UNIT-II

Loaders and Linkers

Loaders:-

A loaders is a part of operating system that is responsible for loading programs and library files. It is one of the Essential Stage to start a program it places program into memory and prepare them for Execution. loading a program involves reading the components of Executable files containg the program instructions in to the memory and carry required tasks prepared for Execution.

The Embedded systems difficulty do not shave leaders instead the code directly Execute from the Rom. In order to load the operating system itself as part of booting a specialized boot strap loader is used. loader is an utility program which takes object as input and loads the Executable code into the memory. Those loader is responsible for initiating the Execution process

Basic function of loaders:-

There are 4 important functions performed by the leader

1. Allocation
2. Linking
3. Relocation
4. loading

1. Allocation:-

It allocates the space for a program into the memory by allocating the size of program

2. Linking:-

It resolves the symbolic references such as code/data b/w the object modules by assigning all the user sub-routines and library subroutine address

3. Relocation:-

Their are some address dependent location in the program. such address constants must be adjusted according to the allocated space is called Reallocation

4. loading:-

It place all machine instructions and data of corresponding programs and sub-routine into the memory. Thus the program now becomes ready for Execution. this

activity is called loading.

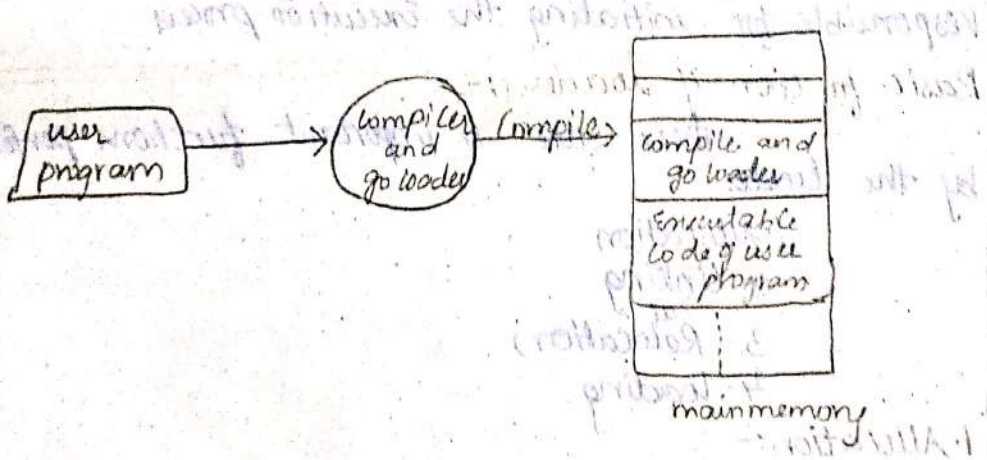
Loader Scheme:-

Based on various functionalities on loaders their are various types of loaders

1. compile and go loader
2. general loader scheme
3. Absolute loader

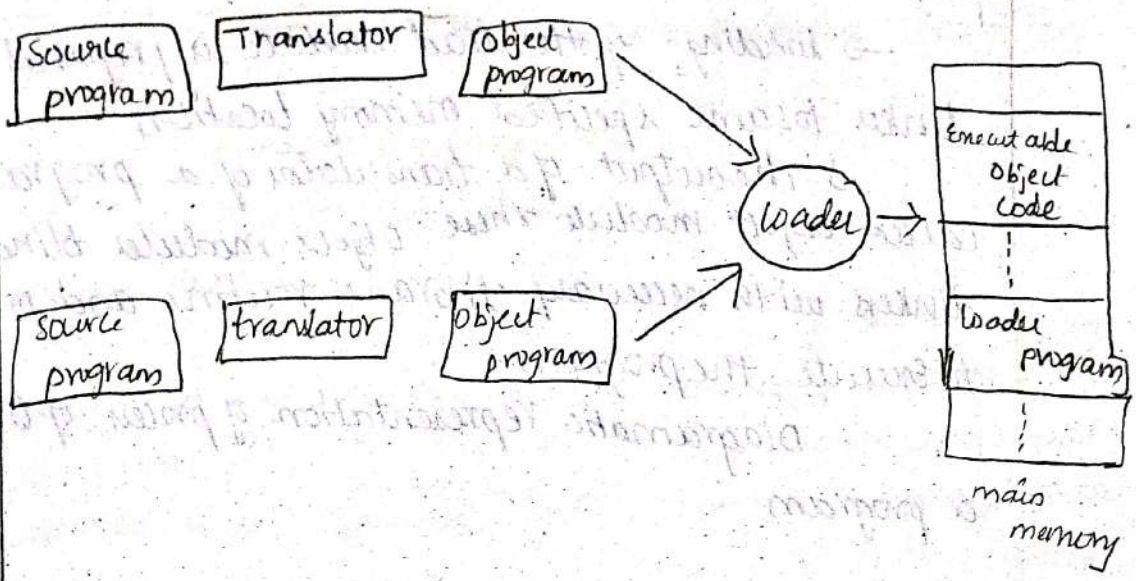
1. Compile and go loader:-

In this type of loader the instruction is red line by line its machine code is optain and it is directly stored in the main memory that means assembler has runs one part of memory. The machine instruction and data directly stored into their asigned memory locations. After complition of assembly prolcss assigned the starting address of the program to the location counter



2. General loader scheme:-

In this loaders scheme the source program is converted into the object program by some translators. the loader accepts this object modules and place machine instructions and data in an Executable form at their assigned memory. The loader occupies some portion of main memory



3. Absolute ~~memory~~ or loader :-

Absolute loader is a kind of loader in which relocated object files are created. loader accepts these files and placed them at specified location in memory. this type of loader is called as absolute loader because no relocation information is need rather it is option from the programmer and assembler.

