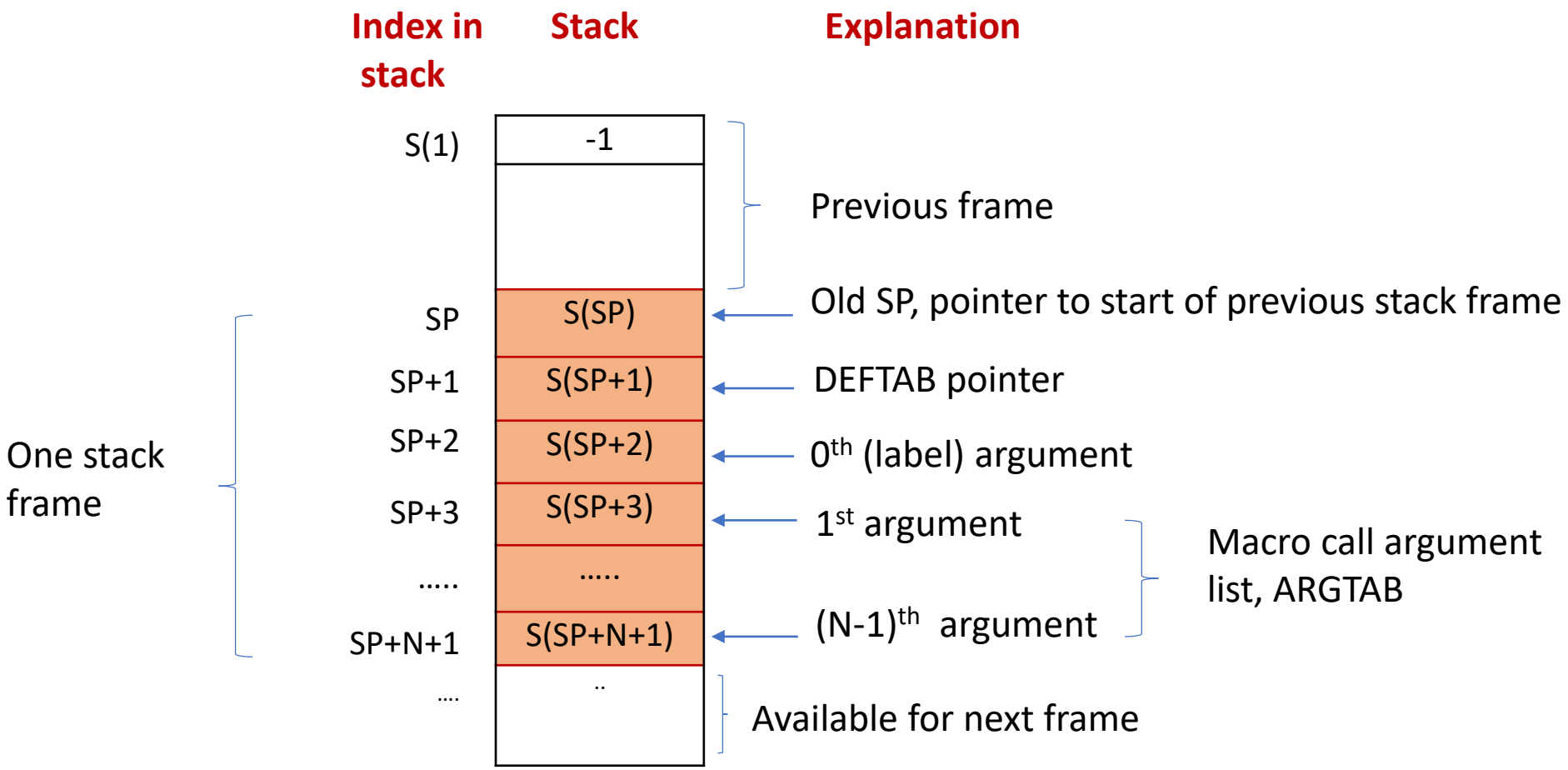
Recursive Macro Expansion

- At the end of this expansion, however, a problem would appear. When the end of the definition of RDCHAR was recognized, EXPANDING would be set to FALSE.
- Thus the macro processor would “forget” that it had been in middle of expanding a macro when it encountered the RDCHAR statement.
- **Solution**
- Use a Stack to save ARGTAB and Use a counter to identify the expansion
- Write the macro processor in a programming language which automatically takes care of the recursive calls thus retaining the local variables.

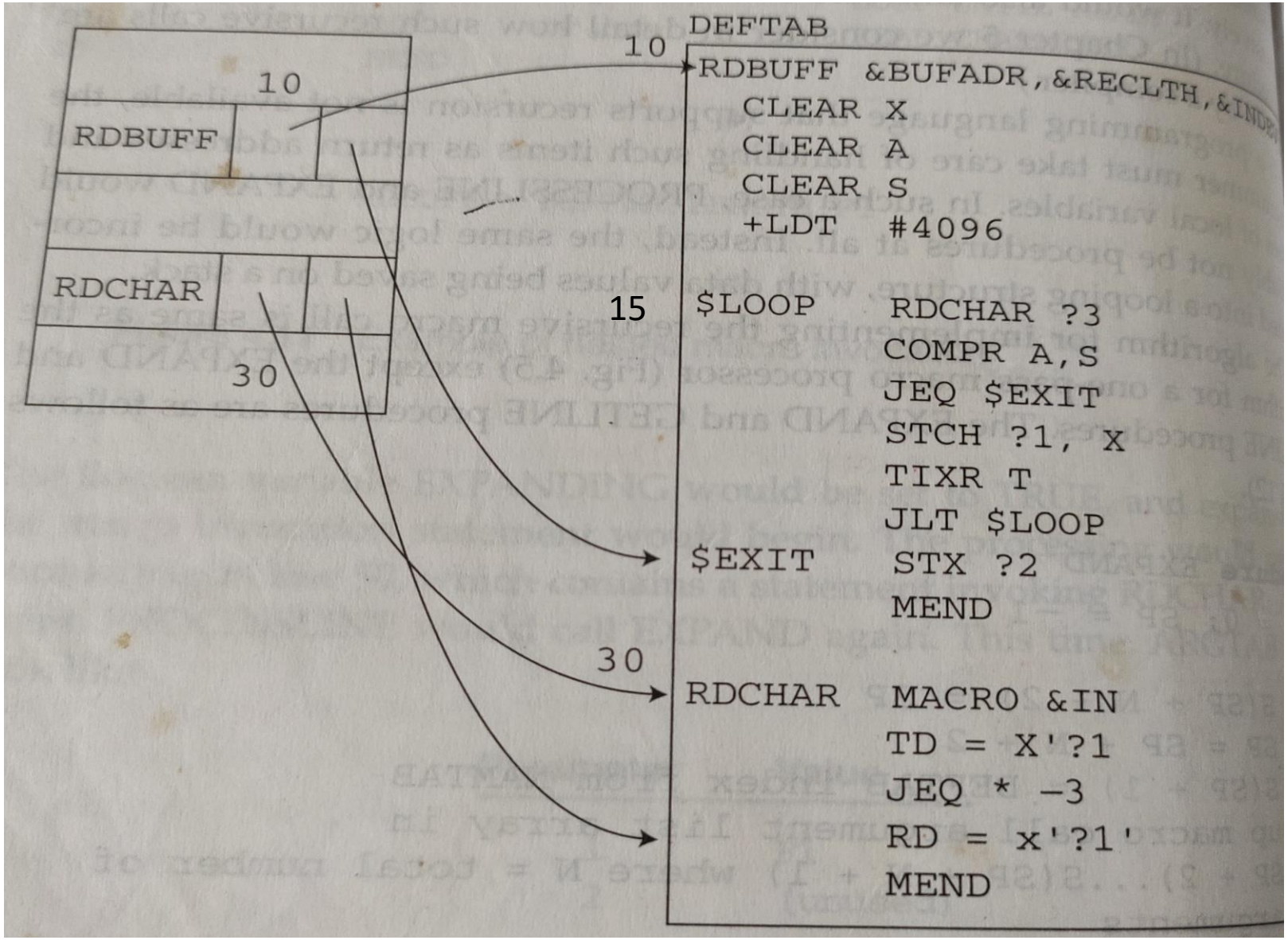
Recursive Macro Expansion

- Recursive procedures usually operate by means of stack
- Stack pointer (SP) indicates the position or frame at the top of the stack
  - SP indicates the beginning of the current stack frame
- S(SP) contains the previous value of SP, for the first frame SP=1 and S(SP) = -1
- S(SP) = -1 indicates EXPANDING=FALSE
- S(SP+1) contains the current value of DEFTAB pointer
- S(SP+2) to S(SP+N+1) contains 0^{th} to $(N-1)^{th}$ arguments

