JAWAHARLAL COLLEGE OF ENGINEERING AND TECHNOLOGY

(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(NBA Accredited)

COURSE MATERIAL

CST 305 SYSTEM SOFTWARE

VISION OF THE INSTITUTION

Jawaharlal College of Engineering and Technology, Mangalam intends to emerge as a centre of excellence imparting high quality education encompassing professional ethics, teaching and research to the students and faculty in the fields of Aeronautical, Electronics, Mechanical, Computer Engineering, Civil, Electrical, Management and other frontier technological areas of knowledge.

MISSION OF THE INSTITUTION

- To become an ultimate destination for acquiring latest and advanced knowledge in the multidisciplinary domains.
- To provide high quality education in engineering and technology through innovative teaching-learning practices, research and consultancy, embedded with professional ethics.
- To promote intellectual curiosity and thirst for acquiring knowledge through outcome-based education.
- To have partnership with industry and reputed institutions to enhance the employability skills of the students and pedagogical pursuits.
- To leverage technologies to solve the real-life societal problems through community services.

ABOUT THE DEPARTMENT

- Established in: 2008
- Courses offered: B.Tech in Computer Science and Engineering
- Affiliated to the A P J Abdul Kalam Technological University.

DEPARTMENT VISION

To produce competent professionals with research and innovative skills, by providing them with the most conducive environment for quality academic and research oriented undergraduate education along with moral values committed to build a vibrant nation.

DEPARTMENT MISSION

- Provide a learning environment to develop creativity and problem-solving skills in a professional manner.
- Expose to latest technologies and tools used in the field of computer science.
- Provide a platform to explore the industries to understand the work culture and expectation of an organization.
- Enhance Industry Institute Interaction program to develop the entrepreneurship skills.
- Develop research interest among students which will impart a better life for the society and the nation.

PROGRAMME EDUCATIONAL OBJECTIVES

Graduates will be able to

- Provide high-quality knowledge in computer science and engineering required for a computer professional to identify and solve problems in various application domains.
- Persist with the ability in innovative ideas in computer support systems and transmit the knowledge and skills for research and advanced learning.
- Manifest the motivational capabilities, and turn on a social and economic commitment to community services.

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.

COURSE OUTCOMES

CST305.1	C303.1	Distinguish software's into system and application software categories.
CST305.2	C303.2	Identify standard and extended architectural features of machines.
CST305.3	C303.3	Identify machine dependent features of system software
CST305.4	C303.4	Identify machine independent features of system software.
CST305.5	C303.5	Design algorithms for system software's and analyse the effect of data structures and understand the features of device drivers and editing & debugging tools

PROGRAM SPECIFIC OUTCOMES (PSO)

The students will be able to

- Use fundamental knowledge of mathematics to solve problems using suitable analysis methods, data structure and algorithms.
- Interpret the basic concepts and methods of computer systems and technical specifications to provide accurate solutions.
- Apply theoretical and practical proficiency with a wide area of programming knowledge, design new ideas and innovations towards research.

CO PO PSO MAPPING

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

Subject Code	Course Code	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CST305.1	C303.1	3	1	-	-	2	-	-	-	-	-	-	2	2	-	-
CST305.2	C303.2	3	3	2	-	-	-	-	-	-	-	-	2		3	
CST305.3	C303.3	2	2	2	-		-	-	-	-	-	-	2			3
CST305.4	C303.4	3	2	-	-	-	-	-	-	-	-	-	2			
CST305.5	C303.5	3	2	2	2	-	2	-	-	-	-	-	2	2	3	
CS303	**C303**	**2.667**	**2**	**2**	**2**	**2**	**2**	-	-	-	-	-	**2**	**2**	**3**	**3**

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)**

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.

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 quotientin ALPHA reminder in DELTA	CO1	K5
5	Illustrate the SIC/XE architecture, Explaining in detaildata and instruction formats.	CO1	K3
6	Describe the format of Object Program generated bythe 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 applicationsoftware.	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 asequence 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 bythe 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

MODULE 1

SOFTWARE

- Set of instructions given to the computer.
- We cannot touch and feel it.
- Developed by writing instructions in programming language.
- Operations of computer are controlled via this.
- If damaged or corrupted, back up copy can be installed again.
- Eg:- Antivirus, Microsoft Office Tools.

HARDWARE

- Physical parts of a computer.
- We can touch and feel it.
- Constructed using physical components.
- Operates under control of software.
- If damaged, can be replaced.
- Eg:- Keyboard, Monitor, Mouse

SOFTWARE vs HARDWARE

SOFTWARE	HARDWARE
1. Collection of instructions that tells computer what to do	1. Physical elements of computer
2. Divided in to a. System Software b. Application Software	2. Categories a. Input Devices. b. Output Devices

c. Utility Software	c. Storage Devices
3. Should be installed in to computer	3. Once software is loaded these can be used.
4. Prone to viruses	4. No virus attacks
5. If damaged/ corrupted reinstallation is possible	5. If damaged, can be replaced.
Eg:- Microsoft Office, Adobe	Eg:- Mouse, Monitor, Keyboard

TYPES OF SOFTWARE

1. System Software:
    - Contains collection of programs that support operation of computer.
    - Helps to run computer hardware and computer system.
    - Handles running of computer hardware.
    - These are of different types"
        a) Operating System
        b) Language Translators
            i. Compiler
            ii. Assembler
            iii. Interpreter
            iv. Macro Processor
        c) Loader
        d) Linker
        e) Debugger
        f) Text Editor

2. Application Software:
    - It allows end users to accomplish one or more specific tasks.
    - Focus on application or problem to be solved.

Operating System

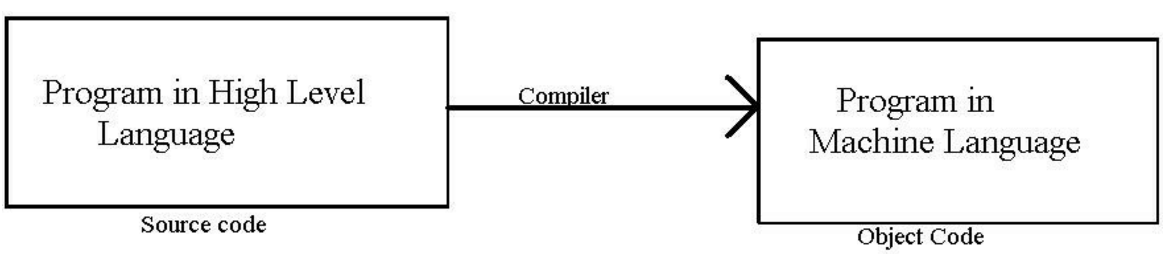
- Acts as interface between user and system.
- Provide user friendly interface.
- Functions:
    - a) Process Management
    - b) Memory Management
    - c) Resource Management
    - d) I/O Operations
    - e) Data Management
    - f) Provide Security for job.