The absolute loader is simple to implement in the scheme which has the following features such as.

→ allocation is done by either programmer (or) assembler

→ linking is done by the programmer (or) assembler

→ Relocation is done by assembler

Linkers :-

A linker (or) link Editor is a computer program that takes one (or) more objects file generated by the compiler and combines them into a single Execution file, library file or another object file.

Concepts of link Editor :-

The Execution of a program can be done with the help of following steps

→ Translation of the program can be done by assembler (or) compile

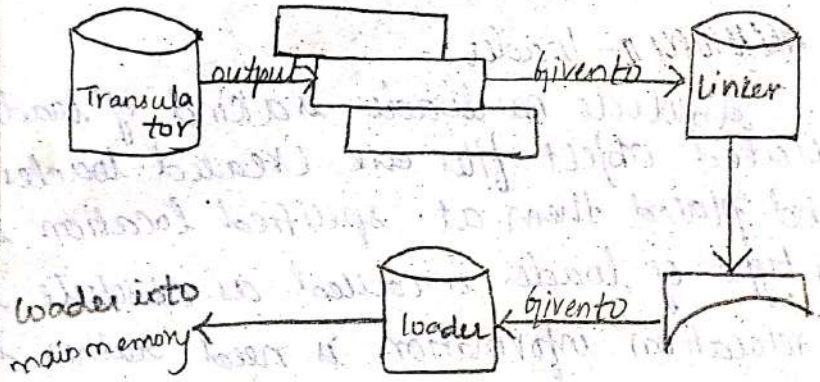
→ Linking of the program with all other progra which are needed for Execution. this preparation of

program is called load module

→ loading of the load module is prepared by a linker to same specified memory location

→ The output of a translator of a program is called object module. These object modules binds by linkers with necessary libraries routine and prepares to execute the program.

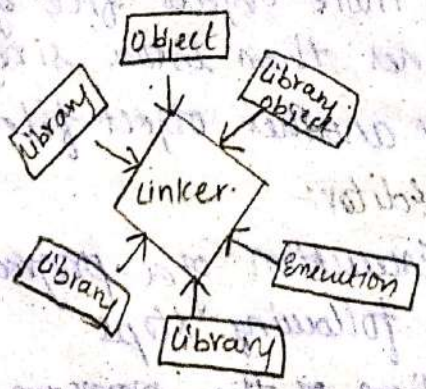
Diagramatic representation of proces of linking a program



Various tasks a linkers:-

→ It prepars a single load modules and adjust all address and subroutines reference with respect to the offset location.

→ to prepare a load module concatinate all object modules and adjust all the operand References as well as External reference to the offset locations



Types of linking :-

Their are two types of linking that are available in linkers. They are

1. Static linking
2. Dynamic linking

1. Static linking :-

Static linking is the of linker copying all # library routines used in program into the Executable code this may require more disk space and memory then dynamic linking but it is more portable since it does not require the presents of library where it runs. static linking is performed as compilation time.

2. Dynamic linking :-

many operating system Environment allows dynamic linking that resolves some undefine symbols until the program is run. It means that the Executable code still contains undefined symbols, a list of objects (or) libraries that provides the definition, loading the program means loading the objects alone with libraries and perform an final linking Dynamic linking needs no linker and it is performed at runtime

Advantages :-

→ It often uses standard system library need to be stored in only one location and no duplication is presents in it

→ If a bugg in a library function is corrected by replacing the library. rall program used by it dynamically until benefit the correct after restarting them.