Recursive Macro Expansion



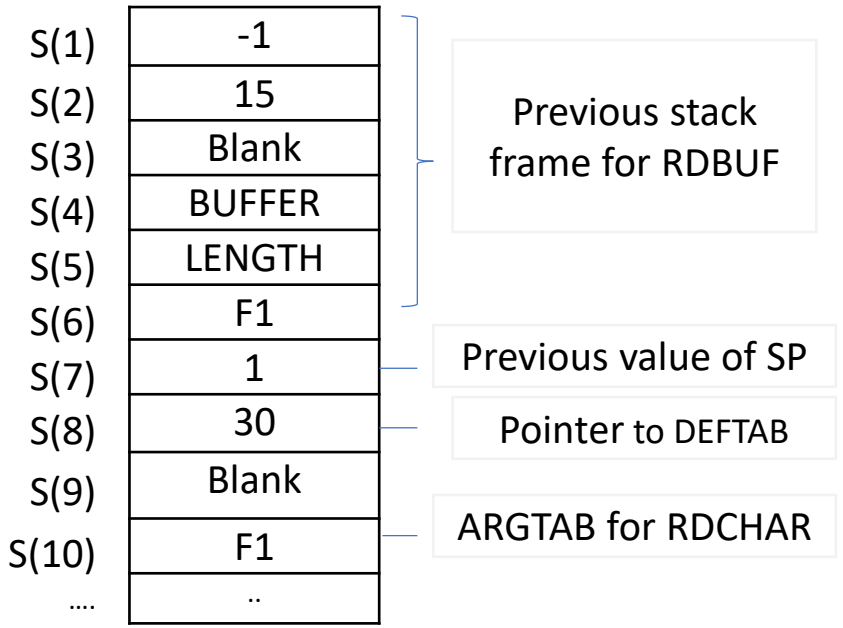
Recursive Macro Expansion



Recursive Macro Expansion

Source statement being processed	**Stack**	**Expanded Source**
Initial state during processing of macro definitions	SP=-1 S(1) -1 S(2)	
RDBUF BUFFER, LENGTH, F1	SP=1, Pointer to old stack frame S(1) -1 S(2) 10 S(3) Blank S(4) BUFFER S(5) LENGTH S(6) F1 .. …	

Recursive Macro Expansion

Source statement being processed	Stack	Expanded Source
RDCHAR F1	SP=7  	CLEAR X CLEAR A CLEAR S +LDT #4096

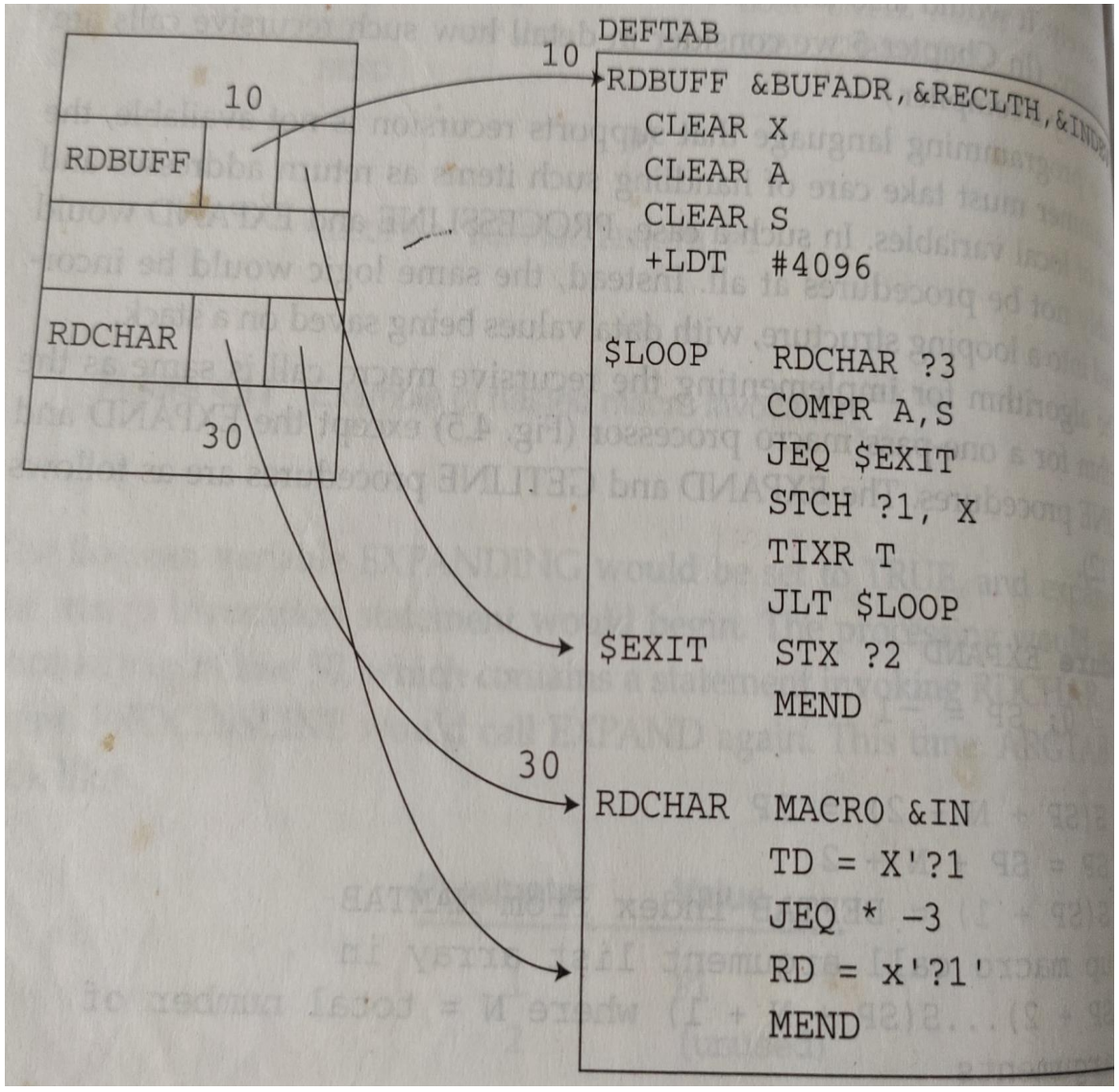
Recursive Macro Expansion

Source statement being processed	Stack	Expanded Source Statement
TD =X'?1'	SP=7 S(1) -1 S(2) 15 S(3) Blank S(4) BUFFER S(5) LENGTH S(6) F1 S(7) 1 S(8) 31 S(9) Blank S(10) F1	TD =X'F1'
JEQ *-3	SP=7 S(8) 32	JEQ *-3
RD =X'?1'	SP=7 S(8) 33	RD =X'F1'