Language Translators

- Program that takes input program in one language and produces an output in another language.

I. Compilers

- Translates program in high level language in to machine level language.
- Conversion or translation is taking place by taking program as whole.
- Bridges the semantic gap between language domain and execution domain.
- Perform syntax analysis, semantics analysis and intermediate code generation.



II. Interpreters

- Translates statement of high level language in to machine level language by taking the program line by line.
- Interpretation cycle includes:

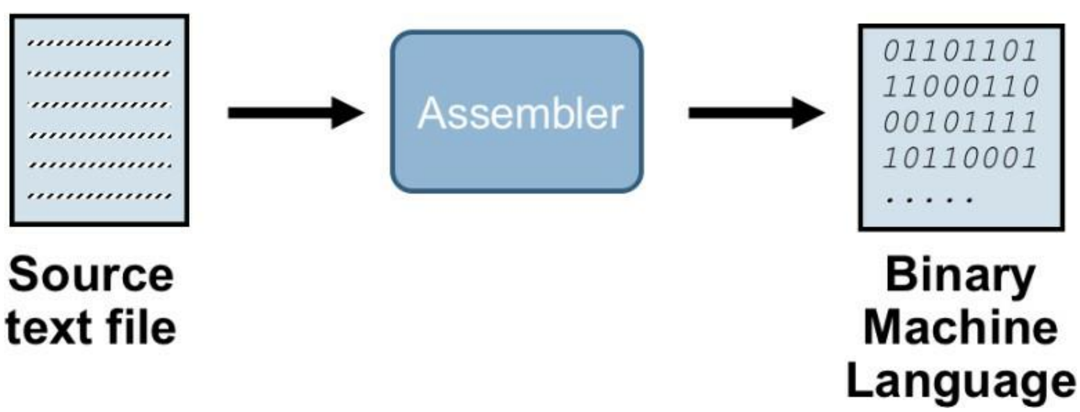
  i) Fetch the statement.

  ii) Analyze the statement and determine its meaning.

  iii) Execute the meaning of statement.

III. Assemblers

- Programmers found it difficult to read or write programs in machine language, so for convenience they used mnemonic symbols for each instruction which is translated to machine language.
- Assemblers translate assembly language to machine language.
- Translate mnemonic code to machine language equivalents.
- Assign machine address to symbol table.



Working:

- Find the required information to perform task.
- Analyze and design suitable data structures to hold and manipulate information.
- Find the process or steps needed to gather information and maintain it.
- Determine processing step required to execute each identified task.

COMPILER vs INTERPRETER vs ASSEMBLER

COMPILER VS INTERPRETER VS ASSEMBLER

COMPILER	INTERPRETER	ASSEMBLER
Software that converts programs written in a high level language into machine language	Software that translates a high level language program into machine language	Software that converts programs written in assembly language into machine language
Converts the whole high level language program to machine language at a time	Converts the high level language program to machine language line by line	Converts assembly language program to machine language
Used by C, C++	Used by Ruby, Perl, Python, PHP	Used by assembly language

<u>Linker</u>

- Process of collecting and combining various pieces of code and data in to single file that can be loaded in to memory and executed.
- Linking performed a compile time, when source code is translated to machine code, at load time, when program is loaded in to memory and executed by loader and at run time by application programs.

Types:

a) Linking Loader: Performs all linking and relocation operations directly in to main memory for execution.

b) Linkage Editor: Produce a linked version of program called as load module or executable image. This load module is written in to file or library for later execution.

c) Dynamic Linker: This linking postpones the linking function until execution time. Also called as dynamic loading.

Loader

- Utility of an operating system.
- Copies program from a storage device to computer's main memory.
- They can replace virtual address with real address.
- They are invisible to user.

Debugger

- An Interactive debugging system provides programmers with facilities that aid in testing and debugging of programs.
- Debugging means locating bugs or faults in program.
- Helps in fixing error.
- Determination of exact nature and location of error in the program.

Device Driver

- It is a software module which manages the communication and control of specific I/O device on type of device.
- Convert logical requests from the user in to specific commands directed to device itself.

Macro Processor

- Macro is the unit of specification of program generation through expansion.
- Macros are special code fragments that are defined once in the program and used by calling them from various places within the program.
- Macro processor is a program that copies stream of text from one place to another, making a systematic set of replacements as it does so.
- They are often embedded in other programs such as assemblers and compilers.

- Before you can use a macro, you must *define* it explicitly with the `#define' directive. `#define' is followed by the name of the macro and then the code it should be an abbreviation for. For example,

  #define BUFFER_SIZE 1020

  defines a macro named `BUFFER_SIZE' as an abbreviation for the text `1020'

Text Editors

- Program that allows the user to create the source program in the form of text in to the main memory.
- Creation, edition, deletion, updating of document or files can be done with the help of text editor.

SIMPLIFIED INSTRUCTIONAL COMPUTER (SIC)

- It is a hypothetical computer that has hardware features which are found in real machines.
- To versions:

  a). SIC Standard Model

  b). SIC/XE (Extra Equipment)

Machine Dependent features of Software System:

1. Assembler: Instruction format, Addressing mode.
2. Compiler: Registers, Machine Instructions.
3. OS: All resources of computing system.

Machine Independent features of Software

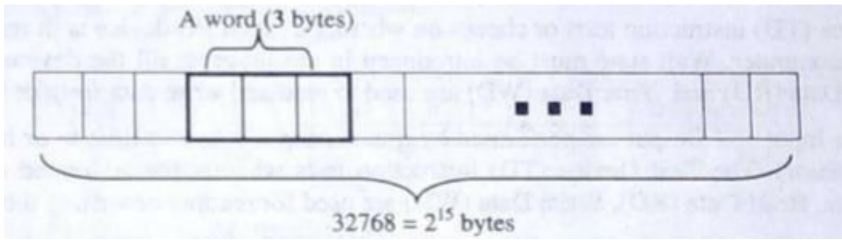
System:

1. General design and logic of assembler.
2. Code optimization in compiler
3. Linking independently assembled subprogram

SIC ARCHITECTURE- STANDARD MODEL

- It has basic addressing, storing most memory addresses in hexadecimal integer format.
- Its machine architecture includes

1. Memory: There are 2^{15} bytes in the computer memory that is 32768 bytes.



2. Register:

   ➢ Used as storage locations that perform some functions.

   ➢ There are 5 registers each of them is of 24 bits length.

Five Registers

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) instruction stores the return address in this register
PC	8	Program counter; contains the address of the next instruction to be fetched for execution
SW	9	Status word; contains a variety of information, including a Condition Code (CC)

3. Data Formats:

   ➢ It supports only the Integer and Character data formats.

