NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE
(NAAC "A" Accredited)
(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

Since 1968

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(2019-SCHEME)
COURSE MATERIALS

CST 305 SYSTEM SOFTWARE

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

- To Impart Quality Education by creative Teaching Learning Process.
- To promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
- To Inculcate Entrepreneurship Skills among Students.
- To cultivate Moral and Ethical Values in their Profession.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

PEO2: Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

PEO3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

PEO4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamworkand leadership qualities.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities

with an understanding of the limitations.

6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOMES

SUBJECT CODE: C303		
COURSE OUTCOMES		
C303.1	K4	**Identify** and classify different software into different categories.
C303.2	K6	**Design**, analyze and implement two pass assembler
C303.3	K6	**Design**, analyze and implement one pass and multi pass assembler.
C303.4	K6	**Design**, analyze and implement linkers and loaders
C303.5	K6	**Design**, analyze and implement macro processors And to critique the features of modern editing /debugging tools.

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

CO Vs PO'S Mapping

CO'S	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C303.1	3	-	-	-	-	-	-	-	-	-	-	2
C303.2	3	3	3	2	-	-	-	-	-	-	-	2
C303.3	3	3	3	2	-	-	-	-	-	-	-	2
C303.4	3	3	3	-	-	-	-	-	-	-	-	2
C303.5	3	3	3	-	-	-	-	-	-	-	-	2
C303	**3**	**3**	**3**	**2**	**-**	**-**	**-**	**-**	**-**	**-**	**-**	**2**

CO PSO'S Mapping

CO'S	PSO1	PSO2	PSO3
C303.1	3	-	-
C303.2	3	2	-
C303.3	3	2	-
C303.4	3	2	-
C303.5	3	2	-
C303	3	2	-

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

SYLLABUS

CST 305	SYSTEM SOFTWARE	Category	L	T	P	Credit	Year of Introduction
		PCC	3	1	0	4	2019

Preamble:

The purpose of this course is to create awareness about the low-level codes which are very close to the hardware and about the environment where programs can be developed and executed. This course helps the learner to understand the machine dependent and machine independent system software features and to design/implement system software like assembler, loader, linker, macroprocessor and device drivers. Study of system software develops ability to design interfaces between software applications and computer hardware.

Prerequisite: A sound knowledge in Data Structures, and Computer Organization

Course Outcomes: After the completion of the course the student will be able to

CO#	Course Outcomes
CO1	Distinguish **softwares** into system and application software categories. **(Cognitive Knowledge Level: Understand)**
CO2	Identify **standard** and extended architectural features of machines. **(Cognitive Knowledge Level: Apply)**
CO3	Identify machine dependent features of system software **(Cognitive Knowledge Level: Apply)**
CO4	Identify machine independent features of system software. **(Cognitive Knowledge Level: Understand)**
CO5	Design **algorithms** for system softwares and analyze the effect of data structures. **(Cognitive Knowledge Level: Apply)**
CO6	Understand the features of device drivers and editing & debugging tools.**(Cognitive Knowledge Level: Understand)**

Syllabus

Module-1 (Introduction)

System Software vs Application Software, Different System Software– Assembler, Linker, Loader, Macro Processor, Text Editor, Debugger, Device Driver, Compiler, Interpreter, Operating System (Basic Concepts only). SIC & SIC/XE Architecture, Addressing modes, SIC & SIC/XE Instruction set , Assembler Directives.

Module-2 (Assembly language programming and Assemblers)

SIC/XE Programming, Basic Functions of Assembler, Assembler Output Format – Header, Text and End Records. Assembler Data Structures, Two Pass Assembler Algorithm, Hand Assembly of SIC/XE Programs.

Module-3 (Assembler Features and Design Options)

Machine Dependent Assembler Features-Instruction Format and Addressing Modes, Program Relocation. Machine Independent Assembler Features –Literals, Symbol Defining Statements, Expressions, Program Blocks, Control Sections and Program Linking. Assembler Design Options- One Pass Assembler, Multi Pass Assembler. Implementation Example-MASM Assembler.

Module-4 (Loader and Linker)

Basic Loader Functions - Design of Absolute Loader, Simple Bootstrap Loader. Machine Dependent Loader Features- Relocation, Program Linking, Algorithm and Data Structures of Two Pass Linking Loader. Machine Independent Loader Features -Automatic Library Search, Loader Options. Loader Design Options.

Module-5 (Macro Preprocessor ,Device driver, Text Editor and Debuggers)

Macro Preprocessor - Macro Instruction Definition and Expansion, One pass Macro processor Algorithm and data structures, Machine Independent Macro Processor Features, Macro processor design options. Device drivers - Anatomy of a device driver, Character and block device drivers, General design of device drivers. Text Editors- Overview of Editing, User Interface, Editor

Structure. Debuggers - Debugging Functions and Capabilities, Relationship with other parts of the system, Debugging Methods- By Induction, Deduction and Backtracking.

Text book

1. Leland L. Beck, System Software: An Introduction to Systems Programming, 3/E, Pearson Education Asia

References

1. D.M. Dhamdhere, Systems Programming and Operating Systems, Second Revised Edition, Tata McGraw Hill.
2. John J. Donovan, Systems Programming, Tata McGraw Hill Edition 1991.
3. George Pajari, Writing UNIX Device Drivers, Addison Wesley Publications (Ebook : http://tocs.ulb.tu-darmstadt.de/197262074.pdf).
4. Peter Abel, IBM PC Assembly Language and Programming, Third Edition, Prentice Hall of India.
5. Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, Linux Device Drivers, Third Edition, O.Reilly Books
6. M. Beck, H. Bohme, M. Dziadzka, et al., Linux Kernel Internals, Second Edition, Addison Wesley Publications,
7. J Nithyashri, System Software, Second Edition, Tata McGraw Hill.
8. The C Preprocessor http://gcc.gnu.org/onlinedocs/gcc-2.95.3/cpp_1.html -

QUESTION BANK

MODULE I			
	QUESTIONS	**CO**	**KL**
1	Define the Functions of an Assembler	CO1	K1
2	List any Four Addressing modes of SIC/XE	CO1	K1
3	Summarize the instruction formats used in SIC	CO1	K2
4	Write the sequence of instructions for SIC/XE to divide BETA by GAMA and to store integer quotient in ALPHA reminder in DELTA	CO1	K5
5	Illustrate the SIC/XE architecture, Explaining in detail data and instruction formats.	CO1	K3
6	Describe the format of Object Program generated by the Two Pass SIC Assembler Algorithm	CO1	K2
7	Summarize debugger, text editor and device driver.	CO1	K2
8	Illustrate the SIC architecture in detail.	CO1	K3
9	Differentiate System software and application software.	CO1	K4
10	Summarize the instruction formats used in SIC/XE	CO1	K2
11	Discuss the SIC/XE memory, registers, data and instruction formats and addressing modes	CO1	K2
12	Let NUMBERS be an array of 100 words. Write a sequence of instructions for SIC and SIC/XE to set all 100 elements of the array to 1.	CO1	K5
MODULE II			
1.	Define the Functions of an Assembler	CO2	K1
2.	Describe Program Relocation	CO2	K2
3.	List Assembler directives in SIC	CO2	K1
4.	Give the Algorithm for Pass1 of two Pass SIC Assembler	CO2	K2
5.	Describe the format of Object Program generated by the Two Pass SIC Assembler Algorithm	CO2	K2
6.	Give the use of SYMTAB and OPTAB	CO2	K2

7	Explain the Algorithm for Pass2 of SIC Assembler	CO2	K5
MODULE III			
1	Define Literals.	CO3	K1
2	With example, write notes on program blocks.	CO3	K2
3	Summarize Symbol defining statements in assemblers.	CO3	K2
4	Give the purpose of EXTREF and EXTDEF assembler directives	CO3	K2
5	Write short notes on MASM Assembler	CO3	K2
6	Give the structure and purpose of Modification record and Define record	CO3	K2
7	Explain the concept of single pass assembler with suitable example	CO3	K5
8	Illustrate control sections and program blocks	CO3	K3
9	Explain in detail about Control section and its different records .	CO3	K5
10	Explain in detail assembler independent features-literals, symbol defining statements and expressions.	CO3	K2
11	Differentiate control sections and program blocks in detail and also point out the assembler directives	CO3	K4
12	Explain the external reference handling of an assembler	CO3	K5
13	Define forward reference. Illustrate the forward reference handling by a single pass assembler.	CO3	K1&K3
MODULE IV			
1	Point out Relocation , Linking and Loading.	CO4	K4
2	Write notes on different loader design options	CO4	K3
3	State and explain two pass algorithm for a linking loader.	CO4	K5
4	Write short note on dynamic linking	CO4	K3
5	Explain detail about machine dependent features of loader.	CO4	K2
6	State and explain pass one algorithm for a linking loader	CO4	K5
7	Write notes in detail about program linking.	CO4	K3
8	Explain with example dynamic linking and automatic library search.	CO4	K2

9	List and explain different loader options	CO4	K1 & K2
MODULE V			
1	Illustrate about recursive macro expansion.	CO5	K3
2	Design an iterative algorithm for a one pass macro processor	CO5	K5
3	Differentiate between a macro and a subroutine. Illustrate macro definition and expansion using an example.	CO5	K4
4	Illustrate about recursive macro expansion.	CO5	K3
5	Write note on conditional macro expansion.	CO5	K3
6	Illustrate the data structure required for a macro processor algorithm and explain the format of each.	CO5	K3
7	Illustrate about macro definion and expansion	CO5	K3
8	Explain keyword macro parameters and how unique label generated in a macro expansion.	CO5	K5
9	Explain the macro processor algorithm	CO5	K5
10	Differentiate between character and block device drivers.	CO5	K4
11	Explain the structure of text editor with the help of a diagram.	CO5	K5
12	Discuss about device drivers with neat sketch.	CO5	K2
13	Explain about debugging and different debugging techniques.	CO5	K5
14	Differentiate Text editor and debugger	CO5	K4
15	Explain the design of driver with diagrammatic representation.	CO5	K5
16	Describe the function and capabilities of interactive debugging system.	CO5	K5
17	Explain different debugging methods in detail. What is a debugger?	CO5	K5

APPENDIX 1	
CONTENT BEYOND THE SYLLABUS	
SL NO	**TOPIC**
1	**commands used in VI text editors.**
2	**Detailed study of structure and record formats of DLL.**

MODULE NOTES

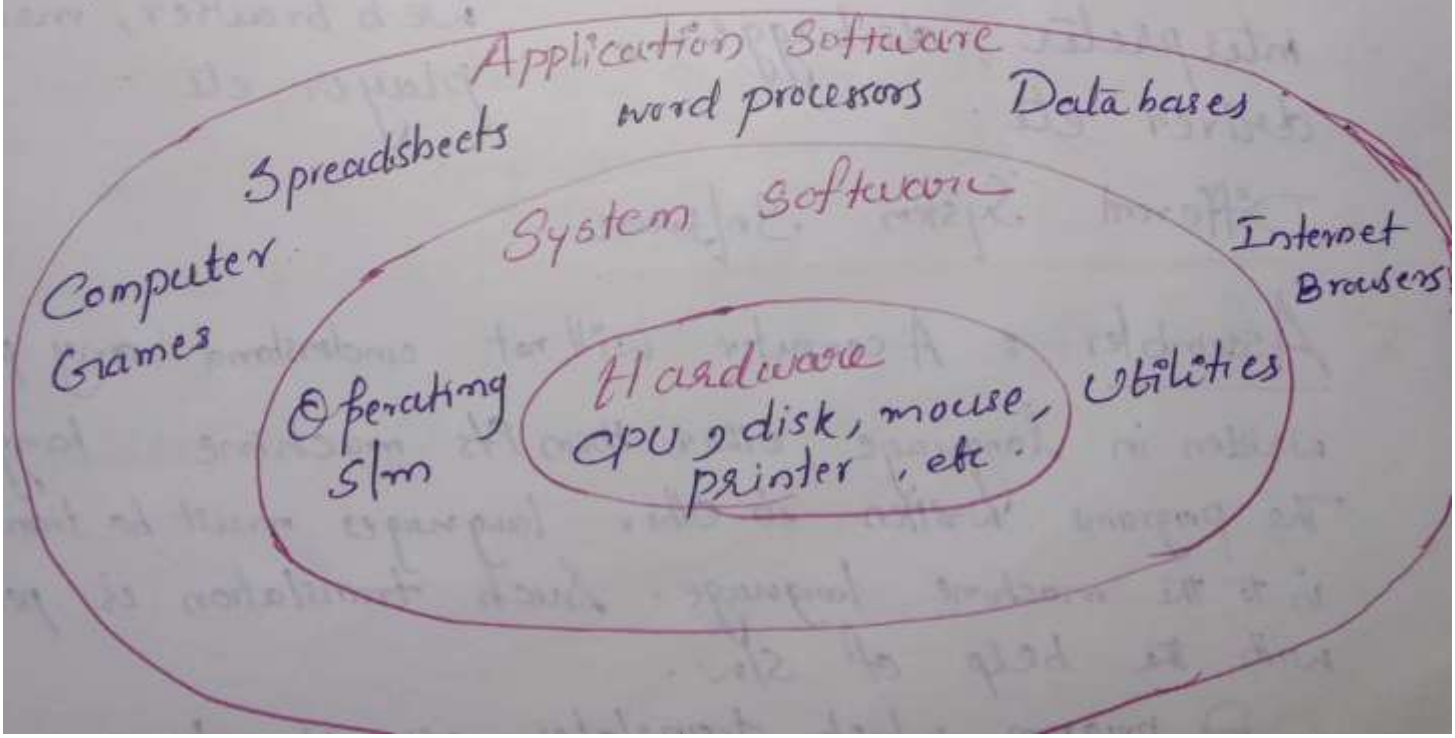
MODULE - I.

System S/W vs. Application Software - Different System software - Assembler, Linker, Loader, Macro processor, Text Editor, Debugger, Device Driver, Compiler, Interpreter, Operating System (Basic Concepts only).

SIC & SIC/XE Architecture, Addressing modes, SIC & SIC/XE Instruction set, Assembler Directives and programming.

⇒ S/m Software vs. Application Software

- System Software is general purpose software which is used to operate computer hardware. It provides platform to run application softwares.
- Application Software is specific purpose s/w which is used by user for performing specific task.



Difference between System software & Application S/w.

System Software	Application Software
1. S/m s/w is used for operating computer h/w.	Application s/w is used by user to perform specific task.
2. S/m S/w s are installed on the computer when OS is installed.	Application S/w s are installed according to user's requirements.
3. In general, the user does not interact with system software because it works in the background.	In general the user interacts with application S/ws.
4. S/m s/w can run independently. It provides platform for running application S/ws.	Application S/w can't run independently. They can't run without the presence of s/m s/w.
5. Some exs of s/m S/ws are compiler, assembler, interpreter, debugger, driver etc.	Some exs. of application S/ws are word processor, web browser, media player etc.

⇒ Different System Software

* Assembler : A computer will not understand any program written in language, other than its machine language. The programs written in other languages must be translated in to the machine language. Such translation is performed with the help of s/w.

- A program which translates assembly language pgm in to a machine language program is called an assemb

If an assembler which runs on a computer and produces the machine codes for the same computer then it is called self assembler or resident assembler.

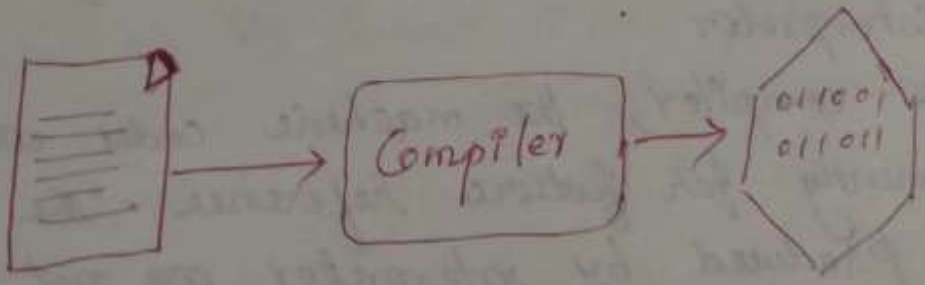
If an assembler that runs on a computer and produces the machine codes for other computer then it is called Cross Assembler.

Assemblers are further divided in to two types: One pass Assembler and Two pass Assembler.

- One pass assembler is the assembler which assigns the memory addresses to the variables and translates the source code in to machine code in the first pass simultaneously.
- A Two pass assembler is the assembler which reads the source code twice. In the first pass, it reads all the variables and assigns them memory addresses, in the second pass, it reads the source code and translates the code in to object code.

Compiler :- It is a program which translates a high level language program in to a machine language program.

- Typically, from high level source code to low level machine code or object code.



- A compiler is more intelligent than an assembler.
- It checks all kinds of errors. But its program run time is more and occupies a longer part of the memory. It has low speed. Because a compiler goes through a the entire program and then translates the entire program in to machine code
- If compiler runs on a computer and produces the machine codes for the same computer then it is known as a self compiler or resident compiler.
- If a compiler runs on a computer and produces the machine codes for the other computer then it is known as a cross compiler.

* Interpreter :- An interpreter is a program which translates statements of a program in to a machine code. It translates only one statement of program, translates it and executes it. Then it reads the next statement of the ~~of the~~ program again translates it and executes it. In this way it proceeds further till all the statements are translated and executed.
- On the otherhand a compiler goes through the entire program and then translates the entire pgrm in to machine codes.
- A compiler is 5 to 25 times faster than an interpreter.
- By the compiler, the machine codes are saved permanently for future reference. The machine codes produced by interpreter are not saved.

- An Interpreter is a small program as compared to compiler. It occupies less memory space, so it can be used in a smaller system which has limited memory space.

Linker :- In high level languages, some built in header files or libraries are stored. These libraries are predefined and these contain basic functions which are essential for executing the program. These functions are linked to the libraries by a program called Linker. If linker does not find a library of a function then it informs to compiler and then compiler generates an error. The compiler automatically invokes the linker as the last step in compiling a program.
- It also links the user defined functions to the user defined libraries.
- Usually a longer program is divided in to smaller subprograms called modules. And these modules must be combined to execute the program. The process of combining the modules is done by the linker.

Loader :- Loader is a program that loads machine codes of a program in to the system memory. In computing, a loader is the part of an OS that is responsible for loading programs. It is one of the essential stages in the process of starting a program.

- It places the programs in to memory and prepares them for execution. Loading a pgm involves reading the contents of executable file in to memory. Once loading is complete the OS starts the pgm by passing control to the loaded pgm code.
- In many OS s the loader is permanently resident in memory.

* Macro processor

A macro processor is a program that copies a stream of text from one place to another, making a systematic set of replacements as it does so.
Macroprocessors are often embedded in other programs such as assemblers and compilers.
- Some times they are stand-alone programs that can be used to process any kind of text.
- A macro processor is a program that reads a file (or files) and scans them for certain keywords. When a keyword is found, it is replaced by some text. The keyword / text combination is called macro.
- A general purpose macro processor or general purpose preprocessor is a macro processor that is not tied to or integrated with a particular language or piece of sfw.
- A simple ex is the C language preprocessor.

Eg:
```
#define MAX 6
int a1;
for (a1=0; a1<MAX; a1++)
{
....;
}
```

In a C pgm, the C preprocessor reads the first line and stores it as a macro definition when it comes across the later reference to MAX in the for loop, it replaces it with the macro definition 6. The output of the C preprocessor is then fed to the C compiler proper.

* Text Editors :-

- Allows to edit a text file
- Common editing features
  - Deleting, replacing, pasting, saving, Searching
- Windows OS - Notepad, Word pad, Microsoft word
- Unix OS - vi, emacs, jed, pico

→ Acts as a primary interface to the computer for all type of knowledge workers" as they compose, organize, study and manipulate computer based information

→ An interactive editor is a computer program that allows a user to create and revise a target document. Documents includes objects such as computer diagrams, text, equations, tables, diagrams, line art & photographs.

In text editors, character strings are the primary elements of the target text

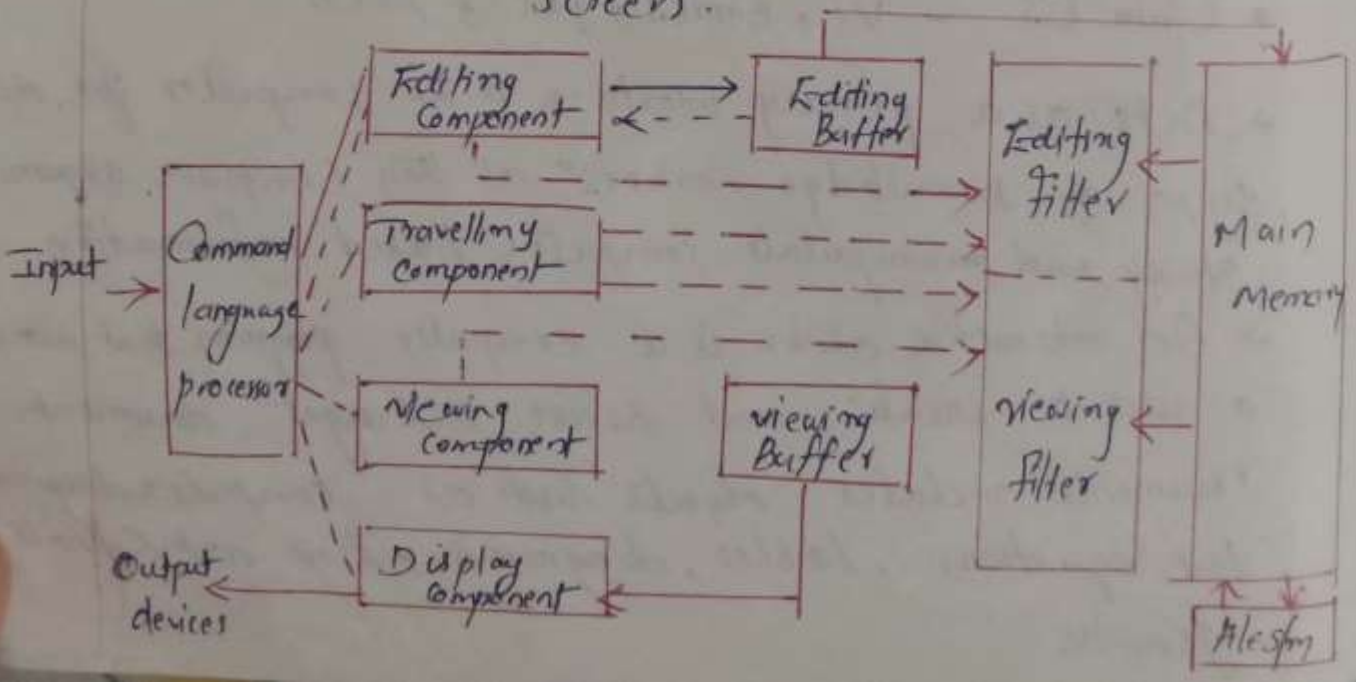
Editing phase involves - insert, delete, replace, move, copy, cut, paste etc.

Document editing process is an interactive user-computer dialogue has four tasks.

1. Select the part of the target document to be viewed and manipulated.
2. Determine how to format this view on-line & how to display it.
3. Specify and execute operations that modify the target document.
4. Update the view appropriately.

The task involves traveling, filtering & formatting.

- Traveling - locate the area of interest.
- Filtering - extracting the relevent subset.
- Formatting - visible representation on a display screen.



Types of Text editors

Depending on the how editing is performed, and the type of output that can be generated, editors can be broadly classified as -

1. Line editors: During original creation lines of text are recognised and delimited by end-of-Line markers, and during subsequent revision, the line must be explicitly specified by line number or by some pattern context. eg. edlin editor in early MS DOS s/ms.

2. Stream Editors: Similar to line editor, but the entire text is treated as a single stream of characters. Hence the location for revision cannot be specified using line numbers. It, either specified by explicit positioning or by using pattern context. eg. sed in Unix / Linux.

- Line editors and stream editors are suitable for text-only documents.

3. Screen Editors: These allow the document to be viewed and operated upon as a two dimensional plane, of which a portion may be displayed at a time. Any portion may be specified for display and location for revision can be specified anywhere with in the displayed portion. eg. Vi, emacs, edt

4. Word processors: provides additional features

to basic screen editors. Usually support non-textual contents and choice of fonts, style etc

5. Structure Editors – These are editors for specific types of documents, so that the editor recognises the structure, syntax of the document being prepared and helps in maintaining that structure/syntax.

Debugger: Debugging means locating (and then removing) bugs ie faults in programs.

The most common steps taken in debugging are to examine the flow of control during execution of the program, examine values of variables at different points in the program, examine the values of parameters passed to functions and values returned by the functions, examine the function call sequence etc

- Usually inserts print statements in the program at various chosen points, that prints values of significant variables or parameters, or some msg that indicates the flow of control.
- Using print statements for debugging a program is often not adequate or convenient. for ex. the programmer may want to change the values of certain variables (or parameters) after observing the execution of the program till some point. For a large program it may be difficult to go back to the source program, make the necessary changes

and rerun the program
- If print statements are placed inside loops, it will produce output every time the loop is executed
To overcome several such drawbacks of debugging by inserting extra statements in the program, there are a kind of tool called debugger that helps in debugging programs by giving the programmer some control over the execution of the program and some means of examining and modifying different program variables during run time

Device Drivers

Device drivers are software modules that can be plugged into an OS to handle a particular device. Operating System takes help from device drivers to handle all I/O devices.

OR-

Device driver is a program that controls a particular type of device that is attached to your computer. There are device drivers for printers, displays, CD-ROM readers, diskette drives and so on. When you buy an OS, many drivers are built in to the product. However, if you later buy a new type of device that the OS didn't anticipate, you'll have to install the new device driver.
A device driver essentially converts the more general I/O instructions of the OS to msgs

that the device type can understand.

- Device drivers encapsulate device dependent code and implement a standard interface in such a way that code contains device specific register reads/writes.
- Device driver is generally written by the devices manufacturer and delivered along with the device on a CD-ROM.

A device driver performs the following jobs

- To accept the request from the device independent SW.
- Interact with the device controller to take and give I/O and perform required error handling
- Making sure that the request is executed successfully.

Operating System

An operating system program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.

Following are the some of important functions of an OS

- Memory management
- processor management
- Device management

- File management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination b/w other s/w & users

The Simplified Instructional Computer (SIC)

SIC has been designed to illustrate the most commonly encountered hardware features and concepts, while avoiding most of the idiosyncrasies that are found in real machines.

- SIC comes in two versions: the standard model and an XE version (XE stands for "extra equipment or extra expensive"). The two versions have been designed to be upward compatible - i.e. an object for the standard SIC will also execute in properly on a SIC/XE system.

SIC Machine Architecture:

Memory:

Memory consists of 8 bit bytes, any three consecutive bytes form a word (24 bits)

- All addresses on SIC are byte addresses; words are

addressed by the location of their lowest numbered byte. There are total of 32,768 (2^15) bytes in the computer memory.

Registers:

There are 5 registers, all of which have special uses. Each register is 24 bits in length.

Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; the Jump to Subroutine (JSUB) instn stores the return add. in this reg.
PC	8	Program Counter; contains the add of the next instn to be fetched for execution.
SW	9	Status word; contains a variety of info; including a condition code (CC)

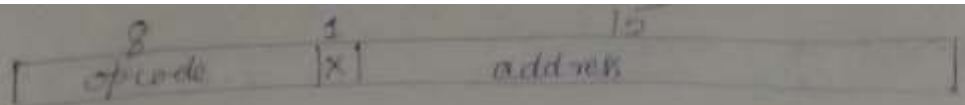
Data formats:-

Integers are stored as 24 bit binary numbers. 2's complement representation is used for negative values.

Characters are stored using their 8 bit ASCII codes. Floating point hardware is not available in the Standard version of SIC.

Instruction formats

All machine instructions on the Standard version of SIC have the following 24 bit format



The flag bit x is used to indicate indexed-addressing mode

Addressing Modes

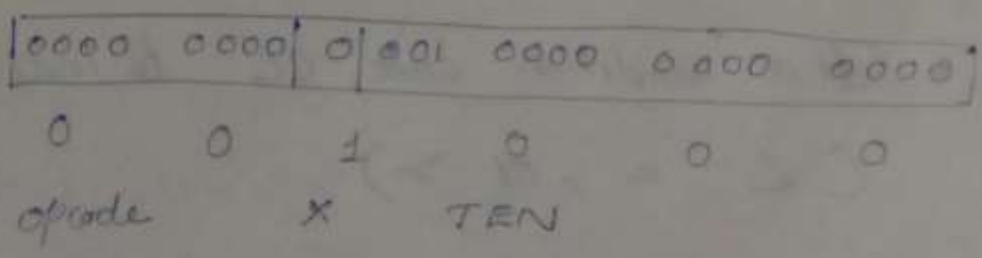
There are two addressing modes available, indicated by the setting of the x bit in the instruction

Mode	Indication	Target address calculation
Direct	$x = 0$	$TA =$ address
Indexed	$x = 1$	$TA =$ address + (X)

Parenthesis are used to indicate the contents of a register or a memory location. for ex, (X) represents the contents of register X

Direct Addressing mode

EX: LDA TEN

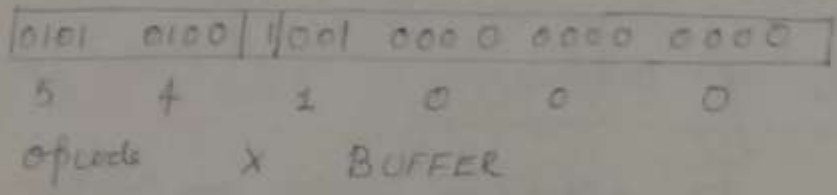


Effective address (EA) = 1000

Content of the address 1000 is loaded to Accumulator

Indexed addressing mode

Ex: STCH BUFFER, X



Effective address (EA) = 1000 + [X]

= 1000 + Content of the index reg X

Instruction set:-

SIC provides a basic set of instructions that are sufficient for most simple tasks. These include

* Load / store registers : LDA, LDX, STA, STX.
* Integer arithmetic : ADD, SUB, MUL, DIV.
  All involve register A and a word in memory, result stored in register A
* COMP : Compares value in register A with a word in memory
* Sets a condition code CC to indicate the result (<, =, or >).
* Conditional Jump Instructions:
  - JLT, JEQ, JGT : can test the setting of CC and jump accordingly

- Two instructions are provided for subroutine linkage

JSUB - jump to the subroutine, placing the return address in register L.