Recursive Macro Expansion

<table>
<tr><th>Source statement being processed</th><th>Stack</th><th>Expanded Source</th></tr>
<tr>
<td>MEND
(Terminates RDCHAR Macro)</td>
<td>SP = 7
S(1) -1
S(2) 15
S(3) Blank
S(4) BUFFER
S(5) LENGTH
S(6) F1
S(7) 1
S(8) 34
S(9) Blank
S(10) F1
…. ..</td>
<td>CLEAR X
CLEAR A
CLEAR S
+LDT #4096
TD =X’F1’
JEQ *-3
RD =X’F1’</td>
</tr>
<tr>
<td>COMPR A,S</td>
<td>SP = -1
S(1) -1
S(2) 16
S(3) Blank
S(4) BUFFER
S(5) LENGTH
S(6) F1
..</td>
<td>CLEAR X
CLEAR A
CLEAR S
+LDT #4096
TD =X’F1’
JEQ *-3
RD =X’F1’
COMPR A, S</td>
</tr>
</table>

Recursive Macro Expansion



1. SP = -1
2. Call RDBUFF BUFFER, LENGTH, F1

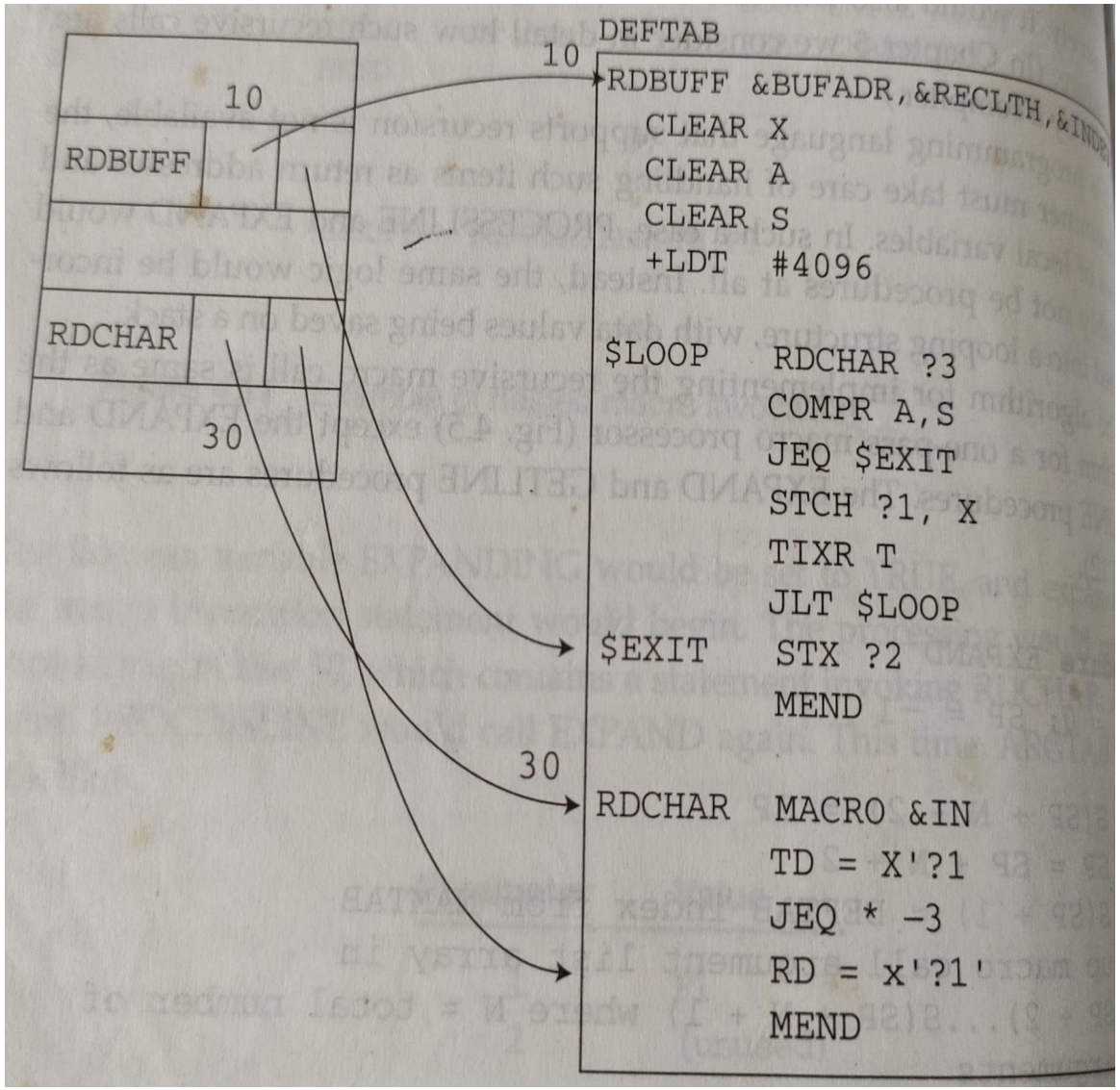
SP = 1

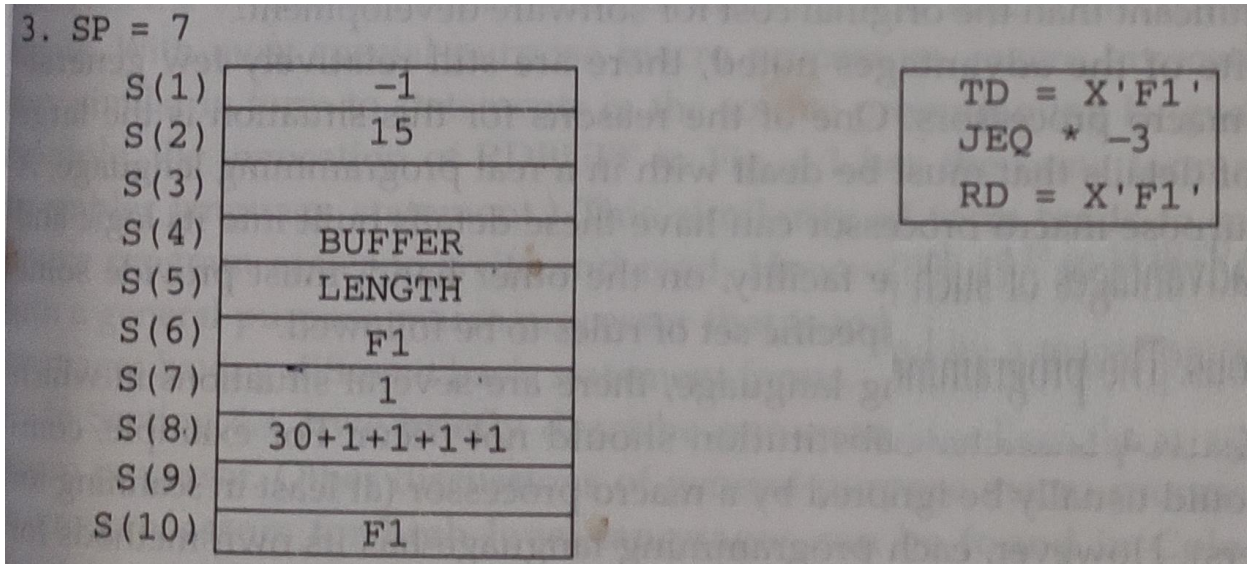
S(1)	-1
S(2)	10+1+1+1+1+1
S(3)	
S(4)	BUFFER
S(5)	LENGTH
S(6)	F1