Disadvantages :-

An incompletable updated library will break the Executable file that depends on the previous version of the library.

---

machine Independent loader features :-

In this section we discuss some loader features that are not related to the machine architecture design loading and linking are often called as

operating system functions.

They are two types of machine independent loader features. they are

1. Automatic ~~the~~ library search
2. loader options.

1. Automatic library search:-

many loaders can automatically incorporate routines from a sub program library into a program being loaded. In most ~~the~~ there is a standard system library that is used. In this way other libraries may be specified by control statements (or) by parameters to the loader. It allows the programmes to use the subroutines from the libraries that are automatically fetched and linked to the main program.

library routines are External interfaces in which the user can include routines to override library routine. Library search is a search of the directly that contains address of the routine linking loaders that suppor the automatic theory ~~see~~ of External symbols but not defined in the primary input. Note that the subroutine fetched from the library contains the External references. It is necessary to repeat library search proces until all the references are resolved. this proces allows the programmers to override standed subroutine in the library by supplying their own subroutine it is possible to search libraries by scaning the records for all the objects.

2. loader options:-

many loader allows the users to specify options that modify the standed proceisor. In this section we discusses some typically loader options and their use. many loaders that have aspecial language is used to specify options. some time their is a seperate input file to the loader that contains control statements that can be embedded in the primary input stream b/w the object program. the programmes can include loader control statements in the source program and the

assembler (or) compiler retains these commands as a part of the object program.

The loader options which are specified using command language. But their are other responsibilities on some system options that are specified as a part of job control language that is processed by operating system. this approach is used when the O.S incorporates the option specified in the control block that is made available to the loader when it is invoked. some of the examples of loader options are include, delete, change etc

machine dependent loader features:-

In this sections we discuss some loaders features that are related to machine architecture which machine dependent.

They are two different types of machine dependent loader features. They are

1. program Relocation
2. program linking.

The absolute loader is certainly simple and efficient. However this scheme has several disadvantages one of the need for programmer to specify the actual address where it will be located in the memory. If we consider a small computer with a smaller memory. This doesn't create much difficulty on a larger and more advanced machine the situation is not Easy we would like to run several independent programs together sharing b/w them. This means that we don't know in advance where a program will be loaded

we consider the design and implementation of more complex loader. the loader will present on SIC/XE system. and these are found in most modern computers. this loader provides program relocation and linking as well as loading function.

1. program relocation

1. loaders that allows a program for relocatable are called Relocating loaders (or) relative loaders.
2. there are 2 methods for specifying program

relocation as a part of object program such as
a. modification records
b. Relocation bits

a. modification records:-
It is the first method which is used for small number of relocation required when relative & immediate addressing modes are extensively used

b. Relocation bits :-
If there are many addresses to be modified a

instead of modification records to specify every relocation
1. Relocation bits are used when the instruction formate is fixed
2. It is used when their is a relocation bits for each word of the object program
3. the Relocation bits are put together into a bit mask. If the relocation bit corresponding to a word of object code

4. A Relocation bit '0' indicates that no modification is necessary & '1' indicates that modification is necessary

2. program linking :-
1. A program is a logical Entity that combines all of the related control sections.
2. control sections could be assembled together if they could be assembled independently of one another
3. control sections are to be linked, relocated and loaded by the loader
4. We consider more complex Example of External references b/w the program and examine the relationship b/w ~~relation~~ relocation and linking.
5. Each program contains a list of items such as list A, list B, list C Etc. the End of these lists are

marked by the labels End A, End B, End C etc

6. the labels on the beginning and of list are External symbols and each program contain some set of references to these External symbols.

7. the instruction operands are represented as Ref 1 through Ref 3. and Ref 4 through Ref 8.

Ref 1

It is a simple reference to a label with in the program. It is assembled as the usual way as a program counter relations instructions. No modification for relocation or linking is necessary

Ref 2 :-

It is processor in similar manner for program A. The second operand consist of External reference with a constant.

Ref 3 :-

It is an immediate operand whose value shows difference b/w list A and End A.

Loader design options :-

The loader design option is a important features for the loader these design options helps the loaders to load a program into memory which is used for execution. The loader design options include relocation and linking, the 2 alternative in the loader design are linking editors and dynamic linking. priority to the loading Editors which performs linking priority to the loading time and dynamic linking is performed at Execution time. there are different types of loader design options available in loader they are

1. linkage Editors
2. Dynamic linking
3. Bootstrape loaders.

