### ****Difference between Assembly and Machine Language****

Programming languages are combination of well-defined instructions used for computers. There are two types of programming language Assembly Language and Machine Language. The programming language was developed in 1950s. The program guides the computer to precede any tasks. The grammatical rules followed in programming language are called syntax. High level language is easily understood by human while low language is difficult to understand.

**Assembly Language**

It was developed in 1940s. Before the invention of Assembly language the computer code were written in 0s and 1s which is difficult for human to understand. Second generation language is known as Assemble Language. Assemble Language is converted into machine code by using an assembler.

**Machine Language**

Machine Language is in the form of 0s and 1s. This language is not easily understandable by the human but this is the real language of computer which the computer understands in the form of bytes. It is first generation language. And it is much difficult than Assembly Language and more time is required for understanding this language.

### ****Assembly Language VS Machine Language****

**Readability:**

The readability of Assembly Language is very high as it is written in English which is understandable by human beings.

The readability of Machine Language is less as compared to Assembly Language because it is written in the form of binary code which normal human being cannot understand. But this is the actual language of computer.

**Platform Dependency:**

The Assembly Language is platform dependent and so at the present time most programs are written in third generation language.

Machine Language varies from platform. Different platform have different machine language code.

**Modifiable:**

 It is necessary for survival of software to evolve.  So in order to avail this advantage the language should be easily modifiable. And Assembly Language is easily Modifiable and its support changes.

Machine Language cannot be changes easily and it does not support modification.

**Quality:**

The risk of occurrence of error is reduced in Assembly Language.

The risk of existence of error is high in Machine Language.

**Memorability:**

Memorability is high in Assembly Language because it is easy to remember variable name instead of binary code.

The binary code cannot be memorized.

**Need of Compiler:**

In case of Assembly Language interpreter is needed which is knows as assembler in case of Assembly Language to convert its code into machine code in the form of bits and bytes.

In case of Machine Language there is no need of compiler or interpreter because it is mother language of computer. So computer does not need any interpreter to understand its own language.

## What is the Difference Between Machine Language and Assembly language?

|  |  |
| --- | --- |
| Machine language is the lowest level programming language where the instructions execute directly by the CPU. | Assembly language is a low-level programming language which requires an assembler to convert to machine code/object code. |
| Machine language is comprehensible only to the computers. | Assembly language is comprehensible to humans. |
| A machine language consists of binary digits. | Assembly language follows a syntax similar to the English language. |
| Machine language varies depending on the platform. | Assembly language consists of a standard set of instructions. |
| Machine language is machine code. | Assembly language is using for microprocessor-based, real-time systems. |

**Features of ML:** The language made up of binary coded instructions built into the hardware of a particular computer and used directly by the computer.

Characteristics of machine language: – Every processor type has its own set of specific machine instructions – The relationship between the processor and the instructions it can carry out is completely integrated – Each machine-language instruction does only one very low-level task

*Machine Language* is the language used by the computer. It is comprised entirely of ONs and OFFs (denoted by 1's and 0's), and varies from one (type of) machine to another.

Our discussion will use *typical* features of machine languages and assembly languages in general, not those for any specific machine.

The general format of an instruction in machine code is

**<operation code> <location/value> <location/value> <location/value>**

Where **<operation code>** is a *binary code* corresponding to some instruction, and **<location/value>** is either a memory location or a numerical value. For example, a typical piece of machine language code might look like this:

**00000100 10000000**  
**00000000 10000001**  
**00000101 10000000**  
**00001111 00000000**

For human eyes, this code would be rendered into a special numbering system called "hex" that makes it easier to read. (We will learn about hex later).

**04 80**  
**00 81**  
**05 80**  
**0F 00**

This particular machine code would read as

**Perform operation 4 on memory location 128**  
**Perform operation 0 on memory location 129**  
**Perform operation 5 on memory location 128**  
**Perform operation 15 on memory location 0**

Recall the machine code from the previous slide:

|  |  |  |
| --- | --- | --- |
| **00000100 10000000** | *meaning* | **Perform operation 4 on memory location 128** |
| **00000000 10000001** | *meaning* | **Perform operation 0 on memory location 129** |
| **00000101 10000000** | *meaning* | **Perform operation 5 on memory location 128** |
| **00001111 00000000** | *meaning* | **Perform operation 15 on memory location 0** |

In order to understand this code, we need know what these various operations actually are. For example, the programmer (who creates the instructions) needs to know that "operation 4" is really "Load."

|  |
| --- |
| **Load the contents of memory location 128 into a register** |
| **Add the contents of memory location 129 to the contents of the register** |
| **Store the result (the value in the register) in memory location 128** |
| **Halt (no memory location needed)** |

What does this code do, anyway? Maybe you can figure it out, but maybe not. In general, giving instructions to a computer through machine language is considered too difficult to be practical these days. Nevertheless, it was the only method available when computers were first developed.