Macros expanded

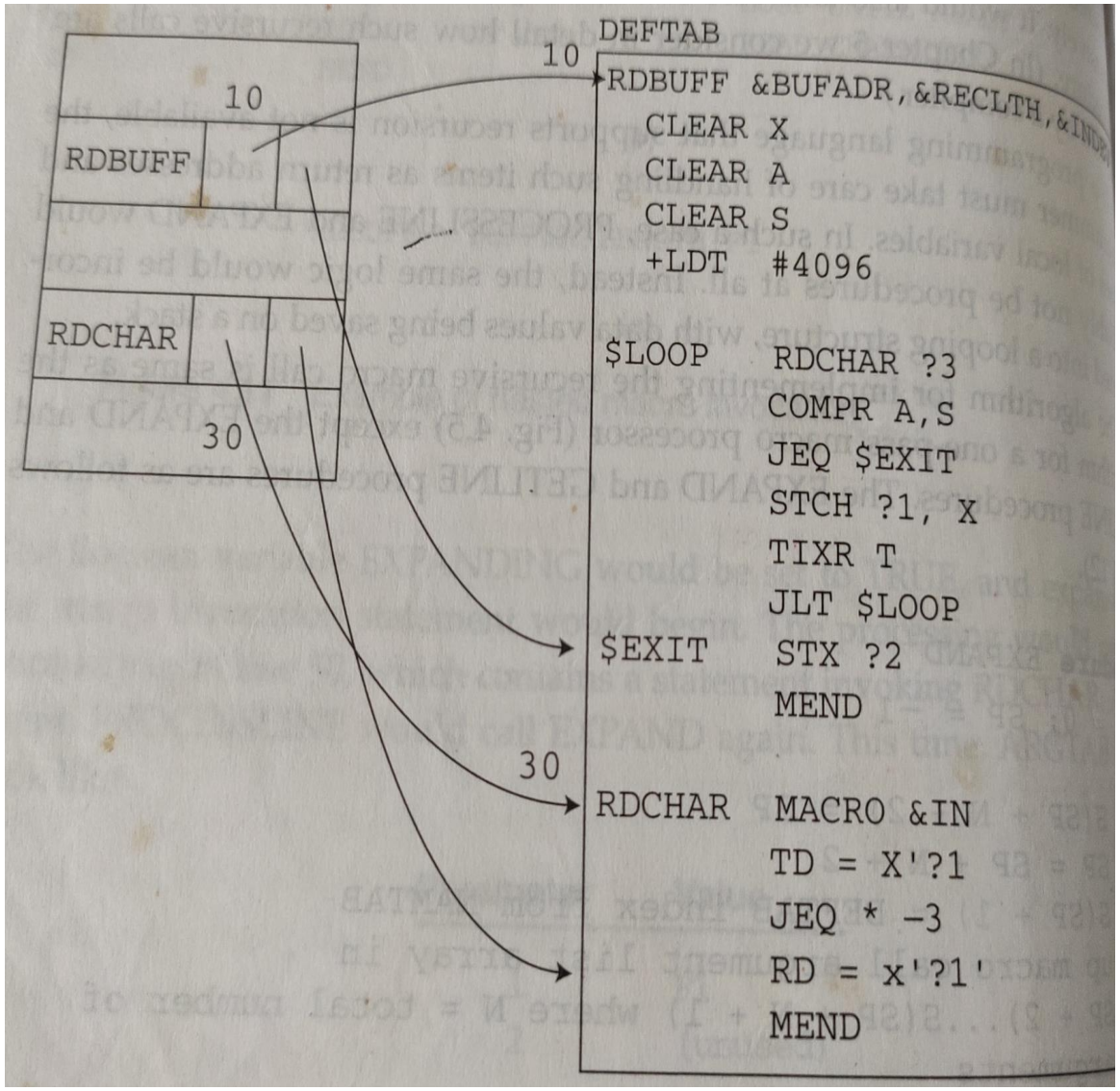
```
CLEAR X
CLEAR A
CLEAR S
+LDT   #4096
```