- There is no hardware support for floating point numbers.
- Integers stored as 24 bit binary numbers.
- Negative values represented as 2's complement.
- Character data stored as 8 bit ASCII codes.

4. Instruction Formats:
   - All machine instructions in the standard version of SIC have the following 24 bit format:

8	1	15
OPCODE	X	Address

   - Flag bit x is used to indicate the indexed addressing mode.

5. Addressing mode: 2 Types

   a) Direct Addressing Mode: Here flag bit x=0

   Target Address= Actual Address

   b) Indexed Addressing Mode: Here flag bit x=1

   Target Address= Actual Address+Index Register (X) contents

   i.e. **Target Address= Address+(X)**

6. Instruction Set:

   a. Data Transfer Instruction: Include instructions that load and store register.

   Eg: LDA, STA, LDX, STX

   b. Arithmetic Operation Instruction: Arithmetic operations can be done which involves register A

   Eg: ADD, SUB, MUL, DIV, COMPR

   c. Conditional Branching Instruction: The conditional jump instruction test the setting of condition code and jumps.

   Eg: JLT, JEQ, JGT

   d. Subroutine Call Instruction: Two instructions are provided to perform subroutine linkage

   i) JSUB: To jump

ii) RSUB: To return

e. Input and Output Instruction:

- I/O operations are executed by transferring a single byte each time.
- Target port is specified by last 8 bits of register A.
- Each device is assigned a unique 8 bit code to send and receive data and control signals.

7. Input and Output:

- Performed by transferring 1 byte at a time to or from right most 8 bits of register A (Accumulator).
- Test Device (TD) instruction tests whether the addressed device is ready to send and receive a byte of data.
- Read Data (RD) and Write Data (WD) is used for reading and writing of data.

8. Data Movement and Storage Definitions:

- LDA, STA, LDX, STX all uses 3 byte word.
- LDCH, STCH are associated with characters which uses 1 byte.
- Storage definitions are:
  a. WORD- ONE WORD CONSTANT
  b. RESW- ONE WORD VARIABLE
  c. BYTE- ONE BYTE CONSTANT
  d. RESB- ONE BYTE VARIABLE

SIC/XE ARCHITECTURE- SIC WITH EXTRA EQUIPMENT

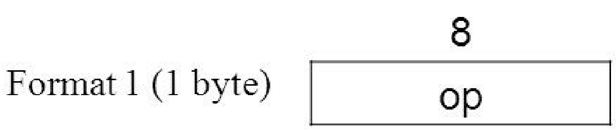
- Architecture is similar to standard model with certain additional components and features.
  1. Memory: Maximum memory available on a SIC/XE system is 1MB (2^{20} bytes)
  2. Registers: Additional B, S, T and F registers are provided by SIC/XE , in addition to the registers of SIC.

Mnemonic	Number	Special use
B	3	Base register
S	4	General working register
T	5	General working register
F	6	Floating-point accumulator (48 bits)

3. Floating point Data type: There is a 48 bit floating point data type, $F*2^{(e-1024)}$

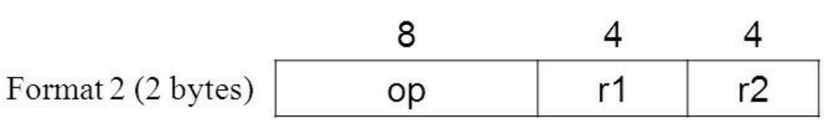
4. Instruction format: New set of instruction formats for SIC/XE are as follows:

   a. Format 1 (1 Byte): Contains only operation code



   Eg: RSUB (Return to Subroutine)

   b. Format 2 (2 Bytes): First 8 bits for operation code, next four for register 1 and following for register 2.



   Eg: COMPR A, S (Compare contents of register A and S)

   c. Format 3 (3 Bytes) : Here e=0

      - First 6 bits contain operation code.
      - Next 6 bits contain flags.
      - Last 12 bits contain displacement for the address of the operand.

- Flags are in order -n, i, x, b, p, e.
- e indicates instruction format.
- Bits i and n are used for target address calculation

	6	1	1	1	1	1	1	12
Format 3 (3 bytes)	op	n	i	x	b	p	e	disp

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

Format 3 has many cases:

i. If i=0, n=1, word given by target address is fetched and value in word is taken as address of operand value- Indirect Addressing (Prefix #).

ii. If i=1, n=0, target address is used as operand value.

Also called Immediate Addressing mode (Prefix #)

a) Case 1: Value contained location in word=operand value

Eg: ADD X, [500]

Here word in location 500 is fetched .

It gives address of first operand, second operand is given in indirect addressing mode.

b) Case 2: Target Address= Operand

Value Eg:- If TA=10, Operand Value

=10

iii. If i=0, n=0 or i=1, n=1 target address is the location of operand. Also called as Simple Addressing.

TA=location of operand

d. Format 4 (4 bytes): Here e=1

- It is same as format 3 with an extra 2 hex digits for address that require more than 12 bits to be represented.

	6	1	1	1	1	1	1	20
Format 4 (4 bytes)	op	n	i	x	b	p	e	address

5. Addressing mode and Flag bits:

a. Direct (x,b and p All set to 0):

- ➢ Operand address goes as it is.
- ➢ n and i are both set to the same value, either 0 or 1.

b. Relative (Either b or p equal to 1 and the other one to 0): Address of operand should be added to the current value stored at the B register (if b=1) or to the value stored at the PC register (if p=1)

c. Immediate (i=1,n=0): The operand value is already enclosed on the instruction.

d. Indirect (i=0, n=1): The operand value points to an address that holds the address for operand value.

e. Indexed (x=1):

- ➢ Value to be added to the value stored at the register x to obtain real address of operand.
- ➢ Can be combined with any of previous mode except

immediate. Indexing is not possible with immediate or indirect addressing

mode.

Two relative addressing modes are:

i) Base relative addressing mode.

ii) Program counter relative addressing mode.

Mode	Indication	Target Address Calculation
Base Relative Addressing Mode	b = 1 P = 0	**TA = Displacement + (B)** B – Base Register Displacement is 12 bit unsigned register. Displacement lies between 0 to 4095
Program Counter Relative Addressing Mode	b = 0 P = 1	**TA = Displacement + (PC)** PC – program counter Displacement is 12 bit signed integer. Displacement lies between – 2048 to 2047.
Direct Addressing Mode	b = 0, P = 0 (Format 4 instruction) b = 0, P = 0 (Format 3 instruction)	**TA = address field of format 4 instruction** TA = Displacement field value of format 3 instruction
Base Relative Indexed Addressing Mode	b = 1, P = 0 X = 1	**TA = Displacement + (B) + (X)** B – Base register X – Index register Displacement is 12 bit unsigned register. Displacement lies between 0 to 4095
Program Counter Relative Indexed Addressing Mode	b = 0, P = 1 X = 1	**TA = Displacement + (PC) + (X)** PC – program counter X – Index Register Displacement is 12 bit signed integer. Displacement lies between – 2048 to 2047

6. Instruction set:

   a. Instruction that load and store new register

‘B’:LDB- Load the register ‘B’ with some

value. Eg: LDBx- Load value of x in to

register B.

b. STB- Store the register ‘B’ content in to some variable.