RSUB - returns by jumping to the address contained in register L.

Input and Output:

- Input and output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of registers
- There are three I/O instructions, each of which specifies the device code as an operand
- The test device (TD) instruction tests whether the addressed device is ready to send or receive a byte of data. The condition code is set to indicate the result of this test.
  - A setting of < means the device is ready to send or receive & = means device is not ready.

→ A program needing to transfer data must wait until the device is ready, then execute the

Read Data (RD) or
Write Data (WD).

→ This sequence must be repeated for each byte of data to be read or written.

SIC / XE Machine Architecture : Memory

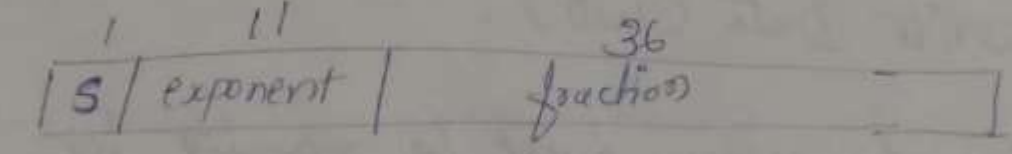
- The maximum memory available on a SIC/XE system is 1 megabyte (2^{20} bytes). This increase leads to a change in instruction formats and addressing modes

Registers :

The following additional registers are provided by SIC/XE

Mnemonic	Number	Special use
B	3	Base register used for addressing
S	4	General working register – no special use
T	5	General working register – no special use
F	6	Floating point accumulator (48 bits)

Data formats: SIC/XE provides the same data formats as the standard version. In addition there is 48 bit floating point data type with the following format



The fraction is interpreted as a value between 0 & 1 for normalized floating point numbers, the high

Older bit of the fraction must be 1. The exponent is interpreted as an unsigned binary number b/w 0 and 2047. If the exponent has value e and fraction has value f, the absolute value of the number represented as,

$$f * 2^{(e-1024)}$$

- The sign of the floating point number is indicated by the value of s (0 = positive, 1 = negative).

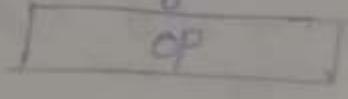
A value of zero is represented by setting all bits (including sign, exponent, fraction) to 0.

Instruction formats :-

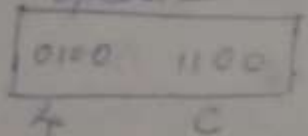
The larger memory available on SIC/XE means that an address field will no longer fit into a 15 bit field, thus the instruction format used on the Standard version of SIC is no longer suitable.

- SIC/XE includes the following instruction formats:

Format 1 (1 byte):

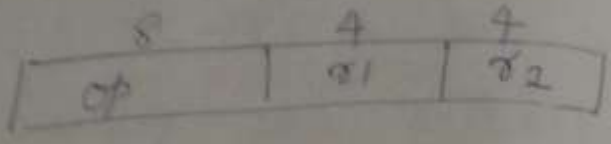


Ex: RSUB (Return to subroutine)

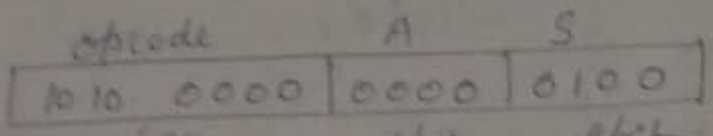


* format 1 & 2 do not reference memory at all
bit e distinguishes between format 3 & 4

Format 2 (2 bytes):



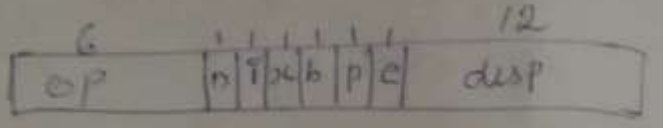
Ex: COMP A,S (compare the contents of regs)



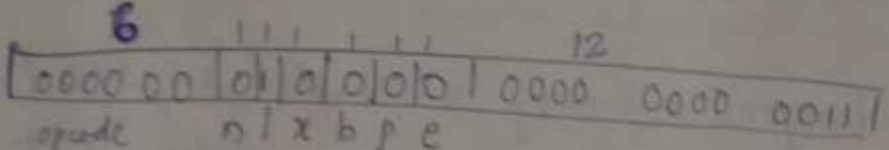
2 bytes

A 0 0 4 — object code

Format 3 (3 bytes):

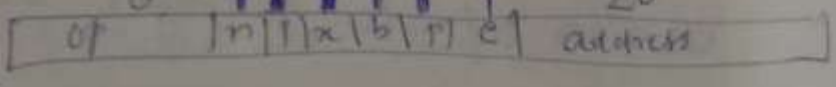


Ex: LDA #3 (Load 3 to Accumulator A)

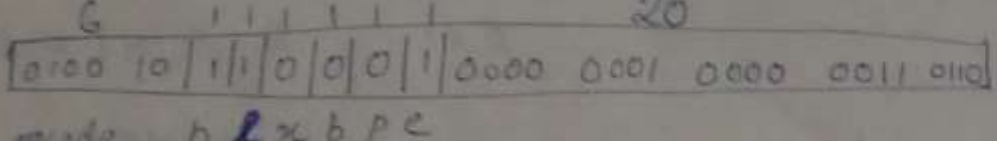


0 1 0 0 0 3 — object code

Format 4 (4 bytes):-



Ex: +JSUB RDREC (Jump to address, 1036).



4 B 1 0 1 0 3 6 Object code

Both format 3 & 4 have six flag values in them, consisting of the following flag bits

- n : Indirect addressing flag
- i : Immediate addressing flag
- x : Indexed addressing flag
- b : Base address relative flag
- p : Program counter relative flag
- e : Format 4 instruction flag

⇒ The SIC/XE has three instruction formats and the Extra Equipment add-on includes a fourth.

- The instruction formats provide a model for memory and data management.
- Each format has a different representation in memory

Format 1 : Consists of 8 bits of allocated memory to store instruction

Format 2 : Consist of 16 bit of allocated memory to store 8 bits of instructions & two 4 bit segments to store operands.

Format 3 : Consist of 6 bits to store an instruction 6 bits of flag values & 12 bits of displacement

Format 4 : Only valid on SIC/XE machines, consist of the same elements as format 3, but instead

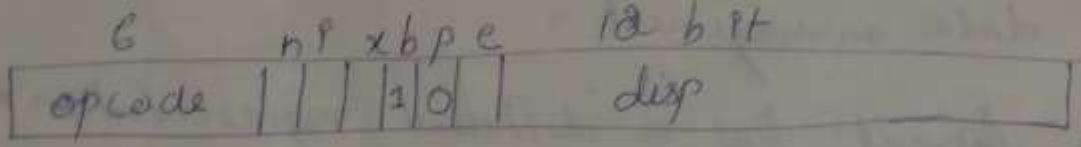
of a 12 bit displacement, stores a 20 bit addres

Addressing Modes:

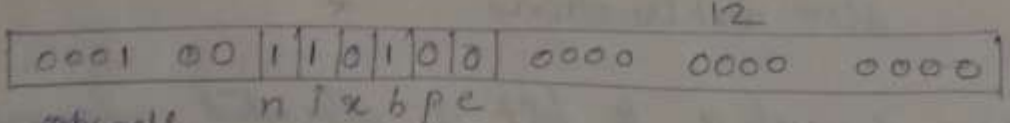
Two new relative addressing modes are available for use with instructions assembled using format 3.

mode	indication	Target address calculation
Base relative	b=1, p=0	TA = (B) + disp (0 ≤ disp ≤ 4095)
Program counter relative	b=0, p=1	TA = (PC) + disp (-2048 ≤ disp ≤ 2047)

Base relative Addressing mode



EX: 1056 STX LENGTH



1 3 4 0 0 0 objectcode

EA = LENGTH = 0033

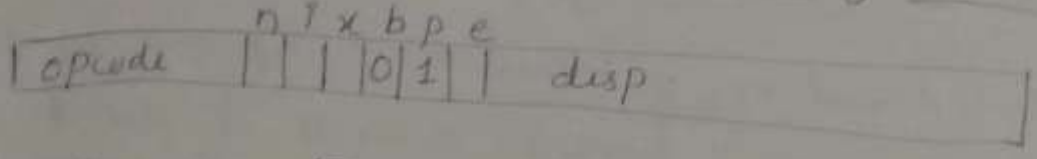
[B] = 0033

disp = 0

EA = disp + [B]

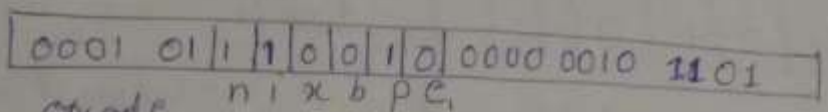
The content of the address 0033 is loaded to the Index register X.

Program counter Relative Addressing Mode



- for pgm counter relative addressing mode, displacement field is interpreted as 12-bit signed integer, with negative values represented in 2's complement notation.

Ex 0000 STL RETADR



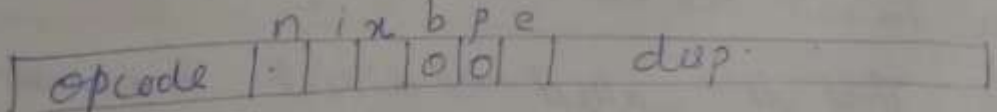
1 7 2 0 2 D object code

EA = RETADR = 0030 EA = (PC) + disp

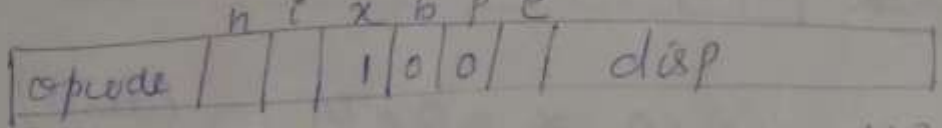
PC = 0003 (address of the next instruction)

disp = 002D

Direct Addressing mode



b = 0 p = 0 TA = disp ($0 \le disp \le 4095$)

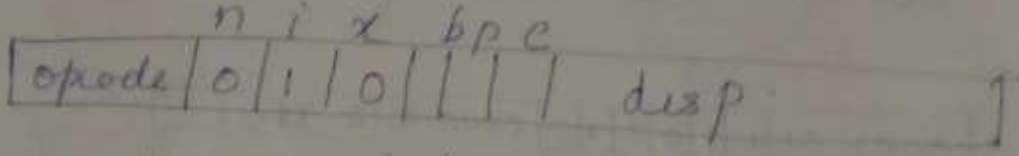


b = 0, p = 0, TA = (x) + disp

(combined with index addressing mode)

Immediate Addressing mode

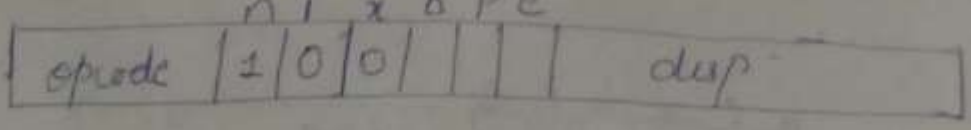
Bits i and n in format 3 & 4 are used to specify how the target address is used. If bit i = 1 and n = 0 target address itself is used as the operand value, no memory reference is performed. This is called immediate addressing



n = 0, i = 1, x = 0, operand = disp

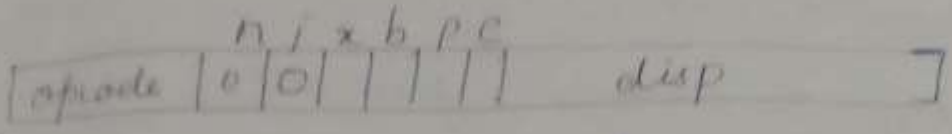
Indirect Addressing mode

If bit i = 0 & n = 1, the word at the location given by the target address is fetched, the value contained in this word is then taken as the address of the operand value. This is called indirect addressing



n = 1, i = 0, x = 0, TA = (disp)

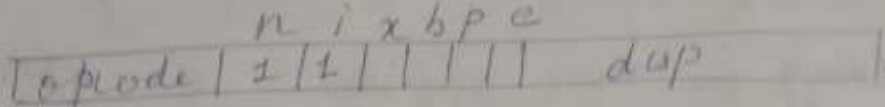
Simple Addressing mode



$i = 0, n = 0$, TA = bpe + disp (SIC std)

opcode = opcode + n+i = SIC std opcode bit are

- If bits i & n are both 0 or both 1, the target address is taken as the location of the operand. We will refer to this as simple addressing. Indexing cannot be used with immediate or indirect addressing modes.



$i = 1, n = 1$ TA = disp (SIC/XE std).

Instruction Set

SIC/XE provides all of the instructions that are available on the standard version.
In addition, there are instructions to load & store the new registers (LDB, STB, etc).
- To perform floating point arithmetic operations
  - ADDF, SUBF, MULF, DIVF.
- Register move - RMO
- Register - to Register arithmetic operations

ADDR, SUBR, MULR, DIVR

- Supervisor call : SVC.

Executing this instruction generates an interrupt that can be used for communication with the OS

Input and Output

The I/O instructions for SIC are also available on SIC/XE.

- There are I/O channels that can be used to perform input and output while the CPU is executing the other instructions.
- This allows overlapping of computing & I/O, resulting in more efficient S/m operation.

The instructions - SIO, TIO & HIO are used to Start, Test and Halt the operation of I/O Channels.

SIC Programming Examples

Fig 1.2 contains exs of data movement operations for SIC and SIC/XE. There are no memory to memory instns, Thus all data movement must be done using registers.

In the fig 1.2a, a 3byte word is moved by

loading it into register A and then storing the register at the desired destination.

In the next line, a single byte of data is moved using the instn LDCH (Load character) and STCH (Store character). These instructions operate by loading & storing the rightmost 8 bit byte of register A; the other bits in reg. A are not affected

Fig - 1-2(a) shows 4 different ways of defining storage for data items in the SIC assembler language

- The statement WORD reserves one word of storage which is initialized to a value defined in the operand field of the statement. Thus the WORD statement in fig 1.2a defines a data word labeled FIVE whose value is initialized to 5.
- The statement RESW reserves one or more words of storage for use by the pgm. for eg: the RESW statement in fig 1.2a defines one word of storage labelled ALPHA, which will be used to hold a value generated by the program.

The statements BYTE and RESB perform simple storage definition functions for data items that are characters instead of words

SIC Programming Example (Fig 1.2a)

- Data movement

```
        LDA     FIVE    load 5 into A
        STA     ALPHA   store in ALPHA
        LDCH    CHARZ   load 'Z' into A
        STCH    C1      store in C1
        .
        .
        .
ALPHA   RESW    1               reserve one word space
FIVE    WORD    5               one word holding 5
CHARZ   BYTE    C'Z'            one-byte constant
C1      RESB    1               one-byte variable
```

In fig-1.2a CHARZ is a 1 byte data item whose value is initialized to the character 'Z'. C1 is a 1-byte variable with no initial value.

Fig 1.2b

SIC/XE version

```
        LDA     #5
        STA     ALPHA
        LDCH    #90
        STCH    C1
        .
        .
        .
ALPHA   RESW    1
C1      RESB    1
```

Load VALUE 5 into Register A
Store in ALPHA
Load ASCII code for 'Z' into reg A
Store in character variable C1

one-word variable
one-byte variable

- The instructions shown in fig 1.2(a) also work on SIC/XE however they would not take advantage of the more advanced hardware features available
- In Ex 1.2 b, the value 5 is loaded in to register using immediate addressing. The operand field for this instruction contains the flag # (which specifies immediate addressing) and the data value to be loaded. Similarly the character "Z" is placed in to reg. A by using immediate addressing to load the value 90, which is the decimal value of the ASCII code that is used internally to represent the character 'Z'.

SIC Programming Example (Fig 1.3a)

- Arithmetic operations: BETA = ALPHA+INCR-1

```
        LDA     ALPHA
        ADD     INCR
        SUB     ONE
        STA     BETA
        LDA     GAMMA
        ADD     INCR
        SUB     ONE
        STA     DELTA
        ...
ONE     WORD    1       one-word constant
ALPHA   RESW    1       one-word variables
BETA    RESW    1
GAMMA   RESW    1
DELTA   RESW    1
INCR    RESW    1
```

SIC/XE Program

```
        LDS     INCR    Load value of INCR in
        LDA     ALPHA   Load ALPHA value in A
        ADDR    S,A     Add the value of INCR
        SUB     #1      Subtract 1
        STA     BETA    Store to BETA
        LDA     GAMMA   Load GAMMA value in A
        ADDR    S,A     Add the value of INCR
        SUB     #1      Subtract 1
        STA     DELTA   Store in DELTA
        ...
        ...             one word variables
ALPHA   RESW    1
BETA    RESW    1       • INCR is
GAMMA   RESW    1         loaded in S
DELTA   RESW    1
INCR    RESW    1       • ADDR is used
```

Fig-1.3(a) shows ex's of arithmetic instructions for SIC. All arithmetic operations are performed using reg A, with the result being left in register A. Thus this sequence of instructions stores the value (ALPHA + INCR − 1) in BETA and the value (GAMMA + INCR − 1) in DELTA.

SIC Programming Example (Fig 1.4a)

- Looping and indexing: copy one string to another

```
        LDX     ZERO        initialize index register to 0
MOVECH  LDCH    STR1,X      load char from STR1 to reg A
        STCH    STR2,X
        TIX     ELEVEN      add 1 to index, compare to 11
        JLT     MOVECH      loop if "less than"
        .
        .
        .
STR1    BYTE    C'TEST STRING'
STR2    RESB    11
ZERO    WORD    0
ELEVEN  WORD    11
```

The index register X is initialized to zero before the loop begins. Thus during the first execution of the loop, the target address for the LDCH instn will be the address of the first byte of STR1.

STCH instn will store the character being copied into the first byte of STR2. The next instn, TIX, performs two functions. First it adds 1 to the value in register X and then it compares the new value of reg X to the value of its operand (in this case, the constant value 11).

SIC/XE Programming Example (Fig 1.4b)

- Looping and indexing: copy one string to another

```
        LDT     #11         initialize register T to 11
        LDX     #0          initialize index register to 0
MOVECH  LDCH    STR1,X      load char from STR1 to reg A
        STCH    STR2,X      store char into STR2
        TIXR    T           add 1 to index, compare to 11
        JLT     MOVECH      loop if "less than" 11
        .
        .
        .
STR1    BYTE    C'TEST STRING'
STR2    RESB    11
```

The condition code is set to indicate the result of the comparison. The JLT instruction jumps if the condition code is set to "less than". Thus the JLT causes a jump back to the beginning of the loop if the new value in Reg X is less than 11. During the second execution of the loop, Reg. X will contain the value 1. Thus TA for the LDCH → 2nd byte of STR1. & TA for STCH → 2nd byte STR2. The TIX instn will again add 1 to the value in Reg X, & the loop will continue in this way until all 11 bytes have been copied from STR1 to STR2.

→ Fig 1.4b Same loop for SIC/XE. The difference is that instn TIXR is used in place of TIX. TIXR works like TIX, except that the value used for comparison is taken from another reg. (Reg T), not from mem. This makes loop more efficient.

- Immediate addressing is used to initialize Reg T to the value 11 & to initialize Reg. X to 0.

⇒ Fig 1.5 contain another ex. of indexing and looping. The variables ALPHA, BETA & GAMMA are arrays of 100 words each.

SIC Programming Example (Fig 1.5a)

```
        LDA     ZERO    initialize index value to 0
        STA     INDEX
ADDLP   LDX     INDEX   load index value to reg X
        LDA     ALPHA,X load word from ALPHA into reg A
        ADD     BETA,X
        STA     GAMMA,X store the result in a word in GAMMA
        LDA     INDEX
        ADD     THREE   add 3 to index value
        STA     INDEX
        COMP    K300    compare new index value to 300
        JLT     ADDLP   loop if less than 300
        ...
        ...
INDEX   RESW    1
ALPHA   RESW    100     array variables—100 words each
BETA    RESW    100
GAMMA   RESW    100
ZERO    WORD    0       one-word constants
THREE   WORD    3
K300    WORD    300
```

SIC/XE Programming Example (Fig 1.5b)

```
        LDS     #3
        LDT     #300
        LDX     #0
ADDLP   LDA     ALPHA,X load from ALPHA to reg A
        ADD     BETA,X
        STA     GAMMA,X store in a word in GAMMA
        ADDR    S,X     add 3 to index value
        COMPR   X,T     compare to 300
        JLT     ADDLP   loop if less than 300
        ...
        ...
ALPHA   RESW    100     array variables—100 words each
BETA    RESW    100
GAMMA   RESW    100
```

o The task of the loop is to add together the corresponding elements of ALPHA and BETA storing the results in the elements of GAMMA

Fig 1.5(a) - define a variable INDEX that holds the value to be used for indexing for each iteration of the loop. This INDEX should be 0 for the starting of the loop (ie first iteration), 3 for second iteration & soon. The first instn in the body of the loop loads the current value of INDEX in to reg. X, it can be used for the TA calculation.

- The next three instns in the loop load a WORD from ALPHA, add the corresponding word from BETA, & store the result in the corresponding word of GAMMA
- The value of INDEX is then loaded in to register A, incremented by 3. and stored back in to INDEX.
- The new value of INDEX is present in reg. A.
- This value is then compared to 300 (length of the arrays in bytes) to determine whether or not to

terminate the loop. If the value of INDEX is less than 300, then all bytes of the arrays have not yet been processed. In that case, the JLT instn causes a jump back to the beginning of the loop, where the new value of INDEX is loaded in to reg X.

- This loop in fig 15(b) for SIC/XE more efficient. The index value is kept permanently in reg X. The amount by which to increment the index value (3) is kept in reg. S, & the reg-to-reg ADDR instn is used to add this increment to reg. X. The value 300 is kept in reg. T, the instn COMPR is used to compare regs. X & T in order to decide when to terminate the loop.

Fig 1.7 → Instructions can be used to read a 100 byte record from an input device in to memory. The read operation is this ex. is placed in a subroutine. This subroutine is called from the main program by using the JSUB (Jump to Subroutine) instruction. At the end of the subroutine there is an RSUB (Return from subroutine) instn, which returns control to the instn that follows the JSUB.

```
        JSUB    READ        CALL READ SUBROUTINE
        .
        .
        .
                            SUBROUTINE TOREAD 100-BYTE RECORD
READ    LDX     ZERO        INITIALIZE INDEX REG. TO 0
RLOOP   TD      INDEV       TEST INPUT DEVICE
        JEQ     RLOOP       LOOP IF DEVICE IS BUSY
        RD      INDEV       READ ONE BYTE INTO REG A
        STCH    RECORD,X    STORE DATA BYTE INTO RECORD
        TIX     K100        ADD 1 TO INDEX & COMPARE to 100
        JLT     RLOOP       LOOP IF INDEX IS LESS THAN 100
        RSUB                EXIT FROM SUBROUTINE
        .
        .
        .
INDEV   BYTE    X'F1'       INPUT DEVILE NUMBER
RECORD  RESB    100         100-BYTE BUFFER FOR INPUT RECORD
                            ONE WORD CONSTANTS
ZERO    WORD    0
K100    WORD    100
```

Fig (a) - 17 (a)

SIC/XE Programming Example (Fig 1.7b)

```
        JSUB    READ        CALL READ SUBROUTINE
        .
        .
        .
                            SUBROUTINE TO READ 100-BYTE RECORD
READ    LDX     #0          INITIALIZE INDEX REGISTER TO 0
        LDT     #100        INITIALIZE REGISTER T TO 100
RLOOP   TD      INDEV       TEST INPUT DEVICE
        JEQ     RLOOP       LOOP IF DEVICE IS BUSY
        RD      INDEV       READ ONE BYTE INTO REGISTER A
        STCH    RECORD,X    STORE DATA BYTE INTO RECORD
        TIXR    T           ADD 1 TO INDEX AND COMPARE TO 100
        JLT     RLOOP       LOOP IF INDEX IS LESS THAN 100
        RSUB                EXIT FROM SUBROUTINE
        .
        .
        .
INDEV   BYTE    X'F1'       INPUT DEVICE NUMBER
RECORD  RESB    100         100-BYTE BUFFER FOR INPUT RECORD
```

o) Write the sequence of instructions for SIC to divide BETA by GAMMA and to store integer quotient in ALPHA, remainder in DELTA

```
LDA   BETA
LDS   GAMMA
DIVR  S,A      { DIVR r1,r2   r2 ← r2/r1
STA   ALPHA
MULR  S,A      r2 ← r2*r1
LDS   BETA
SUBR  A,S      r2 ← r2-r1
STS   DELTA
  :
ALPHA  RESW1
BETA   RESW1
GAMMA  RESW1
DELTA  RESW1
```

o) Let NUMBERS be an array of 100 words. Write a sequence of instructions for SIC & SIC/XE to set all 100 elements of the array to 1.

SIC

```
LDA   ONE
STA   INDEX
LOOP  LDX   INDEX
LDA   ONE
STA   NUMBER, X
LDA   INDEX
ADD   THREE
```

```
STA  INDEX
COMP k300
JLT  LOOP
:
INDEX  RESW 1
NUMBER RESW 100
ONE    WORD 1
k300   WORD 100
THREE  WORD 3
```

SIC/XE

```
LDS  #3
LDT  #300
LDX  #1
LOOP LDA  #1
STA  NUMBER,X
ADDR S,X
COMPR X,T
JLT  LOOP
:
NUMBER RESW 100
```

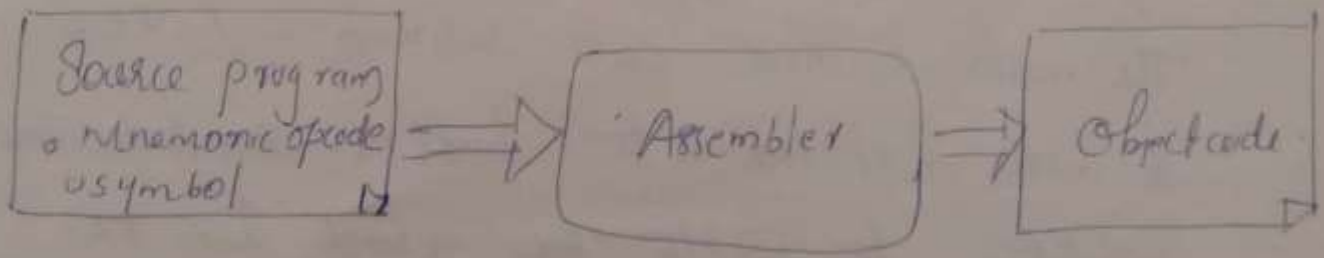
MODULE II.

Basic functions of Assembler, Assembler Output format – Header, Text and End Records – Assembler data structures, Two pass Assembler algorithm, Hand assembly of SIC/XE program, Machine dependent assembler features.

Basic Functions of Assembler :-

Fundamental functions that any assembler must perform, such as translating mnemonic operation codes to their machine language equivalents and assigning machine address to symbolic labels used by the programmer.

- The feature and design of an assembler depend heavily upon the source language it translates and machine language it produces.



Assembler directives :-

- Assembler directives are pseudo instructions

They provide instructions to the assembler itself. They are not translated into machine operation codes.

SIC assembler directives

START : Specify name and starting address for the program

END : Indicate the end of the source program (optionally) specify first executable instruction in the program

BYTE : Generate character or hexadecimal constant occupying many bytes as needed to represent the constant.

WORD : Generate one-word integer constant

RESB : Reserve the indicated number of bytes for a data area.

RESW : Reserve the indicated number of words for a data area.

- The program contains a main routine that reads records from an input device and copies them to an output device.

The main routine calls subroutine RDREC to read a record into buffer and subroutine WRREC to write the record from the buffer to the output device. Each sub routine must transfer the record one

character at a time because the only I/O instructions available are RD and WD. The buffer is necessary because the I/O rates for the two devices, such as disk and a slow printing terminal, may be very different. The end of each record is marked with a null character (hexadecimal 00). If a record is longer than the length of the buffer (4096 bytes) only the first 4096 bytes are copied.

- The end of the file copied is indicated by a zero length record. When the end of file is detected, the pgm writes EOF on the output device and terminates by executing an RSUB instruction. We assume that this pgm is called by the OS using JSUB instruction, then the RSUB will return control to the OS.