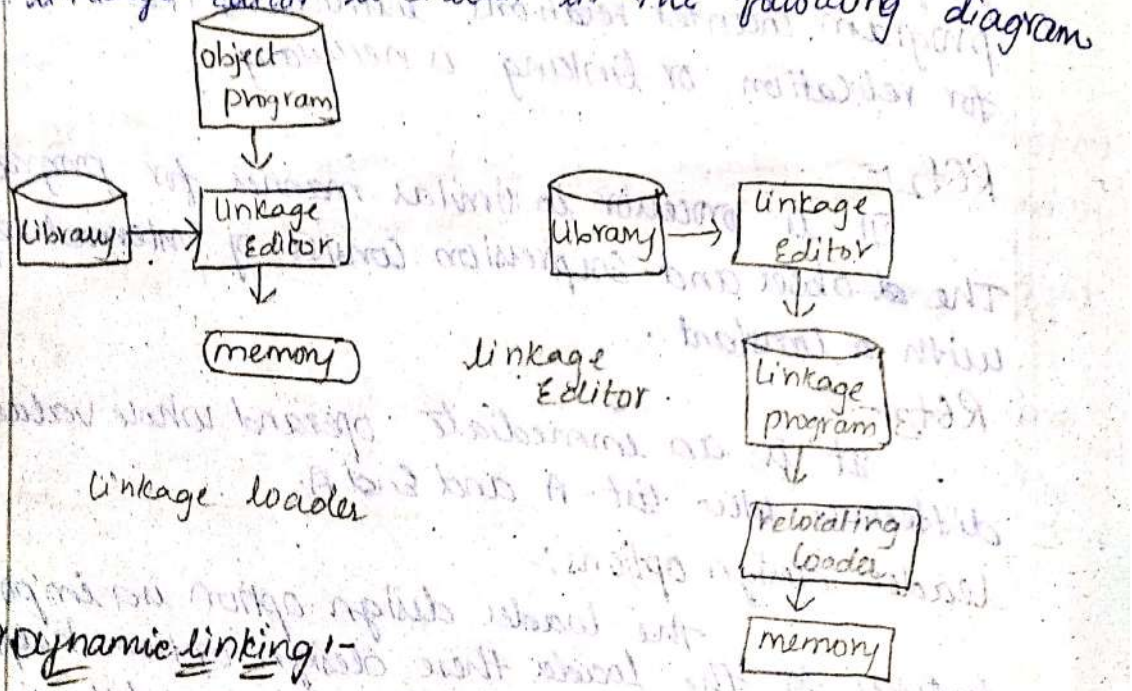
1: Linkage Editors :-

The source program is first assembled or compiled to produce an object program. A linkage loader performs all linking and relocating operations including

automatic library search. A linkage editor produces a linked version of a program called "load module" The linkage editor performs relocation of all control sections in an order to start the linked program.

Linkage Editors are also used to build packages of subroutines that are generally used together. This can be useful when dealing with sub-routine libraries that supports high level programming language.

The differences b/w a linkage loader and linkage Editor is shown in the following diagram



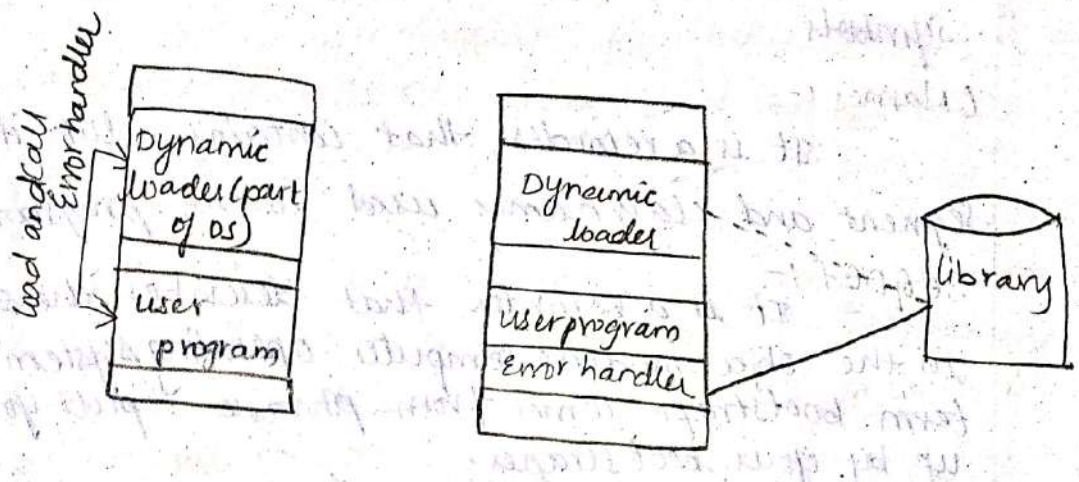
Dynamic linking :-

In this section we discuss scheme that postponed the linking function until the execution time. This type of function is called "dynamic linking" and "dynamic loading". Dynamic linking allows several executing program to share a copy of subroutine. For example, it supports high level language like C and C++ that could be represented as a dynamic link.