**What are macro languages and its features?**

A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. There are two kinds of macros. They differ mostly in what they look like when they are used. Object-like macros resemble data objects when used, function-like macros resemble function calls. . Give symbolic names to constants, so global changes need be made in only one place. . Conditionally compile blocks of code. . Abbreviate or customize the language of frequently used blocks of coding where a subroutine call is not desired. . Improve readability of the code to make its structure and purpose more obvious.

**Features of the Macro Language**

Although subsequent sections go into far more detail on the various elements of the macro language, this section highlights some of the possibilities, with pointers to more information.

macro statements

This section has illustrated only a few of the macro statements, such as %MACRO and %IF-%THEN. Many other macro statements exist, some of which are valid in open code, while others are valid only in macro definitions. For a complete list of macro statements.

macro functions

Macro functions are functions defined by the macro facility. They process one or more arguments and produce a result. For example, the %SUBSTR function creates a substring of another string, while the %UPCASE function converts characters to uppercase. A special category of macro functions, the macro quoting functions, mask special characters so they are not misinterpreted by the macro processor.

There are two special macro functions, %SYSFUNC and %QSYSFUNC, that provide access to SAS language functions or user-written functions generated with SAS/TOOLKIT. You can use %SYSFUNC and %QSYSFUNC with new functions in Base SAS software to obtain the values of SAS host, base, or graphics options. These functions also enable you to open and close SAS data sets, test data set attributes, or read and write to external files. Another special function is %SYSEVALF, which enables your macros to perform floating-point arithmetic.

autocall macros

Autocall macros are macros defined by SAS that perform common tasks, such as trimming leading or trailing blanks from a macro variable's value or returning the data type of a value..

automatic macro variables

Automatic macro variables are macro variables created by the macro processor. For example, SYSDATE contains the date SAS is invoked..

macro facility interfaces

Interfaces with the macro facility provide a dynamic connection between the macro facility and other parts of SAS, such as the DATA step, SCL code, the SQL procedure, and SAS/CONNECT software. For example, you can create macro variables based on values within the DATA step using CALL SYMPUT and retrieve the value of a macro variable stored on a remote host using the %SYSRPUT macro statement.

**A macro facility** is used to interpret macro definitions and expand each macro call as it occurs with the requisite pattern of assembly language statements, providing expanded source code ready for the assembler. Hence, the macro facility is a preprocessor, which interprets all macro calls into assembly code prior to passing the expanded code on to the assembler. A macro facility is an add-on piece of system software, a convenience for the programmer, to facilitate production of multiple lines of commonly occurring code via single macro calls embedded in the programmer's assembly program. The macro preprocessor included with C compilers (with calls such as #include) uses the same idea, albeit assembly language macro facilities predate the similar compiler preprocessors

.

|  |
| --- |
|  |

**What is macro and explain features of macro facility?**

features:- . conditional macro expansion . concatenation of macro parameters . generation of unique labels . macro instruction arguments . expansion time variables . expansion time loops

**Macro and Macro Processors**

**Explain following terms with suitable example.**

1. **Expansion time variable**
2. **Positional parameter**
3. **Semantic expansion**
4. **Macro Pre-processor**
5. **Macro**

* **Expansion time variable**

 Expansion time variables (EV's) are variables which can only be used during the expansion of macro calls.

* A local EV is created for use only during a particular macro call.
* A global EV exists across all macro calls situated in a program and can be used in any macro which has a declaration for it.
* Local and global EV's are created through declaration statements with the following syntax:

                                  LCL <EV specification> [,<EV specification> .. ] GBL <EV specification> [,<EV specification> .. ]

                                  <EV specification> has the syntax &<EV name>, where <EV name> is an ordinary string.

* Values of EV's can be manipulated through  the preprocessor statement SET.
* A SET statement is written as:

                                  EV specification > SET <SET-expression>where < EV specification > appears in the label field and SET in mnemonic field.

* A SET statement assigns value of <SET-expression> to the EV specified in <EV specification>.

**Example**

MACRO

CONSTANTS

LCL &A

&A SET 1

DB &A

&A SET &A+l

DB &A

MEND

The local EV A is created.