   Eg: STBx- Store register ‘B’ content in to variable x.

ii. Instruction those perform floating point Arithmetic operation

   a. ADDF
   b. SUBF
   c. MULF
   d. DIVF

   Here F is the floating point register

   Eg: ADDF, here register’ B’ contents are added with Accumulator content and result is left with accumulator.

iii. Instruction that take operand from

   Register RMO-Register move

   Eg: RMO S,B Register ‘S’ content is moved to ‘B’ register.

iv. Instruction which perform register arithmetic operation

   a. ADDR
   b. SUBR
   c. MULTR
   d. DIVR

   Eg: ADDR S,B

   add value of register B with register Sand store result in register B.

7. Input and Output:

   - The SIC/XE supports all the I/O instructions in the standard version.
   - There are special I/O channels which are utilized for data transfer when CPU is involved in another process at same time.
   - Channels control associated I/O channels.
   - There can be maximum of 16 I/O channels each supporting maximum of 16 devices.RD and WD is used to read and write data from or to specified I/O

devices.

SIO	Instruction is used to Start an I/O Channel number
TIO	Instruction is used to Test an I/O Channel number
HIO	Instruction is used to Halt an I/O Channel number

SIC vs SIC/XE

Basis	SIC	SIC/XE
Registers	Only 5 registers are used, which are A, X, L, SW and PC.	There are 9 registers are used, which are A, X, L, SW, PC, B, T and F.
Floating Point Hardware	There is no floating point hardware.	Floating point hardware is used.
Instruction Format	Only one instruction format is used.	There are four different types of instructions format.
Addressing Modes	There are two addressing modes.	There are many more addressing modes.

Also refer the pdf (Comparison SIC and SIC XE)

ASSEMBLER DIRECTIVES

- Pseudo instructions.
- Provide instruction to assembler itself
- They are not translated in to machine operation code.
- SIC and SIC/XE has following assemble directives:

  START- Specify name and starting address of the

  program

  END- Indicate end of the source program and specify first executable statement in program

  BYTE- Generate character or hexadecimal constant.

  WORD- Generate one word integer constant.

  RESB- Reserves the indicated number of bytes for data area.

  RESW- Reserve the indicated number of words for data area.

<u>Data movement in SIC and SIC/XE</u>

1. Data Movement in SIC

```
        LDA     EIGHT       load constant 8 in to the register A
        STA     FIRST       store in FIRST
        LDCH    CHARZ       load character 'Z' in to register A
        STCH    C1          store in character variable C1
        .
        .
        .
FIRST   RESW    1           One word variable
EIGHT   WORD    8           One word constant
CHARZ   BYTE    C'Z'        One byte constant
C1      RESB    1           One byte variable
```

Note, In SIC:

- RESB and RESW is used for variables
- BYTE and WORD is used for values
- RESB is used for variable for eg: C1
- RESW is used for variables represented using words For eg: FIRST, it is a variable name represented in form of letter/ word. C can be another example which uses the assembler directive RESW
- BYTE is used for character values/constants for eg: char Z
- WORD is used for values expressed in word form, for eg: EIGHT represents value 8 in word form

2. <u>Data Movement in SIC/XE</u>

    - Here immediate addressing scheme is used.

```
        LDA     #8          load value 8 in to the register A
        STA     FIRST       store in FIRST
        LDCH    #90         load ASCI code of 'Z' in to register A
        STCH    C1          store in character variable C1
        .
        .
        .
FIRST   RESW    1           One word variable
C1      RESB    1           One byte variable
```

Note, In SIX/XE:

- The values are represented with a prefix # and in numerical form , eg: #8
- Character values are represented using their ASCII values, eg: for Z we used its ASCII value 90

Arithmetic Operations in SIC and SIC/XE

1. In SIC

```
        LDA     FIRST       load FIRST into register A
        ADD     INCR        add value of INCR
        SUB     ONE         subtract 1
        STA     SECOND      store in SECOND
        LDA     THIRD       load THIRD into register A
        ADD     INCR        add value of INCR
        SUB     ONE         subtract 1
        STA     FOURTH      store in FOURTH
        .
        .
        .
FIRST   RESW    1           One word variable
ONE     WORD    1           One word variable
SECOND  RESW    1           One word variable
THIRD   RESW    1           One word variable
FOURTH  RESW    1           One word variable
INCR    RESW    1           One word variable
```

2. In SIC/XE

```
            LDS     INCR      load value of INCR in to the register S
            LDA     FIRST     load FIRST into register A
            ADDR    S, A      add value of INCR
            SUB     #1        subtract 1
            STA     SECOND    store in SECOND
            LDA     THIRD     load THIRD into register A
            ADDR    S, A      add value of INCR
            SUB     #1        subtract 1
            STA     FOURTH    store in FOURTH
            .
            .
            .
FIRST       RESW    1         One word variable
SECOND      RESW    1         One word variable
THIRD       RESW    1         One word variable
FOURTH      RESW    1         One word variable
INCR        RESW    1         One word variable
```

Input/ Output Operations in SIC and SIC/XE

1. In SIC

```
INLOOP   TD      INDEV    test input device
         JEQ     INLOOP   loop until device is ready
         RD      INDEV    read one byte into register A
         STCH    DATA     store byte that was read
         .
         .
         .
OUTLP    TD      OUTDEV   test output device
         JEQ     OUTLP    load until device is ready
         LDCH    DATA     load data byte into register A
         WD      OUTDEV   write one byte to output device
         .
         .
         .
INDEV    BYTE    X’F1’    input device number
OUTDEV   BYTE    X’05’    output device number
DATA     RESB    1        one byte variable
```

2. In SIC/XE

```
INLOOP   TD      INDEV     test input device
         JEQ     INLOOP    loop until device is ready
         RD      INDEV     read one byte into register A
         STCH    DATA      store byte that was read
         .
         .
         .
OUTLP    TD      OUTDEV    test output device
         JEQ     OUTLP     load until device is ready
         LDCH    DATA      load data byte into register A
         WD      OUTDEV    write one byte to output device
         .
         .
         .
INDEV    BYTE    X’F1’     input device number
OUTDEV   BYTE    X’05’     output device number
DATA     RESB    1         one byte variable
```