Forward reference

```
110                .
115                .     SUBROUTINE TO READ RECORD INTO BUFFER
120                .
125     RDREC      LDX   ZERO      CLEAR LOOP COUNTER
130                LDA   ZERO      CLEAR A TO ZERO
135     RLOOP      TD    INPUT     TEST INPUT DEVICE
140                JEQ   RLOOP     LOOP UNTIL READY
145                RD    INPUT     READ CHARACTER INTO R
150                COMP  ZERO      TEST FOR END OF RECOR
155                JEQ   EXIT      EXIT LOOP IF EOR
160                STCH  BUFFER,X  STORE CHARACTER IN BU
165                TIX   MAXLEN    LOOP UNLESS MAX LENGT
170                JLT   RLOOP       HAS BEEN REACHED
175                STX   LENGTH    SAVE RECORD LENGTH
180     EXIT       RSUB            RETURN TO CALLER
185     INPUT      BYTE  X'F1'     CODE FOR INPUT DEVIC
190     MAXLEN     WORD  4096
195                .

200                .     SUBROUTINE TO WRITE RECORD FROM BUFFER
205                .
210     WRREC      LDX   ZERO      CLEAR LOOP COUNTER
215     WLOOP      TD    OUTPUT    TEST OUTPUT DEVIC
220                JEQ   WLOOP     LOOP UNTIL READY
225                LDCH  BUFFER,X  GET CHARACTER FRO
230                WD    OUTPUT    WRITE CHARACTER
235                TIX   LENGTH    LOOP UNTIL ALL C
240                JLT   WLOOP       HAVE BEEN WRIT
245                RSUB            RETURN TO CALLER
250     OUTPUT     BYTE  X'05'     CODE FOR OUTPUT
255                END   FIRST
```

1. Simple SIC Assembler

Line	Loc	Source Statement			Object code
5	1000	COPY	START	1000	
10	1000	FIRST	STL	RETADR	141033
15	1003	CLOOP	JSUB	RDREC	482039

- The column headed Loc gives the machine address (in hexadecimal) for each part of the assembled Pgm. The Pgm starts at address 1000.

The translation of source pgm to object code requires to accomplish the following functions:

1. Convert Mnemonic operation codes to their machine language equivalents - eg. translate STL to 14 (line 10).
2. Convert symbolic operands to their equivalent machine address - eg. translate RETADR to 1033 (line 10).
3. Build the machine instructions in proper format
4. Convert the data constants specified in the source pgm in to their internal machine representations - eg. translate EOT to 454F46.
5. Write the object program & assembly listing

All of these functions except number 2 can easily be accomplished by the sequential processing of the source program, one line at a time. The translation of addresses, however presents a problem. Consider the statement

```
10    1000   FIRST   STL   RETADR   141033
```

This instruction contains a forward reference - i.e a reference to a label (RETADR) that is defined later in the program.

If we attempt to translate the pgm line by line, will be unable to process this statement because we dont know the address that will be assigned to RETADR.

- Because of this most assemblers make two passes over the source program.

2 passes

First pass: Scan the source program for label definitions and assign addresses (such as Loc column in fig)

Second pass: perform actual translation

In addition to translating the instructions of the Source program, the assembler must process statements called assembler directives (or pseudo-instructions).

- These statements are not translated into machine instructions. Instead they provide instructions to the assembler itself.
- Finally the assembler must write the generated object code on to some output device. This object program will later be loaded into memory for execution.
- The simple object program format we use contains three types of records

- Header
- Text
- End

Header record: contains the program name, starting address and length.

Text record: contain the translated (i.e machine code) instructions and data of the program, together with an indication of the address where

these are to be loaded.

The End record: marks the end of the object program and specifies the address in the program where execution is to begin. (This is taken from the operand of the program's END statement. If no operand is specified, the address of the first executable instruction is used).

The formats we use for these records are as follows. The details of the formats (column numbers, etc) are arbitrary. The ~~instruction~~ information contained in these records must be present in the object pgm.

Header record:

Col. 1	H
Col. 2-7	program name
Col. 8-13	Starting address of the object program (hexadecimal)
Col. 14-19	Length of the object program in bytes (hexadecimal)

Text record:

Col. 1 T

Col. 2-7 Starting address for object code in this record (hexadecimal)

Col. 8-9 Length of object code in this record in bytes (hexadecimal)

Col. 10-69 Object code, represented in hexadecimal (2 columns per byte of object code).

End Record

Col. 1 E

Col. 2-7 Address of first executable instruction in object program (hexadecimal).

Eg. object code:

- General description of the functions of the two passes of simple assembler.

Pass 1 (define Symbols):

1. Assign addresses to all statements in the program
2. Save the values (addresses) assigned to all labels for use in Pass 2.
3. Perform some processing of assembler directives (This includes processing that affects address assignment, such as determining the lengths of data areas defined by BYTE, RESW, etc).

Pass 2: (Assemble instructions and generate object program)

1. Assemble instructions (translating operation codes and looking up addresses)
2. Generate data values defined by BYTE, WORD, etc
3. Perform processing of assembler directives not done during Pass 1.
4. Write the object program and the assembly listing

Assembler Data Structures and Two pass

Algorithm:-

- Simple assembler uses two major internal data structures:
  - Operation code Table (OPTAB)
  - Symbol Table (SYMTAB)
- OPTAB is used to lookup mnemonic operation codes and translate them to their machine language equivalents.
- SYMTAB is used to store values (addresses) assigned to labels.
- Location Counter LOCCTR → This is a variable that is used to help in the assignment of addresses.

LOCCTR initialized to the beginning address specified in the START statement. After each source statement is processed, the length of the assembled instruction or data area to be generated is added to LOCCTR.

i.e. LOCCTR ← LOCCTR + (instruction length).

- The current value of LOCCTR gives the address to the label encountered.

- The Operation Code Table must contain the mnemonic operation code and its machine language equivalent. During pass 1, OPTAB is used to look up and validate operation codes in the source program. In pass 2, it is used to translate the operation codes to machine language.

- In SIC assembler, both of these processes could be done together in either pass 1 or pass 2.
- OPTAB is organized as a hash table, with mnemonic operation code as the key. [Hash table provides fast retrieval with a minimum of searching].
- OPTAB is a static table - i.e entries are not normally added or deleted from it.

⇒ The Symbol Table (SYMTAB) includes the name and address for each label in the source program, together with flags to indicate error conditions (e.g. a symbol defined in two different places).
This table may also contain other information about the data area or instruction labeled - for eg. its type or length.

During pass1 of the assembler, labels are entered into SYMTAB as they are encountered in the source pgm, along with their assigned addresses from LOCCTR). During pass2, symbols used as operands are looked up in SYMTAB to obtain the address to be inserted in the assembled instructions

- SYSTAB is usually organized as a hash table for efficiency of insertion and retrieval.
- SYSTAB is used heavily throughout the assembly, care should be taken in the selection of a hashing function
- programmers often select many labels that have similar characteristics - for ex, labels that start or end with the same characters (like LOOP1, LOOP2)
- So the hashing function used perform well with such non-random keys.

PPT

It is possible for both passes of the assembler to read the original Source program as input. However there is a certain information (Such as location counter values and error flags for statements) that can communicated between two passes. For this reason, pass 1 usually writes an intermediate file that contains each Source statement together with its assigned address, error indicators, etc. This file is used as the input to pass 2. This working copy of the Source program can also be used to retain the results of certain operations that may be performed during Pass 1. (Such as scanning operand field for symbols & addressing flags), So these need not be performed again during pass 2. Similarly pointers in to OPTAB and SYMTAB may be retained for each operation code & Symbol used. This avoids the need to repeat many of the table - Searching operations.

Pass1

```
begin
  read first input line
  if OPCODE = 'START' then
    begin
      save #[OPERAND] as starting address
      initialize LOCCTR to starting address
      WRITE line to intermediate file
      read next input line
    end {if START}
  else
    initialize LOCCTR to 0
    while OPCODE ≠ 'END' do
      begin
        if this is not a comment line then
          begin
            if there is a symbol in the LABEL field then
              begin
                Search SYMTAB for LABEL
                  if found then
                    set error flag (duplicate symbol)
                  else
                    insert (LABEL, LOCCTR) into SYMTAB
                end {if symbol}
              Search OPTAB for OPCODE
              if found then
                  add 3 {instruction length} to LOCCTR
              else if OPCODE = 'WORD' then
                  add 3 to LOCCTR
              else if OPCODE = 'RESW' then
                  add 3 * #[OPERAND] to LOCCTR
              else if OPCODE = 'RESB' then
                  add #[OPERAND] to LOCCTR
              else if OPCODE = 'BYTE' then
                begin
                  find length of constant in bytes
                  add length to LOCCTR
                end {of BYTE}
              else
                  set error flag (invalid operation code)
            end {if not a comment}
          WRITE line to intermediate file
          read next input line
      end {while not END}
    WRITE last line to intermediate file
    Save (LOCCTR - starting address) as program length
  end {pass 1}
```

pass2

```
begin read first input line {from intermediate file}
  if OPCODE = 'START' then
  begin write listing line
        read next input line
  end {if START}
  Write Header record to object program
  initialize first Text record
  while OPCODE ≠ 'END' do
     begin
        if this is not a comment line then
        begin Search OPTAB for OPCODE
           if found then
              begin
                 if there is a symbol in OPERAND field then
                 begin
                    Search SYMTAB for operand
                    if found then
                       Store symbol value as operand address
                    else begin
                          Store 0 as operand address
                          Set error flag (undefined symbol)
                         end
                 end {if symbol}
              else
                 Store 0 as operand address
                 assemble the object code instruction
           end {if opcode found}
        else if OPCODE = 'BYTE' OR 'WORD' then
           convert constant to object code
        if object code will not fit in to the current Text record then
        begin
           write Text record to object program
           initialize new Text record
        end
           add object code to Text record
        end {if not comment}
        write listing line
        read next input line
     end {while not END}
  write last Text record to object program
  write End record to object program
  write last listing line
end {pass2}
```

MACHINE DEPENDENT ASSEMBLER FEATURES

- Consider the design and implementation of an assembler for SIC/XE.
- In assembler language the following addressing indicates that -
  - Indirect addressing
    Adding prefix @ to operand. (line 70)
  - Immediate Operands
    Adding the prefix # to operand (lines, 12, 25, 55)
  - Base relative addressing
    Assembler directive BASE (Lines 12 & 13)
  - Extended format
    Adding the prefix + to OP code (lines 15, 35, 65).

The use of register to register instructions, faster and don't require another memory reference.

```
5   COPY    START   0           COPY FILE FROM input to output
10  FIRST   STL     RETADR      Save Return address
12          LDB     #LENGTH     Establish BASE Register
13          BASE    LENGTH
15  CLOOP   +JSUB   RDREC       Read input Record
20          LDA     LENGTH      Test for EOF (Length = 0)
25          COMP    #0
30          JEQ     ENDFIL      Exit if EOF found
35          +JSUB   WRREC       Write output Record
40          J       CLOOP       Loop
45  ENDFIL  LDA     EOF         Insert EOF file marker
50          STA     BUFFER
55          LDA     #3          Set Length = 3
60          STA     LENGTH
65          +JSUB   WRREC       Write EOF
70          J       @RETADR     Return to caller
80  EOF     BYTE    C'EOF'
```

```
95  RETADR   RESW   1
100 LENGTH   RESW   1      Length of Record
105 BUFFER   RESB   4096   4096 - Byte Buffer area
```

- The assembler directive BASE (Line 13) is used in conjunction with base relative addressing.
- If the displacement required for both program-counter relative and base relative addressing are too large to fit in to a 3-byte instⁿ, then the 4 byte extended format (format 4) must be used.

i.e. pc relative / Base relative addressing	op m
• Extended format	+ op m
• Indirect addressing	op @m
• Immediate addressing	op #c
• Index addressing	op m, X
• register to register	COMPR

⇒ SIC/XE involve the use of register to register instns (in place of reg to memory instns).
eg. COMP ZERO ⇒ COMPR A,S.

⇒ Immediate & Indirect addressing improve the execution speed of the program.

⇒ Reg. to reg. instns are faster than the corresponding reg. to memory operations because they are shorter, they do not require another mry reference

⇒ Using immediate addressing, the operand is already present as part of the instⁿ and need not be fetched from anywhere

◦ The use of indirect addressing often avoids the need for another instn. (eg. "return" operation on line 70).

Instruction formats and Addressing modes (Hand Assembly):-

In this section we consider mainly,

◦ Translation of the source statements, handling of different instruction formats & different addressing modes.

◦ The START statement specifies a beginning pgm address of 0. for the purpose of instn assembly the pgm will be translated exactly as if it were really to be loaded at machine addr 0.

◦ Register translation :

Register name (A, X, L, B, S, T, F, PC, SW) and their values (0, 1, 2, 3, 4, 5, 6, 8, 9)

. The conversion of register mnemonics to numbers can be done with a separate table (symbol table) for this purpose. To do this SYMTAB would be preloaded with the register names and their values.

◦ Register to Register instructions (Ex: CLEAR, and COMPR): - The assembler must simply convert the mnemonic operation code to machine language (using OPTAB). And change each register mnemonic to its numeric equivalent [This translation is done during pass 2 (the above point)]

Address translation:

- Most of the register to memory instructions are assembled using either program counter relative or base relative addressing.
- The assembler calculate a displacement to be assembled as part of the object instn. This is computed so that the correct target address results when the displacement is added to the contents of the pgm counter (PC) or the base register (B).

The resulting displacement is -

Format 3 : 12 bit disp (address) field.

- PC-relative : -2048 to 2047
- Base relative : 0 to 4095

Format 4: 20 bit address field.

- If neither pgm counter relative nor base relative addressing can be used (because displacement too large), then the 4 byte extended instruction format (format 4) must be used.
- It is large enough to contain full memory address.
- no displacement to be calculated in this case.

Ex: 15 0006 CLOOP +JSUB RDREC 4B101036

In this instn the operand address is 1036.

The full address is stored in the instruction, with bit e set to 1 to indicate extended instruction format.

The programmer must specify the extended format by using the prefix + (as on line 15). If extended format is not specified our assembler first attempt translate the instn using pgm counter relative addressing. If it is not possible, attempts to use base relative. If neither form of relative addressing is applicable and extended format is not specified, then the instn cannot be properly assembled. In this case, the assembler must generate an error msg.

- Now the displacement calculation for pgm counter and base relative addressing modes of format 3.

Ex. 10 0000 FIRST STL RETADR 17202D.

- During execution of instructions on SIC, the pgm counter is advanced after each instruction is fetched and before it is executed. Thus during the execution of the STL instn, the PC will contain the address of the next instn (i.e. 0003). From the Loc column of the listing, RETADR (line 95) is assigned address 0030. (The assembler would get this add. from SYMTAB).

The displacement we need is the listn is

$30 - 3 = 2D$.

0011 0000 −
0011
0010 1101

At the execution time, the target address calculation performed will be (PC) + disp.

ie 0011 + (PC)
0010 1101 disp) ie 0030.
0011 0000 − (30).

The displacement calculation process for base relative addressing is much the same as for pgm counter relative addressing. The main difference is that the assembler knows what the contents of the pgm counter will be at execution time. The base reg. on the other hand, is under control of the programmer. Therefore, the programmer must tell the assembler what the base registers will contain during execution of the pgm so that the assembler can compute the displacements. This is done with the assembler directive BASE.

The statement BASE LENGTH (line 13) informs the assembler that the base register will contain the address of LENGTH.

The preceding instruction (LDB #LENGTH) loads this value to the register during program execution)

To 00 2A J @RETADR

RETADR @ is at address 0030

TA = (PC) + disp. [disp = 0030 − 002D = (0003)$_{16}$]

∴ = 002D + 003

= 0030 [address of the RETADR].

op	ni	xbpe	disp
001111	10	0010	003

=> 3E 2003.

Program Relocation :-

- It is desirable to load and run several programs at the same time.
- The system must be able to load programs in to memory wherever there is a room
- The exact starting address of the program is not known until load time.
- The previous program is an example of an absolute program (or absolute assembly). This pgm must be loaded at address 1000 (the address that was specified at assembly time). in order to execute properly.

55 101B LDA THREE 00 102D

calculate based on the starting address 1000

Reload the pgm starting at 3000

55 101B LDA THREE 00 302D

The absolute address should be modified

This instr is translated as 00102D, specifying that reg. A is to be loaded from memory address 102D

Suppose we attempt to load & execute the pgm at 2000 instead of address 1000. If we do this, address 102D will not contain the value that we expect - in fact, it will probably be part of some other user's program.

Obviously we need to make change in the address portion of this instn so we can load and execute our pgm at address 2000.

- The assembler doesnot know the actual location where the pgm will be loaded.
- The assembler can identify for the loader those parts of the pgm that need modification.
- An object program that contains the information necessary to perform this kind of modification is called a relocatable program.

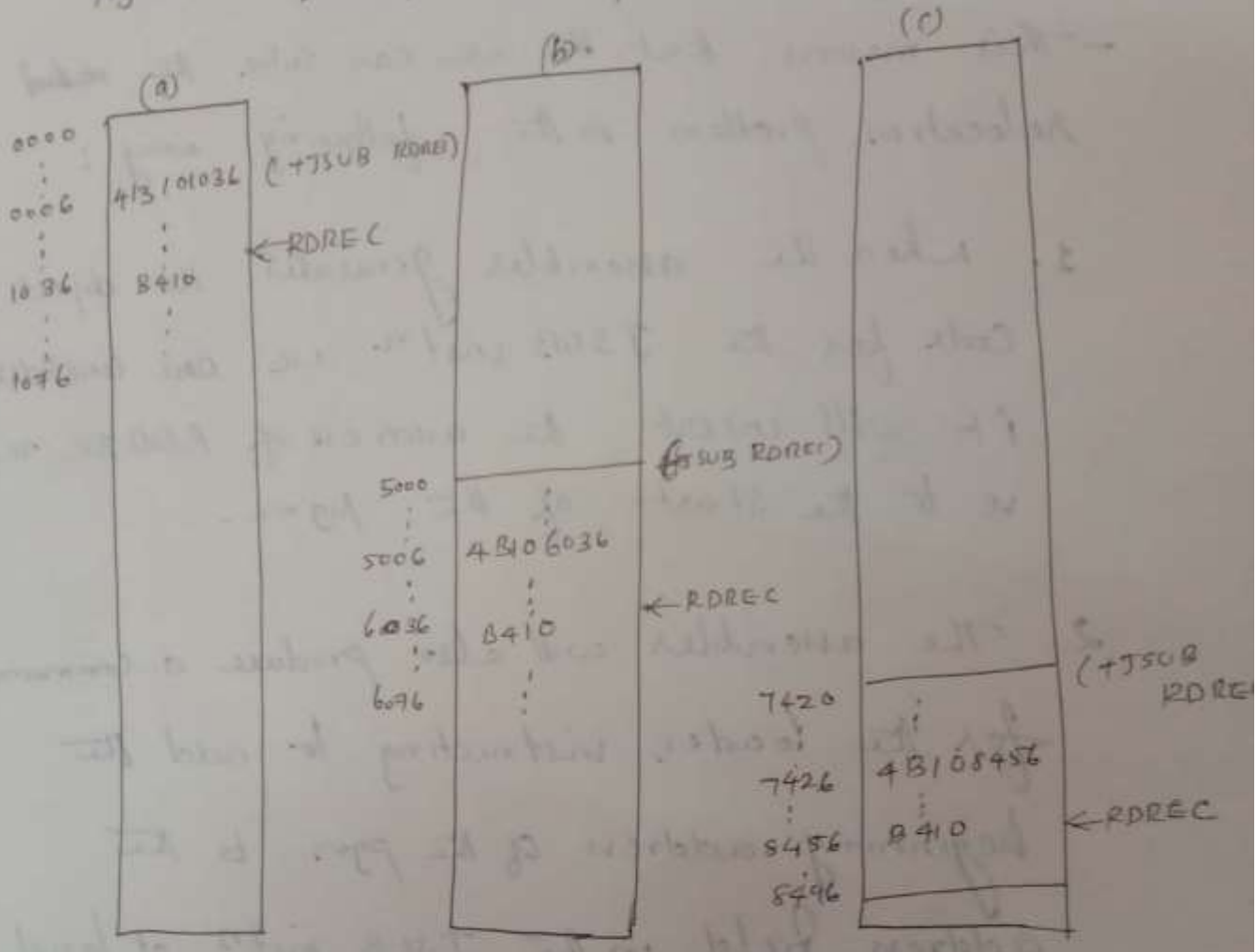
Ex. fig. 2.6 line is —

0006 CLOOP +JSUB RDREC 4B101036

fig. 2.7(a) shows this pgm loaded at

beginning at address 0000.
The JSUB instn from line 15 is loaded at address 0006.
- The address field of the instruction labeled RDREC contains 1036, which is the address of the instruction labeled RDREC.

Fig 2.7 Examples of pgm Relocation.



Now Suppose that we want to load this pgm beginning at address 5000, as fig 2-7 (b).
The address of the instn labeled RDREC is then 6036. JSUB instn modified as shown to contain this new address.

Like wise, if we loaded the pgm beginning at address 4420 (Fig. 2.7 C), the JSUB instn would need to be changed to 4B10 8456 to correspond to the new address of RDREC.

- No matter where the pgm is loaded, RDREC is always 1036 bytes past the starting address of the pgm.
- This means that we can solve the relocation problem in the following way:

1. When the assembler generates the object code for the JSUB instn we are considering it will insert the address of RDREC relative to the start of the pgm.

2. The assembler will also produce a command for the loader, instructing to add the beginning address of the pgm to the address field in the JSUB instn at load time.

MODULE III

Assembler Design options:

Machine Independent assembler features - program blocks, Control sections, Assembler design options - Algorithm for single pass assembler, Multipass assembler, Implementation example of MASM assembler

2.3. Machine Independent Assembler features

. Some common assembler features that are not closely related to machine architecture.

2.3.1 - Implementation of literals within an assembler, including required data structure & processing logic.

2.3.2 - two assembler directives (EQU and ORG) whose main function is the definition of symbols.

2.3.2 - The use of expressions in assembler language statements. (different types of expressions and their evaluation rule.

2.3.4 & 2.3.5 - introduce ~~different~~ important topics of program blocks and control sections

2.3.1 : Literals:

It is often convenient for the programmer to be able to write the value of a constant operand as a part of the instruction that uses it. This avoids having to define the constant elsewhere

in the program and make up a label for it. Such an operand is called a literal because the value is stated "literally" in the instruction.

- A literal identified with the prefix =

```
45   0014   ENDFIL   LDA   =C'EOF'   032010
```

- Specifies a 3 byte operand whose value is the character string EOF.

```
25   1062   WLOOP    TD    =X'05'    E 32011
```

- Specifies a 1 byte literal with the hexadecimal value 05

- The difference between literal operands(=) and immediate operands (#)
- With the immediate addressing, the operand value is assembled as part of the machine instruction, no memory reference
- With a literal, the assembler generates the specified value as a constant at some other memory location. The address of this generated constant is used as the TA for the machine instruction, using PC relative or base relative addressing with memory reference.

Literal pools: All of the literal operands used in a pgm are gathered together into one or more literal pools. Normally literals are placed into a pool at the end of the pgm. The assembly listing of a program containing literals usually includes a listing of this literal pool, which shows the assigned addresses and the generated data values.

- The assembler directive LTORG, it creates a literal pool that contains all of the literal operands used since the previous LTORG (or the beginning of the pgm). This literal pool is placed in the object program at the location where the LTORG directive was encountered. (Fig 2.10) - line 93

⇒ If we had not used the LTORG statement on line 93, the literal =C'EOF' would be placed in the pool at the end of the pgm. This literal pool would begin at address 1073. This means that literal operand would be placed too far away from the instn referencing. It makes problem, to use PC relative addressing or Base relative addressing to generate an object code

- LTORG desirable to keep the literal operand close to the instruction that uses it.

⇒ most assembler recognize duplicate literals - i.e the same literal used in more than one place in the pgm - and store only one copy of the specified data value. for eg. the literal =X'05' is used in one pgm in lines 215 & 230.

However, only one data area with this value is generated. Both instructions refer to the same address in the literal pool for their operand.

⇒ The data structure needed to handle literal operands is LITTAB. For each literal used, LITTAB includes literal name, operand value, length and the address assigned to operand. LITTAB is organized as hash table, using literal name or value as key.

⇒ During pass 1, the assembler searches LITTAB for the specified literal name. If it is present no action is needed else the literal is added to LITTAB. When LTORG statement or end of pgm is encountered, assembler scans LITTAB entries and address is assigned to each entry in table.

⇒ During pass 2, the operand address for use in generating object code is obtained by searching LITTAB for each literal operand encountered.

2.3.2. Symbol Defining statements

Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values. The assembler directive generally used is EQU (for "equate").

SYMBOL EQU value

This stmt defines the given symbol (i.e, enters it in to SYMTAB) and assigns to it the value specified. The value may be given as a constant or as any expression involving constants and previously defined symbols.

- Improved readability in place of numeric values.

for eg. on line 133 in fig 2.5

+LDT #4096

to load the value 4096 in to reg. T. This value represents the maximum length record we could read with subroutine RDREC.

On assigning

MAXLEN EQU 4096

line 133 can be written as

+LDT #MAXLEN

When the Assembler encounters the EQU stmt, it enters MAXLEN in to SYMTAB (with the value 4096)

During Assembly of the LDT instruction the assembler searches SYMTAB for the symbol MAXLEN, using its value as the operand in the instruction.

- Another common use of EQU is in defining mnemonic names for registers
  - Registers A, X, L can be used by No's 0, 1, 2.
- Suppose that the assembler expected the register numbers instead of names in an instruction like RMO (Register move)

  ie RMO 0, 1 ⟹
  RMO A, X

  ```
  A   EQU  0
  X   EQU  1
  L   EQU  2
  ```

- The standard names of (base, index) reflect the usage of registers

```
BASE   EQU  R1
Count  EQU  R2
INDEX  EQU  R3
```

⇒ 'ORG' assembler directive can be used to indirectly assign values to Symbols.

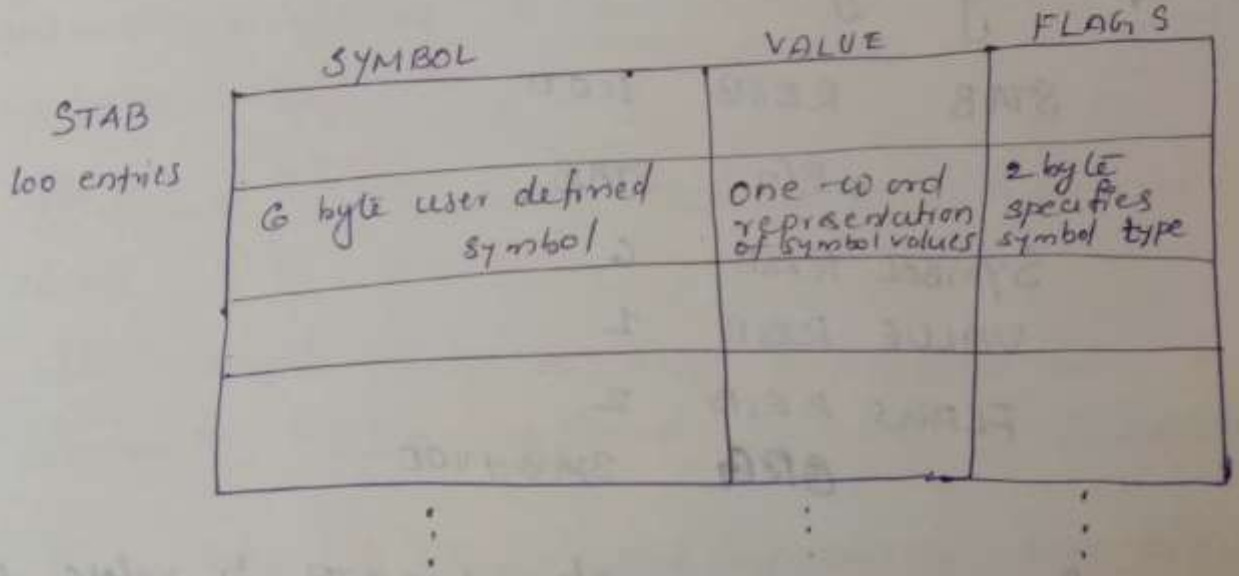
ORG 'Value'

Where value is a constant OR an expression.

- When this statement is encountered during

assembly of a program, the assembler resets its location counter (LOCCTR) to the specified value. Since the values of symbols used as labels are taken from LOCCTR, the ORG statement will affect the values of all labels defined until the next ORG.

for a symbol table with following structure



we could reserve space for this table with the statement → STAB RESB 1100

- To refer to the fields SYMBOL, VALUE, FLAGS individually, we must define these labels - one way of doing this using EQU statements.

```
SYMBOL   EQU   STAB      (1100)
VALUE    EQU   STAB+6    (1106)
FLAGS    EQU   STAB+9    (1109)
```

This statement LDA VALUE, X to fetch the value VALUE field from the table entry indicated by the contents of reg X. The above method simply defines the labels It does not make the structure of table clear.

- The same symbol definition using ORG is the following way.

```
STAB    RESB    1100
        ORG     STAB
SYMBOL  RESB    6
VALUE   RESB    1
FLAGS   RESB    2
        ORG     STAB+1100
```

- first ORG resets LOCCTR to value of STAB (Starting address of table).
- The label SYMBOL will get current value of LOCCTR as its value.
- The label VALUE is assigned address (STAB+6). LOCCTR is advanced to it.
- FLAGS is assigned STAB+9.
- This definition makes it clear each entry in STAB consists of 6-byte symbol, followed by one-word VALUE, followed by 2-byte flags.