* The first SET statement assigns the value '1' to it.
* The first DB statement thus declares a byte constant ‘1’.
* The second SET statement assigns the value '2' to A and the second DB statement declares a constant '2'

**Positional parameters**

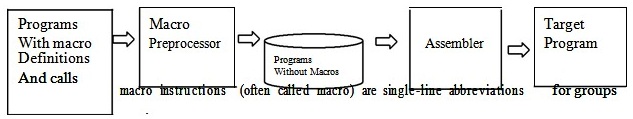
* For positional formal parameters, the specification <parameter kind> of syntax rule is simply omitted. Thus, a positional formal parameter is written as &<parameter name>, e.g., &SAMPLE where SAMPLE is the name of a parameter.
* In a call on a macro using positional parameters the<actual Parameter specification> is an ordinary string.
* The value of a positional formal parameter XYZ is determined by the rule of positional association as follows:
* Find the ordinal position of XYZ in the list of formal parameters in the macro prototype statement.
* Find the actual parameter specification that occupies the same ordinal position in the list of actual parameters in the macro call statement. If it is an ordinary string ABC, the value of formal parameter XYZ would be ABC.

**Semantic Expansion**

* Implies generation of instructions tailored to the requirements of a specific usage.
* Semantic expansion is characterized by the fact that different uses of a macro can lead to codes which differ in the number, sequence and op-codes of instructions.
* Eg: Generation of type specific instructions for manipulation of byte and word operands.

**Macro-preprocessor**

* The macro preprocessor accepts an assembly program containing definitions and calls and translates it into an assembly program which does not contain any macro definitions and calls.
* The program form output by the macro preprocessor can be handed over to an assembler to obtain the target program.



 For every occurrence of this one-line macro instruction within a program, the instruction Must be replaced by the entire block.

* The advantages of using macro are as follows:
* Simplify and reduce the amount of repetitive coding.
* Reduce the possibility of errors caused by repetitive coding. Make an assembly program more readable.

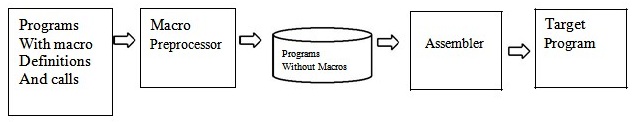
**What is macro preprocessor? Explain steps of macro preprocessor design.**

**OR**

**Explain design specification tasks for macro preprocessor with suitable example.**

**Macro-preprocessor**

* The macro preprocessor accepts an assembly program containing definitions and calls and translates it into an assembly program which does not contain any macro definitions and calls.
* The program form output by the macro preprocessor can be handed over to an assembler to obtain the target program.



* Macro preprocessors are vital for processing all programs that contain macro definitions and/or calls.
* Language translators such as assemblers and compilers cannot directly generate the target Code from the programs containing definitions and calls for macros.
* Therefore, most language Processing activities by assemblers and compilers preprocess these programs through macro Processors.
* A macro preprocessor essentially accepts an assembly program with macro definitions and calls as its input and processes it into an equivalent expanded assembly program with no macro definitions and calls.
* The macro preprocessor output program is then passed over to an assembler to generate the target object program.

**Recognize macro calls:**

* A table is maintained to store names of all macros defined in a program. Such a table is called Macro Name Table (MNT) in which an entry is made for every macro definition being processed.
* During processing program statements, a match is done to compare strings in the mnemonic field with entries in the MNT. A successful match in the MNT indicates that the statement is a macro call.

**Determine the values of formal parameters:**

* A table called Actual Parameter Table (APT) holds the values of formal parameters during the expansion of a macro call. The entry into this table will be in pair of the form (<formal parameter name>, <value>).
* A table called Parameter Default Table (PDT) contains information about default parameters stored as pairs of the form (<formal parameter name>, <default value>) for each macro defined in the program. If the programmer does not specify value for any or some parameters, its corresponding default value is copied from PDT to APT.

**Maintain the values of expansion time variables declared in a macro:**

* A table called Expansion time Variable Table (EVT) maintains information about expansion variables in the form (<EV name>, <value>).
* It is used when a preprocessor statement or a model statement during expansion refers to an EV.

**Organize expansion time control flow:**

* A table called Macro Definition Table (MDT) is used to store the body of a macro.
* The flow of control determines when a model statement from the MDT is to be visited for expansion during macro expansion. MEC {Macro Expansion Counter) is defined and initialized to the first statement of the macro body in the MDT. MDT is updated following an expansion of a model statement by a macro preprocessor.

**Determine the values of sequencing symbols:**

* A table called Sequencing Symbols Table (SST) maintains information about sequencing symbols in pairs of the form(<sequencing symbol name>, <MDT entry#>)where <MDT entry #> denotes the index of the MDT entry containing the model statement with the sequencing symbol.
* Entries are made on encountering a statement with the sequencing symbol in their label field or on reading a reference prior to its definition.