MODULE -2

ASSEMBLERS-1

2.1 Basic Assembler Functions:

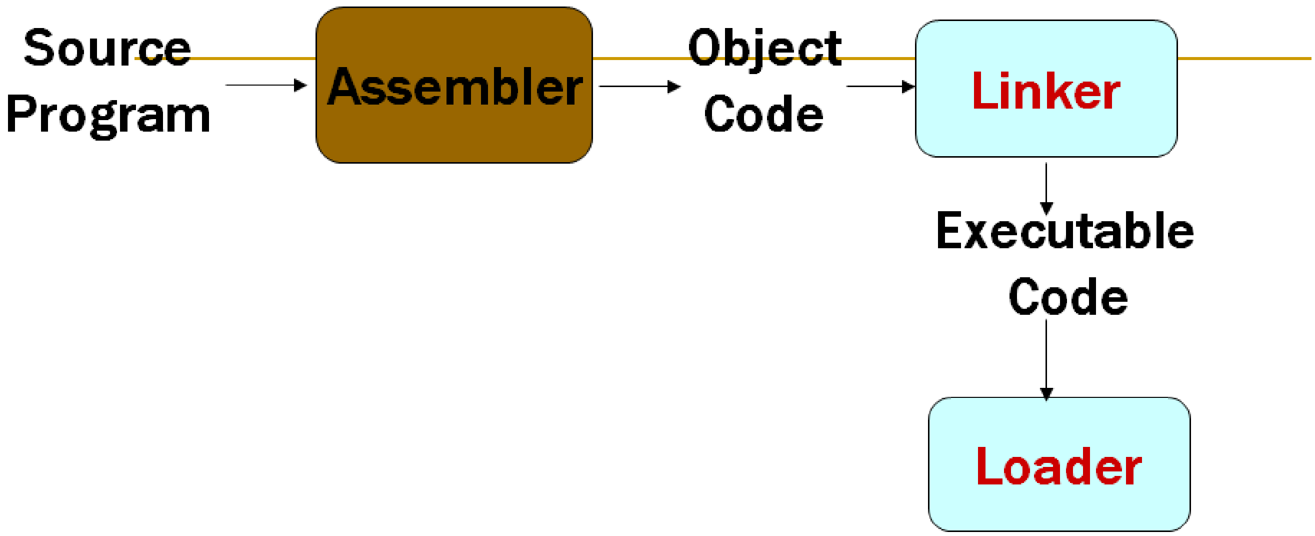


Figure 1

- Figure 2 shows SIC program which contains a main routine that reads records from an input device (F1) and copies that to an output device (05) . This main routine calls subroutine RDREC to read a record into a buffer and subroutine WRREC to write the record from the buffer to the output device. Each subroutine must transfer one byte at a time. The end of each record is marked with a null character(hexa decimal 00). The end of the file to be copied is indicated by a zero length record. When the end of the file is detected the program writes EOF on the output device. And terminates by executing RSUB instruction and returns to the OS. Length of the buffer is 4096 bytes.

Line   Source statement

```
COPY     START   1000         COPY FILE FROM INPUT TO OUTPUT
FIRST    STL     RETADR       SAVE RETURN ADDRESS
CLOOP    JSUB    RDREC        READ INPUT RECORD
         LDA     LENGTH       TEST FOR EOF (LENGTH = 0)
         COMP    ZERO
         JEQ     ENDFIL       EXIT IF EOF FOUND
         JSUB    WRREC        WRITE OUTPUT RECORD
         J       CLOOP        LOOP
ENDFIL   LDA     EOF          INSERT END OF FILE MARKER
         STA     BUFFER
         LDA     THREE        SET LENGTH = 3
         STA     LENGTH
         JSUB    WRREC        WRITE EOF
         LDL     RETADR       GET RETURN ADDRESS
         RSUB                 RETURN TO CALLER
EOF      BYTE    C'EOF'
THREE    WORD    3
ZERO     WORD    0
RETADR   RESW    1
LENGTH   RESW    1            LENGTH OF RECORD
BUFFER   RESB    4096         4096-BYTE BUFFER AREA
.
.        SUBROUTINE TO READ RECORD INTO BUFFER
.
RDREC    LDX     ZERO         CLEAR LOOP COUNTER
         LDA     ZERO         CLEAR A TO ZERO
RLOOP    TD      INPUT        TEST INPUT DEVICE
         JEQ     RLOOP        LOOP UNTIL READY
         RD      INPUT        READ CHARACTER INTO REGISTER A
         COMP    ZERO         TEST FOR END OF RECORD (X'00')
         JEQ     EXIT         EXIT LOOP IF EOR
         STCH    BUFFER,X     STORE CHARACTER IN BUFFER
         TIX     MAXLEN       LOOP UNLESS MAX LENGTH
         JLT     RLOOP          HAS BEEN REACHED
EXIT     STX     LENGTH       SAVE RECORD LENGTH
         RSUB                 RETURN TO CALLER
INPUT    BYTE    X'F1'        CODE FOR INPUT DEVICE
MAXLEN   WORD    4096
```

```
        .          SUBROUTINE TO WRITE RECORD FROM BUFFER
        .
WRREC   LDX        ZERO                CLEAR LOOP COUNTER
WLOOP   TD         OUTPUT              TEST OUTPUT DEVICE
        JEQ        WLOOP               LOOP UNTIL READY
        LDCH       BUFFER,X            GET CHARACTER FROM BUFFER
        WD         OUTPUT              WRITE CHARACTER
        TIX        LENGTH              LOOP UNTIL ALL CHARACTERS
        JLT        WLOOP                 HAVE BEEN WRITTEN
        RSUB                           RETURN TO CALLER
OUTPUT  BYTE       X'05'               CODE FOR OUTPUT DEVICE
        END        FIRST
```

Figure 2.1 Example of a SIC assembler language program.

SIC Assembler Directive:

- In addition to the machine instructions assembler directives are also used in programs. Assembler directives are pseudo instructions. They provide instructions to the assembler itself. They are not translated into machine code.

START – Specify name and starting address for the program.
END – Indicate the end o the source program and(optionally) specify the first executable instruction in the program.
BYTE – Generate character or hexadecimal constant , occupying as 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.

A Simple SIC Assembler

- Figure 3 shows the same program as in figure 2 with the generated object code for each statement.