2.

* The last 'ORG' sets LOCCTR back to its previous value. So any labels on subsequent statements (which are not part of STABS) are assigned proper address.

2.3.3. Expressions

Assemblers generally allow arithmetic expressions formed according to the normal rules using the operators +, −, * and /.

- Division is usually defined to produce an integer result.
- Individual terms in the expression may be constants, user-defined symbols or special terms.
- The values of terms and expressions are either relative or absolute. (independent of program location).
  - A constant is an absolute term.
  - Labels on instructions and data areas, and references to the location counter value are relative terms.
  - A symbol whose value given by EQU may be either an absolute term or a relative term depending upon the expression used to define its value.

Expressions are classified as either absolute expressions or relative expressions depending upon the type of value they produce.

- An expression that contains only absolute terms is an absolute expression.

- A relative term or expression represents some value that may be written as (S+R), where S is the starting address of the program and R is the value of the term or expression relative to the starting address. Thus the relative term usually represents some location with in the pgm.
- When relative terms are paired with opposite signs, the result is an absolute value.

Ex. 107 MAXLEN EQU BUFEND-BUFFER

- Both BUFEND and BUFFER are relative terms.
- The expression represents absolute value; it the difference between the two addresses
  - Loc = 1000 (Hex).
- BUFEND + BUFFER, 100 - BUFFER, 3* BUFFER represent neither absolute values nor locations.

- To determine the type of an expression, we must keep track of the types of all symbols defined in the program. For this purpose we need a flag in the symbol table to indicate type of value

(absolute or relative) in addition to the value itself.

Some symbol in the entries are like

symbol	TYPE	value.
RETADR	R	0030
BUFFER	R	0036
BUFEND	R	1036
MAXLEN	A	1000

- with this information the assembler can easily determine the type of each expression used.

2.3.4. Program Blocks

- The Source program logically contained main, subroutines, data areas. They were handled by the assembler as one entity, resulting in a single block of object code. With in this object program the generated machine instructions & data appeared in the same order as they were written in the Source programs.
- Flexible handling of the Source & Object programs. provide by many assemblers. These assemblers allow the generated machine instructions and data to appear in the object program in a different order from the corresponding Source statements.
- Other features result in the creation of the several independent parts of the object program.

→ Program blocks to refer to segments of code that are rearranged within a single object program unit.

→ Control Sections to refer to segments of code that are translated into independent object program units.

Fig 2.11 is ex: pgm written using 3 program blocks.

- The first (unnamed) program block contains the executable instructions of the program.
- The second (CDATA) contains all data areas that are a few words or less in length.
- The third (CBLKS) contains all data areas that consists of larger blocks of memory.

- The assembler directive USE indicates which portions of the source program belong to the various blocks. At the beginning of the pgm, statements are assumed to be part of the unnamed (default) block. If no USE statements are included, the entire program belongs to this single block.

eg: In fig 2.11 line 92 → USE CDATA ⇒

- Each program block may contain several separate segments.

The assembler will rearrange these segments to gather together the pieces of each block.

- The assembler accomplishes this logical arrangement of code by maintaining a separate location counter for each program block.
- The location counter for a block is initialized to 0 when the block is first begun.
- The current value of this location counter is saved when switching to another block and the saved value is restored when resuming the previous block.
- During pass 1, each label in the pgm is assigned an address. When labels are entered in to the symbol table, block name or number is stored along with the relative address.

At the end of the pass 1 the latest value of the location counter for each block indicates the length of that block.

At the end of the pass 1 assembler constructs a table that contains the starting addresses and lengths for all blocks.

Block name	Block number	Address	length
default	0	0000	0066
CDATA	1	0066	000B
CBLKS	2	0071	1000

During pass 2, the assembler needs the address for each symbol relative to the start of the object pgm (not the start of an individual program block). This is found from the information in SYMTAB. The assembler simply adds the location of the symbol, relative to the start of its block, to the assigned block starting address.

Ex:

20 0006 0 LDA LENGTH 032060

SYMTAB stores the value of the operand (the symbol LENGTH) as relative location 0003 within program block 1 (CDATA). The starting address for CDATA is 0066.

Thus TA = 0003 + 0066 = 0069.

⇒ The assembler directive USE indicates which portions of the source pgm belong to the various blocks.
The USE stmt on line 92. -[USE CDATA] signals the beginning of the block named CDATA. Source stmts are associated with this block until the USE stmt on line 103 [USE CBLKS], which begins the block named CBLKS. The USE stmt may also indicate the continuation of a previously begun block. Thus the stmt on line 123 [USE] resumes the default block. & the stmt on line 183 [USE CDATA] resumes the block named CDATA.

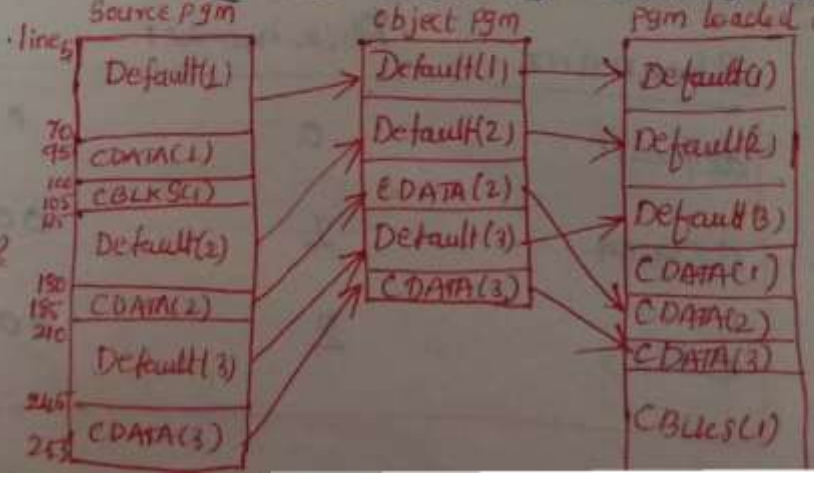


fig. 2.14 pgm blocks from fig 2.11 traced through the assembly & loading process.

2.3.5 Control Sections & Pgm linking

A control section is a part of the program that maintains its identity after assembly; each such control section can be loaded and relocated independently of the others.

- Different control sections are most often used for subroutines or other logical subdivisions of a pgm.
- The programmer can assemble, load and manipulate each of these control sections separately.
- Flexibility is a major benefit of using control sections.
- When control sections form logically related parts of a program, thus it needs linking them together. Instructions in one control section need to refer to instructions or data located in another section. Because control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. The assembler has no idea where any other control section will be located at execution time. Such references b/w control sections are called external references. The assembler generates information for each external reference that will allow the loader to perform the required linking.

External Reference Handling of Assembler

Ex. 2.15 example program includes 3 control sections (5-107) - START statement identifies the beginning

of assembly and gives a name (COPY) to the first control section.

(109-190) → CSECT (line 109) → indicates the second Control section named RDREC

(193-255) → CSECT start on line 193 begins the Control Section named WRREC.

- The assembler establishes a separate Location counter for each C.S (beginning at 0).

o Control Sections differ from pgm blocks in that they are handled separately by the assembler.

o Symbols that are defined in one C.S may not be used directly by another control section. They must be identified as external references for the loader to handle.

Two assembler directives to identify the references are: EXTDEF (External Definition).
EXTREF (External Reference).

* EXTDEF Statement in a control section names symbols called external symbols that are defined in this control section and may be used by other sections.

Ex in fig 2-15

```
5    COPY    START   0
6            EXTDEF  BUFFER, BUFEND, LENGTH
             :
```

```
100  LENGTH    RESW   1
     LTORG
103
105  BUFFER    RESB   4096
106  BUFFEND   EQU    *
```

- Control section names (in this case COPY, RDREC, and WRREC) do not need to be named in an EXTDEF statement because they are automatically considered to be external symbols.

* EXTREF statement names symbols that are used in this control section and are defined elsewhere.

for ex. the symbols BUFFER, BUFEND and length are defined in the CS COPY and made available to the other sections.

fig. 2.15

```
5    COPY      START   0
                EXTDEF  BUFFER, BUFEND,
6                       LENGTH
                EXTREF  RDREC, WRREC
7
:
:
109  RDREC     CSECT
:
                EXTREF  BUFFER, LENGTH,
120                     BUFEND.
:
193  WRREC     CSECT
```

Consider the instruction in fig 2.16.

15 0003 CLOOP +JSUB RDREC 4B 100000

The operand (RDREC) is named in the EXTREF statement for the control section, so this is an external reference. The assembler has no idea where the control section containing RDREC will be loaded, so it cannot be assemble the address for this instruction.

- The assembler inserts an address of zero and passes information to the loader, which will cause the proper address to be inserted at load time.
- The extended format (4) must be used to provide room for the actual address to be inserted.
- The two new record types are Define and Refer. → Define record gives information about external symbols that are defined in this control section - ie symbols named by EXTDEF.

A Refer record lists symbols that are used as external references by the control section - ie symbols named by EXTREF.

The format of these record as follows.

Define record:

col. 1	D
col. 2-7	Name of external symbol defined in this C.S
col. 8-13	Relative address of symbol with in this C.S (Hex)
col. 14-73	Repeat information in Col. 2-13 for other external Symbols.

Refer record

col. 1	R
col. 2-7	Name of external symbol referred to in this C.S
col. 8-13	Name of other external reference symbols.

The other information needed for program linking is added to the Modification record type. The new format is as follows

Modification record (revised):

col. 1	M
col. 2-7	Starting address of the field to be modified, relative to the beginning of the control section (hexadecimal).
col. 8-9	Length of the field to be modified, in half bytes (hexadecimal).
col. 10	Modification flag (+ or -)
col. 11-16	External symbols whose value is to be added to or subtracted from the indicated field.

- The address field for the JSUB instn on line 15 begins at relative address 0004. Its initial value in the object pgm is zero. The modification record in CS copy specifies that address of RDREC is to be added to this field

thus producing the correct machine instruction for execution.

- The other two modification records is COPY perform similar functions for the instns on line 65 & 35.

2.4 ASSEMBLER DESIGN OPTIONS

- Two alternatives to the standard two-pass assembler logic.

2.4.1 - One-pass assembler - used when it is necessary to avoid the second pass over the source pgm.

2.4.2 - Multipass assembler - extension to the two pass logic that allows to handle forward references during symbol definition.

2.4.1 : One pass Assemblers

- The main problem in trying to assemble a program in one-pass involves forward references. [symbols have not yet been defined in the source program. Thus the assembler doesn't know what address to insert in the translated instruction]

- It is easy to eliminate forward references to data items; for that all such areas

be defined in the source pgm before they are referenced. (Eg 2.18)

- Forward references to labels on instructions cannot be eliminated easily (the logic of the pgm often requires a forward jump).

There are two main types of one-pass Assembler:

1) Produces object code directly in memory for immediate execution
2) Object program for later execution

1⇒ For the first kind of assembler, no object program is written out, and no loader is needed. called Load and go Assembler.

- Because the object program is produced in memory rather than being written out on secondary storage, the handling of forward reference becomes less difficult.
- The assembler simply generates object code instns as it scans the source program. If an instn operand is a symbol that has not yet been defined, the operand address is omitted when the instn is assembled.
- The symbol used as an operand is entered in to the symbol table. This entry is flagged to indicate that the symbol is undefined.

- The address of the operand ... instruction that refers to the undefined symbol is added to a list of forward references associated with the symbol table entry.
- When the definition for a symbol is encountered, the forward reference list for that symbol is scanned, and the proper address is inserted in to any instns previously generated.
- Fig 2.19(a) shows the object code and symbol table entries as they would be after scanning line 40 in fig. 2.18. The first forward reference occurred on line 15. Since the operand (RDREC) was not yet defined, the instruction was assembled with no value assigned as the operand address (denoted in the fig. by ---). RDREC was then entered in to SYMTAB as an undefined symbol (indicated by *). The address of the operand field of the instn (2013) was inserted in a list associated with RDREC.

Symbol	Value
LENGTH	100C
RDREC	* → 2013 | 0
THREE	1003
ZERO	1006
WRREC	* → 201F | 0
EOF	1000
ENDFIL	* → 2016 | 0
RETADR	1009
BUFFER	100F
CLOOP	2012
FIRST	200F

Mry address	Contents			
1000	454F 4600	00B60000	00XXXXXX	XXXXXX
1010	XXXXXXXX	XXXXX	XXXXX	XXXXX
⋮				
2000	X XXX	X XXX	XXXX	XXX
2010				
2020				
⋮				

2.19(a). Object code in mry and symbol table entries for the pgm fig. 2.18 (after scanning line 40).

- fig - 2.19(b), which corresponds to the situation after scanning line 160. Some of the forward references have been resolved by this time, while others have been added.
- At the end of the program, any SYMTAB entries that are still marked with * indicate undefined symbols. These should be flagged by assembler as errors.

Mrry address	Contents
1000	454F4600 0003 0000 00xxxxxx xxxxx
1010	x xxxxxxx xxxxxxxx xxxxxxxx xxxxxxx
⋮	⋮
2000	xxxxxx xxxx xx xxxxxxx xxxxxx
2010	
2020	
2030	
2040	
2050	
⋮	

Symbol	value
LENGTH	100C
RDREC	203D
THREE	1003
ZERO	1006
WRREC	* → 2015 → 2039
EOF	1000
ENDFIL	2024
RETADR	1009
BUFFER	100F
CLOOP	2012
FIRST	2000
MAXLEN	2039
INPUT	2039
EXIT	* → 2056
R LOOP	2043

Fig. 2.19(b). object code in mrry and symbol table entries for pgm in fig. 2.18 after scanning line 160.

- When the symbol ENDFIL was defined (line 45) - (2024), the assembler placed its value in the SYMTAB entry. It then inserted this value in to the inst? operand field (2012).
→ RDREC — 203D @ 2013.
→ Meanwhile the two fwd references have been added WRREC (line 65) and EXIT (line 155).

When the end of the program is encountered, the assembly is complete. If no errors have occured, the assembler searches SYMTAB for the value of the symbol named in the END stmt. and jumps to this location to begin execution of the assembled pgm.

2) One pass assemblers that produce object pgm for follow later execution follow a slightly different procedure. (User loader service)

- Forward references are entered into lists as before. But when the definition of a symbol is encountered instructions that made forward reference to that symbol may no longer be available in memory for modification. They will already have been written out as part of Text record in object program.
- The assembler must generate another Text record with correct operand address. When the pgm is loaded this address will inserted into to the inserted instruction by the action of the loader.
- Fig. 2.20 The second text record contains the object code generated by from lines 10 through 40 in fig. 2.18

Fig 2.20

1). T,001000,09,454F46,000003,000000

- The operand address for the instns on line 15/

30, 35 have been generated 0000.

2) T^002000F,15^14 1009^480000^001000^2810064^30 0000^
line 10 line 15 line 30

→ On executing line 45, ENDFIL is defined, the assembler generates third Text Record

– This specifies the value 2024 (address of ENDFIL) is to be loaded at location 201C (operand add. field on line 30).

– when the pgm loaded, this value 2024 will replace the 0000 previously loaded.

3) T^00201C^02^2024
Add. of operand field of JEQ in line 30
Add. of ENDFIL.

The other forward references in the pgm are handled in exactly the same way. In effect the services of the loader are being used to complete forward references.

MULTIPASS ASSEMBLERS

It can make as many passes as are needed to process the definitions of symbols. It is not necessary for such an assembler to make more than two passes over the entire program.

Eg shows the sequence of symbol defining statements that involve forward references.

Ex.

1) HALFSZ EQU MAXLEN/2

2) MAXLEN EQU BUFFEND - BUFFER

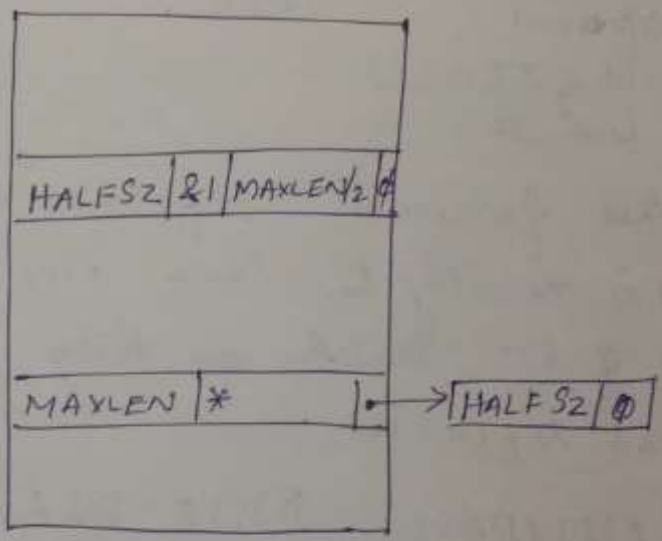
3) PREVBT EQU BUFFER-1

⋮

4) BUFFER RESB ~~4096~~ 1034

5) BUFEND EQU *

Fig - below shows - that it display symbol table entries resulting from pass 1 processing of the statement.



HALFSZ EQU MAXLEN/2.

MAXLEN has not yet been defined, so no value for HALFSZ can be computed. The defining expression for HALFSZ is stored in the symbol table in place of its value.

- The entry &1 indicates that one symbol in the defining expression is undefined.

- In an actual implementation, this definition must be stored at some other location. SYMTAB contains pointer to the defining expression.
- The symbol MAXLEN entered in the symbol table, with the flag * identifying it as undefined.
- This entry also includes as list of symbols whose values depend on MAXLEN.

Fig - C

Fig - D

BUFEND	*		→ MAXLEN | Ø
HALFSZ	$1	MAXLEN/2	Ø
PREVBT	1033		Ø
MAXLEN	$1	BUFFEND-BUFFER	→ HALFSZ | Ø
BUFFER	1034		Ø

Fig F

BUFFEND	2034		Ø
HALFSZ	500		Ø
PREVBT	1033		Ø
MAXLEN	1000		Ø
BUFFER	1034		Ø

In fig c - There are two undefined symbol involved in the definition: BUFFER and BUFFEND. These are entered in to SYMTAB with list indicating the dependence of MAXLEN upon them. Similarly the definition of PREVBT causes this symbol to be added to the list of dependencies on BUFFER [Fig d].

- The definition of BUFFER on line 4, begins evaluation of these some symbols.
- Let us assume that when line 4 is read, the location counter contains the hexadecimal value of 1034. This address stored as the value of BUFFER. The assembler then examines the list of symbols that are dependent on BUFFER.
- The symbol table entry for the first symbol in this list (MAXLEN) shows that it enters a value for MAXLEN causes the evaluation of the symbol in its list (HALFSZ). In fig (7) this completes the symbol def' process. If any symbols remained undefined at the end of the pgm, the assembler would flag them as errors.

Implementation Example

- Example of assembler for real machines
- focus on main interested features.

MASM Assembler

- An MASM assembler language program is written as a collection of segments.
- Each segment is defined as belonging to a particular class, corresponding to its contents.
- Commonly used classes are: CODE, DATA, CONST & STACK.

During program execution, segments are addressed via the x86 segment registers.

- Code segments are addressed using register CS. and stack segments are addressed using registers SS.
- These segment registers automatically set by the system loader when a program is loaded for execution.
- Register CS is set to indicate the segment that contains the starting label specified in the END stmt of a pgm.
- Register SS is to set to indicate the last stack segment processed by the loader.
- Data Segments (including constant segments) are normally addressed using DS, ES, FS or GS.
- The segment register to be used can be specified explicitly by the programmer (by writing it as part of the assembler language instn).
- If the programmer doesnot specify a segment reg, one is selected by the assembler.

By default, the assembler assumes that all references to data segments use register DS. This can be changed by the assembler directive ASSUME.

For ex. ASSUME ES: DATASEG2

tells the assembler to assume that reg. ES indicates the segment DATASEG2. Thus any references to labels that are defined in DATASEG2 will be assembled using reg. ES.
- It also possible to Collect several segments into group.
- Registers ES, DS, FS & GS loaded by the pgm before they can be used to address data segments

The instn →
```
MOV AX, DATASEG2
MOV ES, AX.
```

would set to ES to indicate the dataseg. DATASEG2.

- ASSUME tells MASM the contents of a segment reg. the pgmer must provide instns in to load this registers when the pgm is executed.

- Jump instructions are assembled in two different ways depending on whether the target of the jump is in the same code segment as the jump instn.
- A near jump is a jump to a target in the same code segment.
- A far jump is a jump to a target in a different code segment.
- A near jump is assembled using the current code segment reg CS.
- A far jump must be assembled using a different segment reg. which is specified in an instn prefix.

Ex: JMP TARGET [forward references to labels

in the pgm can cause problems.

If the defn of the label TARGET occurs in the pgm before the JMP instn, the assembler can tell whether this is a near jump or far jump.

If this is a forward reference to TARGET, the assembler doesnot know how many bytes to reserve for the instn.

By default MASM assumes that a forward jump is a near jump. If the target of the jump is another code segment, the programmer must warn the assembler by writing

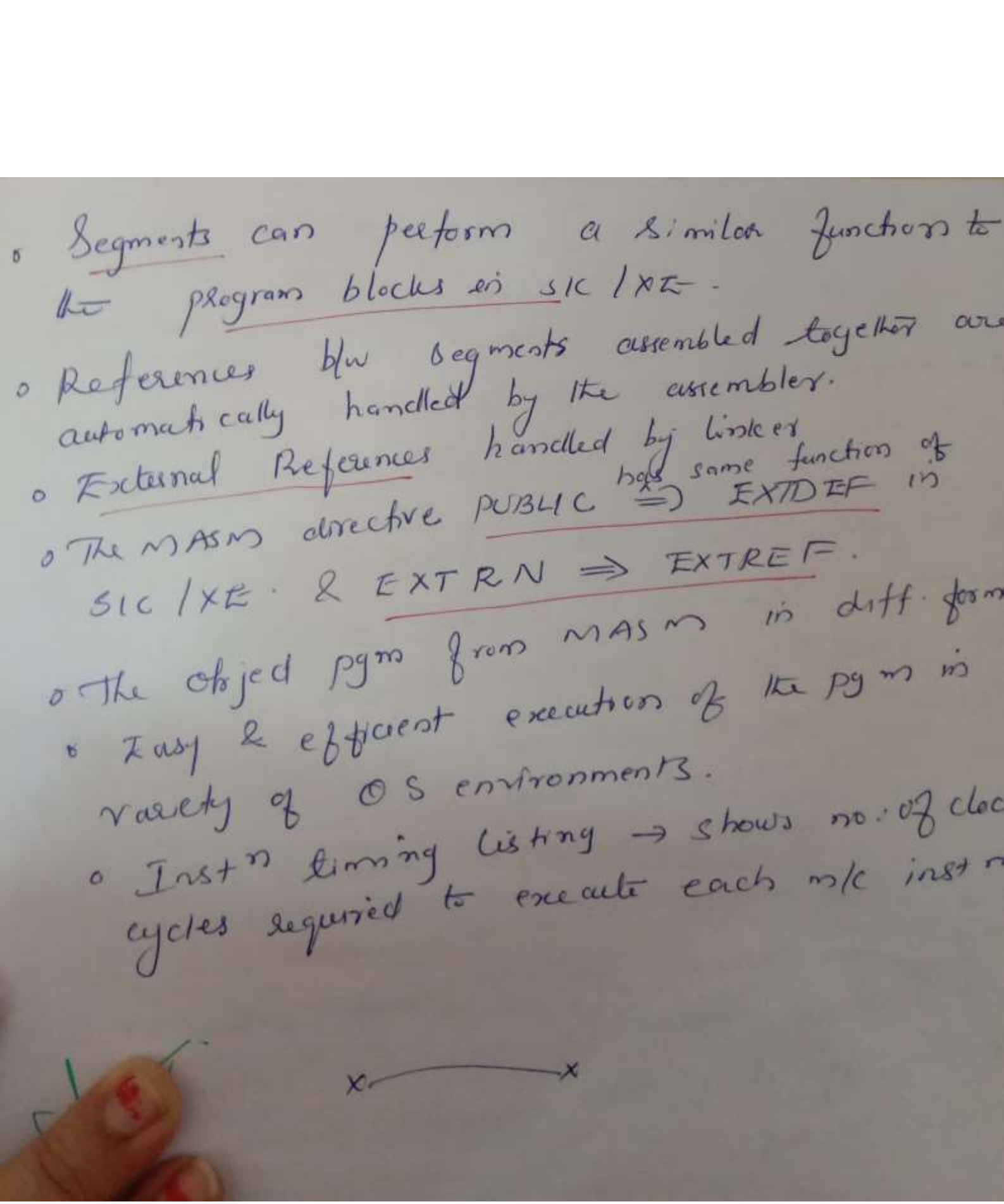
JMP FAR PTR TARGET

If the jump address is with in 128 bytes of the current instn, the programmer can specify the shorter (2-byte) near jump by writing

JMP SHORT TARGET.

- Similarity b/w the far jump & the forward reference in SIC/XE that require the use of extended format instns.
- pass 1 of an x86 assembler Complex than pass 1 of a SIC assembler. because analyze the opecands of an instn. & operation code table are more complicated.

- Segments can perform a similar function to the program blocks in SIC/XE.
- References b/w segments assembled together are automatically handled by the assembler.
- External References handled by linker
- The MASM directive PUBLIC ⇒ EXTDEF in SIC/XE & EXTRN ⇒ EXTREF. has same function of
- The objed pgm from MASM is diff. from
- Easy & efficient execution of the pgm in variety of OS environments.
- Instn timing listing → shows no. of clo cycles required to execute each m/c inst



MODULE IV

Linker & Loader:

1) Loading :- which brings the object program into memory for execution

2) Relocation :- Modifies the object program so that it can be loaded at an address different from the location originally specified.

3) Linking :- Combines two or more separate object programs and supplies the information needed to allow references between them.

- A Loader is a system program that performs the loading function. Many loaders also support relocation and linking. Some systems have a linker (linkage editor) to perform the linking operations and a separate loader to handle relocation and loading

3.1 Basic Loader functions

The fundamental functions of a loader - bringing an object program into memory and starting its execution

3.1.1 Design of an absolute Loader

Loader doesnot need to perform functions like linking and program relocation, its operation is very simple.

All functions are accomplished in a single pass.

- The Header record is checked to verify that the correct program has been presented for loading. (and that it is to the available mry).
- As each text record is read, the object code it contains is moved to the indicated address in mry.
- When the End record is encountered, the loader jumps to the specified address to begin execution of the loaded pgm.

memory address	contents			
0000	xxxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮				
1000	14 103348	20390010	36281030	30101548
1010	⋮	⋮	⋮	⋮
1020	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮
2070	xxxxxxxx 2C103638	xxxxxxxx 20644C00	xxxxxxxx 0005	xxxxxxxx

Fig 3.1(b).

- Representation of the pgm in fig (2.2) in modules

```
HCOPY 001000 001077
T^001000^1E^141033^482039^001036^281030^301015^482061
⋮
E^001000
```

Fig 3.1(a) → object program.

- In the object program each byte of assembled code

is given using the hexadecimal representation in character form. for ex. the machine operation code for an STL instn represented by the pair of characters "1" & "4" when these are read by the loader, they occupy two bytes of memory.

- In the instn as loaded for execution, this operation code must be stored in a single byte with hexadecimal value 14.
- Thus each pair of bytes from the object program record must be packed together in to 1 byte during loading.

fig. 3.1 (a) each printed character represent 1 byte of the object pgm record.

fig 3.1(b), each printed character represents one hexadecimal digit in mry.

- This method is ineffficient in terms of space & execution time
- Therefore most machines store object programs in a binary form, with each byte of object code stored as a single byte in the object pgm.
- Fig - 3.2 shows an algorithm for the absolute loader.

```
begin
    read Header record
    verify program name & length
    read first Text record
    while record type ≠ 'E' do
    begin
        { if object code is in character
          form, convert in to internal
          representation
        }
```

move object code to specified location
in memory.
read next object program record
end
Jump to address specified in End record.
end.

3.1.2 A Simple Bootstrap Loader

When a computer is first turned on or restarted a special type of absolute loader called a bootstrap loader is executed.

- This bootstrap loads the first program to be run by the computer — usually an Operating system.
- The bootstrap begins at address 0 in the memory of machine. It loads the OS starting at address 80.
- Each byte of the object code to be loaded is represented on device F1 as two hexadecimal digits. (It is like a Text record of a SIC pgm).
- The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80.
- After all of the object code from device F1 has been loaded, the bootstrap jumps to address 80, which begins the execution of the pgm that was loaded.

→ The work of the bootstrap loader is performed by the subroutine GETC.

Subroutine GETC:-

- The subroutine GETC reads one character from device F1 and converts it from ASCII character code to the value of hexadecimal digit that is represented by that character. for ex, the ASCII code for the character "0" (hexadecimal 30) is converted to the numeric value 0.
- The bootstrap ignores any control bytes that are read.
- The main loop of the bootstrap keeps the address of the next memory location to be loaded in register X. GETC is used to read and convert a pair of characters from device F1.
- These two ~~de~~ hexadecimal digit values are combined in to a single byte.
- The resulting byte is stored at the address currently in register X, using STCH instn that refers to location 0 using indexed addressing.
- The TIXR instruction is then used to add 1 to the value in register X.