**Perform expansion of a model statement:**

* The expansion task has the following steps:
* MEC points to the entry in the MDT table with the model statements.
* APT and EVT provide the values of the formal parameters and EVs, respectively.
* SST enables identifying the model statement and defining sequencing.

**Define macros of your choice to illustrate nested calls to these macros. Also show their corresponding expansion.**

* A model statement in a macro may constitute a call on another macro.Such calls are known as nested macro calls.
* Macro containing the nested call is the outer macro and, Macro called is inner macro. They follow LIFO rule.

**EXAMPLE**

+ MOVEM BREG, TMP

+ MOVER BREG, X

+ ADD BREG, Y

+ MOVEM BREG, X

+ MOVER BREG, TMP

MACRO

COMPUTE &FIRST, &SECOND

MOVEM BREG, TMP

INCR\_D &FIRST, &SECOND, REG=BREG

MOVER BREG, TMP

MEND

COMPUTE X, Y:

+ MOVEM BREG, TMP [1]

+ INCR\_D X, Y

• + MOVER BREG, X [2]

• + ADD BREG, Y [3]

• + MOVEM BREG, X [4]

+ MOVER BREG, TMP [5]

**What are advanced macro programming facilities? Explain with example.**

**OR  
Explain with examples - expansion time variables, expansion time statements - AIF and AGO for macro programming. Show their usage for expansion time loop by giving example.**

**AIF**

* An AIF statement has the syntax:
* Where <expression>is a relational expression involving ordinary strings, formal parameters and their attributes, and expansion time variables.
* If the relational expression evaluates to true, expansion time control is transferred to the statement containing <sequencing symbol> in its label field.

**AGO**

* An AGO statement has the syntax: AGO <sequencing symbol>
* It unconditionally transfers expansion time control to the statement containing <sequencing symbol> in its label field.

**Expansion time loops or (Explain expansion time loop)**

* It is often necessary to generate many similar statements during the expansion of a macro.
* This can be achieved by writing similar model statements in the macro.
* Expansion time loops can be written using expansion time variables (EV’s) and expansion time control transfer statements AIF and AGO.

**Example**

MACRO

CLEAR &X, &N

LCL &M

&M SET

MOVER AREG, =’0’

.MORE MOVEM AREG, &X+&M

&M SET &M + 1

AIF (&M NEN)

.MORE

MEND

* The LCL statement declares M to be a local EV.
* At the start of expansion of the call, M is initialized to zero.
* The expansion of model statement MOVEM, AREG, &X+&M thus leads to generation of the statement MOVEM AREG, B.
* The value of M is incremented by 1 and the model statement MOVEM. is expanded repeatedly until its value equals the value of N.

**Expansion time variable**

* Expansion time variables (EV's) are variables which can only be used during the expansion of macro calls.
* A local EV is created for use only during a particular macro call.
* A global EV exists across all macro calls situated in a program and can be used in any macro which has a declaration for it.
* Local and global EV's are created through declaration statements with the following syntax:

                   LCL <EV specification> [,<EV specification> .. ] GBL <EV specification> [,<EV specification> .. ]

                   <EV specification> has the syntax &<EV name>, where <EV name> is an ordinary string.

* Values of EV's can be manipulated through  the preprocessor statement SET.
* A SET statement is written as:

                      EV specification > SET <SET-expression>where < EV specification > appears in the label field and SET in mnemonic field.

* A SET statement assigns value of <SET-expression> to the EV specified in <EV specification>.

**Example**

MACRO

CONSTANTS

LCL &A

&A SET 1

DB &A

&A SET &A+l

DB &A

MEND

The local EV A is created.

* The first SET statement assigns the value '1' to it.
* The first DB statement thus declares a byte constant ‘1’.
* The second SET statement assigns the value '2' to A and the second DB statement declares a constant '2'.

**Attributes of formal parameter**

* An attribute is written using the syntax <attribute name> ‘ <formal parameter spec>
* It represents information about the value of the formal parameter, i.e. about the corresponding actual parameter.
* The type, length and size attributes have the names T, L and S.

Example

MACRO

DCL\_CONST &A

AIF (L'&A EQ 1) .NEXT

--

.NEXT --

--

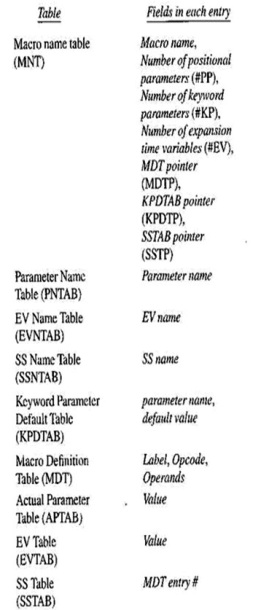
MEND

* Here expansion time control is transferred to the statement having .NEXT field only if the actual Parameter corresponding to the formal parameter length of ' 1'.