Loc	Source statement			Object code
1000	COPY	START	1000	
1000	FIRST	STL	RETADR	141033
1003	CLOOP	JSUB	RDREC	482039
1006		LDA	LENGTH	001036
1009		COMP	ZERO	281030
100C		JEQ	ENDFIL	301015
100F		JSUB	WRREC	482061
1012		J	CLOOP	3C1003
1015	ENDFIL	LDA	EOF	00102A
1018		STA	BUFFER	0C1039
101B		LDA	THREE	00102D
101E		STA	LENGTH	0C1036
1021		JSUB	WRREC	482061
1024		LDL	RETADR	081033
1027		RSUB		4C0000
102A	EOF	BYTE	C'EOF'	454F46
102D	THREE	WORD	3	000003
1030	ZERO	WORD	0	000000
1033	RETADR	RESW	1	
1036	LENGTH	RESW	1	
1039	BUFFER	RESB	4096	
	.			
	.	SUBROUTINE TO READ RECORD INTO BUFFER		
	.			
2039	RDREC	LDX	ZERO	041030
203C		LDA	ZERO	001030
203F	RLOOP	TD	INPUT	E0205D
2042		JEQ	RLOOP	30203F
2045		RD	INPUT	D8205D
2048		COMP	ZERO	281030
204B		JEQ	EXIT	302057
204E		STCH	BUFFER,X	549039
2051		TIX	MAXLEN	2C205E
2054		JLT	RLOOP	38203F
2057	EXIT	STX	LENGTH	101036
205A		RSUB		4C0000
205D	INPUT	BYTE	X'F1'	F1
205E	MAXLEN	WORD	4096	001000
	.			

```
200                  .          SUBROUTINE TO WRITE RECORD FROM BUFFER
205                  .
210     2061    WRREC    LDX     ZERO              041030
215     2064    WLOOP    TD      OUTPUT            E02079
220     2067             JEQ     WLOOP             302064
225     206A             LDCH    BUFFER,X          509039
230     206D             WD      OUTPUT            DC2079
235     2070             TIX     LENGTH            2C1036
240     2073             JLT     WLOOP             382064
245     2076             RSUB                      4C0000
250     2079    OUTPUT   BYTE    X'05'             05
255                      END     FIRST
```

Figure 2.2 Program from Fig. 2.1 with object code.

- The translation of source program to object code requires to accomplish the following **basic functions:**

1. Convert mnemonic operation codes to their machine language equivalents. Eg: translate STL to 14.
2. Convert symbolic operands to their equivalent machine addresses. Eg: translate RETADR to 1033
3. Build the machine instructions in the proper format
4. Convert the data constants specified in the source program into their internal machine representations.- eg: translate EOF to 454F46
5. Write the object program and assembly listing.

- All these functions except the second one can be easily accomplished by sequential processing of the source program, one line at a time.

- Consider the following:

```
10    1000    FIRST     STL    RETADR     141033
--
--
--
--
95    1033    RETADR    RESW   1
```

The instruction(line 10) contains a forward reference, that is a reference to a label that is defined later. So can not process the statement . So most of the assemblers makes two passes. The first pass scans the program for labels and assign addresses. The second pass performs the actual translation.

- The assembler must process assembler directives. They are not translated into machine language. But they provide instructions to assembler itself.
- Finally the assembler must write the generated object code to some output device. The object program will later be loaded into memory for execution.

Object Program format

- The simple object program contains three types of records: Header record, Text record and end record.
- The header record contains the starting address and length. Text record contains the translated instructions and data of the program, together with an indication of the addresses where these are to be loaded. The end record marks the end of the object program and specifies the address where the execution is to begin.

The format of each record is as given below.

Header record:

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

- Figure 2.3 shows the object program corresponding to figure 2.2. The ^symbol is used to separate the fields.

```
H^COPY  ^001000^00107A
T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^00102A^0C1039^00102D
T^00101E^15^0C1036^482061^081033^4C0000^454F46^000003^000000
T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^549039^2C205E^38203F
T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^509039^DC2079^2C1036
T^002073^07^382064^4C0000^05
E^001000
```

Figure 2.3 Object program corresponding to Fig. 2.2.

- The assembler can be designed either as a single pass assembler or as a two pass assembler. The general description of both passes is as given below:
  - Pass 1 (define symbols)
    - Assign addresses to all statements in the program
    - Save the addresses assigned to all labels for use in Pass 2
    - Perform some processing of assembler directives, including those for address assignment, such as BYTE and RESW etc.
  - Pass 2 (assemble instructions and generate object program)
    - Assemble instructions (generate opcode and look up addresses)
    - Generate data values defined by BYTE, WORD
    - Perform processing of assembler directives not done during Pass 1
    - Write the object program and the assembly listing

Assembler Algorithms and Data structure

The simple assembler uses two major internal data structures: the operation Code Table (OPTAB) and the Symbol Table (SYMTAB).

OPTAB:

- It is used to lookup mnemonic operation codes and translates them to their machine language equivalents. In more complex assemblers the table also contains information about instruction format and length.
- In pass 1 the OPTAB is used to look up and validate the operation code in the source program. In pass 2, it is used to translate the operation codes to machine language. In simple SIC machine this process can be performed in either in pass 1 or in pass 2. But for machine like SIC/XE that has instructions of different lengths, we must search OPTAB in the first pass to find the instruction length for incrementing LOCCTR.
- In pass 2 we take the information from OPTAB to tell us which instruction format to use in assembling the instruction, and any peculiarities of the object code instruction.
- OPTAB is usually organized as a hash table, with mnemonic operation code as the key. The hash table organization is particularly appropriate, since it provides fast retrieval with a minimum of searching. Most of the cases the OPTAB is a static table- that is, entries are not normally added to or deleted from it. In such cases it is possible to design a special hashing function or other data structure to give optimum performance for the particular set of keys being stored.

SYMTAB:

- This table includes the name and value for each label in the source program, together with flags to indicate the error conditions (e.g., if a symbol is defined in two different places).
- During Pass 1: labels are entered into the symbol table along with their assigned address value as they are encountered. All the symbols address value should get resolved at the pass 1.
- During Pass 2: Symbols used as operands are looked up the symbol table to obtain the address value to be inserted in the assembled instructions.
- SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval. Since entries are rarely deleted, efficiency of deletion is the important criteria for optimization.

- Both pass 1 and pass 2 require reading the source program. Apart from this an intermediate file is created by pass 1 that contains each source statement together with its assigned address, error indicators, etc. This file is one of the inputs to the pass 2.

LOCCTR:

- Apart from the SYMTAB and OPTAB, this is another important variable which helps in the assignment of the addresses. LOCCTR is initialized to the beginning address mentioned in the START statement of the program. After each statement is processed, the length of the assembled instruction is added to the LOCCTR to make it point to the next instruction. Whenever a label is encountered in an instruction the LOCCTR value gives the address to be associated with that label.

The Algorithm for Pass 1:

  - The algorithm scans the first statement START and saves the operand field (the address) as the starting address of the program. Initializes the LOCCTR value to this address. This line is written to the intermediate line.
  - If no operand is mentioned the LOCCTR is initialized to zero. If a label is encountered, the symbol has to be entered in the symbol table along with its associated address value.
  - If the symbol already exists that indicates an entry of the same symbol already exists. So an error flag is set indicating a duplication of the symbol.