- Bootstrap loader GETC subroutine shown in Fig 3.3 P.No. 134.

3.2 MACHINE DEPENDENT LOADER FEATURES

- Absolute loader has several disadvantages.
- In a simple computer with small memory, the programmer to specify (when the pgm is assembled) the actual address at which it'll be loaded is to memory. Starting address of user pgm known in advance.
- On a larger and more ~~one~~ advanced machine, several independant pgms run together, sharing memory and other resources between them.
- Do not know the starting address in advance.
- Thus efficient sharing of the machine requires that we write relocatable pgms instead of absolute ones.
- Writing absolute pgms also makes it difficult to use subroutine libraries efficiently.
- ~~So~~ If we use more complex loaders, it will support
  - Relocation (machine dependent – 3.2.1)
  - Linking (not m/c dependent – 3.2.2)

3.2.1 Relocation

Loaders that allow for program relocation are called relocating loaders or relative loaders.

There are two methods for specifying relocation.

1) Modification record:

It is used to describe each part of the object code that must be changed when program is relocated. In fig - 3.4 shows SIC/XE pgm.

- Most of the instructions in this pgm use relative or immediate addressing.
- fig - 3.4 lines 15, 35, 265 contains actual address. Thus these are the only items whose values are affected by relocation.

Line	Loc	Source Statement		Object code.
1) 15	0006	CLOOP +JSUB	RDREC	4B101036
:				
2) 35	0013	+JSUB	WRREC	4B10105D
:				
3) 65	0026	+JSUB	WRREC	4B10105D
		Subroutine to read record in to Buffer		
125	1036	RDREC CLEAR	X	B410
:	:			
		Subroutine to write Record from Buffer		
210	105D	WRREC CLEAR	X	B410

(Fig - 3.4)

- Fig - 3.5 shows the object program corresponding to the source in fig 3.4.

There is one modification record for each value that must be changed during relocation. (in this case three instrs).

- Each modification record specifies the starting address and length of the field whose value is to be altered. It then describes the modification to be performed.
- In this example, all modifications add the value of the symbol COPY, which represents the starting address of the pgm. (fig - 3.6).

```
H,COPY,0000000,001077
T,.....
T,.....
:
M,000007,05+COPY
M,000014,05+COPY
M,000027,05+COPY
E,000000
```

Fig - 3.5 object pgm with relocation by Modification records

Fig-3.6 SIC/XE rebcation loader Algorithm.

```
begin
    get PROGADDR from operating system
    while not end of input do
        begin
            read next record
            while record type ≠ 'E' do
            begin
                read next input record
                while record type = 'T' then
                begin
                    move object code from record to location
                        ADDR + specified address.
                end
                while record type = 'M'
                    add PROGADDR at the location PROGADDR
                        specified address.
        end
end
```

2) Relocation list:-

fig 3.7 shows SIC version, where almost all instructions except RSUB uses direct addressing. (fixed instn format)

In such case, the modification record number increases

- In relocation list technique there is a relocation bit associated with each word of object code. Thus one relocation bit for each instruction.

```
H,COPY ,000000,001OTA
T,000000,1EFFC,14 0033, ....
:
:
E,000000
```

fig - 3.8 [object pgm with relocation by bit mask]

- Std SIC machine doesn't using relative addressing.

There are no modification records.

- The relocation bits are gathered together into a bit mask following the length indicator in each Text record. In fig 3.8 this mask is represented (in characters form) as three hexadecimal digits.
- If the relocation bit corresponding to a word of object code is set to 1, the program's starting address is to be added to this word when the pgm is relocated
- A bit value of 0 indicates that no modification is necessary.
- If a Text record contains fewer than 12 words of object code, the bits corresponding to unused words are set to 0.
- for ex. the bit mask FFC (representing the bit string 111111111100) in the first Text record specifies that all 10 words of object code are to be modified during relocation.

2.2 Program Linking :-

The goal of program linking is to resolve the problems with external references (EXTREF) & external definitions (EXTDEF) from different control sections.

The EXTDEF statement in a control section names symbols, called external symbols that are defined in this (present) control section & may be used by other sections.

EXTREF - Statement names symbols used (in this) present control section and are defined elsewhere

- In fig-3.10 - three programs named as PROGA, PROGB, and PROGC, which are separately assembled and each of which consists of a single control section.

LISTA, ENDA in PROGA.
LISTB, ENDB in PROGB
LISTC, ENDC in PROGC are external definitions in each of the control sections.

Similarly LISTB, ENDB, LISTC & ENDC in PROGA.
LISTA, ENDA, LISTC & ENDC in PROGB
LISTA, ENDA, LISTB & ENDB in PROGC

are the external references.

These sample programs given here are used to illustrate linking & relocation.

Consider first the reference marked REF1.

0020 REF1 LDA LISTA 032010

⇒ For the first pgm (PROGA),

- REF1 is simply a reference to a label within the pgm.
- It is assembled in the usual way as a pc relative instruction.
- No modification for relocation or linking is necessary.

⇒ In PROGB, the same operand refers to an external symbol.

ie 0036 REF1 +LDA LISTA

- The assembler uses an extended-format instruction with address field set to 0000.
- The object program for PROGB (Fig - 3.11) contains a modification record instructing the loader to add the value of the symbol LISTA to this address field when the pgm is linked.

ie M^000037^05^+LISTA

⇒ For PROGC, REF1 is handled in exactly the same way.

ie 0018 REF1 +LDA LISTA 03100000

Modification record is that

M^000019^05^+LISTA

⇒ REF3 is an immediate operand whose value is to be the difference b/w ENDA and LISTA.

0027 REF3 LDX #ENDA-LISTA 050014

- In PROGA, the assembler has all of the information necessary to compute this value.

During the assembly of PROGB (and PROGC), the values of the labels are unknown. In these pgms the expression must be assembled as an external reference (with two modification records) even though the final result will be an absolute value independent of the locations at which the programs are loaded.

⇒ Consider REF4 of PROGA.

The assembler for PROGA can evaluate all of the expression in REF4 except for the value of LISTC.

0054 REF4 WORD ENDA-LISTA+LISTC 000014

This results in an initial value of '000014'H and one modification record.

M, 00005 4, 06, +LISTC

⇒ REF4 of PROGB, 0070 REF4 WORD ENDA-LISTA+LISTC

The same expression in PROGB contains no terms that can be evaluated by the assembler. The object code therefore contains an initial value of 000000 and three Modification records.

M, 000070, 06 + ENDA
M, 000070, 06 − LISTA
M, 000070, 06 + LISTC

⇒ REF4 of PROGC.

0042 REF4 WORD ENDA-LISTA+LISTC 000000

for PROGC, the assembler can supply the value of LISTC relative to the beginning of the pgm.

(but not the actual address, which is not known until the program is loaded). The initial value of this data word contains the relative address of LISTC ('000030'H). Modification records instruct the loader to add the beginning address of the program (ie the value of PROGC), to add the value of ENDA, and to subtract the value of LISTA.

- Fig - (3.12 A) shows the memory view after loading 3 programs

PROGA	4000
PROGB	4063
PROGC	40E2.

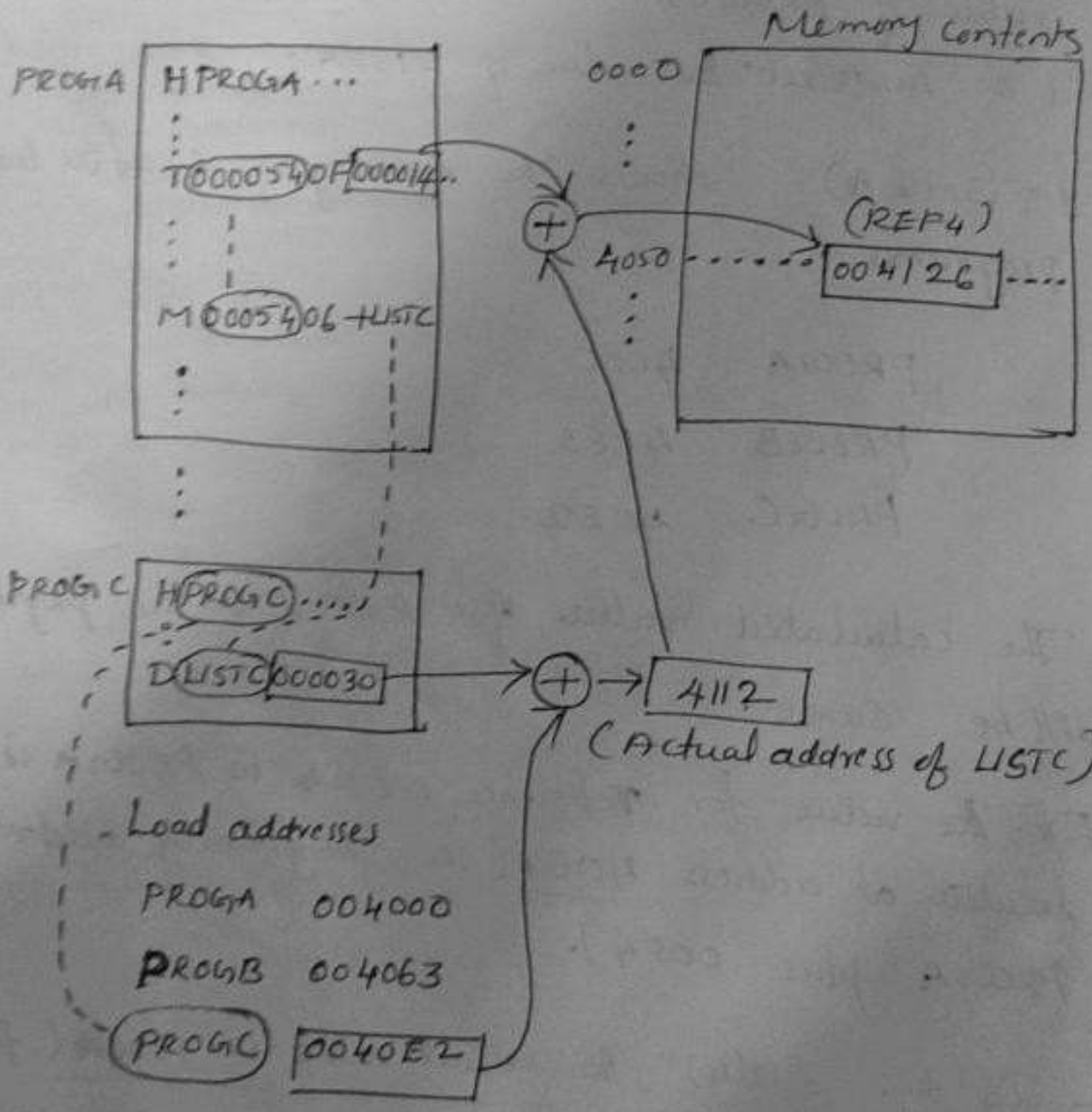
The calculated value for REF4 in all programs will be same.

Ex the value for reference REF4 in PROGA is located at address 4054 (the beginning address of PROGA plus 0054).

- In fig - 3.12(b), the initial value is 000014 (from text record). To this is added the address assigned to LISTC which is 4112 (0040E2 + 000030) same in PROGB & PROGC.
- for ex the value for reference REF4 in PROGA is located at address 4054 (4000 + 54). The address assigned to list, LISTC, which is 4112 (beginning address of PROGC + 30 ie 40E2 + 30)

To the initial value (000000), the loader adds the value ENDA (4054) & LISTC (4112) & subtracts LISTA (4000). The computation for REF4 is PROGA, PROGB & PROGC results in same value.

ENDA-LISTA +LISTC



3.12 (b) Relocation and linking operations performed on REF4 from PROGA.

3.2.3 Algorithm & Data Structures for a Linking Loader:-

The algorithm and for a linking loader is considerably more complicated than the absolute loader algorithm. The input to such a loader consist of a set of object programs that are to be linked together.

- The required linking operation cannot be performed until an address is assigned to the external symbol involved.
- A linking loader usually makes two passes over its input, just as an assembler does.

  pass 1: assigns addresses to all external symbols.
  pass 2: performs the actual loading, relocation, and linking.

- The main data structure needed for our linking loader is an external symbol table (ESTAB).
- This table, which is analogous to SYMTAB in assembler algorithm, is used to store the name and address of each external symbol in the set of control sections being loaded.
- The table also defines in which control section the symbol is defined. A hashed organization is used for this table.
- Two other important variables are PROGADDR (program address) and CSADDR (control section address).

- PROGADDR is the beginning address in memory where the linked program is to be loaded. Its value is supplied to the loader by the OS.

- CSADDR contains the starting address assigned to the control section currently being scanned by the loader.
This value is added to all relative addresses within the control section to convert them to actual addresses.

- Algorithm in fig ——

- During pass1, the loader is concerned only with Header and Define record types in the control sections.

1) The beginning load address for the linked program (PROGADDR) is obtained from the OS. This becomes the starting address (CSADDR) for the first control section in the input sequence.
2) The control section name from Header Record is entered in to ESTAB, with the value given by the CSADDR. All external symbols appearing in the Define Record for the control section are also entered in to ESTAB. Their addresses are obtained by adding the value specified in the DefineRecord to CSADDR.

Pass 1:

```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR (for first control section)
while not end of input do
   begin
      read next input record (Header record for control section)
      set CSLTH to control section length
      search ESTAB for control section name
      if found then
         set error flag (duplicate external symbol)
      else
         enter control section name into ESTAB with value CSADDR
      while record type ≠ 'E' do
         begin
            read next input record
            if record type = 'D' then
               for each symbol in the record do
                  begin
                     search ESTAB for symbol name
                     if found then
                        set error flag (duplicate external symbol)
                     else
                        enter symbol into ESTAB with value
                           (CSADDR + indicated address)
                  end (for)
         end (while ≠ 'E')
      add CSLTH to CSADDR (starting address for next control section)
   end (while not EOF)
end (Pass 1)
```

Figure 3.11(a) Algorithm for Pass 1 of a linking loader.

Pass 2:

```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
   begin
      read next input record (Header record)
      set CSLTH to control section length
      while record type ≠ 'E' do
         begin
            read next input record
            if record type = 'T' then
               begin
                  {if object code is in character form, convert
                     into internal representation}
                  move object code from record to location
                     (CSADDR + specified address)
               end {if 'T'}
            else if record type = 'M' then
               begin
                  search ESTAB for modifying symbol name
                  if found then
                     add or subtract symbol value at location
                        (CSADDR + specified address)
                  else
                     set error flag (undefined external symbol)
               end  {if 'M'}
         end {while ≠ 'E'}
      if an address is specified (in End record) then
         set EXECADDR to (CSADDR + specified address)
      add CSLTH to CSADDR
   end  {while not EOF}
jump to location given by EXECADDR (to start execution of loaded program)
end {Pass 2}
```

3) When the END record is read, the control section length CSLTH (which was saved from the End record) is added to CSADDR. This calculation gives the starting address for the next control section in sequence.

- At the end of the Pass1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each.
- Many loaders include as an option the ability to print a loadmap that shows these symbols and their addresses. (3.11 & 3.12 figs).
- Pass 2, of our loader performs the actual loading, relocation and linking of the pgm.

1) As each Text record is read, the object code is moved to the specified address (plus the current value of CSADDR).

2) When a Modification record is encountered, the symbol whose value is to be used for modification is looked up in ESTAB

3) This value is then added to or subtracted from the indicated location in the memory

4) The last step performed by the loader is usually the transferring of control to the loaded

program to begin execution.

- The end record for each control section may contain the address of the first instruction in that C.S to be executed. Our loader takes this as the transfer point to begin execution.

- If more than one control section specifies a transfer address, the loader arbitrarily uses the last one encountered.

- If no C.S contains a transfer address, the loader uses the beginning of the linked pgm (i.e PROGADDR) as the transfer point.

Normally, a transfer address would be placed in the END record for a main pgm, but not for a subroutine.

- This algorithm can be made more efficient. Assign a reference number, which is used (instead of symbol name) in modification records, to each external symbol referred to in a control section.

- Suppose we always assign the reference number 01 to the control section name.

3.3 MACHINE INDEPENDENT LOADER FEATURES.

- Loading and Linking are often thought of as an OS service functions. Therefore, most loaders includes fewer different features than are found in a typical assembler.

They include the use of an automatic library search process for handling external reference and some common options that can be selected at the time of loading and linking.

3.3.1 Automatic Library Search

- Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded.
- (The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program, and loaded. The programmer only needs to mention the subroutine names as external references in the source program. This feature is referred as automatic library search)
- Linking loaders that support automatic library search must keep track of external symbols that are referred to, but not defined, in the primary input to the loader. The symbols from each refer record are entered into ESTAB. These entries are marked to indicate that the symbols has not yet been

- when the definition is encountered, the address assigned to the symbol is filled in to complete the entry.

- At the end of the pass 1 the symbols in ESTAB that remain undefined represent unresolved external references. The loader searches the library or libraries specified for routines that contain the definitions of these symbols, and processes the subroutine found by this search exactly as if they had been part of the primary i/p stream.

- The subroutines fetched from a library themselves contain external references. It is therefore necessary to repeat the library search process until all the references are resolved. If unresolved external references remain after the library search is completed, these must be treated as error.

3.3.2 Loader Options:-

Many loaders have a special command language that is used to specify options. Sometimes there is a separate input file to the loader that contains such control statements. Sometimes these statements can be included in the primary input stream b/w object programs.

→ Typical loader option 1: allows the selection of alternative sources of i/p.

Ex. INCLUDE program-name (library-name)

- It direct the loader to read the designated object program from a library and treat it as if it were part of the primary loader input.

- Loader option 2: allows the user to delete external symbols or entire control sections ex.

. DELETE Csect-Name

It Instruct the loader to delete the named control section(s) from the set of programs being loaded.

. CHANGE name1, name2

might cause the external symbol name1 to be changed to name2 wherever it appears in the object programs.

- Loader option 3: Involves automatic inclusion of library routines to satisfy external references.

Ex. LIBRARY MYLIB

Such user defined libraries are normally searched before the standard system libraries. This allows the user to use special versions of the standard routines.

NOCALL STDDEV, PLOT, CORREL

To instruct the loader that these external references are to be remain unresolved. This avoids the overhead of loading & linking unneeded routines, and saves the memory space that would otherwise be required

Consider fig. 2.17 COPY as main pgm and RDREC, WRREC as subprograms.

Suppose that a set of utility programs (subroutines) is made available on the computer s/m. Two of these READ & WRITE are designed to perform the same function as RDREC and WRREC. It would be desirable to change the source pgm of COPY to use these utility routines, READ & WRITE.

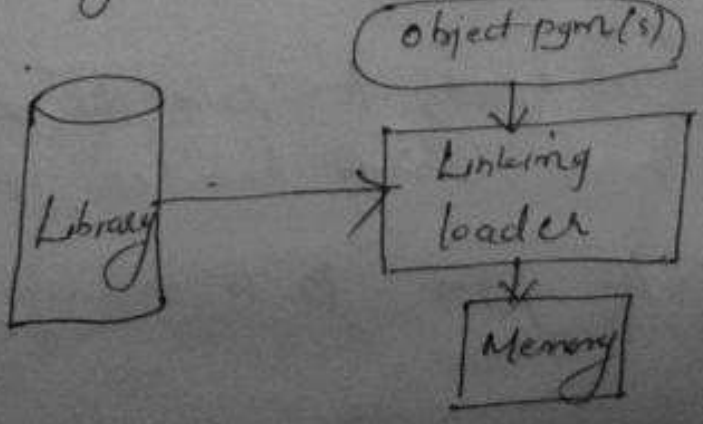
So the loader commands:

```
INCLUDE READ (UTLIB)
INCLUDE WRITE (UTLIB)
DELETE RDREC, WRREC
CHANGE RDREC, READ
CHANGE WRREC, WRITE
```

These commands would direct the loader to include Control Section READ & WRITE from library UTLIB & to delete control sections RDREC & WRREC from the load. The CHANGE commands would cause all external references to symbol RDREC / WRREC to be changed to refer to symbol READ / WRITE.

3.4 LOADER DESIGN OPTIONS

- Linking loader performs all linking and relocation at load time.
There are two alternatives: Linkage editors which peeforms linking prior to load time & dynamic linking, in which the linking function is performed at execution time.
- precondition: The source program is first assembled or compiled, producing an object program.
A linking loader peeforms all linking and relocation operations, including automatic library search, if specified, and loads the linked pgm directly into memory for execution.
- A linkage Editor produces a linked version of the pgm (load module or executable image) which is written to a file or library for later execution.
- The essential difference b/w linkage editor & a linking loader is illustrated as



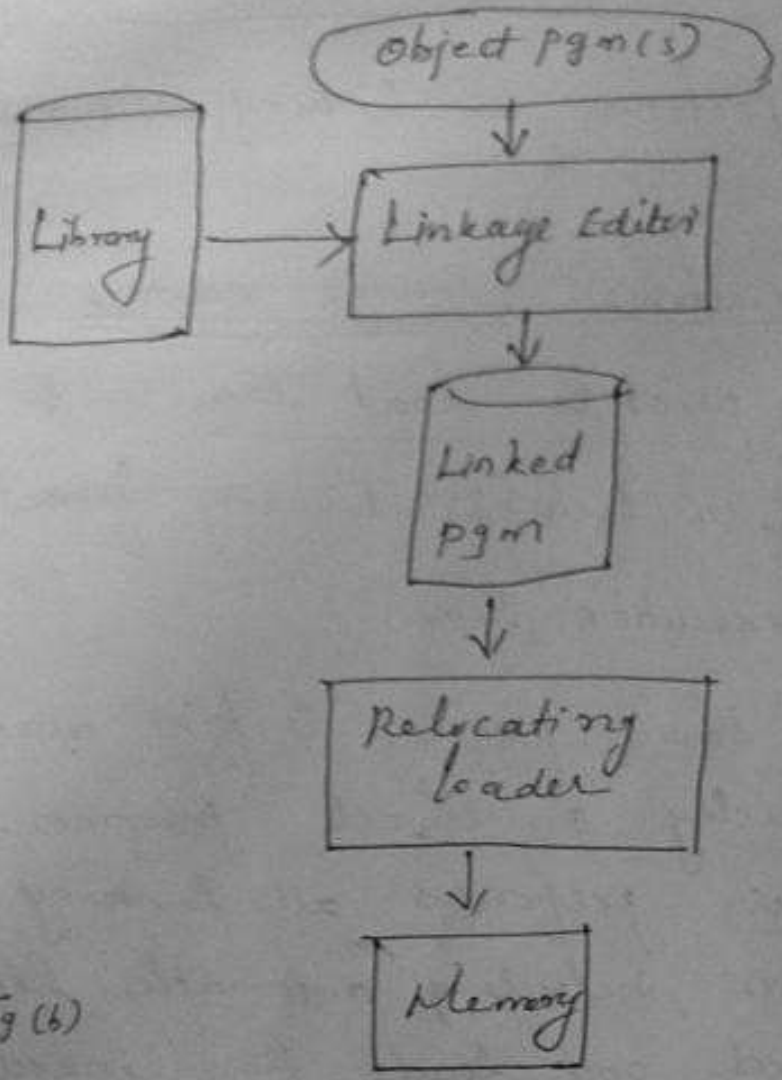


fig (b)

fig: Processing of an object pgm using
(a) linking loader
(b) Linkage editor.

3.4 / Linkage Editors :-

The source program is first assembled or compiled producing an object pgm (which may contain several different control sections).

- A linkage editor produces a linked version of the pgm (often called a load module or an executable image), which is written to a file or library for later execution.

- When the user is ready to run the linked pgm, a simple relocating loader can be used to load the pgm into memory.

- The linkage editor performs relocation of all control sections relative to the start of the linked program. Thus all items that need to be modified at load time have values that are relative to the start of the linked program.
- This means that the loading can be accomplished in one pass with no external symbol table required.
- If a program is to be executed many times without being reassembled, the use of linkage editor substantially reduces the overhead required.
- Linkage editors can perform many useful functions besides simply preparing an object pgm for execution.

  Ex. a typical sequence of linkage editor commands used:

```
[INCLUDE  PLANNER (PROGLIB)
 DELETE   PROJECT  {delete from existing PLANNER}
 INCLUDE  PROJECT  (NEWLIB)  {include new version}
 REPLACE  PLANNER (PROGLIB). ]
```

- Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. This can be useful when dealing with subroutine libraries that support high level programming languages.
- Linkage editors often include variety of

other options and commands like those in linking loaders.

- Compared to linking loaders, linkage editors in general tend to offer more flexibility and control.

3.4.2 Dynamic Linking :-

- Linkage editors perform linking operations before the program is loaded for execution
- Linking loaders perform these same operations at load time.
- Dynamic linking, dynamic loading, or load on call - postpones the linking function until execution time: a subroutine is loaded and linked to the rest of the program when it is first called.
- Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library, ex. runtime support routines for a high level language like C.
- With a program that allows its user to interactively call any of the subroutines of a large mathematical and statistical library, all of the library subroutines

could potentially be needed, but only a few will actually be used in any one execution. Dynamic linking can avoid the necessity of loading the entire library for each execution except those necessary subroutines.

Fig 3.14 illustrates a method in which routines that are to be dynamically loaded must be called via an OS service request.

Fig - 3.14 (a): Instead of executing JSUB instn referring to an external symbol, the pgm makes a load-and-call service request to OS. The parameter of this request is the symbolic name of the routine to be called.

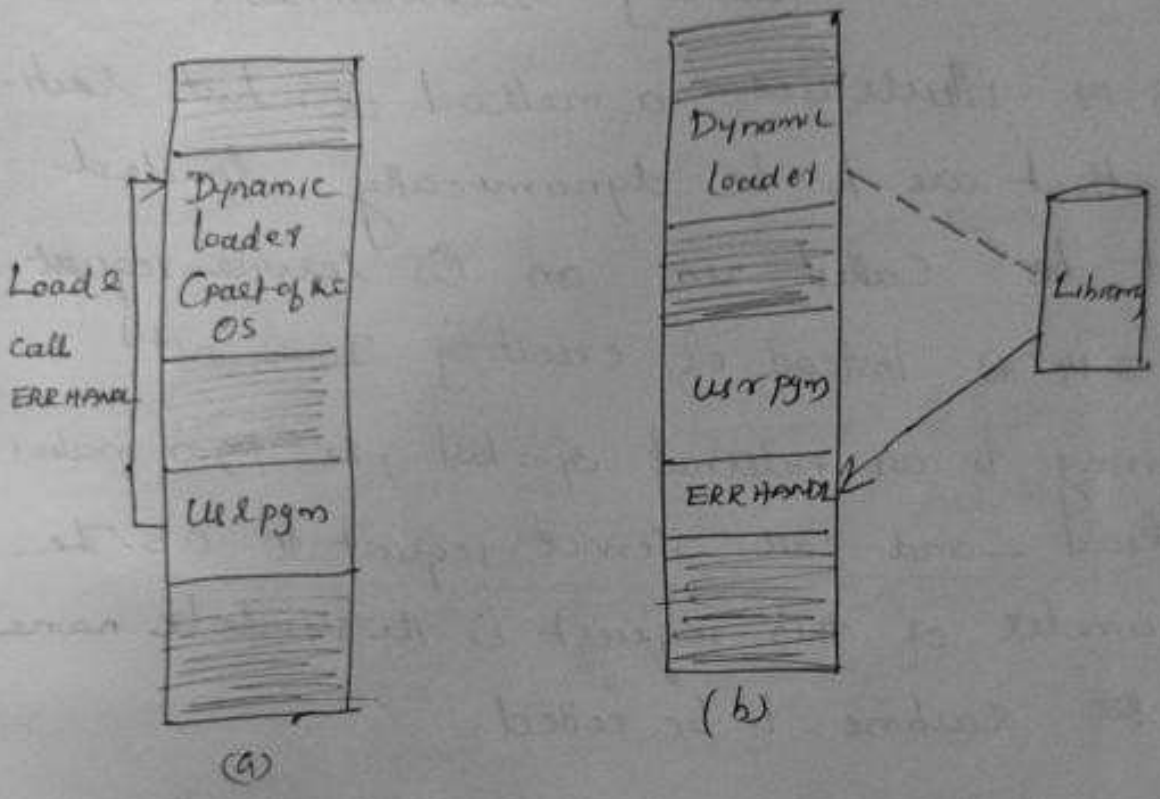
Fig - 3.14 (b) OS examines its internal tables to determine whether or not the routine is already loaded. If necessary, the routine is loaded from the specified user or system libraries.

Fig 3.14 (c) : Control is then passed from OS to the routine being called.

Fig. 3.14 (d) : When the called subroutine completes it processing, it returns to its caller (ie OS) OS then returns control to the pgm that

issued the request.

Fig. 3.14 (e) : If a subroutine is still in memory a second call to it may not require another load operation. Control may simply be passed from the dynamic loader to the called routine.