**Explain use and field of following macro.**

**KPDTAB, MDT, EVTAB, SSTAB**

* EV name are entered in EVNTAB while processing EV declarations.
* SS name are entered in SSNTAB while processing an SS reference or definition, whichever occur earlier.
* Each entry contains three pointers
* MDTP,KPDTP and SSTP which are pointers to
* MDT, KPDTAB and SSNTAB for the macro respectively.
* KPDTAB are constructed by processing the prototype statement.
* Entries are added to EVNTAB and SSNTAB as EV declarations and SS definitions/references are encountered.
* MDT entries are constructed while processing model statements and preprocessor statements in macro body.
* SSTAB entries, when the definition of sequencing symbol in encountered



**Explain following facilities for expansion time loop with example.**

**(1)REPT statement                       (2) IRP statement**

* **REPT statement**

* Syntax: REPT <expression>
* < expression > should evaluate to a numerical value during macro expansion.
* The statements between REPT and an ENDM statement would be processed for expansion <expression> number of times.

**Example**

MACRO

CONST10

LCL &M

&M SET 1

REPT 10

DC &M?

&M SETA &M+1

ENDM

MEND

* **IRP statement**

* IRP <formal parameter>, <argument-list>

           Formal parameter mentioned in the statement takes successive values from the argument list.

* The statements between the IRP and ENDM statements are expanded once.

**Example:**

MACRO

CONSTS &M, &N, &Z

IRP &Z, &M=7, &N

DC ‘&Z

ENDM

MEND

* A MACRO call CONSTS 4, 10 leads to declaration of 3 constants with the values 4, 7 and 10.

**Draw a flowchart and explain simple one pass macro processor.**

* A one-pass macro processor is another design option available for macro processing.
* The restriction in working with one-pass macro processors is that they strictly require the definition of a macro to appear always before any statements that invoke that macro in the program.

The important data structures required in a one-pass macro processor are:

**DEFTAB (Definition Table):**

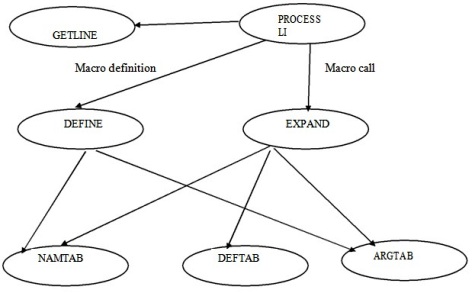
* It is a definition table that is used to store the macro definition including macro prototype and macro body.
* Comment lines are not included here, and references to the parameters use positional notation for efficiency in substituting arguments.

**NAMTAB (Name Table):**

* This table is used for storing macros names. It serves as an index to DEFTAB and maintains pointers that point to the beginning and end of the macro definition in DEFTAB.

**ARGTAB (Argument Table):**

* It maintains arguments according to their positions in the Argument list.
* During expansion, the arguments from this table are substituted for the corresponding parameters in the macro body.
* One-pass Macro Processor scheme is presented as below.



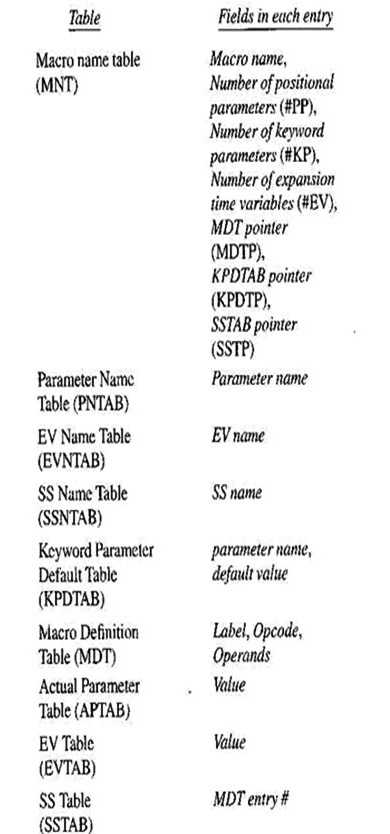
**Describe tasks and data structures considered for the design of a macro preprocessor.**