In the object oriented system, dynamic linking is used for references to an object. The allows the implementation of the object and its methods at the time of program execution. It also offers some other advantages over other type of linking such that dynamic linking provides the ability to load routines when they are needed. If the subroutines are large we have external references this can result in

There are different machanisms that can be used to accomplish actual loading and linking of subroutines.

The following diagram illustrates amethod in which subroutines are dynamically linked.



3. Bootstrape loader:-

Bootstrape loader is a program. It is a bootstrape that resides in computers Eprom or Rom which are non volatile memory. It is automatically executed by the processor while turning on the compiler. It leads the hard drive to continue the proces of loading.

→ LNAMES, SEDEF, GRPDEF } Segment definition and grouping

→ LEDATA, LIDATA } Translated instruction and data

→ FIXUPP — Relocation and loading information

→ MODEND — END of the object module.

Theader:-

It is the recorder that specifies the name of object module and translator header

Typedefi:-

It is a recorder that contains the information of both PUBDEF and EXTDEF.

PUBDEF :-
It is the records that contains a list of External symbols called public names that are defined in the object module

EXTDEF :-
It is a records that contains list of External symbols

LNAMES :-
It is a records that contains a list of all Segment and class names used in the program

SEGDEF :-
It is a records that describes the segment in the object module. Computer operating system. The term bootstrape comes from phrase "pull yourself up by your bootstraps".

An ideal computer with no programs in memory and there is no need for program relocation we can simply specify the absolute address whatever the program is first loaded that occupies a predefined location in the memory.

In some compiler absolute loader program permantly resides in a ~~ROM~~ rom. The program is executed directly in the Rom. In other computer, the program is copied from Rom to main memory and executed. If the loading proces requires more instructions it can read in a single record. The first record causes the reading of others and then read other record.

Implementation Examples:-

In this section we briefly examine linkers and loaders for actual computers. We also point out the areas in which the linker or loader design is related to the assembler design architecture of the computer machine.

The Example for linker and loader are pentium spark, ms-DOS linkers and sun operating system linker

MS-DOS Linker:-

It describes some of the features of microsoft ms-dos linker for pentium and other X-86 machines. most ms-DOS compilers and assembles produce object modules not the executable machine language programs but also describes the structure of the program.

ms-DOS is a linkage editor that combines one or more object modules to produce a complete executable program. This executable program has a file name extention as "exe or". link". It also combines a the translated program with other modules. The following table represted a typical ms-DOS object modules is as follows.

Record	description
→ Theader	Translate headers
→ TYPEDEF, PUBDEF, EXTDEF	External symbols and references
→ LNAMES, SEGDEF, GRPDEF	Segment definition and grouping
→ LEDATA, LIDATA	Translated instruction and data
→ FIXUPP	Relocation and linking information
→ MODEND	End of the object module

THEADR:-
It is the recorder that specifies the name of object module and translator loades

Type def:-
It is a recorder that contains the information of both PUBDEF and EXTDEF

PUBDEF:-
It is the recorder that contains a list of External symbols called public names that are defined in the object module

EXTDEF:-
It is a recorder that contains list of External symbols

LNAMES:-
It is a recorder that contains a list of all Segment and class names used in the program

SEGDEF:-
It is a recorder that describes the segments in the object module

GRPDEF:-
It is a recorder that specifies how the segments are combined into groups

LEDATA:-
It is a recorder that contains translation instruction and data from the source program

LIDATA:-
It is a recorder that specifies translation instruction data that occurs in a representation pattern

FIXUPP:-
It is a recorder that is used to resolve External reference to carry out address modification that are associated with relocation and grouping of segment with in a program

MODEND:-
It is record that specifies End of the object module

Sun OS linker:-

Sun operating system actually provides 2 different linkers such as

1. Link Editor
2. Run time linker (dynamic linking)
3.

1. Link Editor:-

The link editor is most commonly involved in the process of compiling a program. It takes one or more object modules produced by the assembles and compilers and continue them to produce a single output module. The module may be one of the following