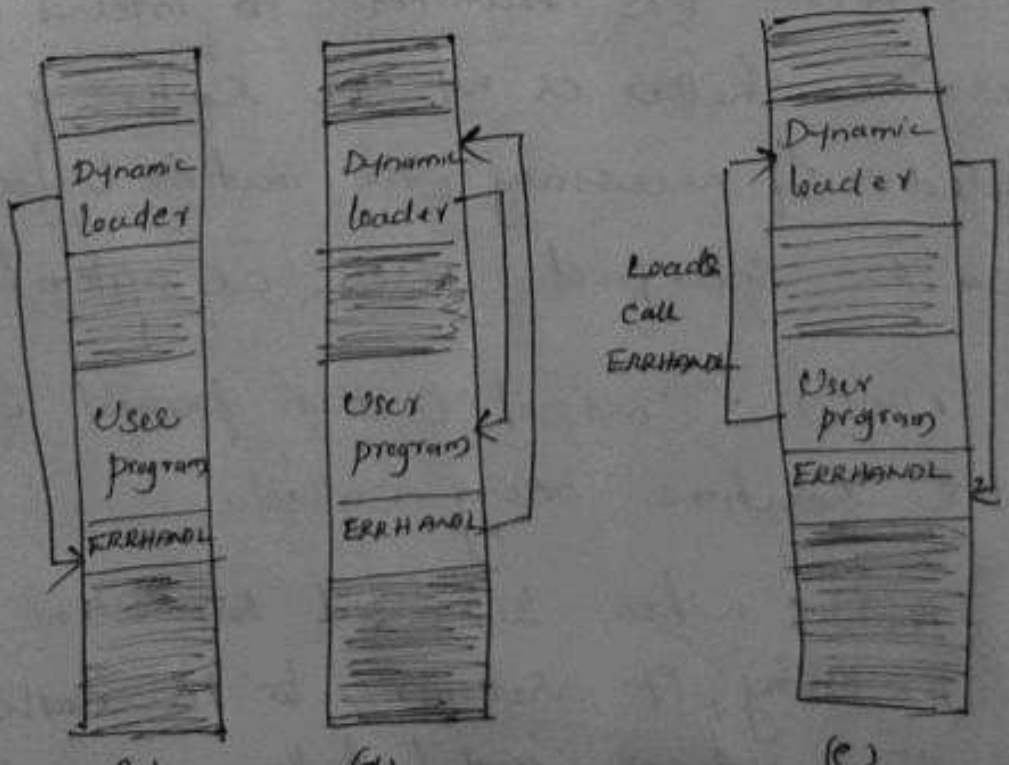


Fig. 3.14 Loading & calling of a subroutine using dynamic linking

Boot Strap loaders

- Given an idle computer with no program in memory, how do we get things started?
- On some computers, an absolute loader pgm is permanently resident in a read-only memory (ROM). When some hardware signal occurs, the machine begins to execute this ROM pgm. This is referred to as a bootstrap loader.

9/10/18

UNIT V

Macro Processor:

Macro instruction definition and expansion
One pass macroprocessor algorithm and Data Structures
Machine independent Macro processor features.
Macro processor design options:

- A macro instruction (abbreviated to macro) is simply a notational convenience for the programmer
- A macro represents a commonly used group of statements in the source programming language.

Expanding macros:-

- The macro processor replaces each macro instruction with the corresponding group of source language statements. This is called expanding the macros.

for ex. Suppose that to save the contents of all registers before calling a subprogram. On SIC/XE this would require a sequence of seven instructions (STA, STB, etc).

- Using a macro instruction, the programmer could simply write one statement like SAVEREGS

This macro instruction expanded into seven assembler language instructions needed to save the register contents.

- The design of a macro processor is not directly related to the architecture of the computer on which it is to run.
- The most common use of macro processors is in assembler language programming.
- Macro processors can also be used with high-level programming languages, OS command languages, etc.

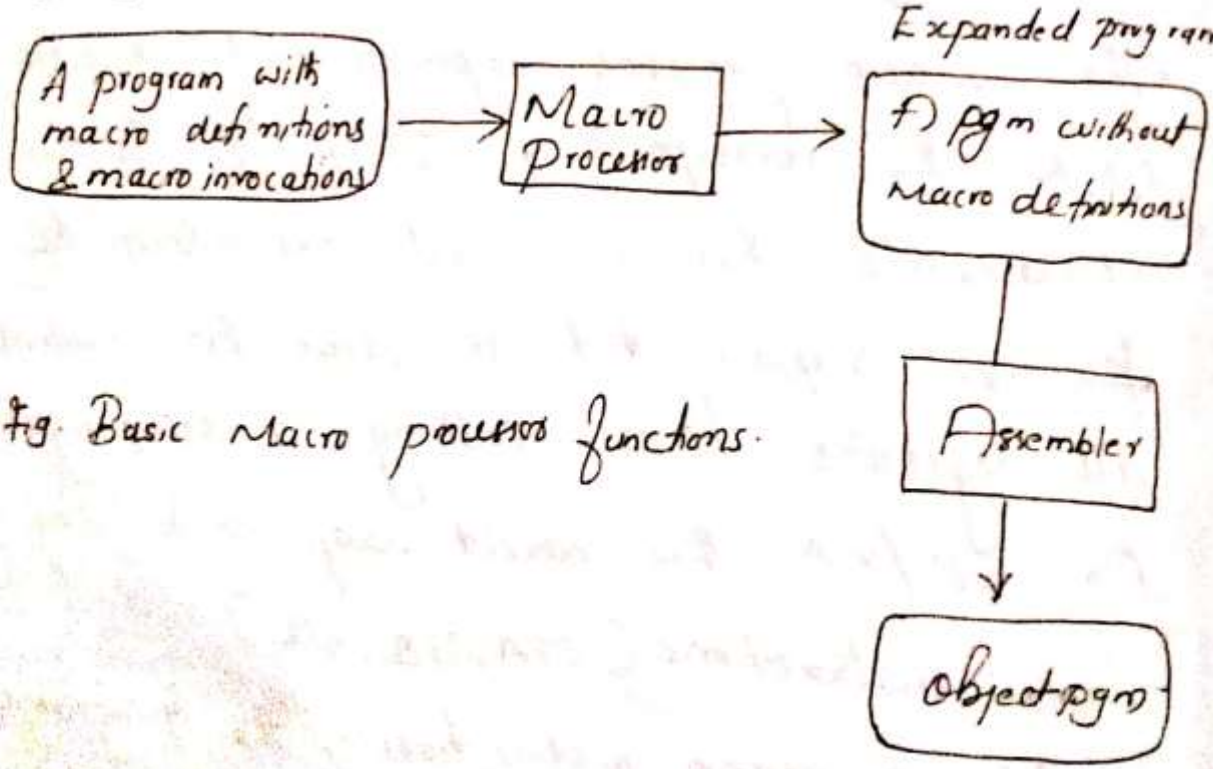


fig. Basic Macro processor functions.

4.1 BASIC MACRO PROCESSOR FUNCTIONS

- fundamental functions of that are common to all macro processors

4.1.1 → Process of macro definition, invocation and expansion with substitution of parameters
. These are illustrated with SIC/XE assembler language

4.1.2 → One pass algorithm for simple macro processors & data structures needed for macro processing

4.1.1. Macro Definition and Expansion :-

fig-4.1 shows an ex. of a SIC/XE pgm using macro instructions. This pgm defines and uses two macro instructions, RDBUFF and CRBUFF.
[RDBUFF macro is similar to the RDREC subroutine]

- The definitions of these macro instructions appear in the source pgm following the start 'START' statement.

```
Line 10 : RDBUFF  MACRO  &INDEV, &BUFFER, &RECLTH
          |       :
          MEND    MEND
```

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

- The first MACRO statement in (line 10) identifies the beginning of a MACRO definition.

The symbol in the label field (RDBUFF) is the name of the macro and entries in the operand field identifies the parameters of the macro instruction

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

```
Line 10   RDBUFF   MACRO   &INDEV, &BUFADR, &RECLTH
                          (parameters)
   15     .
   20     MACRO TO READ  RECORD INTO BUFFER
   25     :
   30     CLEAR  X
          CLEAR  A
          :
          RD  =X'&INDEV'
          COMP  A, S
          :
   95     MEND  -> End of macro definition.
```

(body)

∴ The macro name & parameters define a pattern or prototype for the macro instruction used by the programmer.

→ Following the macro directive are the statements that make up the body of the macro definition (Line 15 through 90). These are statements that will be generated as the expansion of the macro.

→ The MEND assembler directive (Line 95) marks the end of the macro definition.

→ The definition of the WRBUFF macro (lines 100 through 160) follows a similar pattern.

```
Line      Source statement
  5     COPY      START   0               COPY FILE FROM INPUT TO OUTPUT
 10     RDBUFF    MACRO   &INDEV,&BUFADR,&RECLTH
 15     .
 20     .         MACRO TO READ RECORD INTO BUFFER
 25     .
 30               CLEAR   X               CLEAR LOOP COUNTER
 35               CLEAR   A
 40               CLEAR   S
 45               +LDT    #4096           GET MAXIMUM RECORD LENGTH
 50               TD      =X'&INDEV'      TEST INPUT DEVICE
 55               JEQ     *-3             LOOP UNTIL READY
 60               RD      =X'&INDEV'      READ CHARACTER INTO REG A
 65               COMPR   A,S             TEST FOR END OF RECORD
 70               JEQ     *+11            EXIT LOOP IF EOR
 75               STCH    &BUFADR,X       STORE CHARACTER IN BUFFER
 80               TIXR    T               LOOP UNLESS MAXIMUM LENGTH
 85               JLT     *-19              HAS BEEN REACHED
 90               STX     &RECLTH         SAVE RECORD LENGTH
 95               MEND
100     WRBUFF    MACRO   &OUTDEV,&BUFADR,&RECLTH
105     .
110     .         MACRO TO WRITE RECORD FROM BUFFER
115     .
120               CLEAR   X               CLEAR LOOP COUNTER
125               LDT     &RECLTH
130               LDCH    &BUFADR,X       GET CHARACTER FROM BUFFER
135               TD      =X'&OUTDEV'     TEST OUTPUT DEVICE
140               JEQ     *-3             LOOP UNTIL READY
145               WD      =X'&OUTDEV'     WRITE CHARACTER
150               TIXR    T               LOOP UNTIL ALL CHARACTERS
155               JLT     *-14              HAVE BEEN WRITTEN
160               MEND
165     .
170     .         MAIN PROGRAM
175     .
180     FIRST     STL     RETADR          SAVE RETURN ADDRESS
190     CLOOP     RDBUFF  F1,BUFFER,LENGTH  READ RECORD INTO BUFFER
195               LDA     LENGTH          TEST FOR END OF FILE
200               COMP    #0
205               JEQ     ENDFIL          EXIT IF EOF FOUND
210               WRBUFF  05,BUFFER,LENGTH  WRITE OUTPUT RECORD
215               J       CLOOP           LOOP
220     ENDFIL    WRBUFF  05,EOF,THREE    INSERT EOF MARKER
225               J       @RETADR
230     EOF       BYTE    C'EOF'
235     THREE     WORD    3
240     RETADR    RESW    1
245     LENGTH    RESW    1               LENGTH OF RECORD
250     BUFFER    RESB    4096            4096-BYTE BUFFER AREA
255               END     FIRST
```

Figure 4.1 Use of macros in a SIC/XE program.

```
Line        Source statement

5     COPY     START    0                  COPY FILE FROM INPUT TO OUTPUT
10    FIRST    STL      RETADR             SAVE RETURN ADDRESS
15    .CLOOP   RDBUFF   F1,BUFFER,LENGTH   READ RECORD INTO BUFFER
15a   CLOOP    CLEAR    X                  CLEAR LOOP COUNTER
15b            CLEAR    A
15c            CLEAR    S
15d            +LDT     #4096              SET MAXIMUM RECORD LENGTH
15e            TD       =X'F1'             TEST INPUT DEVICE
15f            JEQ      *-3                LOOP UNTIL READY
15g            RD       =X'F1'             READ CHARACTER INTO REG A
15h            COMPR    A,S                TEST FOR END OF RECORD
15i            JEQ      *+11               EXIT LOOP IF EOR
15j            STCH     BUFFER,X           STORE CHARACTER IN BUFFER
15k            TIXR     T                  LOOP UNLESS MAXIMUM LENGTH
15l            JLT      *-19                 HAS BEEN REACHED
15m            STX      LENGTH             SAVE RECORD LENGTH
20             LDA      LENGTH             TEST FOR END OF FILE
25             COMP     #0
30             JEQ      ENDFIL             EXIT IF EOF FOUND
35             WRBUFF   05,BUFFER,LENGTH   WRITE OUTPUT RECORD
35a            CLEAR    X                  CLEAR LOOP COUNTER
35b            LDT      LENGTH
35c            LDCH     BUFFER,X           GET CHARACTER FROM BUFFER
35d            TD       =X'05'             TEST OUTPUT DEVICE
35e            JEQ      *-3                LOOP UNTIL READY
35f            WD       =X'05'             WRITE CHARACTER
35g            TIXR     T                  LOOP UNTIL ALL CHARACTERS
35h            JLT      *-14                 HAVE BEEN WRITTEN
40             J        CLOOP              LOOP
45    .ENDFIL  WRBUFF   05,EOF,THREE       INSERT EOF MARKER
45a   ENDFIL   CLEAR    X                  CLEAR LOOP COUNTER
45b            LDT      THREE
45c            LDCH     EOF,X              GET CHARACTER FROM BUFFER
45d            TD       =X'05'             TEST OUTPUT DEVICE
45e            JEQ      *-3                LOOP UNTIL READY
45f            WD       =X'05'             WRITE CHARACTER
45g            TIXR     T                  LOOP UNTIL ALL CHARACTERS
45h            JLT      *-14                 HAVE BEEN WRITTEN
50             J        @RETADR
55    EOF      BYTE     C'EOF'
60    THREE    WORD     3
65    RETADR   RESW     1
70    LENGTH   RESW     1                  LENGTH OF RECORD
75    BUFFER   RESB     4096               4096-BYTE BUFFER AREA
80             END      FIRST
```

Figure 4.2 Program from Fig. 4.1 with macros expanded

- The main program begins on line 180. The statement on line 190 is a macro invocation statement that gives the name of the macro instruction being invoked and the arguments to be used in expanding the macro.
- A macro invocation statement is often referred to as a macro call.

line 190 CLOOP RDBUFF F1, BUFFER, LENGTH

↓ name of macro

↓ arguments

The pgm in fig. 4.1 supplied as i/p to a macro processor. Fig 4.2 shows the output that would be generated.

- The macro instruction definitions have been deleted since they are no longer needed after the macros are expanded

- Each macro invocation statement have been expanded into the statements that forms the body of the macro, with the arguments from the macro invocation substituted for the parameters in the macro prototype.
- The arguments and parameters are associated with one another according to their positions.

- The first argument in the macro invocation corresponds to the first parameter in the macro prototype, & soon
- In expanding RDBUFF F1, BUFFER, LENGTH
  F1 is substituted for &INDEV
  BUFFER is " " &BUFADR
  LENGTH is " " &RECLTH

By Hp

In fig 4.2
Lines 190 a through 190m show the complete expansion of the macro invocation on line 190 (fig 4.1)

- The comment lines with the macro body (i.e macro to read RECORD INTO BUFFER (4.1)] have been deleted, but comments on individual statements retained.
- The label on the macro invocation stmt (CLOOP) has been retained as a label on the first stmt generated in the macro expansion. This allows the programmer to use a macro instruction in the program exactly the same way as an assembler language mnemonic.
- After macro processing the expanded file (4.2) can be used as input to the assembler. The macro invocation statement will be treated as comments (because, these statements need not be assembled).

Macro defn

```
eg.  WRBUFF  MACRO  &OUTDEV, &BUFADR, &RECLTH
         CLEAR X
         LDT  &RECLTH
         LDCH  &BUFADR
         TD   =X'&OUTDEV'
         JEQ  *-3
         WD   =X'&OUTDEV'
         TIXR T
         JLT  *-14
```

Macro call :-

- CURBUFF 05 , BUFFER , LENGTH

On Expanding

```
CLEAR  X
LDT    LENGTH
LDCH   BUFFER,X
TD     =X '05'
JEQ    *-3
WD     =X '05'
TIXR   T
JLT    *-14
```

Macro Processor Algorithm & Data Structures

It is easy to design two pass macro processor

- Pass 1:
  - All macro definitions are processed

- Pass 2:
  - All macro invocation statements are expanded

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

For ex. the two macro instruction definitions in fig 4.3. The body of the first macro (RDBUFF) contains statements that define RDBUFF, WRBUFF and other macro instructions for a SIC system.

- The body of the second macro instruction (MACROX) define these same macro for a SIC/XE s/m
- A pgm could run on a standard SIC s/m could invoke MACROS to define other utility macro instns
- A pgm for a SIC/XE s/m could invoke MACROX to define these same macros in IBM XE versions
- Same pgm could run on either a standard SIC machine or a SIC/XE machine

→ Defining MACROS or MACROX does not define RDBUFF (or) other macro instructions. These definitions are processed only when an invocation of MACROS or MACROX is expanded

```
1   MACROS    MACRO  { Defines SIC Std Version Macros}
2   RDBUFF    MACRO  &INDEV, &BUFFADR, &RECLTH
                 .
                 .
                 .
3                MEND   { End of RDBUFF}
4   WRBUFF    MACRO  &OUTDEV, &BUFFADR, &RECLTH
                 .
                 .
                 .
5                MEND   { End of WRBUFF}
                 .
                 .
                 .
6                MEND   { End of MACROS}
```

(a)

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

(b)

Fig. 4.3 Ex. of the definition of macros within a macro body.

* Sub-macro definitions are only processed when an Invocation of their Super-macros are expanded.

⟹ A <u>one pass macro</u> processor - that can alternate between macro definition and macro expansion is able to handle macros like in fig 4.3

- In one pass structure, the definition of a macro must appear in the source pgm before any statements that invoke that macro.

3 Data Structures for the macroprocessor.

- DEFTAB (Definition table)
- NAMTAB (Name table)
- ARGTAB (Argument table).

- The macro definitions (the statements that make up the macro definition table (DEFTAB), which contains the macro prototype and the statements that make up the macro body (with a few modifications)
- Comment lines from the macro definitions are not entered into DEFTAB because they will not be part of the macro expansion.
- References to the macro instruction parameters are converted to a positional notation for efficiency in substituting arguments.

⇒ The macro names are also entered in a NAMTAB, which serves as an index to DEFTAB. For each macro instruction defined, NAMTAB contains pointers to the beginning and end of the definition in DEFTAB

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

Fig 4.4 shows the portions of the contents of these tables during the processing of the program.

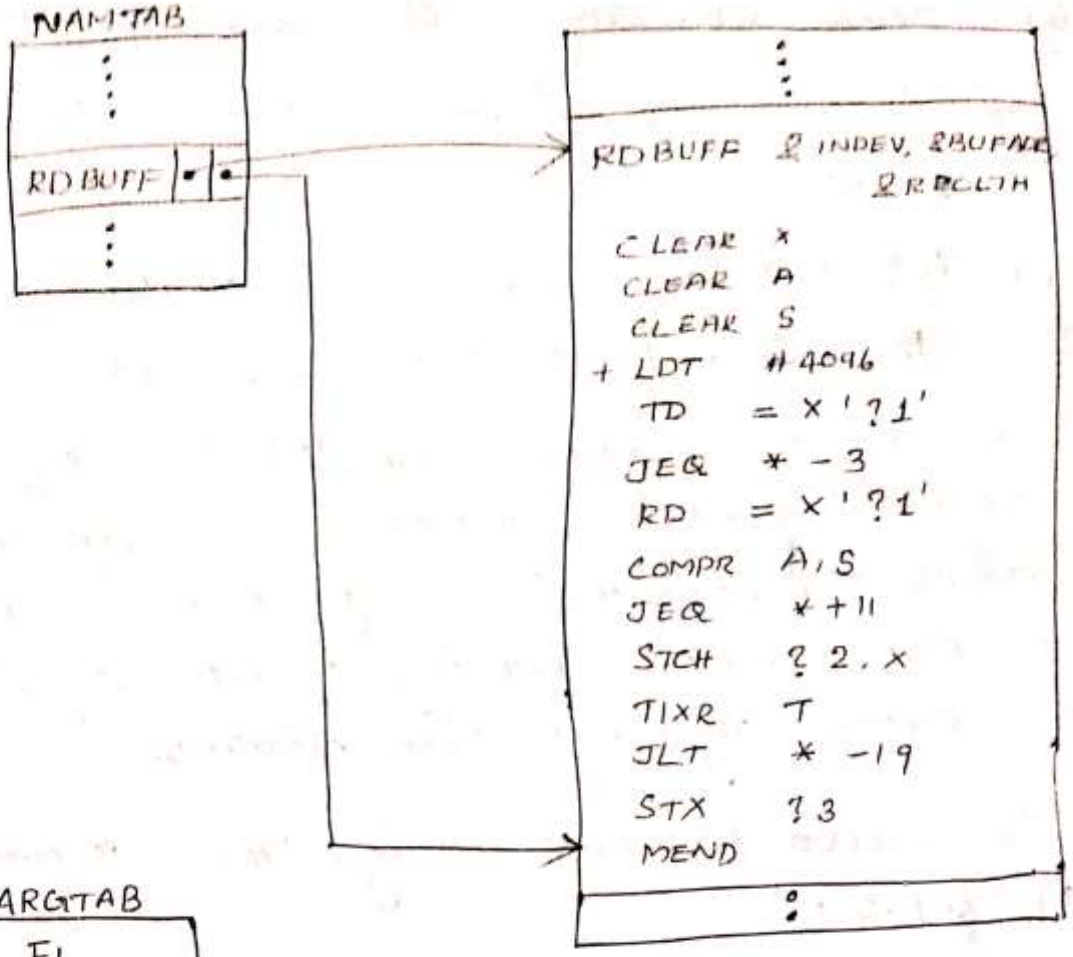


Fig. 4.4 contents of macro processor tables for the pgm in fig 4.1 (a) entries in NAMTAB, and DEFTAB defining macro RDBUFF, (b) entries in ARGTAB for invocation of RDBUFF on line 190.

- fig (a) shows the definition of RDBUFF stored in DEFTAB, with an entry in NAMTAB identifying the beginning and end of the definition.
- The positional notation that has been used for the parameters: the parameter &INDEV has been converted to ?1 (indicating the first parameter in the prototype).

&BUFADR is converted to ?2 (Second parameter)

&RECLTH " ?3 (third parameter)

fig (4.6) shows ARGTAB if it would appear during expansion of the RDBUFF stmt on line 190.

For this invocation, the first argument is ?1, the second is BUFFER, etc.

This scheme makes substitution of macro arguments much more efficient.

- When the ?n notation is recognized as a line from DEFTAB, a simple indexing operation supplies the proper argument from ARGTAB.
- The macro processor algorithm is presented in fig 4.5
- The procedure DEFINE, which is called when the beginning of a macro defn is recognized, makes the appropriate entries in DEFTAB & NAMTAB.
- EXPAND is called to setup the argument values in ARGTAB and expand a macro invocation stmt. The procedure GETLINE which is 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 being expanded), or from the i/p file, depending upon whether a Boolean variable EXPANDING is set to TRUE or FALSE.

Fig 4.5 Algorithm for a one-pass macro processor

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


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

Figure 4.5 Algorithm for a one-pass macro processor.

```
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}
```

Big Top

```
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}
```

Figure 4.5 (*cont'd*)

This algorithm's: the handling of macro definitions within macros (in fig 4.3). when a macro def^n is entered into a DEFTAB, the normal approach would be to continue until an MEND. directive is reached.

The MEND on line 3 (which actually marks the end of the definition of RDBUFF). would be taken as the end of the definition of MACROS.

To solve this problem, our 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. This process is very much like matching left and right parenthesis when scanning an arithmetic expression.

- Most macro processors allow the definitions of commonly used macro instructions to appear in a standard system library, rather than in the source pgm. This makes the use of such macros much more convenient. Definitions are retrieved from this library as they are needed during macro processing.

21/11/20

4.2 MACHINE INDEPENDENT MACRO PROCESSOR FEATURES.

The design of macro processor doesnot depend on the architecture of the machine.
- This section explains some extended features for the macro processor. These features are:

- Concatenation of Macro parameters
- Generation of unique labels
- Conditional macro expansion
- keyword macro parameters.

4.2.1 concatenation of macro parameters

Most macro processors allows parameters to be concatenated with other character strings

- Suppose that, for ex, a pgm contains a series of variables named by the symbols XA1, XA2, XA3, ... another series named by XB1, XB2, XB3 ... etc. If similar processing Ps to be performed on each series of variables, the programmer put this as a macro instruction. The parameter to such macro instruction would specify the series of variables to be operated on (A, B, etc). The macro processor would use this parameter to construct the symbols required in the macro expansion (XA1, XB1, etc).

— Suppose that the parameter to such macro instruction is named &ID. The body of the macro defn contain a stmt like

LDA X&ID1

In which parameter ID is concatenated after the character string X and before the character string 1.

- problem with this statement is, the beginning of the macro parameter is identified by the starting symbol & ; however, the end of the parameter is not marked.

- The operand in the first represent the character string X followed by the parameter &ID1.

- If the parameter macro definition contained both &ID and &ID1 as parameter the situation would be unavoidably ambiguous

→ Most macro processors deal with this problem by providing a special <u>concatenation operator</u>
In the SIC macro language, this operator is the character →. Thus the previous stmt written as

```
LDA   X&ID→1
```

So that the end of the parameter &ID is clearly identified. The macro processor deletes all occurrences of the concatenation operator immediately after performing parameter substitution so the character → will not appear in the macro expansion.

Fig 4.6 (a) shows macro defn that uses the concatenation operator. 4.6 (b) & (c) shows macro invocations. Stmts and corresponding macro expansion

```
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
```

(4.6) (a)

SUM A

↓

LDA XA1
ADD XA2
ADD XA3
STA XA5

4(b)

SUM BETA

↓

LDA XBETA1
ADD XBETA2
ADD XBETA3
STA XBETA5

4.(c)

fig Concatenation of macro parameters.

4.2.2 Generation of Unique Labels

Consider the definition of WRBUFF in fig 4.1. If a label were placed on the TD instruction on line 135, this label would be defined twice - for each invocation of WRBUFF. This duplicate definition would prevent correct assembly of the resulting expanded pgm.

```
ii   WRBUFF  MACRO  &OUTDEV, &BUFFADR, &RECLTH
     CLOOP
@CLOOP  TD   =X'&OUTDEV'
        JEQ  CLOOP
```

In fig 4.1, the WRBUFF is called twice. Thus this statement is expanded twice, one for each invocation of WRBUFF.

```
ii  1) WRBUFF  05, BUFFER, LENGTH
       CLOOP   TD  =X'05'
               JEQ  CLOOP

   2)  WRBUFF  05, EOF, THREE
       CLOOP   TD  =X'05'
               JEQ  CLOOP      ; expanded stmts.
```

Fig - 4.7 illustrates one technique for generating unique labels within a macro expansion. A definition of the RDBUFF macro is shown in fig 4.7 (a). Labels used within the macro body begin with a special character $.

Fig. 4.7 (b) shows a macro invocation statement & the resulting macro expansion.

```
4(a)  RDBUFF  MACRO   &INDEV, &BUFADR, &RECLTH
              CLEAR   X
              :
      $LOOP   TD      =X'&INDEV'
              :
              JLT     $LOOP

4(b)  RDBUFF  F1, BUFFER, LENGTH
      CLEAR   X
      :
$AALOOP TD    =X'F1'
      JEQ     $AALOOP
```

Fig Generation of unique labels within macro expansion

Each symbol beginning with $ has been modified by replacing $ with $AA. More generally, the character $ will be replaced by $XX, where XX is a two character alphanumeric counter of the number of macro instructions expanded. For the first macro expansion in a pgm, XX will have the value AA.
- For succeeding macro expansions, XX will be set to AB, AC, etc.

Biju Kp

(If only alphabetic and numeric characters are allowed in XX, such a two character counter provides for as many as 1296 macro expansions in a single pgm.

This results in the generation of unique labels for each expansion of a macro instruction.

4.2.3 Conditional Macro Expansion:-

Most macro processors, modify the sequence of statements generated for a macro expansion, depending on the arguments supplied in the macro invocation.

- Such a capability adds greatly to the power & flexibility of a macro language.
- This section present a typical set of conditional macro expansion statements.

- The use of one type of conditional macro expansion statement is illustrated in fig 4.8. fig. 4.8(a) shows definitions of a macro RDBUFF.

This definition of RDBUFF has two additional parameters &EOR, which specifies a hexadecimal character code that marks the end of a record. & &MAXLTH, which specifies the maximum length record that can be read.

```
25  RDBUFF   MACRO   &INDEV, &BUFFADR, &RECLTH,
                                  &EOR, &MAXLTH
26           IF      (&EOR NE '')
27  &EORCK   SET     1
28           ENDIF
30           CLEAR   X
35           CLEAR   A
38           IF      (&EORCK EQ 1)
40           LDCH    =X'&EOR'
42           RMO     A,S
43           ENDIF
44           IF      (&MAXLTH EQ '')
45           +LDT    #4096
46           ELSE
47           +LDT    #&MAXLTH
48           ENDIF
50  $LOOP    TD      =X'&INDEV'
55           JEQ     $LOOP
60           RD      =X'&INDEV'
63           IF      (&EORCK EQ 1)
65           COMPR   A,S
70           JEQ     $EXIT
73           ENDIF
75           STCH    &BUFFADR,X
80           TIXR    T
85           JLT     $LOOP
90  $EXIT    STX     &RECLTH
95           MEND
```

4.8 (a)

The statement on lines 44 through 45 of this defn illustrates a simple macro-time conditional structure. The IF statement evaluates a Boolean expression that is its operand.

If the value of this expression is TRUE, the statements following the IF are generated until an ELSE is encountered. Otherwise these statements are skipped, & the statements following the ELSE are generated

- The ENDIF statement terminates the conditional expression that was begun by the IF statement.
- Thus if the parameter &MAXLTH is equal to the null string, the statement on line 45 is generated. Otherwise the stmt on line 47 is generated.