* To obtain a detailed design of the data structure it is necessary to apply the practical criteria of processing efficiency and memory requirements.
* The table APT, PDT and EVT contain pairs which are searched using the first component of the pairs as a key-the formal parameter name is used as the key to obtain its value from APT.
* This search can be eliminated if the position of an entity within a table is known when its value is accessed.
* The value of formal parameter ABC is needed while expanding a model statement using it MOVER AREG, &ABC
* Let the pair (ABC, 5) occupy entry #5 in APT. the search in APT can be avoided if the model statement appears as

           MOVER

           AREG, (P, 5)

* In the MDT, where (P, 5) stand for the word „parameter #5?.
* The first component of the pairs stored in APT is no longer used during macro expansion e.g. the information (P,5) appearing in model statement is sufficient to access the value of formal parameter ABC,APT containing (<formal parameter name>,<value>) pairs is replaced by another table called APTAB which Only contains <value>?s.
* Ordinal number are assigned to all parameters of macro, a table named parameter name table (PNTAB) is used for this purpose.
* Parameter name are entered in PNTAB in same order in which they appear in the prototype statement.
* The information (<formal parameter name>, <value>) in APT has been split into two tables.
* PNTAB-which contains formal parameter names
* APTAB-which contains formal parameter values
* PNTAB is used while processing a macro definition while APTAB is used during macro expansion.
* Similar analysis leads to splitting of EVT into EVNTAB and EVTAB and SST into SSNTAB and SSTAB.
* EV name are entered in EVNTAB while processing EV declarations.
* SS name are entered in SSNTAB while processing an SS reference or definition, whichever occur earlier.
* The positional parameter of macro appear before keyword parameters in the prototype statement.
* If macro have p positional parameter and k keyword parameters, then keyword parameters have the ordinal number p+1, p+2 ...P+k
* Due to this numbering redundancies appear in PDT.
* Entry only needs to exist for parameter number p+1, P+2 ...P+k
* So, replace parameter default table (PDT) by a keyword parameter default table (KPDTAB), this table have only k entries.
* MNT has entries for all macros defined in a program, each entry contains three pointers MDTP, KPDTP and SSTP which are pointers to MDT, KPDTAB and SSNTAB for the macro respectively.



**Explain attributes of formal parameters, default specifications of parameter and semantic expansion for macro by giving examples.**

**OR**

**Explain positional parameters, keyword parameters and default value parameters for macros.**

**Types of formal parameters**

**Positional parameters**

* For positional formal parameters, the specification <parameter kind> of syntax rule is simply omitted. Thus, a positional formal parameter is written as &<parameter name>, e.g., &SAMPLE where SAMPLE is the name of a parameter.
* In a call on a macro using positional parameters the<actual Parameter specification> is an ordinary string.
* The value of a positional formal parameter XYZ is determined by the rule of positional association as follows:
* Find the ordinal position of XYZ in the list of formal parameters in the macro prototype statement.
* Find the actual parameter specification that occupies the same ordinal position in the list of actual parameters in the macro call statement. If it is an ordinary string ABC, the value of formal parameter XYZ would be ABC.

**Keyword parameters**

* For keyword parameters, the specification <parameter kind> is the string '=' in syntax rule.
* The <actual parameter specification> is written as <formal parameter name> = <ordinary string>. The value of a formal parameter is determined by the rule of keyword association as follows:
* Find the actual parameter specification which has the form XYZ= <ordinary string>.
* If the <ordinary string> in the specification is some string ABC, the value of formal parameter XYZ would be ABC.

**Specifying default values of parameters**

* If a parameter has the same value in most calls on a macro, this value can be specified as its default value in the macro definition itself.
* If a macro call does not explicitly specify the value of the parameter, the Preprocessor uses its default value; otherwise, it uses the value specified in the macro call.
* This way, a programmer would have to specify a value of the parameter only when it differs from its default value specified in the macro definition.
* Default values of keyword parameters can be specified by extending the syntax of formal parameter specification as follows:& < parameter name > [< parameter kind > [< default value >]]

**Macros with mixed parameter lists**

* A macro definition may use both positional and keyword parameters. In such a case, all positional parameters must precede all keyword parameters in a macro call.
* For example, in the macro call SUMUP A,B,G=20,H=X
* A and B are positional parameters while G and H are keyword parameters. Correspondence between actual and formal parameters is established by applying the rules governing positional and keyword parameters separately.

II

A **one-pass macro processor** is another design option available for macro processing. The restriction in working with one-pass macro processors is that they strictly require the definition of a macro to appear always before any statements that invoke that macro in the program.