1. Are loctable objects module that is for further link editing.
2. A static executable module that contains symbolic references and ready to run
3. A dynamic executable module that contains some symbolic references that may be need at runtime
4. A shared object module that provides the services that can be bound at runtime to one or more dynamic executable modules. An object module contains one or more sections which represents the instructions and data from the source program Each section has a set of attributes such as executables and writable it also includes a list of relocation and linking operations that need to be performed and symtab decides the symbols used in the operations.

for Example, there are 24 relocation loaders that are used in spare assembler. sun operating system linkers is implemented on X-86 system that uses 11 different set of codes.

2. Run time linker:-

The sun operating system runtime linker is used to bind dynamic executables and shared objects at runtime

UNIT-II

Chapter-II

MACRO PROCESSOR

* Macro processor:-

A macro processor represents a commonly used group of statements in the source programming language. It replaces each macro instruction with the corresponding group of source language statement. This is called "expanding the macro". Thus the macro instruction allows the programs to write short hand version of the program and leaves mechanical details to be handled by the macroprocessor. The most common use of macro processor in Assembly language programming.

We use SIC assembler language examples to illustrate most of the concepts. macro processor can also used in high level programming language like C, OS & commanding languages like lex, unix ... etc

Basic macro processor functions:-

It describes the process of macro definition invocation and expansion with substitution of parameters. These function are illustrated with examples using SIC/XE assembler language by using macro definition and expansion.

macro processor algorithm & data structure represents a pass 1 algorithm for a simple micro processor together with a description of data structure needed for macro processing.

Macro Definition & example:-

The following example of SIC/XE program using macro instruction is

shown below. This program has the numbering scheme used for the source statements that has been changed. This program is defene and uses two macro functions such as: * RDBuff
* WRBuff

The function and logic of the RDBuff macros are similar to these of RDREC subroutine

ex: use of macros in SIC/XE program

LINE	SOURCE STATEMENT		description
5	COPY	START 0	copy from I/P to O/P
10	RDBUFF	MACRO INDEX, BUFF ADDR, RECLTH	
15		MACRO	TO read record in to Buffer
20		—	—
25		—	—
30		CLEAR X	clear loop counter
35		CLEAR A	—
40		CLEAR S	—
45		HDA #4096	Set maximum record length
150		TD =X&INDEX	Test input device
55		JEQ *-3	Loop until Ready
60		RD =X&INDEX	Read a character in to register A
65		COMPR A, S	—
70		JEQ *+11	exit loop
75		STCH &BUFFER, X	store a character in a Buffer
80		TIXR T	loop unless maximum length has been Reached

```
85      JLT     *-19            -
90      MEND    &RECLTH         save record length
95      MACRO   &OUTDEV         -
WRBUFF
100     -       &BUFFER,&RECLTH -
105     MACRO   -               To write record into
                                BUFFER
110     CLEAR   X               clear loop counter
115     LDT     &RECLTH         -
120     LDCH    &BUFFER,X       Get a character
                                from BUFFER
125     TD      =X'OUTDEV'      write a character
130     JEQ     *-3             loop until ready
135     WD      =X'OUTDEV'      write character
140     TIXR    T               loop until all
                                characters has been
                                reached
145     JLT     *-14            -
150     MEND    -               End of macro
155     MAIN    PROGRAM         -
160FIRST STL    RETADDR         Save Return address
165CLOOP RDBUFF &1,&BUFFER,&LENGTH READ record into buffer
170     LDA     LENGTH          Test for end of file
175     COMP    #0              -
180     JEQ     ENDFIL          Exit if not found
185     WRBUFF  05,BUFFER,LENGTH write output record
190     J       CLOOP           -
195     WRBUFF  05,EOF,THREE    -
                RETADDR
200     J       @RETADDR        -
205     WORD    3               -
THREE
210     RESW    1               Length of record
215     RESW    1               4096 bytes buffer Area
220     RESB    4096            -
225     MEND    -               end of MACRO
```

Two new assembler directives [MACRO and MEND] are used in macro definition. The first macro statement identifies the beginning of macro defenition. The symbol in the field RDBUFF is the name of the macro and the entries in the operand field identifies the parameters of macro instructio

The macro name and parameters define a pattern or prototype for the macro instructions used by the programmer. The MEND assembler directive specifies the end of the macro definition. The defenition of WRBUFF macro follows a similar pattern.

The main program itself begins at line. The statement at line is a macro invocation statement that gives the name of the macro instruction being invoked and the argument to be used in expanding macros.