⇒ f) Similar structure appears on lines 26 through 28. Here, another macro processor directive (SET) is used. This SET stmt assigns the value 1 to &EORCK. The symbol &EORCK is a macro time variable (also called a set symbol), which can be used to store working values during the macro expansion.

- Any symbol that begins with the character & and that is not a macro instruction parameter is assumed to be a macro time variable. All such variables are initialized to a value of 0.

Fig - 4.8(b-d) shows the expansion of three different macro invocation statements that illustrate the operation of the IF statements in fig(4.8(a))

- The implementation of the conditional macro expansion is that the macro processor must contain a symbol table that contains

the values of all macro time variables used. The table is used to 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

- When an IF statement is encountered during the expansion of a macro, the specified Boolean expression evaluated. If the value of this expn is TRUE, the macro processor continues to process lines from DEFTAB until it encounters the next ELSE or ENDIF stmt. 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 the value of the specified Boolean expression is FALSE, the macro processor skips ahead in DEFTAB until it finds the next ELSE or ENDIF stmt. The macro processor then resumes the normal macro expansion.

- It is extremely important to understand that the testing of Boolean expression in IF stmts occurs at the time macros are expanded. By the time the pgm is assembled, all such decisions have been made. The Macro-time IF-ELSE-ENDIF structure provides a mechanism for either generating or skipping selected

Statements in the macro body

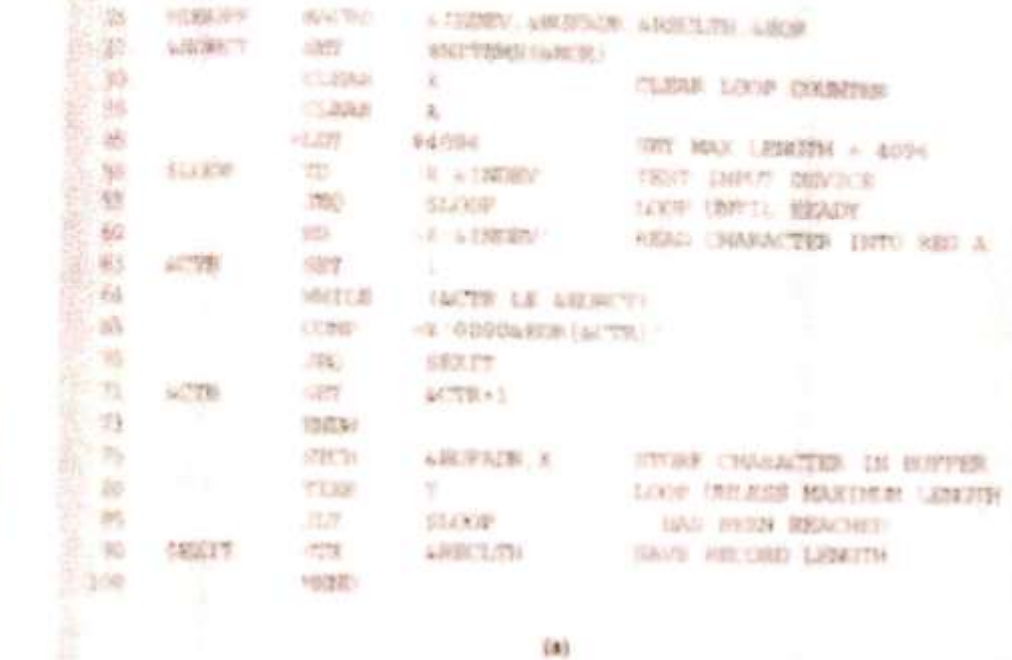
fig 4.8 (b) - Expanding the macro from 4.

```
RDBUFF  F3 , BUF,REL , 04, 2048
        CLEAR   X
        CLEAR   A
        LDCH    =X '04'
        RMO     A, S
        +LDT    #2048
$AALoop TD      =X'F3'
        JEQ     $AALoop
        RD      =X'F3'
        COMPR   A, S
        JEQ     $AA EXIT
        STCH    BUF, X
        TIXR    T
        JLT     $AALOOP
$AAEXIT STX     REC L.
        MEND.
```

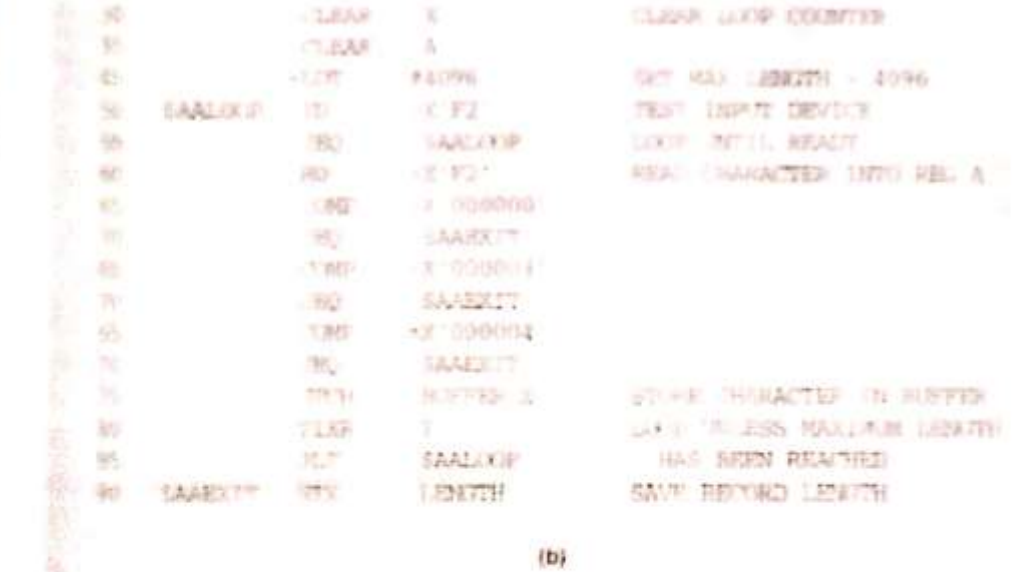
— A different type of conditional macro expansion statement is illustrated in fig 4.9. Fig 4.9 (a) shows another definition of RDBUFF. The purpose & fun. of the macro are the same before.

— With this definition, the programmer can specify a list of end-of-record characters. In the macro invocation stmt in fig 4.9(b), there is a list (00,03,04) corresponding to the parameter &EOR. Any one of these characters interpreted as marking the end of a record.

— The max. record length always 4096.



(a)



(b)

Figure 4.9 Use of macro-time looping statements

The defn in fig 4.9(a) uses a macro time looping statement WHILE. The WHILE stmt specifies that the following lines, upto the next ENDW stmt, are to be generated repeatedly as long as particular condition is true. The both testing of this condition and the looping are done while the macro is being expanded. The conditions to be tested involve macro time variables and arguments.

The use of the WHILE - ENDW structure is illustrated on lines 64 through 73. of fig 4.9(a) The macro time variable &EORCT has previously been set (line 27) to the value %NITEMS (&EOR). %NITEMS is a macro processor function that returns as its value number of members in an argument list.
For ex. If the argument corresponding to &EOR is (00, 03, 04), then %NITEMS (&EOR) has the value 3.

- The macro-time variable &CTR is used to count the no: of times the lines following the WHILE stmt have been generated. The value of &CTR is initialized to 1 (line 63) & incremented by 1 each time through the loop (line 71) The WHILE stmt itself 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.

- The 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 & so on.

- The implementation of a macro-time looping stmt: when a WHILE statement is encountered during macro expansion, the specified Boolean expression is evaluated. If the value of this expression is FALSE the macroprocessor skips ahead in DEFTAB until it finds the next ENDW statement, & then resumes normal macro expansion. If the value of the Boolean expression is TRUE, the macro processor continues to process lines from DEFTAB in the usual way until the next ENDW stmt. When the ENDW is encountered, the macro processor returns to the preceding WHILE, re-evaluates the Boolean expression and takes action based on the new value of this expression.

Keyword Macro parameters

- Parameters and arguments are associated with each other according to their positions in the macro prototype and the macro invocation statement. With positional parameters, the programmer must be careful to specify the arguments in the proper order. If an argument is omitted, the macro invocation statement must contain a null argument (two consecutive commas) to maintain the correct argument positions.

- Positional parameters are quite suitable for most macro instructions. However, if a macro has a large no. of parameters, and only a few of these are given values in a typical invocation, a different form of parameter specification is more useful.

- For ex, Suppose that a certain macro instruction GENER has 10 possible parameters but in a particular invocation of the macro only the third & ninth parameters are to be specified. If positional parameters were used, the macro invocation statement would be

GENER ,, DIRECT,,,,,,3

Using a different form of parameter specification called Keyword parameters.

each argument value is written with a keyword that names the corresponding parameter. Arguments may appear in any order. If the third parameter in the previous ex, is named &TYPE and the fourth parameter is named &CHANNEL, the macro invocation stmt would be

GENER TYPE =DIRECT, CHANNEL=3

This stmt is easier to read, & much less error prone than the positional version

Eg:

→ RDBUFF MACRO &INDEV=F1, &BUFADR=, &RECLTH=, &REQR=4, &MAXLTH=4096

[Fig-4.10a]

In the above macro defn, each parameter name is followed by an equal sign, which identifies a keyword parameter. After the equal sign, a default value is specified for some of parameters (&INDEV=F1,). This parameter is assumed to have this default value if its name doesnot appear in the macro invocation statement. There is no default value for the parameter &BUFADR

for the macrocall

RDBUFF BUFADR=BUFFER, RECLTH=LENGTH

CLEAR X
CLEAR A
:

(Fig 4.10b)

Fig 4.10(b) all the default values are accepted. Fig(c) the value of RNDEV is specified as F3, & the value of REOR is specified as ß null. These values overrides the corresponding defaults.

The arguments may appear in any order in the macro invocation statement.

Fig 4.10.c

```
RDBUFF    BUFLTH = LENGTH, BUFADR = BUFFER,
                  EOR = , RNDEV = F3.

          CLEAR X
          CLEAR A
            :
            :
EXIT  STX  LENGTH.
```

4.3 MACRO PROCESSOR DESIGN OPTIONS

- Major design options for a macro processor.
- One pass macro processor algorithm doesnot work properly if a macro invocation statement appears with in the body of the macro instruction.

4.3.1 Examines the problems created by such macro invocation statements and possibilities for the solution of these problems.

4.3.2 - general purpose macro processors are not tied to any particular language

4.3.1 Recursive Macro Expansion

Fig. 4.11 shows an example of - invocation of one macro by another.

- It - There are chances where a macro invocation statement appears with in the body of macro instruction ie invocation of one macro by other macro.

Ex. 4.11 Example of of nested macro invocation.

```
RDBUFF   MACRO    &BUFADR, &RECLTH, &INDEV
         MACRO TO READ RECORD INTO BUFFER
         CLEAR    X
         CLEAR    A
         CLEAR    S
       + LDT      #4096
$LOOP    RDCHAR   &INDEV
         COMP     A,S
         JEQ      $EXIT
         STCH     &BUFFER, X
         TIXR     T
         JLT      $LOOP
$EXIT    STX      &RECLTH
         MEND.
```

(a)

```
RDCHAR   MACRO    &IN
         MACRO TO READ CHARACTER INTO REGISTER A
         TD       =X '&IN'
         JEQ      *-3
         RD       =X '&IN'
         MEND
```

(b)

RDBUFF BUFFER, LENGTH, F1 → when a such macro

invocation statement is encountered, the procedure EXPAND would be called. The arguments from the macro invocation would be entered in ARGTAB as follows:

parameter	value
1	BUFFER
2	LENGTH
3	F1
4	(UNUSED)
:	:

The boolean variable 'Expanding' would be set to TRUE and the expansion of macro 'RDBUFF' would begin. The processing would proceed normally until the statement 'RDCHAR &INDEV', when the macro RDCHAR is called. At that point, PROCESSLINE would call EXPAND again. This time, ARGTAB look like

parameter	value
1	F1
2	(unused)
.	.

The expansion of RDCHAR would also proceed normally. At the end of this definition of RDCHAR was recognized, EXPANDING would be set to FALSE. Thus the macro processor would "forget" that it had been in the middle of expanding a macro ~~process~~ when it

encountered the RDCHAR statement. In addition the arguments from the original macro invocation (RDBUFF) would be lost because the values in PREVTHIS were overwritten with the arguments from the invocation of RDCHAR.

When the RDBUFF macro invocation is encountered EXPAND is called. Later it calls PROCESSLINE for RDCHAR READEV, which results in another call to EXPAND before a return is made from the original call. These problems solve the macro processor being written in a programming language that allows recursive calls.

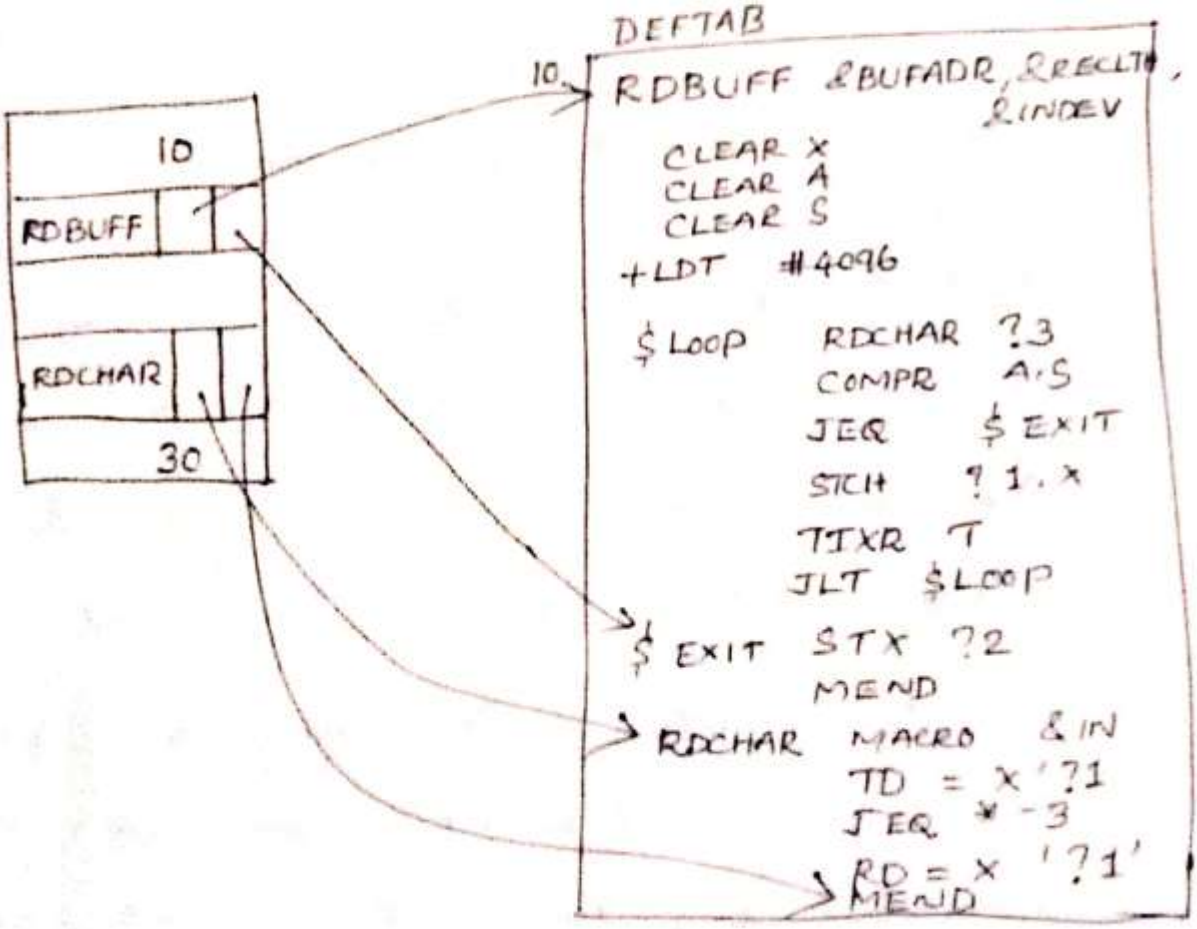


Fig 4.12.

Raju Kp

General purpose Macro processors

- A general purpose macro processors are not dependent on any particular programming language, but can be used with a variety of different languages.

Advantages

1) The programmer doesn't need to learn about a different macro facility for each compiler or assembly language.

2) The costs involved in producing a general purpose macro process are somewhat greater than those for developing a language specific processor. However this expense doesn't need to be repeated for each language.

- A general purpose facility must provide some way for a user to define the specific set of rules to be followed.

• Comments should usually be ignored by a macro processor. However each programming language has its own method for identifying comments.

next is related to the grouping of together terms, expressions or stmts. Some A general purpose macroprocessor do these grouping by scanning source statements

- Some languages use keywords such as begin and end for grouping statements.
- others use special characters such as { and }

- A more general problem involves, the token of the programming languages. for ex. identifiers, constants, operators & keywords.

Languages differ in the restrictions on the length of identifiers and rules for the formation of constants '**' in FORTRAN may be treated by a macroprocessor as two seperate characters rather than as a single operator.

- Another problem involves the syntax used for macro definitions & macro invocation statements.

Macro processing within Language Translators

Preprocessor: They process macro definitions and expand macro invocations producing an expanded version of the source program. This expanded program is then used as i/p to an assembler or compiler.

The macro procesing functions are combined with in the language translators (assembler) itself. The simplest method of achieving this sort of combination is a line-by-line macro processor.

Using this approach, the macro processor reads the source program statements and perform all of its functions. However, the o/p lines are passed to the language translator as they are generated, one at a time, instead of being written to an expanded Source source file.

This line by line approach has several advantages.

1) It avoids making an extra pass over the source pgm, thus more efficient

than using a macro preprocessor.

2) Some of the data structures used by macro preprocessor & language translator can be combined Eg OPTAB in assembler and NAMTAB in the macroprocessor would be implemented in same table.

Disadvantages

1) They must be specifically designed & written to work with a particular implementation of an assembler or compiler.

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

Device Drivers :-
Anatomy of a device driver, Character and

MODULE VI

Device Drivers :-
Anatomy of a device driver, Character and block device drivers, General design of device drivers.

Text Editors :-
Overview of editing, User Interface, Editor Structure,

Debuggers :-
Debugging functions and capabilities, Relationship with other parts of the system, Debugging methods - By induction, Deduction and backtracking.

DEVICE DRIVERS:

A device driver is a program that controls a particular type of device that is attached to your computer. There are device drivers for printers displays, CDROM readers, diskette drives, and so on. When you buy an operating system many device drivers are built in to the product.

A device driver is a particular form of S/W application that is designated to enable interaction with H/W devices. Without the required device driver, the corresponding H/W device fails to work.

Biji Kp

A device driver is a particular form of S/W application that is designated to enable interaction with H/W devices. Without the required device driver, the corresponding H/W device fails to work.

A device driver usually communicates with the h/w by means of the communication subsystem or computer bus to which the h/w is connected. Device drivers are operating s/m specific and h/w dependent.

- A device driver acts as a translator b/w the h/w device and the pgms or operating system that use it.
- The sole purpose of a device driver is to instruct a computer on how to communicate with the I/P & O/P device by translating OS's I/O instructions in to a language that a device can understand.
  - There are various types of device drivers for I/O devices such as keyboards, mice, CD/DVD drives, controllers, printers, graphics cards & ports.
  - It is essential that a computer have the correct device drivers for all its parts to keep the s/m running safely and efficiently When first turning on a computer, the OS works with the device drivers and the basic I/O s/m (OS) to perform h/w tasks. Without a device driver the OS would not be able to communicate with the I/O devices

- Not only physical h/w devices rely on a device driver to function, but s/w components do as well.
- Most programs access devices by using general commands, the device driver translates the language in to specialized commands for the device.

- Devices are, in general, complex and hard to use. Devices are controlled by communicating with device controllers. through device controller registers, or sending them commands in the format of a msg.

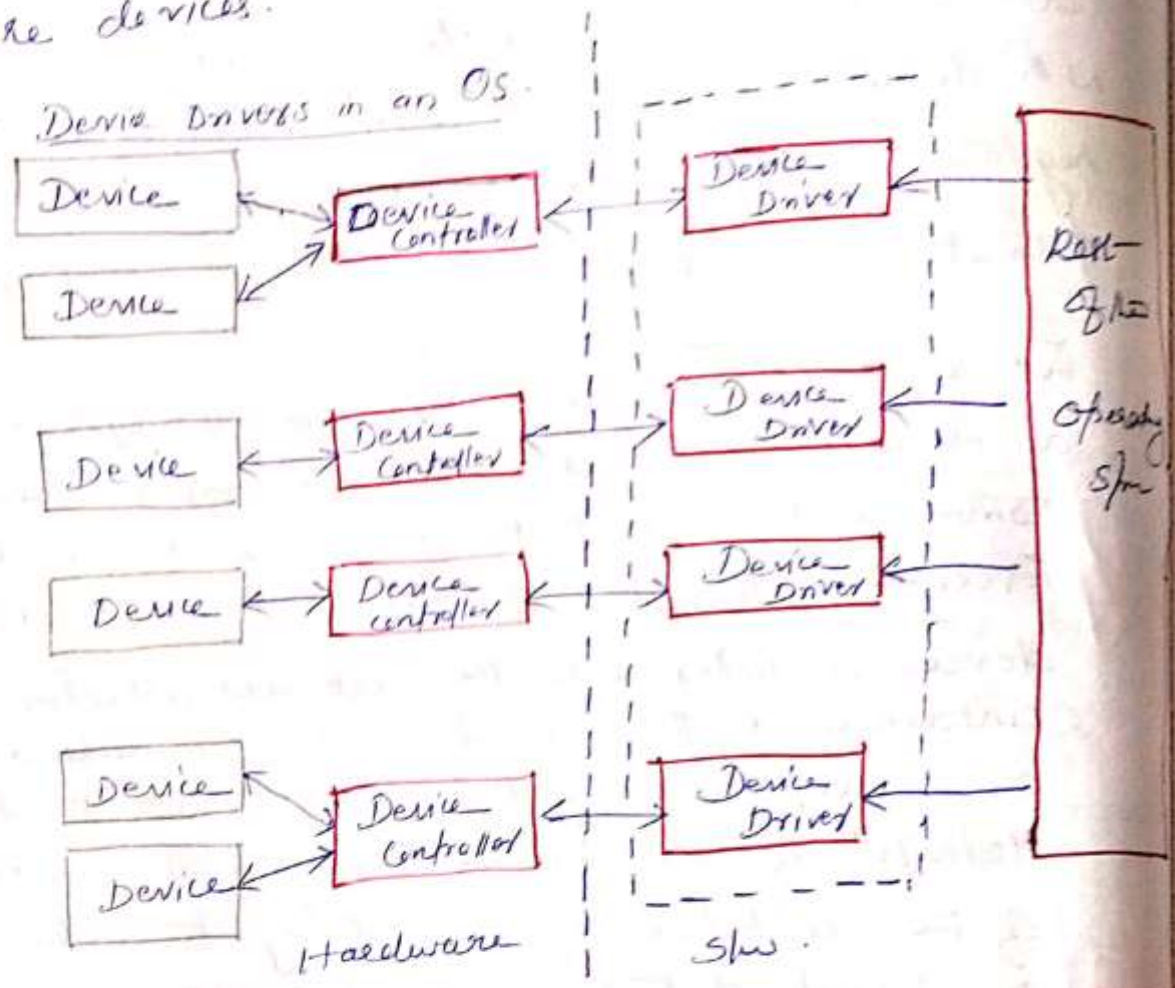
In order to manage this complexity, we create a module for each device controller whose job is to communicate with that particular device controller. Everything that the s/m needs to know about that device controller and the devices attached to it is contained in this module. This module is called a device driver. The device driver will know the details of how the device controller works (Address of the controller registers, bit layout in the registers, the format of the command msgs, error codes, operations codes, etc).

- A device driver knows how to control one device controllers (and the devices connected to it) and it also knows how to communicate the rest of the OS.
- A device driver is an interface module that communicates in the devices language on one side and the operating s/m's language

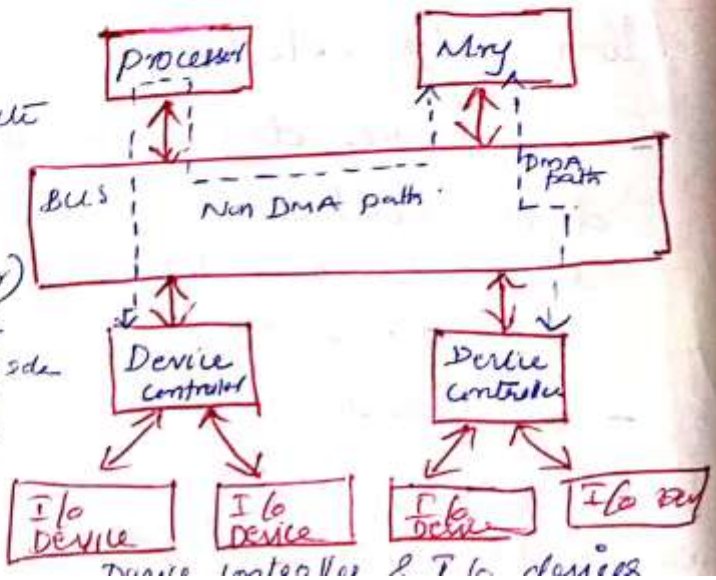
on other side. It is the s/w equivalent to the device controller, and that is why there is a device dri... for every device controller.

Fig - shows the relationship b/w the devices, device controller, device driver and rest of the OS. Each device controller is connected to one or more devices.

fig - Device Drivers in an OS.

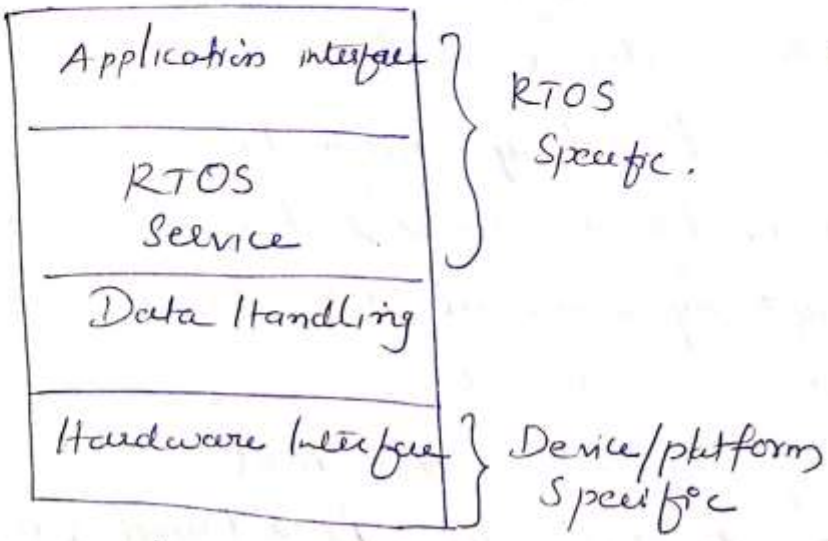


- Device controllers is an intermediate electronic device used to communicate b/w the I/o devices and computer (processor) On one side, it knows how to communicate with the I/o devices, computers S/m (usually over the s/m bus).



- The controller is on a card that plugs directly into the s/m bus, & there is a cable from the controller to each device it controls). [DMA - directly read & write in the mry]

Typical parts of a device driver



- Application interface
  - calling conventions
  - parameters
    - Device identification
    - mry allocation
- RTOS service
  - Semaphores
  - queues
  - mry allocation
- Data Handling
- Hardware interface
  - Device setup
  - Read/write/control

- RTOS have std interface to the device driver like create(), open(), read(), write().

- Device Driver interface :-

A device driver is a s/w module that defines an interface, that is set of procedures can be called.

or

A device driver contains all the s/w routines that are needed to be able to use the device. It contains no: of main routines like a initialization routine, is used to setup a device, a Reading routine that is used to be able to read data from the device, and write routine to be able to write data to device

Two common device driver interfaces:

- int deviceDriver.open (int deviceNumber) –

This call is made once before the device is used and allows the device driver to do any necessary initialization on the device. The device Number indicates which device is to be opened if the device driver is handling more than one device. In many devices (disks, for ex) this procedure does not do anything since no initialization is necessary. But other devices (for ex, some printers) require an initialization sequence when they are powered up. The returned value is success code for the open, which could fail if because the device number is invalid or the device is not ready to use.

- int DeviceDriver.close (int DeviceNumber): – This ca made once when the s/m is finished with the de

Design of Device Driver !

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

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

The device drivers can then be called from the application using RTOS standard calls. The RTOS creates during its own initialization tables that contain function pointers to all the device driver's routines. But as device drivers are initialized after the RTOS has been initialized you can in your device driver use the functionality of the RTOS.

When you design your system, you also have to specify which type of device driver design you need. Should the device driver be interrupt driven, which is most common today, or should the application be polling the device? It of course depends on the device itself, but also on your deadlines in your system. But you also need to specify if your device driver should called synchronously or asynchronously.

Synchronous Device Driver

When a task calls a synchronous device driver it means that the task will wait until the device has some data that it can give to the task, see figure 2. In this example the task is blocked on a semaphore until the driver has been able to read any data from the device.