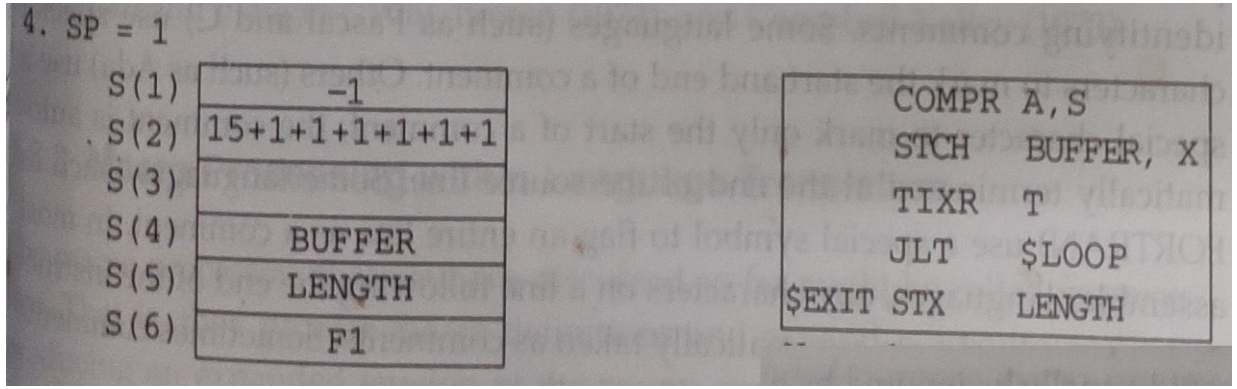
Recursive Macro Expansion





Recursive Macro Expansion



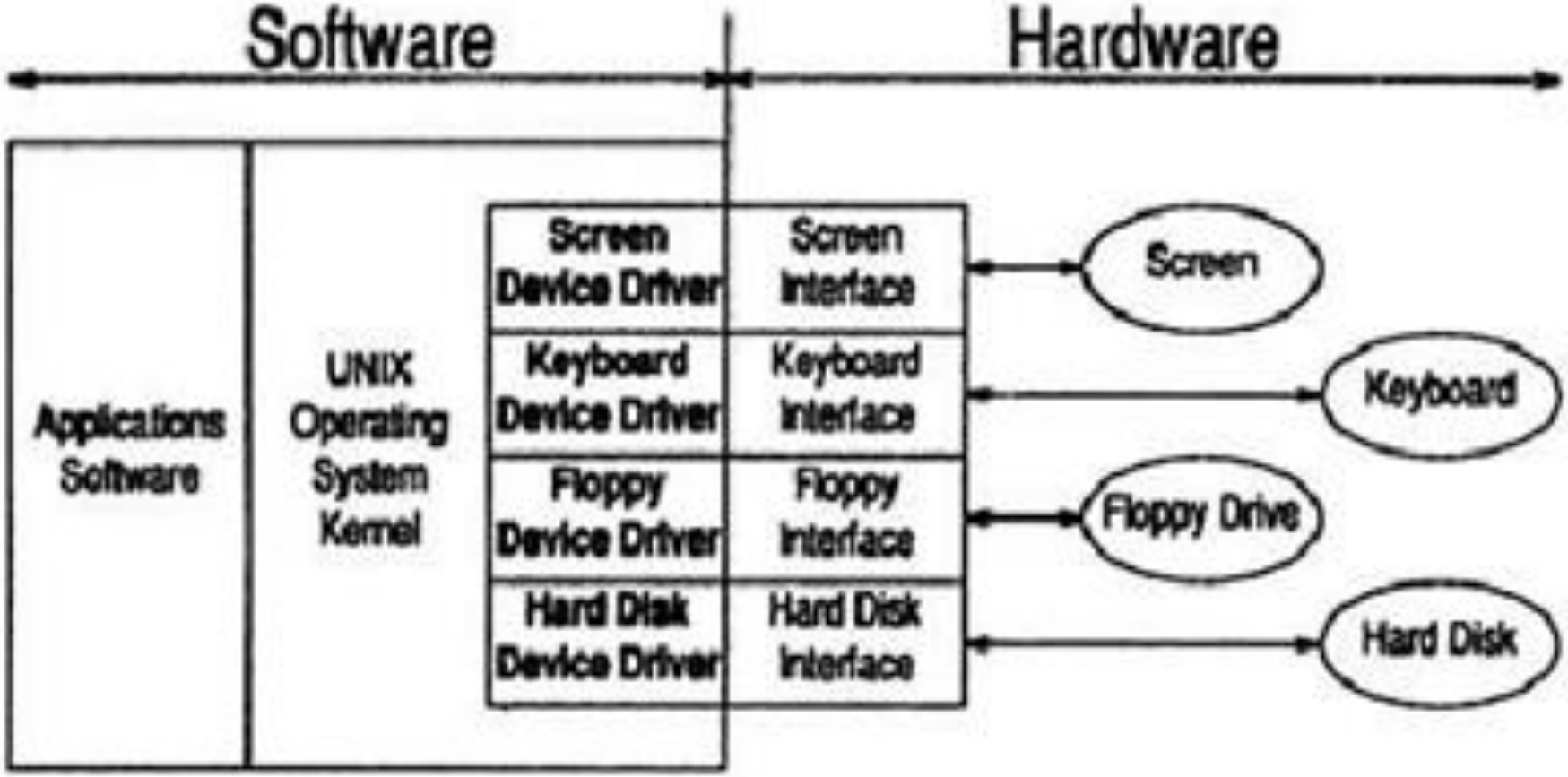


Device Driver

Device Driver

- Device driver or software driver is a computer program that operates or controls a particular type of device that is attached to a computer.
  - Resides within the kernel and is the software interface to a hardware device or devices.
  - i.e., a device driver is glue between an operating system and its I/O devices.
- A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware connects.
- Drivers are hardware-dependent and operating- system-specific.
- A typical operating system has many device drivers built into it.
- A device driver converts general IO instructions into device specific operations.
- Device driver acts as translators, converting the generic requests received from the operating system into commands that specific peripheral controllers can understand.

Device Driver



Device Driver

- A device driver is usually part of the OS kernel
  - Compiled with the OS
  - Dynamically loaded into the OS during execution
- Each device driver handles
  - one device type (e.g., mouse)
  - one class of closely related devices (e.g. SCSI disk driver to handle multiple disks of different sizes and different speeds)
- The application software makes system calls to the operating system requesting services.
- The operating system analyses these requests to the appropriate device driver.
- The device driver in turn analyses the request from the operating system and, when necessary, issues commands to the hardware interface to perform the operations needed to service the request.

Device Driver

- Without device drivers the operating system would be responsible for talking directly to the hardware.
  - This requires the operating system's designer to include support for all of the devices that users might want to connect to their computer.
  - Adding support for a new device would mean modifying the operating system itself.
- All the device driver writer has to do is to take requests from the operating system and to manipulate the hardware in order to fulfil the request.
- Drivers are hardware-dependent and operating system specific

Typical Device Driver Design

- Operating system and driver communication
  - Commands and data between OS and device drivers
- Driver and hardware communication
  - Commands and data between driver and hardware
- Driver operations
  - Initialize devices
  - Interpreting commands from OS
  - Schedule multiple outstanding requests
  - Manage data transfers
  - Accept and process interrupts
  - Maintain the integrity of driver and kernel data structures

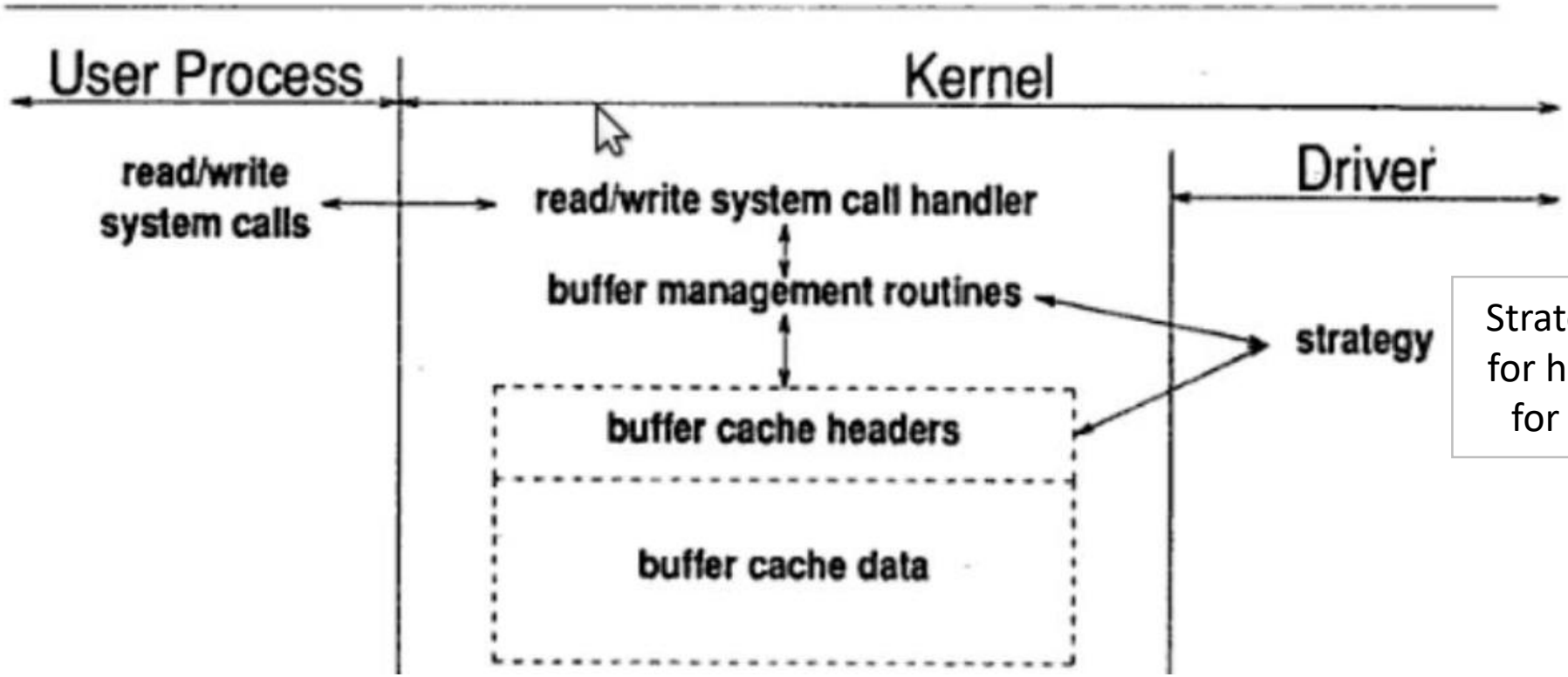
Device Driver

- Device drivers can be classified as:
  - Block device drivers
  - Character device drivers

Block Device Driver

- Block device drivers are particularly well-suited for devices that contain file system, eg: disk drives
- Have a block oriented nature; data is provided by the device in blocks
- Communicates with the operating system through a collection of fixed size buffers
- When using a block device, due to its large data throughput, a cache mechanism is used between the user process and the block device.
- OS manages a cache of these buffers and attempt to manage the user request for data by accessing buffers in the cache
- When accessing data on a block device, the system first reads the data on the block device into a high-speed buffer in the form of a data block, and then provides it to the user.
- Drivers handle requests from the OS to fill or empty fixed size buffers
- For block devices, all I/O occurs through the buffer cache.