* macro processor algorithm and data structure

It is easy to design to pass macro processor in which all macro definitions are produced during to first pass and all macro invocation statements are expanded during second pass. For example the two macro instruction defenition is as follows.

The body of first macro contains RDBUFF WRBUFF and the other macro instructions for SIC system. The body of second macro defines the same macros for SIC/XE systems

```
*MACROS MACRO  defines SIC standard version
                macros
* RDBUFF MACRO &BUFFER,&BUFFERADDR,&RECLENGTH
*       MEND  END OF RDBUFF
* WRBUFF MACRO &OUTDEV,&BUFFERADDR,
                &RECLENGTH
```

* MEND {END of MRBuff}
* MEND {END of MACRO}

There are three main data structures involved in a macro processor. The macro definition are stored in defenition table [DEFTAB] which contains macro prototype and the statements that makes up the macro body. third data structure are used [The macro names are entered in Name argument table [ARGTAB] which is used during the expansion of macro invocation. NAM ARGT]

The macronames are entered in the Name table [NAMTAB] that contains pointer to the begining and ending of DEFTAB

* Algorithm for pass 1 macro processor:-

```
Begin (macro processor)
Expanding := false
While opcode ≠ end do
 begin
 GETLINE
 PROCESS LINE
 end
 {
 while
  {
 end
 }
 macro processor
 }
 produce PROCESSLINE
 Begin
 SEARCH NAMTAB for opcode
  if found then
  EXPAND
  else if opcode="macro the
  DEFINE
  else write source line to expand
 end
 } PROCESS LINE
```

* Machine Independent macro processor features:-

We have discussed several extensions to the basic macro processor functions. There are three different types of machine independent macro processor features they are:- (1) concatenation of macro parameters (2) Generation of unique labels (3) conditional macro expansion

(1) concatenation of macro parameters:-

It describes a method for concatinating macro instruction parameters with other character string. for example, a program containg a serees of variables named by symbols XA1, XA2, XA3---- and another series named by XB1, XB2, XB3---- .If similar processing to be performed on each series of variables the programmer might want to incorporate this processing into a macro instruction. The parameters to such a macro instruction could specify a series of variables to be operated on A, B, c.

Suppose that the parameter to such the macro instruction is named as &id. The body of the macro definition contains &id and &id1 as parameters, this situation would be unavoidably ambiguous.

Most macro processor deals with these problems by providing a specal concatenation operators. In the SIC macro language these operator is the character and it can be written as LDA X&id->1

(2) Generation of unique Labels:-

It describes one method for [concatenating macro instruction parameters with other character strings].

Generating unique labels with macro expression. In general it is not possible to the body of a macro instruction that containg the labels. This leads to the use of relative addressing at the source statement level. This label will be defined twice for each invocation of WRBUFF. These duplication definition would prevent current assembly language program of the resulting expanded program.

The following examples illustrates one technique for generating unique labels with a macro expression. The definition of RDBuff is as follows.

```
Line  SOURCE STATEMENT        Description
5     COPY   START  0          copy from file I/P to O/P
10    RDBUFF macro Index,
             BuffADDR
             RECLENGTH        To record into Buffer
15    MACRO
20    ---    ---
25    ---    ---
30    CLEAR  X                clear loop counter
35    CLEAR  A
40    CLEAR  S
45    HDA    #4096            Set maximum record length
50    TD     =X,&Index        Test for input device
55    JEQ    *-3              loop until Ready
60    RD     =X,&Index        Read character into
65    COMPR  A,S              Register A
70    JEQ    *+11             EXIT loop
75    STCH   &Buffer,X        store a character in Buffer
80    TIXR   T                Loop until maximum
85    JLT    X-19             length is reached
90    STX    &Reclength       Save record length
95    MEND                    End of MACRO
```

conditional macro expression:-

Most macro processor can modify the sequence of statements generated by a macro expansion. It depends on the arguments supplied in the macro invocation such capability adds greately to the power and flexibility of a macro language. The use of one type of can determal macro expansion statement defines the definition of RDBuff has two additional parameters. LEOR which specifies the hexadecimal character code that maxes the end of the record. MAXLTH which specifies the maximum length of the record that can be Read. The set statement assign the value 1 to LEOR the Symbol represents a macrotime variable which can be used to store the working values during the macro expansion. Any Symbol that begins with the character \ (' that is not a macro instruction parameter. It assumes to be a macro time variable all such variables are initialized to a value '0'.