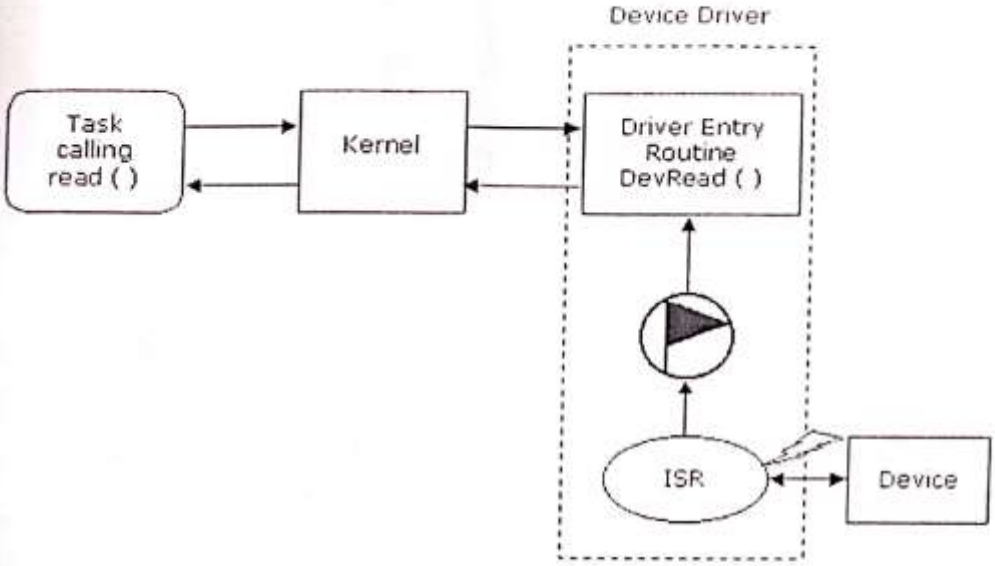


Figure 2. Synchronous device driver
The Task calls the device driver via a kernel device call. The Device Entry Routine gets blocked on a semaphore and blocks the task in that way. When an input comes from the device, it generates an interrupt and the ISR releases the semaphore and the Device Entry Routine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

Asynchronous Device Driver

When a task calls an asynchronous device driver it means that the task will only check if the device has some data that it can give to the task, see figure 3. In this example the task is just checking if there is a message in the queue. The device driver can independently of the task send data into queue.

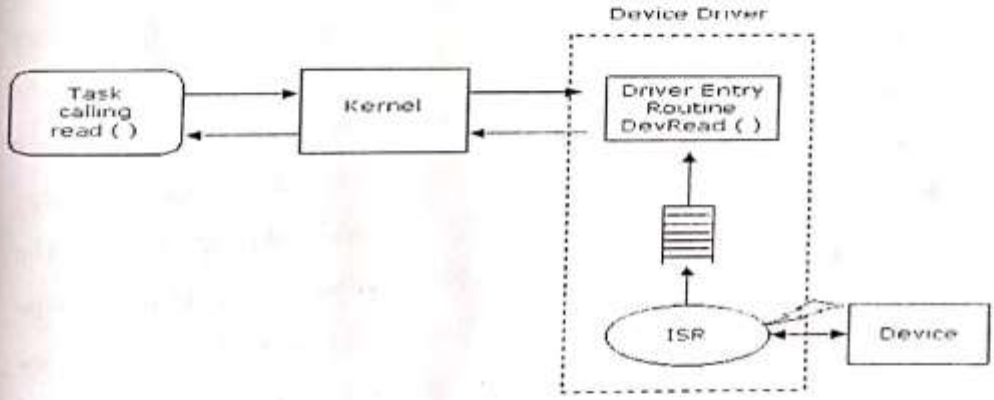


Figure 3. Asynchronous device driver
The Task calls the device driver via a kernel device call. The Device Entry Routine receives a message from the queue and returns the data to the task. When new data arrives from the device the ISR puts the data in a message and sends the message to the queue.

Serial Input and Output Data Spooler

If the device driver should be able to handle blocks of data by itself, the device driver needs to have internal buffers for storing data. Two examples of a design like this are shown in figure 5 & 6.

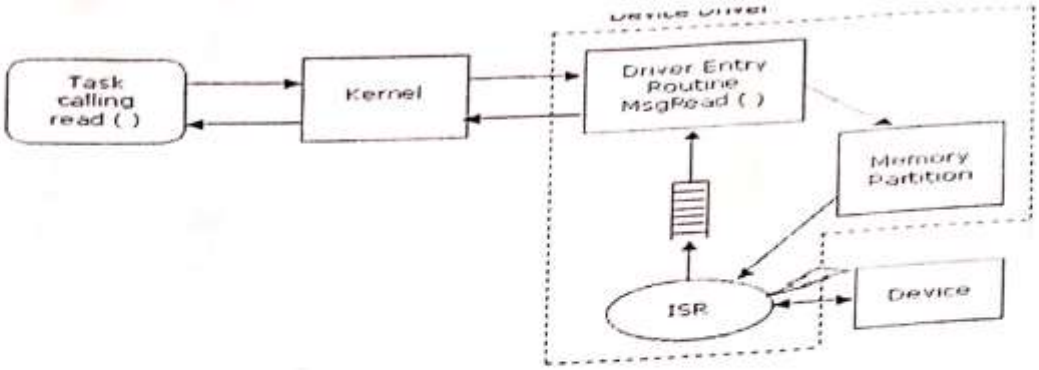


Figure 5. Serial Input Data Spooler
The Task calls the device driver via a Kernel device call. The Device Entry Routine receives a message from the queue and returns the data to the task and returns the buffer to memory partition. When new data arrives from the device the ISR allocates a buffer from a memory partition, puts the data in the buffer and puts a pointer to the buffer in a message and sends the message to the queue.

Structure of a Device Driver

The device header is a formatted table of information that the OS needs to setup and use the device driver property. The strategy and interrupt procedures are called by the OS. The rest of the driver is composed of routines that can be called with in the driver.

* Two categories of Device Drivers

Block Device Driver

Block devices

- Organize data in fixed size blocks.
- Transfers are in units of blocks
- Blocks have addresses and data are therefore addressable.

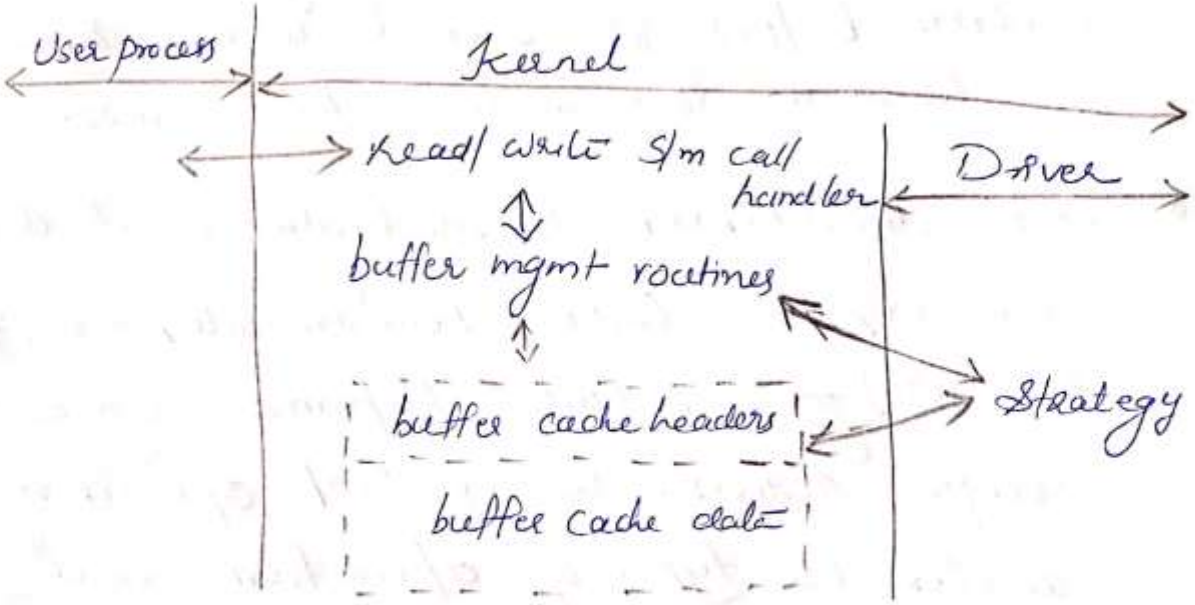
Eg. hard disk, USB disks, CDROMs.

And their device drivers will have the same interface, which we will call the block device interface.

The Block device interface

- int deviceDriver.read (int deviceNumber, int deviceAddress, char * bufferAddress) ⇒ This call reads a block of information from address deviceAddress and writes it in to memory at address bufferAddress.
- int deviceDriver.write (int deviceNumber, int deviceAddress, char * bufferAddress) :-

  This call reads a block of information from memory at address bufferAddress and writes it to the disk block at address deviceAddress.
- int deviceDriver.seek (int deviceNumber, int deviceAddress) → This call moves the read/write heads to the correct cylinder to read the block at address deviceAddress.



- Block drivers communicate with OS through a collection of fixed-sized buffers.

Biji kp

Character device Driver

- Character devices:
  - Delivers or accepts a stream of characters, no block structure
  - Not addressable, no seeks
  - can read/write from stream or write to stream.
  - Printers, network, interfaces, terminals.

- Int deviceDriver.read (int deviceNumber, int numberOfBytes, char * bufferAddress) – This call reads numberOfBytes bytes from character stream of the device and writes them in to memory at address bufferAddress.

- int deviceDriver.write (int deviceNumber, int numberOfBytes, char * bufferAddress) – This call reads numberOfBytes bytes from mry address bufferAddress and writes them to the character stream of the device.

- int deviceDriver.DeviceControl (int deviceNumber, int ControlOperationCode, int OperationData) – This call performs some device specific action. The control operation code indicates the type of operation, and operation Data is the data to use, if necessary.

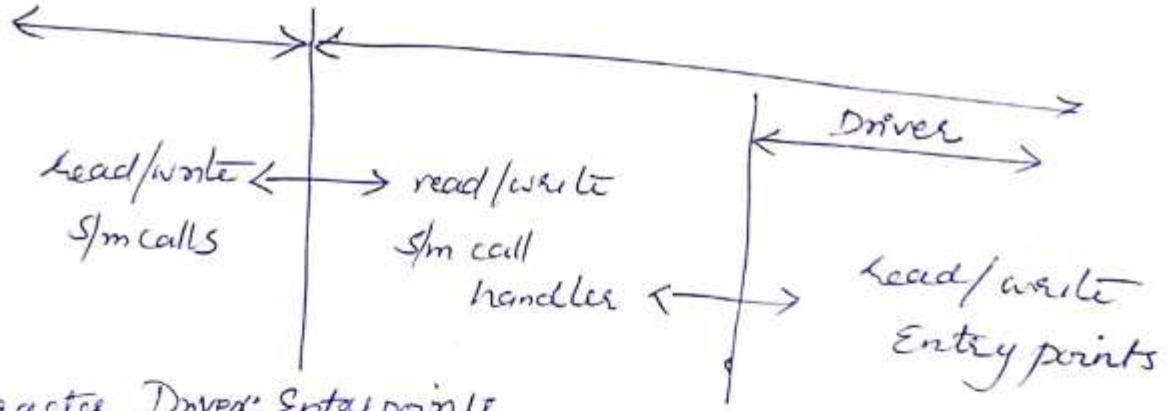
→ To write characters on a terminal write the

the keyboard with the read procedure.

Data from character devices does not have an address
Character devices read / write next data.

- The Device Control procedure is general, it is there in every character device driver has one, but device specific
Since the meaning of a call to Device Control will differs from device to device. Each device driver will implement a set of Device control commands.
ex: tape have set of commands
- Printer have set of commands

- the syntax of the call is the same for each character device, but the semantics of the call can be different for each driver.



- Character Driver: Entry points

init(): initialize h/w
start(): Boot time initializ^n
open(dev, flag, id): initialization for read/write
close(dev, flag, id): Release resource after read/write
read/write: data transfer.

- Block driver Entry points: init(), open(), close(), strategy() - Responsible for handling requests for data and replace both the read & write entry points found in character drivers.
print() - Used by kernel report the problems related to the driver.

Editors and Debugging Systems

This Chapter gives you...

- Text editors
- Interactive Debugging Systems

4.0 Introduction

An Interactive text editor has become an important part of almost any computing environment. Text editor acts as a primary interface to the computer for all type of "knowledge workers" as they compose, organize, study, and manipulate computer-based information.

An interactive debugging system provides programmers with facilities that aid in testing and debugging of programs. Many such systems are available during these days. Our discussion is broad in scope, giving the overview of interactive debugging systems – not specific to any particular existing system.

4.1 Text Editors

An Interactive text editor has become an important part of almost any computing environment. Text editor acts as a primary interface to the computer for all type of "knowledge workers" as they compose, organize, study, and manipulate computer-based information.

A text editor allows you to edit a text file (create, modify etc,..). For example the Interactive text editors on Windows OS - Notepad, WordPad, Microsoft Word, and text editors on UNIX OS - vi, emacs, jed, pico.

Normally, the common editing features associated with text editors are, Moving the cursor, Deleting, Replacing, Pasting, Searching, Searching and replacing, Saving and loading, and, Miscellaneous(e.g. quitting).

4.1.1 Overview of the editing process

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

Document-editing process in an interactive user-computer dialogue has four tasks

- Select the part of the target document to be viewed and manipulated
- Determine how to format this view on-line and how to display it
- Specify and execute operations that modify the target document
- Update the view appropriately

The above task involves traveling, filtering and formatting. Editing phase involves – insert, delete, replace, move, copy, cut, paste, etc…

- Traveling – locate the area of interest
- Filtering - extracting the relevant subset
- Formatting – visible representation on a display screen

There are two types of editors. Manuscript-oriented editor and program oriented editors. Manuscript-oriented editor is associated with characters, words, lines, sentences and paragraphs. Program-oriented editors are associated with identifiers, keywords, statements. User wish – what he wants – formatted.

4.1.2 User Interface

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

The user interface is concerned with, the input devices, the output devices and, the interaction language. The input devices are used to enter elements of text being edited, to enter commands. The output devices, lets the user view the elements being edited and the results of the editing operations and, the interaction language provides communication with the editor.

Input Devices are divided into three categories, text devices, button devices and, locator devices. Text Devices are keyboard. Button Devices are special function keys, symbols on the screen. Locator Devices are mouse, data tablet. There are voice input devices which translates spoken words to their textual equivalents.

Output Devices are Teletypewriters (first output devices), Glass teletypes (Cathode ray tube (CRT) technology), Advanced CRT terminals, TFT Monitors (Wysiwyg) and Printers (Hard-copy).

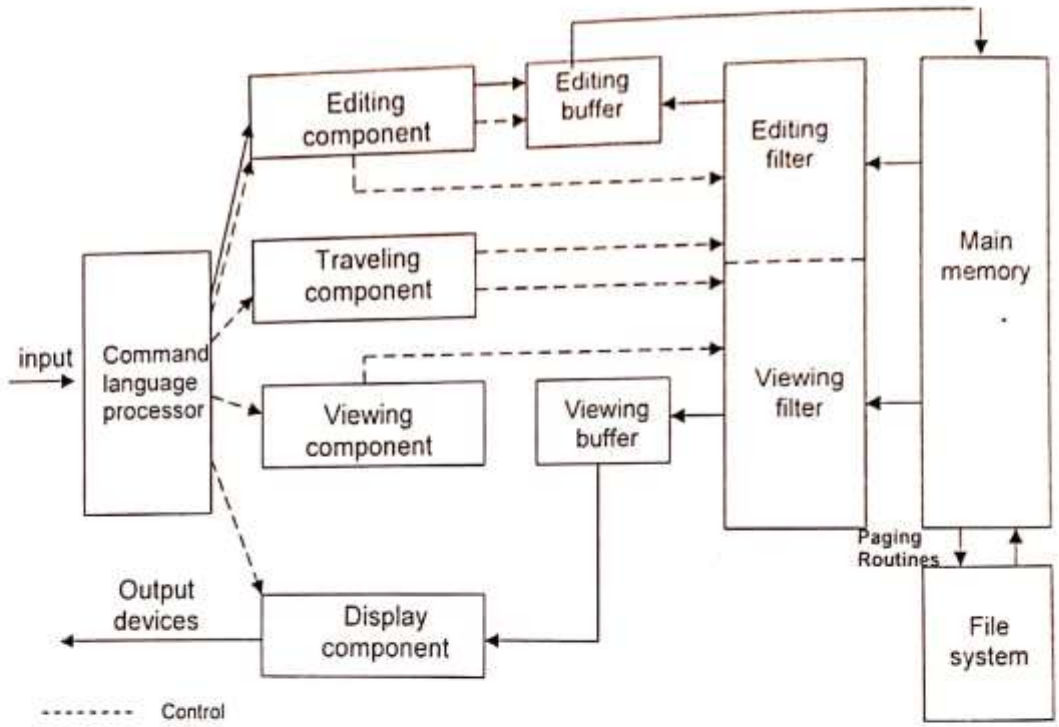
The interaction language could be, typing oriented or text command oriented and menu-oriented user interface. Typing oriented or text command oriented interaction was with oldest editors, in the form of use of commands, use of function keys, control keys etc.,

Menu-oriented user interface has menu with a multiple choice set of text strings or icons. Display area for text is limited. Menus can be turned on or off.

4.1.3 Editor Structure

Most text editors have a structure similar to that shown in the following figure. That is most text editors have a structure similar to shown in the figure regardless of features and the computers

Command language Processor accepts command, uses semantic routines – performs functions such as editing and viewing. The semantic routines involve traveling, editing, viewing and display functions.



Typical Editor Structure

Editing operations are specified explicitly by the user and display operations are specified implicitly by the editor. Traveling and viewing operations may be invoked either explicitly by the user or implicitly by the editing operations.

In editing a document, the start of the area to be edited is determined by the current editing pointer maintained by the editing component. Editing component is a collection of modules dealing with editing tasks. Current editing pointer can be set or reset due to next paragraph, next screen, cut paragraph, paste paragraph etc....

When editing command is issued, editing component invokes the editing filter – generates a new editing buffer – contains part of the document to be edited from current editing pointer. Filtering and editing may be interleaved, with no explicit editor buffer being created.

In viewing a document, the start of the area to be viewed is determined by the current viewing pointer maintained by the viewing component. Viewing component is a collection of modules responsible for determining the next view. Current viewing pointer can be set or reset as a result of previous editing operation.

When display needs to be updated, viewing component invokes the viewing filter – generates a new viewing buffer – contains part of the document to be viewed from current viewing pointer. In case of line editors – viewing buffer may contain the current line, Screen editors - viewing buffer contains a rectangular cutout of the quarter plane of the text. Viewing buffer is then passed to the display component of the editor, which produces a display by mapping the buffer to a rectangular subset of the screen – called a window. The editing and viewing buffers may be identical or may be completely disjoint. Identical – user edits the text directly on the screen. Disjoint – Find and Replace (For example, there are 150 lines of text, user is in 100th line, decides to change all occurrences of 'text editor' with 'editor'). The editing and viewing buffers can also be partially overlap, or one may be completely contained in the other. Windows typically cover entire screen or a rectangular portion of it. May show different portions of the same file or portions of different file. Inter-file editing operations are possible.

The components of the editor deal with a user document on two levels: In main memory and in the disk file system. Loading an entire document into main memory may be infeasible – only part is loaded – demand paging is used – uses editor paging routines. Documents may not be stored sequentially as a string of characters. Uses separate editor data structure that allows addition, deletion, and modification with a minimum of I/O and character movement.

4.1.4 Types of editors based on computing environment

Editors function in three basic types of computing environments: Time sharing, Stand-alone, and Distributed. Each type of environment imposes some constraints on the design of an editor.

In time sharing environment, editor must function swiftly within the context of the load on the computer's processor, memory and I/O devices. In stand-alone environment, editors on stand-alone system are built with all the functions to carry out editing and viewing operations – The help of the OS may also be taken to carry out some tasks like demand paging. In distributed environment, editor has both functions of stand-alone editor, to run independently on each user's machine and like a time sharing editor, contend for shared resources such as files.

4.2 Interactive Debugging Systems

An interactive debugging system provides programmers with facilities that aid in testing and debugging of programs. Many such systems are available during these days. Our discussion is broad in scope, giving the overview of interactive debugging systems – not specific to any particular existing system.

Here we discuss

- Introducing important functions and capabilities of IDS
- Relationship of IDS to other parts of the system
- The nature of the user interface for IDS

4.2.1 Debugging Functions and Capabilities

One important requirement of any IDS is the observation and control of the flow of program execution. Setting break points – execution is suspended, use debugging commands to analyze the progress of the program, résumé execution of the program. Setting some conditional expressions, evaluated during the debugging session, program execution is suspended, when conditions are met, analysis is made, later execution is resumed.

A Debugging system should also provide functions such as 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 – procedure, branch, individual instruction, and so on… Traceback can show the path by which the current statement in the program was reached. It can also show which statements have modified a given variable or parameter. The statements are displayed rather than as hexadecimal displacements

4.2.2 Program-Display capabilities

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

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

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

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

Examples of assignment statements

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

```
Cobol   -  IF X NOT EQUAL TO Z
Fortran -  IF ( X.NE.Z)
C       -  IF ( X <> Z)
```

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

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

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

- Invariant expressions can be removed from loops
- Separate loops can be combined into a single loop
- Redundant expression may be eliminated
- Elimination of unnecessary branch instructions

Leads to rearrangement of segments of code in the program. All these optimizations create problems for the debugger, and should be handled carefully.

4.2.3 Relationship with Other Parts of the System

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

Debugging is more important at production time than it is at application-development time. When an application fails during a production run, work dependent on that application stops. The debugger must also exist in a way that is consistent with the security and integrity components of the system. The debugger must coordinate its activities with those of existing and future language compilers and interpreters.

4.2.4 User-Interface Criteria

Debugging systems should be simple in its organization and familiar in its language, closely reflect common user tasks. The simple organization contribute greatly to ease of training and ease of use. The user interaction should make use of full-screen displays and windowing-systems as much as possible. With menus and full-screen editors, the user has far less information to enter and remember. There should be complete functional equivalence between commands and menus – user where unable to use full-screen IDSs may use commands. The command language should have a clear, logical and simple syntax; command formats should be as flexible as possible. Any good IDSs should have an on-line HELP facility. HELP should be accessible from any state of the debugging session.

CONTENT BEYOND THE SYLLABUS

Elaborate commands used in VI text editors.

There are many ways to edit files in Unix. Editing files using the screen-oriented text editor **vi** is one of the best ways. This editor enables you to edit lines in context with other lines in the file.

An improved version of the vi editor which is called the **VIM** has also been made available now. Here, VIM stands for **Vi IM**proved.

vi is generally considered the de facto standard in Unix editors because −

- It's usually available on all the flavors of Unix system.
- Its implementations are very similar across the board.
- It requires very few resources.
- It is more user-friendly than other editors such as the **ed** or the **ex**.

You can use the **vi** editor to edit an existing file or to create a new file from scratch. You can also use this editor to just read a text file.

Starting the vi Editor

The following table lists out the basic commands to use the vi editor −

Sr.No.	Command & Description
1	**vi filename** Creates a new file if it already does not exist, otherwise opens an existing file.
2	**vi -R filename** Opens an existing file in the read-only mode.
3	**view filename** Opens an existing file in the read-only mode.

Following is an example to create a new file **testfile** if it already does not exist in the current working directory −

```
$vi testfile
```

The above command will generate the following output −

```
|
~
~
~
~
~
~
~
~
~
~
~
~
"testfile" [New File]
```

You will notice a **tilde** (~) on each line following the cursor. A tilde represents an unused line. If a line does not begin with a tilde and appears to be blank, there is a space, tab, newline, or some other non-viewable character present.

You now have one open file to start working on. Before proceeding further, let us understand a few important concepts.

Operation Modes

While working with the vi editor, we usually come across the following two modes −

- **Command mode** − This mode enables you to perform administrative tasks such as saving the files, executing the commands, moving the cursor, cutting (yanking) and pasting the lines or words, as well as finding and replacing. In this mode, whatever you type is interpreted as a command.
- **Insert mode** − This mode enables you to insert text into the file. Everything that's typed in this mode is interpreted as input and placed in the file.

vi always starts in the **command mode**. To enter text, you must be in the insert mode for which simply type **i**. To come out of the insert mode, press the **Esc** key, which will take you back to the command mode.

Hint − If you are not sure which mode you are in, press the Esc key twice; this will take you to the command mode. You open a file using the vi editor. Start by typing some characters and then come to the command mode to understand the difference.

Detailed study of structure and record formats of DLL.

Dynamic Link Library (DLL) is Microsoft's implementation of the shared library concept. A DLL file contains code and data that can be used by multiple programs at the same time, hence it promotes code reuse and modularization. This brief tutorial provides an overview of Windows DLL along with its usage.

Dynamic linking is a mechanism that links applications to libraries at run time. The libraries remain in their own files and are not copied into the executable files of the applications. DLLs link to an application when the application is run, rather than when it is created. DLLs may contain links to other DLLs.

Many times, DLLs are placed in files with different extensions such as **.exe**, **.drv** or **.dll**.

Advantages of DLL

Given below are a few advantages of having DLL files.

Uses fewer resources

DLL files don't get loaded into the RAM together with the main program; they don't occupy space unless required. When a DLL file is needed, it is loaded and run. For example, as long as a user of Microsoft Word is editing a document, the printer DLL file is not required in RAM. If the user decides to print the document, then the Word application causes the printer DLL file to be loaded and run.

Promotes modular architecture

A DLL helps promote developing modular programs. It helps you develop large programs that require multiple language versions or a program that requires modular architecture. An example of a modular program is an accounting program having many modules that can be dynamically loaded at run-time.

Aid easy deployment and installation

When a function within a DLL needs an update or a fix, the deployment and installation of the DLL does not require the program to be relinked with the DLL. Additionally, if multiple programs use the same DLL, then all of them get benefited from the update or the fix. This issue may occur more frequently when you use a third-party DLL that is regularly updated or fixed.

Applications and DLLs can link to other DLLs automatically, if the DLL linkage is specified in the IMPORTS section of the module definition file as a part of the compile. Else, you can explicitly load them using the Windows LoadLibrary function.

Important DLL Files

Mentioned below are some important **dll** files which user should know for programming −

- **COMDLG32.DLL** − Controls the dialog boxes.
- **GDI32.DLL** − Contains numerous functions for drawing graphics, displaying text, and managing fonts.
- **KERNEL32.DLL** − Contains hundreds of functions for the management of memory and various processes.
- **USER32.DLL** − Contains numerous user interface functions. Involved in the creation of program windows and their interactions with each other.

Types of DLLs

When you load a DLL in an application, two methods of linking let you call the exported DLL functions. The two methods of linking are −

- load-time dynamic linking, and
- run-time dynamic linking.

Load-time dynamic linking

In load-time dynamic linking, an application makes explicit calls to the exported DLL functions like local functions. To use load-time dynamic linking, provide a header (.h) file and an import library (.lib) file, when you compile and link the application. When you do this, the linker will provide the system with the information that is required to load the DLL and resolve the exported DLL function locations at load time.

Runtime dynamic linking

In runtime dynamic linking, an application calls either the LoadLibrary function or the LoadLibraryEx function to load the DLL at runtime. After the DLL is successfully loaded, you use the GetProcAddress function, to obtain the address of the exported DLL function that you want to call. When you use runtime dynamic linking, you do not need an import library file.

The following list describes the application criteria for choosing between load-time dynamic linking and runtime dynamic linking −

- **Startup performance** − If the initial startup performance of the application is important, you should use run-time dynamic linking.
- **Ease of use** − In load-time dynamic linking, the exported DLL functions are like local functions. It helps you call these functions easily.

- **Application logic** − In runtime dynamic linking, an application can branch to load different modules as required. This is important when you develop multiple-language versions.

The DLL Entry Point

When you create a DLL, you can optionally specify an entry point function. The entry point function is called when processes or threads attach themselves to the DLL or detach themselves from the DLL. You can use the entry point function to initialize or destroy data structures as required by the DLL.

Additionally, if the application is multithreaded, you can use thread local storage (TLS) to allocate memory that is private to each thread in the entry point function. The following code is an example of the DLL entry point function.

```
BOOL APIENTRY DllMain(
  HANDLE hModule,  // Handle to DLL module
  DWORD ul_reason_for_call,
  LPVOID lpReserved )     // Reserved
{
  switch ( ul_reason_for_call )
  {
    case DLL_PROCESS_ATTACHED:
    // A process is loading the DLL. break;
    case DLL_THREAD_ATTACHED:
    // A process is creating a new thread.
    break;
    case DLL_THREAD_DETACH:
    // A thread exits normally.
    break;
    case DLL_PROCESS_DETACH:
        // A process unloads the DLL.
        break;
```

```
   } return TRUE;

}
```

When the entry point function returns a FALSE value, the application will not start if you are using load-time dynamic linking. If you are using runtime dynamic linking, only the individual DLL will not load.

The entry point function should only perform simple initialization tasks and should not call any other DLL loading or termination functions. For example, in the entry point function, you should not directly or indirectly call the **LoadLibrary** function or the **LoadLibraryEx** function. Additionally, you should not call the **FreeLibrary** function when the process is terminating.