Block Device Driver



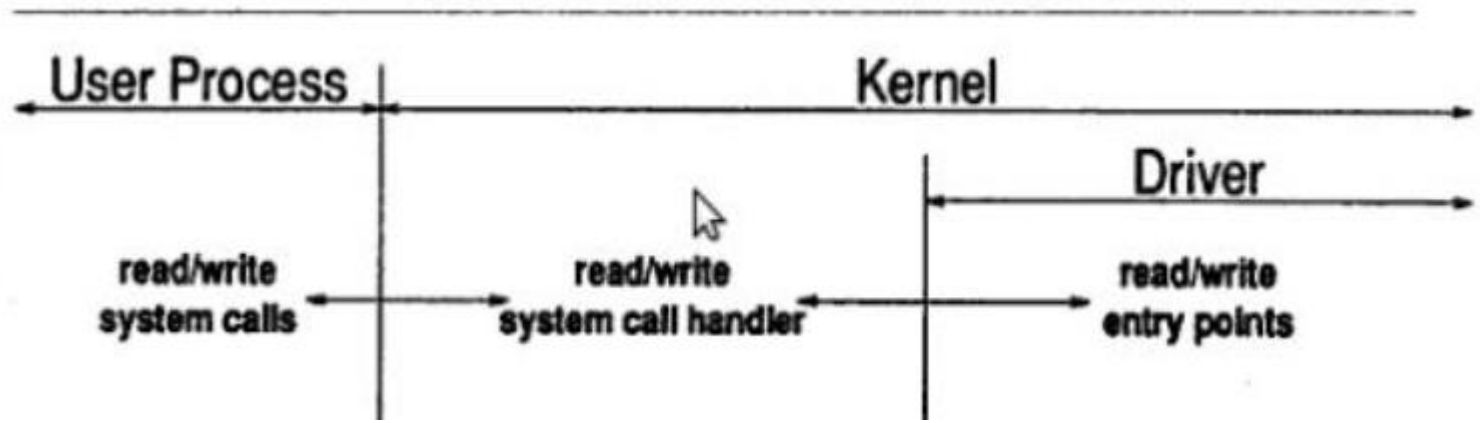
Strategy: Responsible for handling requests for data (in or out)

- Ref: https://www.slideserve.com/kita/device-drivers

Device Driver

- **Character Device Driver**
- Character devices have a stream oriented nature where data is accessed as a stream of bytes
- Can handle data of arbitrary size and can be used to support any type of device
- Character device drivers can be used
  - for devices such as a line printer that handles one character at a time or
  - for devices that handle data in chunks smaller or larger than the standard fixed size buffers used by block drivers.
  - For example, tape drivers frequently perform I/O in 10K chunks.
- Character device driver can be used when it is necessary to copy data directly to or from a user process.
- Because of their flexibility in handling I/O, many drivers are character drivers.
- I/O request is passed directly to the driver and the driver is responsible for transferring the data directly to and from the user process
- Line printers, interactive terminals, and graphics displays are examples of devices that require character device drivers.

Character Device Driver



- Ref: https://www.slideserve.com/kita/device-drivers

Device Driver

Character Driver

- Transfers data to and from user process
- Appropriate for devices that transfer data in variably sized blocks, possibly as small as a single byte
- Cannot be used to support a device that contains a UNIX file system
- Ideally suited to support printer, serial ports, sound card, keyboard, parallel ports

Block Driver

- Transfers data to and from kernel's buffer cache
- Inappropriate for devices that transfer data in variably sized blocks, possibly as small as a single byte
- Can be used to support a device that contains a UNIX file system
- Ideally suited to support disk drives, hard drives

Anatomy of device driver

- A UNIX device driver is a collection of functions written in C that can be called by the OS using C standard function calling mechanism
  - These routines are often called as ENTRY POINTS
  - init(), open(), close(), read/write calls (strategy in the case of block device drivers) etc.
- Driver is the set of entry points that can be called by the operating system
- A driver can also contain
  - Data structure private to the driver
  - References to kernel data structures external to drivers
  - Routines private to drivers (not entry points)
- Most device drivers are written as a single source file
- The initial part of a driver is sometimes called the PROLOGUE

Anatomy of device driver

- The PROLOGUE is everything before the first routine and contains
  - #include directives referencing header files which defines various kernel data structures and devices
  - #define directives that provide mnemonic names for various constants used in the driver
  - Declaration of variables and data structures

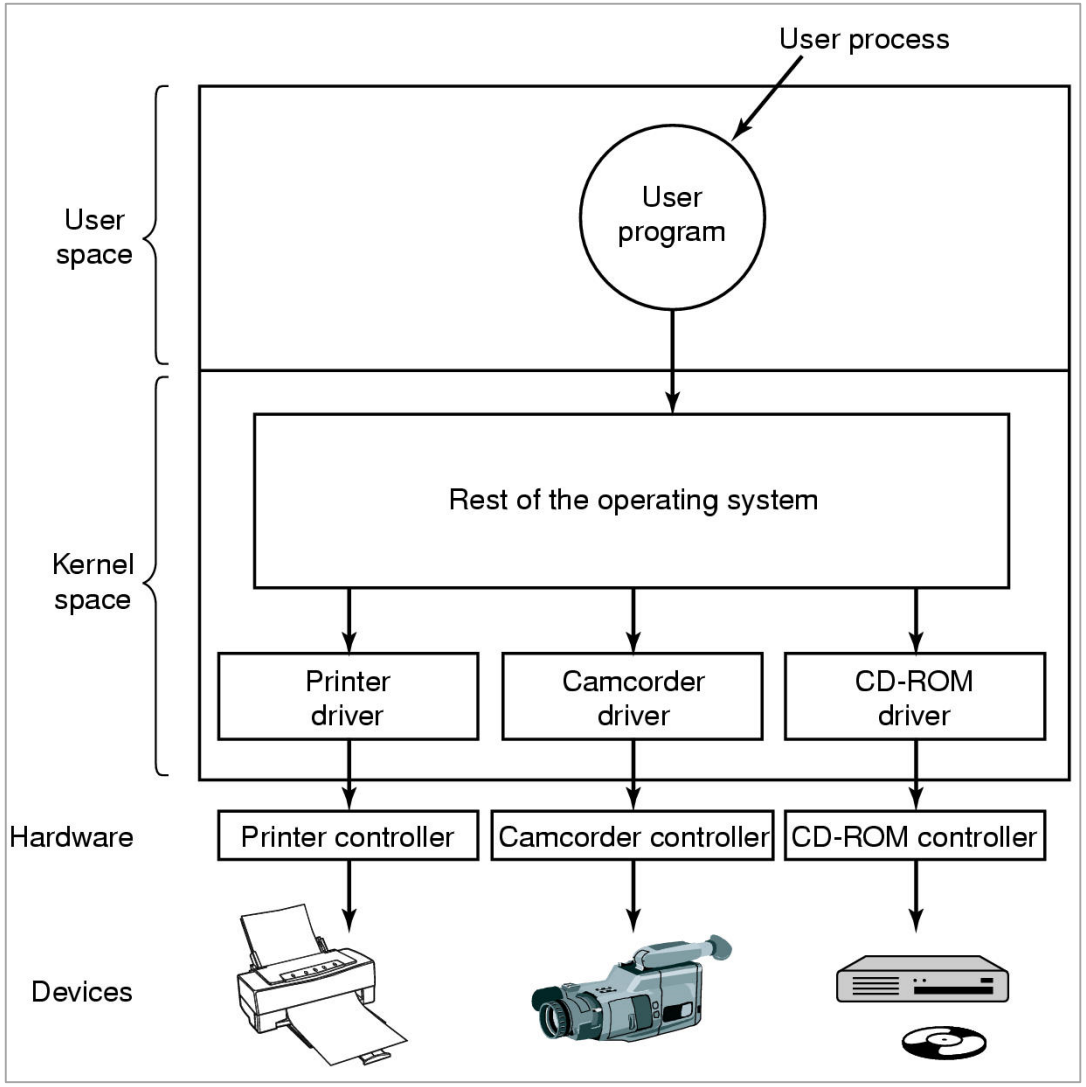
- The remaining parts of the drivers are entry points (C functions referenced by OS) and routines (C functions private to the driver)