Pass 1:

```
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 {if 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}
```

- It next checks for the mnemonic code, it searches for this code in the OPTAB. If found then the length of the instruction is added to the LOCCTR to make it point to the next instruction.
- If the opcode is the directive WORD it adds a value 3 to the LOCCTR. If it is RESW, it needs to add the number of data word to the LOCCTR. If it is BYTE it adds the length of the constant in bytes to the LOCCTR, if RESB it adds number of bytes.
- If it is END directive then it is the end of the program it finds the length of the program by evaluating current LOCCTR – the starting address mentioned in the operand field of the END directive. Each processed line is written to the intermediate file.

The Algorithm for Pass 2:

```
Pass 2:

begin
   read first input line {fror
   if OPCODE = 'START' then
      begin
         write listing line
         read next input line
      end {if START}
   write Header record to obje
   initialize first Text recor
   while OPCODE ≠ 'END' do
      begin
         if this is not a cor
            begin
               search OPTAB
               if found then
                  begin
                     if there is a symbol
                        begin
                           search SYMTAB
                           if found then
                              store symb
                           else
                              begin
                                 store (
                                 set err
                              end
                        end {if symbol}
                     else
                        store 0 as operan
                     assemble the object (
                  end {if opcode found}
               else if OPCODE = 'BYTE' or
                  convert constant to obje
```

```
               if object code will not fit
                  begin
                     write Text record to
                     initialize new Text r
                  end
               add object code to Text reco
            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 {Pass 2}
```

- Here the first input line is read from the intermediate file. If the opcode is START, then this line is directly written to the list file.
- A header record is written in the object program which gives the starting address and the length of the program (which is calculated during pass 1). Then the first text record is initialized. Comment lines are ignored. In the instruction, for the opcode the OPTAB is searched to find the object code.
- If a symbol is there in the operand field, the symbol table is searched to get the address value for this which gets attached to the object code of the opcode. If the address not found then zero value is stored as operands address. An error flag is set indicating it as undefined. If symbol itself is not found then store 0 as operand address and the object code instruction is assembled.
- If the opcode is BYTE or WORD, then the constant value is converted

to its equivalent object code(for example, for character EOF, its equivalent hexadecimal value '454f46' is stored). If the object code cannot fit into the current text record, a new text record is created and the rest of the instructions object code is listed. The text records are written to the object program. Once the whole program is assemble and when the END directive is encountered, the End record is written.

Machine-Dependent Assembler Features:

In this section we consider the design and implementation of SIC/XE assembler.

- Instruction formats and addressing modes
- Program relocation.

Instruction formats and Addressing Modes

1. **Translation of Register to Register instructions**
   In this the assembler must simply convert the opcode to machine language and change each register to its numeric value.
   Eg:
   COMPR A, S A004
   (The opcode for COMPR is A0 , the number of register A is 0 and register S is 4.)
2. **Translation of Format 4 instructions**
   This format contains 20 bit address field . No displacement is calculated.
   Eg:
   CLOOP +JSUB RDREC 4B101036
   Here the opcode for JSUB instruction is 48 and the address of RDREC is 1036. Write the instruction format and set the bits n, i and e to 1.
   (If neither immediate nor indirect mode is used set the bits n and i to 1. Format 4 is identified by the prefix + . If format 4 is not specified assembler first attempts to translate the instruction using program counter relative addressing. If this is not possible, (because the required displacement is out of range), the assembler then attempts to use base relative addressing. If neither form of relative addressing is applicable and the extended format is not specified then the instruction can not be properly assembled. In this case the assembler must generate an error message.)

3. **Translation PC relative instructions**

In this format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value. In PC relative addressing made TA = disp + [PC]

disp = TA –[PC]

Eg:1

0000 FIRST STL RETADR 172

$(14)_{16}$ 1 1 0 0 1 0 $(02D)_{16}$

displacement= RETADR - PC = 30-3 = 2D

Eg: 2

0017 J CLOOP 3

$(3C)_{16}$ 1 1 0 0 1 0 $(FEC)_{16}$

displacement= CLOOP-PC= 6 - 1A= -14

4. **Translation of Base relative instructions**

In this format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value. In Base relative addressing made TA = disp + [B]

disp = TA –[B]

The displacement calculation process for base relative addressing is much the same as for PC relative addressing. In this the programmer must tell the assembler what the base register will contain during execution of the program so that assembler can compute displacements. This is done with the assembler directive BASE. For example, the statement BASE LENGTH informs the assembler that the base register will contain the address of LENGTH. The register B will contain this address until another BASE statement is encountered.

If the base register has to be used for another purpose the programmer must use NOBASE directive to inform the assembler that the contents of the base register is not used for addressing.

```
            LDB    #LEN
            BASE   LENG
104E        STCH   BUFF

( 54 )16     1 1 1 1 0 0    ( 003
 (54)         1 1 1 0 1 0    0036
displacement= BUFFER - B = (
```

5. Translation of Immediate addressing

In this no memory reference is involved. Convert the immediate operand into its internal representation and insert it into its internal representation.

Eg:

```
◆   0020       LDA    #3        010003

( 00 )16    0 1 0 0 0 0   ( 003 ) 16

◆   103C       +LDT   #4096     75101000

( 74 )16    0 1 0 0 0 1   ( 01000 ) 16
```

6. Translation involving indirect addressing

In this the displacement is computed to produce the target address.. Then bit n is set to 1. The example given below is indirect and PC relative.

Eg:

```
002A        J       @RETA

( 3C )16     1 0 0 0 1 0    ( 003 )
```

- TA=RETADR=0030
- TA=(PC)+disp=002D+0003

Program Relocation

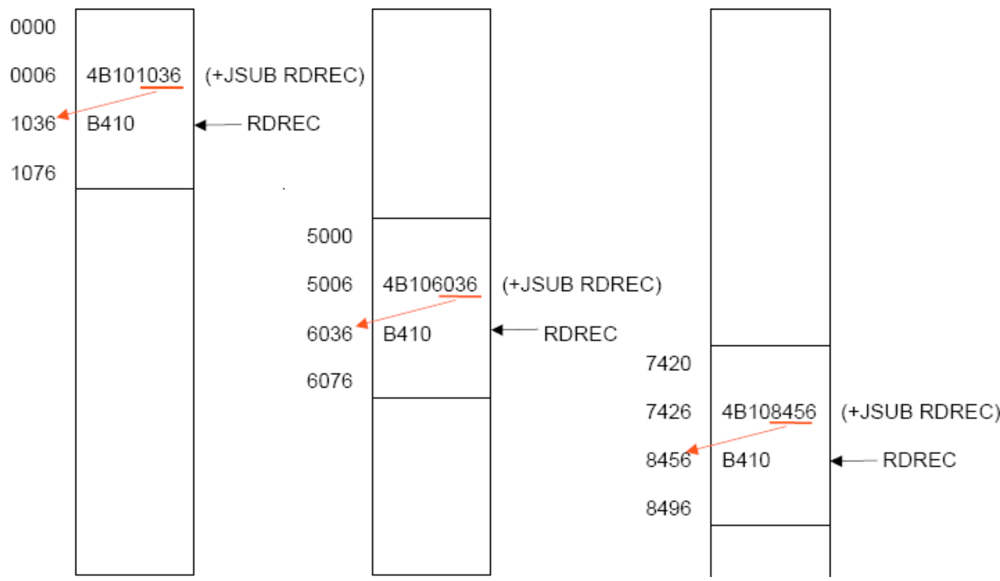
- Sometimes it is required to load and run several programs at the same time. The system must be able to load these programs wherever there is place in the memory. Therefore the exact starting is not known until the load time.