###### **The important data structures required in a one-pass macro processor are:**

* **DEFTAB (Definition Table):** It is a definition table that used to store the macro definition including macro prototype and macro body. Comment lines are not included here, and references to the parameters use a positional notation for efficiency in substituting arguments.
* **NAMTAB (Name Table):** This table used for storing macros names. It serves as an index to DEFTAB and maintains pointers that point to the beginning and end of the macro definition in DEFTAB.
* **ARGTAB (Argument Table):** It maintains arguments according to their positions in the argument list. During expansion, the arguments from this table substituted for the corresponding parameters in the macro body.
* One-pass Macro Processor scheme presented as below.

## https://i2.wp.com/freestudy9.com/wp-content/uploads/2017/10/fgfd.jpg?resize=365%2C182Dynamic Linking

##### **Static Linking One-pass Macro Processors**

* In static linking, the liner links all modules of a program before its execution begins; it produces a binary program that does not contain any unresolved external references.
* If statically linked programs use the same module from a library, each program will get a private copy of the module.
* If many programs that use the module are in execution at the same time, many copies of the module might be present in memory.

##### **Dynamic Linking One-pass Macro Processors**

* Dynamic linking performed during execution of a binary program.
* The linker invoked when an unresolved external reference and resumes execution of the program.
* This arrangement has several benefits concerning use, sharing and updating of library modules.
* If the module referenced by a program has already been linked to another program that in execution, a copy of the module would exist in memory. The same copy of the module could line to his program as well, thus saving memory.
* To facilitate dynamic linking, each program first processed by the static linker.
* The static linker links each external reference in the program to a dummy module whose

**Explain two pass macro processor with Flowchart and databases.**

it is used for for identifying the macro name and performing expansion.

**Features of macro processor:**

1. Recoganized the macro definition
2. Save macro definition
3. Recoganized the macro call
4. Perform macro expansion

**Forward reference Problem**

The assembler specifies that the macro definition should occur anywhere in the program .

So there can be chances of macro call before it’s definition witch gives rise to the forwards reference problem od macro

Due to witch macro is devided into two passes

1. PASS 1-

Recoganize macro definition save macro definition

1. PASS 2-

Recoganize macro call perform macro expansion

**Databases required for pass 2**

In pass2 we perform recognize macro call and perform macro expansion

1.COPY FILE

It is a file it contains the out put given from PASS1

2.MNT

It is used for recognizing macro name

3.MDT

It is used to perform macro EXPANSION

4.MDTP

It is used to point to the index of MDT .