Device Driver

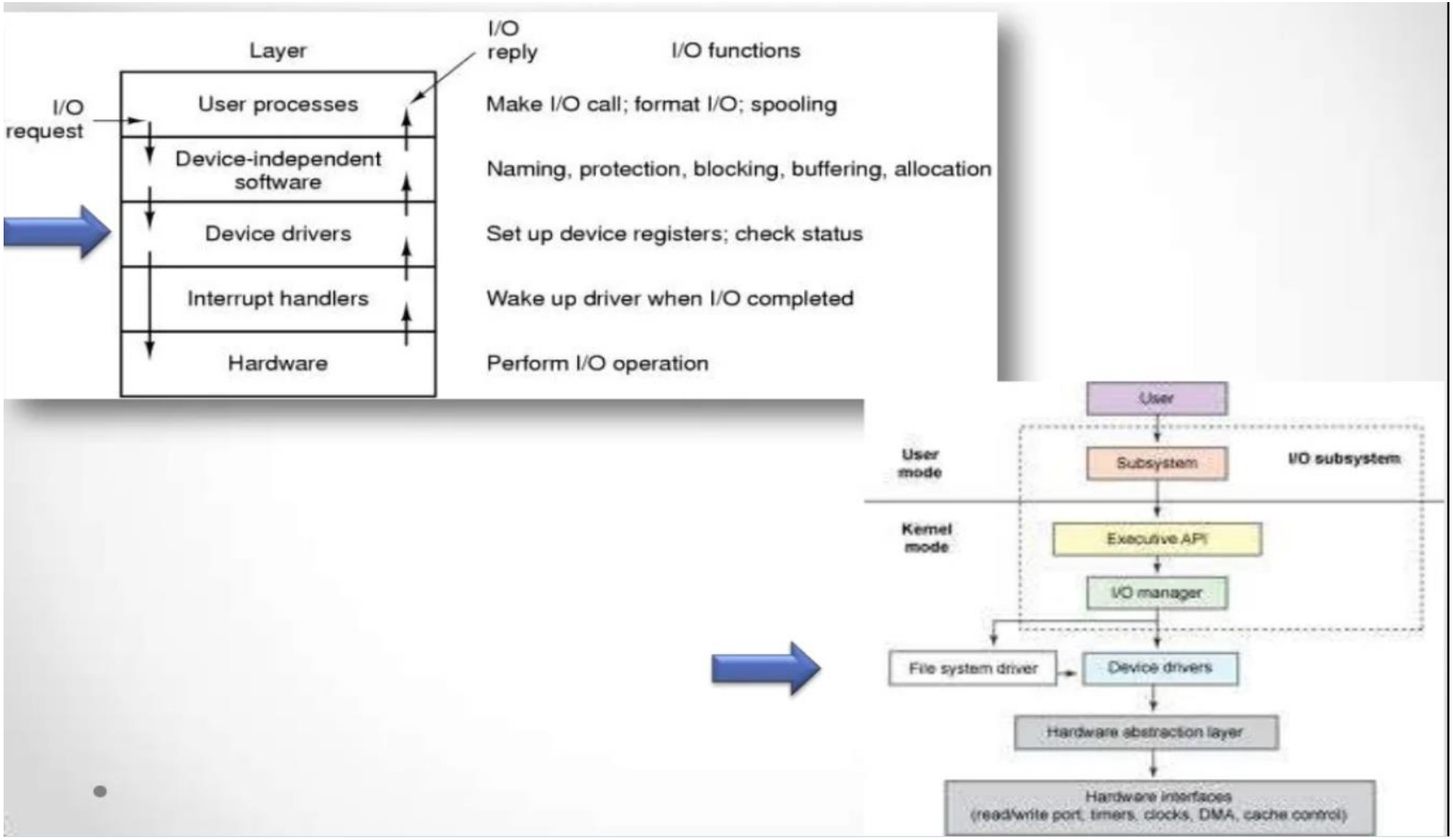
- Functions in device driver
  - Accept abstract *read and write requests* from the device independent layer above;
  - *Initialize the device*;
  - Manage *power requirements* and log events
  - *Check input parameters* if they are valid
  - Translate valid input from abstract to concrete terms
    - e.g., *convert linear block number into the head, track, sector and cylinder number* for disk access
  - Check the device if it is in use (i.e., *check the status bit*)
  - Control the device by *issuing a sequence of commands*. The driver determines what commands will be issued.

Device Driver

- Communications between drivers and device controllers goes over the bus
- Device-specific code to control an IO device, is usually written by device's manufacturer
  - Each controller has some device registers used to give it commands. The number of device registers and the nature of commands vary from device to device (e.g., mouse driver accepts information from the mouse how far it has moved, disk driver has to know about sectors, tracks, heads, etc.).



Device Driver: Location



Block Device Driver

- Block device drivers are particularly well-suited for devices that contain file system, eg: disk drives
- Users access the device through the file system, so the device driver implements a calling interface for the file system.
- Have a block oriented nature; data is provided by the device in blocks
- Communicates with the operating system through a collection of fixed size buffers
- When using a block device, due to its large data throughput, in order to be able to use the data on the block device efficiently, a cache mechanism is used between the user process and the block device.
- OS manages a cache of these buffers and attempt to manage the user request for data by accessing buffers in the cache
- When accessing data on a block device, the system first reads the data on the block device into a high-speed buffer in the form of a data block, and then provides it to the user.
- Drivers handle requests from the OS to fill or empty fixed size buffers
- For block devices, all I/O occurs through the buffer cache.

Debuggers

Debuggers

- Debugging means locating (and then removing) bugs, i.e., 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 used in context of errors that manifest while running the program **during testing or during actual use.**
- The most common steps taken in debugging are
  - 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.

Debugging

- In the absence of other mechanisms, one usually inserts statements in the program at various carefully chosen points, that prints values of significant variables or parameters, or some message that indicates the flow of control (or function call sequence).
- When such a modified version of the program is run, the information output by the extra statements gives clue to the errors.
- An interactive debugging system provides programmers with facilities that aid in the testing and debugging of programs.

Debugging Functions and Capabilities

1. Set of unit test functions

a) Execution sequencing

- Observation and control of the flow of program execution.
- The program may be halted after a fixed no. of instructions are executed

b) Break Points

- Defined by the programmer which causes execution to be suspended when a specific point in the program is reached
- After execution is suspended, other debugging commands can be used to analyze the progress of the program and to diagnose error detected; then execution of the program can be resumed.

c) Conditional expressions

- Programmer can define some conditional expressions that are continuously evaluated during the debugging sessions

d) Gaits

- Given a good graphic representation of the program progress, it may even be useful to enable the program to run at various speeds called.