* Implementation examples:-

⇒ ANSI C macro language:-

(1) It describes some of the macro programming features of ANSI C programming language.

(2) In the ANSIC programming language definition and invocation of macros are handled by a preprocessor. This preprocessor is generally not integrated with the rest of compiler. The example of ANSIC macro definition is as follows

#define NULL 0

#define Eof -1

These definitions appear in the program, every occurance of null will be placed by 0 and every occurance of eof will be replaced by -1:

(3) It is also possible to use macros to make the limited changes in the syntax of the language.

For Ex:-

After defining the macro # define EOG 0, A programmer can write while(I!=EOG) The macro processor could convert these into while(I=0)

(4) The ANSI C preprocessor also provides conditional compilation statements. These statements are used to provide macro that is defined atleast one

```
#if def BUFFER-SIZE
#define BUFFER-SIZE 1024
#end if
```

The hash defined will be processed only if Buffer-Size has not already been defined. conditionaly are also promised to control the inclusion of debugging statements in the program.

⇒ ELENO MACRO processor:-

Some of the features of ELENO general purpose macro processor is as follows.

(1) ELENO was developed has a research tool but not as a commercial software products.

(2) However the same design and implementation techniques could be used in developing other general purpose macro processor.

(3) ELENO are composed as a header and a body with a most macro processor.
(4) However the header is not required to have only special forms it consist of a sequence of keywords and parameters that can be identify by special character "%".
(5) The only restriction is that atleast one of the first two tokens in a macro header must be a keyword, not a parameter
(6) A macro invocation is a sequence of tokens that matches the macro header

$$\%.1 = \%.2 + \%.3$$

It could be invoked as $\alpha = \beta + \delta$.

(7) The following example show how ELENO could be used with different language

$$\%.1 = ABS\ diff[\%.2, \%.3]$$

(8) If the macro is to be used with c language its body might be defined as

$$\%.1 = (\%.2) > (\%.3)\ ?\ (\%.2) - (\%.3) : (\%.3) - (\%.2)$$

Macro processor design option:-

In this section we discuss some major design options for a macro processor. There are three design option that are included in macro-processor. They are

(1) Recursive macro Expansion
(2) General purpose macro parameters
(3) Macro processor within language translators

⇒ Recursive macro expansion:-

It examines the problems created by macro invocation statements and suggest some possibilities for the solution of these problems. The definition of RDBUff is same. It represents the order of

parameters has been changed. However we have assured that a related macro RDCHAR already exist. The purpose of RDCHAR is to read one character from a specified device into the register. It is convinent to use a macro like RDCHAR in the definition of RDBUFF so that the programmer who is designing RDBUFF need not worry about the details of the device access and control. The advantage of using RDCHAR would be greater on a more complex machine where the read a single character might be longer and more complicated than our simple line version, Recursive macro expansion is used when a macro function should be called again & again to expand the instructions and to perform the operations

```
Line    SOURCE statement
 5      RDCHAR  MACRO &IN
10      MACRO   TO Read character in to Register A
15              TD      =X'&IN' Test input device
20              JEQ     *-3 loop until ready
25              RD      =X'&IN' Read a character
30      MEND    END of MACRO
```

⇒ General purpose macro parameters:-

The use of macro processor is an AID to an assembler programming language such MACRO processors are combined with assembler macro processor which has been developed for high level programming language. This special purpose macro processor are similar in general functions approach. These are not dependent on any particular language that can be used with a variety of different languages.

There are several situations in which normal macro parameter should not occur.

For example, commands are usually ignored by a macro processor however each programming language has its own method for identifying commands. Some language use special character to mark start of command automatically terminated at the end of sourceline.

The general problem involves tokens of the programming languages. For example identifiers, constants, operators, variables and keywords languages differ on their restrictions, length of the identifiers and rules for the formation of constants.

⇒ Macro processor within language translators:-

This combines macro processor instruction with the language translator. The simplest method of achieving this sort of combination is a line by line macro processor. Using this approach, the macro processor reads the source program statements and all of its functions line by line approach.

Advantages:-

⇒ It avoids taking an extra pass over the source program. For example, OPTAB in assembler and NAMTAB with macro processor, to be implemented in the same table.

⇒ This includes such operation as the standard input lines, searching tables and converting numeric values from external to internal representation.

DisAdvantages:-

⇒ These are specifically designed and written to work with a particular implementation of an assembler or compiler.

⇒ The cost of macro processor development is added, to the cost of language translator which results in expensive piece of software.

⇒ In addition, the assembler and compiler will be considerably larger and more complex if macro processor were used.