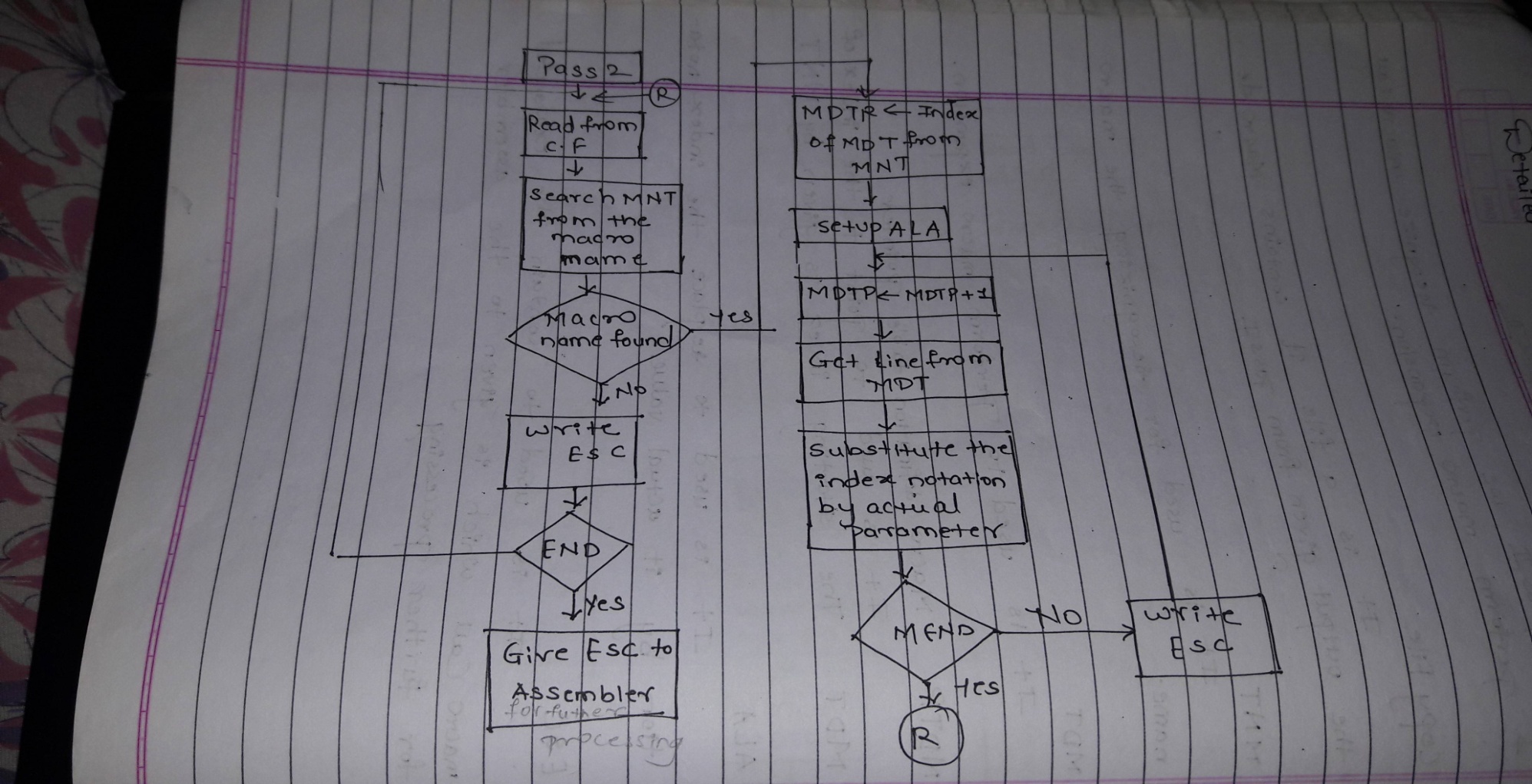
The starting index is given by MNT

5.ALA

It is used to replace the index notation by it actual value

6.ESC

It is used to contain the expanded macro call which is given to the assembler for further processing



.DESIGN:Two –pass Macro Pre-processor

##### **Pass 0 of Assembler Design: Two-pass Macro Preprocessor**

The activities of a pass-0 macro processor is given in the following steps:  
1. Read and examine the next source statement.  
2. If MACRO statement, continue reading the source and copy the entire macro definition to the MDT. Go to Step 1.  
3. If the statement is a pass-0 directive, execute it. Go to Step 1. (These directives are written to the new source file in a unique manner (different from normal directives). They are only needed for the listing in pass 2.

4. If the statement contains a macro name, it must perform expansion, that is, read model statements from the MDT corresponding to the call, substitute parameters, and write each statement to the new source file (or execute it if it is a pass-0 directive). Go to Step 1.  
5. For any other statement, write the statement to the new source file. Go to Step 1.  
6. If the current statement contains the END directive, stop (end of pass 0).

##### **The assembler will be in one of the three modes: Design: Two-pass Macro Preprocessor**

* In the normal mode, the assembler will read statement lines from the source file and write them to the new source file. There is no translation or any change in the statements. In the macro definition mode, the assembler will continuously copy the source file to the MDT.
* In the macro expansion mode, the assembler will read statements from the MDT, substitute parameters, and write them to the new source file. Nested macros can be implemented using the Definition and Expansion (DE) mode.

##### **Pass 1 of Macro Processor – Processing Macro Definitions**

1.Initialize MDTC and MNTC.

2. Read the next source statement of the program.

3.If the statement contains MACRO pseudo-op. go to Step 6.

4.Output the instruction of the program.

5. If the statement contains END pseudo-op, go to Pass 2, else go to Step 2.

6. Read the next source statement of the program.

7. Make an entry of the macro name and MTDC into MNT at location MNTC and increment the MNTC by 1.

8. Prepare the parameter (arguments) list array.

9. Enter the macro name into the MDT and increment the MTDC by 1.

10. Read the next card and substitute index for the parameters (arguments).

11. Enter the statement into the MDT and increment the MDT by 1.

12. If MEND pseudo-op found, go to Step 2, else go to Step 10.

##### **Pass 2 of Macro Processor – Processing for Calls and Expansion of Macro**

1.Read the next source statement copied bypass 1.

2.Search into the MNT for a record and evaluate the operation code.

3.If the operation code has a macro name, go to Step 5.

4.Write the statement to the expanded source file.

5.If END pseudo-op found, pass the entire expanded code to the assembler for assembling and stop. Else go to Step 1.

6. Update the MDTP to the MDT index from the MNT entry.

7. Prepare the parameter (argument) list array.

8. Increment the MDTP by 1.

9. Read the statement from the current MDT and substitute actual parameters (arguments) from the macro call.  
10. If the statement contains MEND pseudo-op, go to Step 1, else write the expanded source code and go to Step 8.