Debugging Functions and Capabilities

2. **Tracing and traceback**

- 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 namely module, subroutine, branch instruction and so on.
- Traceback can show the path by which the current statement is reached.
  - For a given variable or parameter, traceback can show which statements modified it.
  - Such information should be displayed a symbolically

Debugging Functions and Capabilities

3. **Program display capabilities**

- A debugging system should have good program display capabilities
  - The program being debugged should be displayed completely statement numbers.
  - The user should be able to control the level at which this display occurs
    - Display the program as it was originally written as macro expansion and so on

4. Incremental Aspects

- Able to modify and incrementally recompile during debugging
- The system should save all debugging information (break points, display nodes etc.) across such recompilation so that the programmer need not issue all these debugging commands
- Keep track of any changes made during debugging section
  - It should be possible to symbolically display or modify the contents of any of the variables and constants in the program and then resume execution

Debugging Functions and Capabilities

- In providing the functions, a debugging system should consider the language in which the program being debugged is written.
- A single debugging tool that is applicable to multilingual situations is needed
  - When the debugger receives control, the execution of the program being debugged is suspended temporarily
  - The debugger must determine the language in which the program is written and set the ***context*** accordingly
  - The debugger should switch context when a program written in one language calls a program written in another language.
  - The user should be informed about the changes in context

Debugging Functions and Capabilities

- **Assignment statements** that change the values of variables should be processed according to the syntax and semantics of the source programming language
  - COBOL: MOVE 3.5 TO A
  - FORTRAN: A=3.5
- Conditional expressions
  - COBOL: IF A NOT EQUAL TO B
  - FORTARN: IF (A.NE.B)
- Similar differences exist with the form of statement labels, keywords and so on
- The debugger must have access to information gathered by the language translator

Debugging Functions and Capabilities

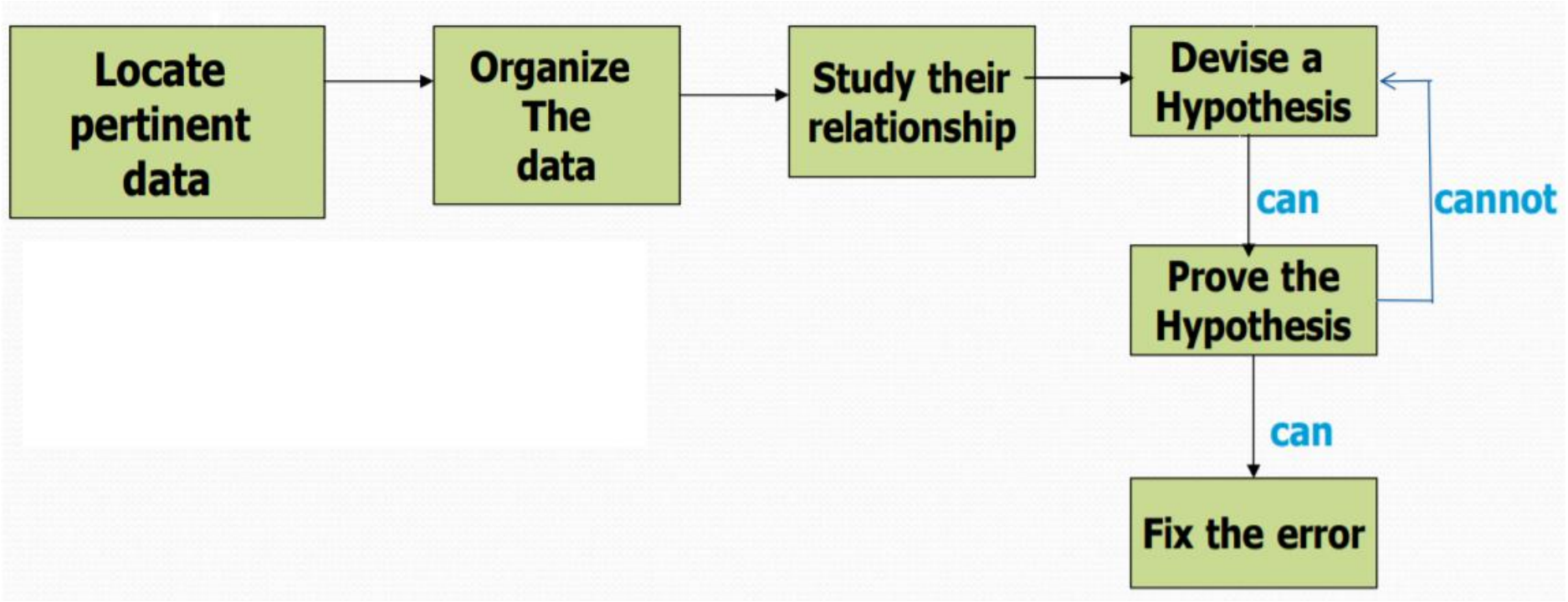
- **Optimization**
- The debugging system should be able to deal with optimized code
- Many optimizations involve rearrangement of segments of code
  - Invariant expressions can be removed from loops
  - separate loops can be combined into a single loop
  - redundant expressions may be eliminated
  - unnecessary branch instructions may be eliminated

Debugging Strategies

- As debugging is a difficult and time-consuming task, it is essential to develop a proper debugging strategy.
- This strategy helps in performing the process of debugging easily and efficiently.
- The commonly used debugging strategies are
  - induction strategy
  - deduction strategy
  - backtracking strategy

Debugging by Induction

- This strategy is a *disciplined thought process*
- Start with the clues and look for the relationships among the clues
- This strategy assumes that once the symptoms of the errors are identified, and the relationships between them are established, the errors can be easily detected by just looking at the symptoms and the relationships.



Debugging by Induction

1. **Locating relevant data**
   - Collect all available data or symptoms about the problem to identify the functions, which are executed correctly and incorrectly.
2. **Organizing data**
   - The collected data is organized according to importance (eg. of structured organization: table)
   - The data is structured to allow one to observe patterns of particular importance and search for contradictions
   - The data can consist of possible symptoms of errors, their location in the program, the time at which the symptoms occur during the execution of the program and the effect of these symptoms on the program.
3. **Devising hypothesis**
   - The relationships among the symptoms are studied and a hypothesis is devised that provides the hints about the possible causes of errors.
4. **Proving hypothesis**
   - Done by comparing the data in the hypothesis with the original data to ensure that the hypothesis explains the existence of hints completely.
   - In case, the hypothesis is unable to explain the existence of hints, it is either incomplete or contains multiple errors in it.

Debugging by Deduction

- In this strategy, first step is to identify all the possible causes and then using the data, each cause is analyzed and eliminated if it is found invalid.
  - Start with set of suspects and use process of elimination and refinement
- Deduction strategy is also based on some assumptions. To use this strategy following steps are followed.

1. Identifying the possible causes or hypotheses
2. Eliminating possible causes using the data
3. Refining the hypothesis
4. Proving the hypothesis

Debugging by Deduction

1. **Identifying the possible causes or hypotheses**
   - A list of all the possible causes of errors is formed.
   - Using this list, the available data can be easily structured and analyzed.
2. **Eliminating possible causes using the data**
   - The list is examined to recognize the most probable cause of errors and the rest of the causes are deleted.
3. **Refining the hypothesis**
   - By analyzing the possible causes one by one and looking for contradiction leads to elimination of invalid causes.
   - This results in a refined hypothesis containing few specific possible causes.
4. **Proving the hypothesis**

Backtracking Strategy

- This method is effectively used for locating errors in small programs.
- When an error has occurred, one needs to start tracing the program backward one step at a time evaluating the values of all variables until the cause of error is found.
- This strategy is useful but in a large program with many thousands lines of code, the number of backward paths increases and becomes unmanageably large.