- Absolute Program- In this the address is mentioned during assembling itself. This is called *Absolute Assembly.*

  Eg: Consider the instruction:

  101B LDA THREE 00102D

- This statement says that the register A is loaded with the value stored at location 102D. Suppose it is decided to load and execute the program at location 2000 instead of location 1000.
- Then at address 102D the required value which needs to be loaded in the register A is no more available. The address also gets changed relative to the displacement of the program. Hence we need to make some changes in the address portion of the instruction so that we can load and execute the program at location 2000.
- Apart from the instruction which will undergo a change in their operand address value as the program load address changes. There exist some parts in the program which will remain same regardless of where the program is being loaded.
- Since assembler will not know actual location where the program will get loaded, it cannot make the necessary changes in the addresses used in the program. However, the assembler identifies for the loader those parts of the program which need modification.
- An object program that has the information necessary to perform this kind of modification is called the **relocatable program**.



- The above diagram shows the concept of relocation. Initially the program is loaded at location 0000. The instruction JSUB is loaded at location 0006.
- The address field of this instruction contains 01036, which is the address of the instruction labeled RDREC. The second figure shows that if the program is to be loaded at new location 5000.
- The address of the instruction JSUB gets modified to new location 6036. Likewise the third figure shows that if the program is relocated at location 7420, the JSUB instruction would need to be changed to 4B108456 that correspond to the new address of RDREC.
- The only part of the program that require modification at load time are those that specify direct addresses(format 4 instructions). The rest of the instructions need not be modified. The instructions which doesn't require modification are the ones that is not a memory address (immediate addressing) and PC-relative, Base-relative instructions.
- For an address label, its address is assigned relative to the start of the program (START 0). The assembler produces a *Modification record* to store the starting location and the length of the address field to be modified. The command for the loader must also be a part of the object program. The Modification has the following format:

Modification record

Col. 1	M
Col. 2-7	Starting location of the address field to be modified, relative to the beginning of the program (Hex)
Col. 8-9	Length of the address field to be modified, in half-bytes (Hex)

One modification record is created for each address to be modified The length is stored in half-bytes (4 bits) The starting location is the location of the byte containing the leftmost bits of the address field to be modified. If the field contains an odd number of half-bytes, the starting location begins in the middle of the first byte.

Eg: Consider the instruction

CLOOP +JSUB RDREC 4B101036

where RDREC is at the address 1036. The modification record for this instruction can be written as

M00000705

- There is one modification record for each address field that needs to be changed when the program is relocated(ie. For each format 4 instructions in the program).

Module-3

Machine-Dependent Assembler Features:

In this section we consider the design and implementation of SIC/XE assembler.

- Instruction formats and addressing modes
- Program relocation.

Instruction formats and Addressing Modes

1. **Translation of Register to Register instructions**
   In this the assembler must simply convert the opcode to machine language and change each register to its numeric value.
   Eg:
   COMPR A, S A004
   (The opcode for COMPR is A0 , the number of register A is 0 and register S is 4.)
2. **Translation of Format 4 instructions**
   This format contains 20 bit address field . No displacement is calculated.
   Eg:
   CLOOP +JSUB RDREC 4B101036

Here the opcode for JSUB instruction is 48 and the address of RDREC is 1036. Write the instruction format and set the bits n, i and e to 1.

(If neither immediate nor indirect mode is used set the bits n and i to 1. Format 4 is identified by the prefix + . If format 4 is not specified assembler first attempts to translate the instruction using program counter relative addressing. If this is not possible, (because the required displacement is out of range), the assembler then attempts to use base relative addressing. If neither form of relative addressing is applicable and the extended format is not specified then the instruction can not be properly assembled. In this case the assembler must generate an error message.)

3. Translation PC relative instructions

In this format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value. In PC relative addressing made TA = disp + [PC]

disp = TA –[PC]

Eg:1

0000 FIRST STL RETADR 17202D

$(14)_{16}$ 1 1 0 0 1 0 $(02D)_{16}$

- displacement= RETADR - PC = 30-3 = 2D

Eg: 2

0017 J CLOOP 3F2FEC

$(3C)_{16}$ 1 1 0 0 1 0 $(FEC)_{16}$

- displacement= CLOOP-PC= 6 - 1A= -14= FEC

4. Translation of Base relative instructions

In this format-3 instruction format is used. The instruction contains the opcode followed by a 12-bit displacement value. In Base relative addressing made TA = disp + [B]

disp = TA –[B]

The displacement calculation process for base relative addressing is much the same as for PC relative addressing. In this the programmer must tell the assembler what the base register will contain during execution of the program so that assembler can compute displacements. This is done with the assembler directive BASE. For example, the statement BASE LENGTH informs the assembler that the base register will contain the address of LENGTH. The register B will contain this address until another BASE statement is encountered.

If the base register has to be used for another purpose the programmer must use NOBASE directive to inform the assembler that the contents of the base register is not used for addressing.

LDB #LENGTH
BASE LENGTH
104E STCH BUFFER, X 57C003

$(54)_{16}$ 1 1 1 1 0 0 $(003)_{16}$
(54) 1 1 1 0 1 0 0036-1051= -$101B_{16}$
displacement= BUFFER - B = 0036 - 0033 = 3

5. **Translation of Immediate addressing**
In this no memory reference is involved. Convert the immediate operand into its internal representation and insert it into its internal representation.
Eg:

- 0020 LDA #3 010003

$(00)_{16}$ 0 1 0 0 0 0 $(003)_{16}$

- 103C +LDT #4096 75101000

$(74)_{16}$ 0 1 0 0 0 1 $(01000)_{16}$

6. **Translation involving indirect addressing**
In this the displacement is computed to produce the target address.. Then bit n is set to 1. The example given below is indirect and PC relative.

Eg:

002A J @RETADR 3E2003

$(3C)_{16}$ 1 0 0 0 1 0 $(003)_{16}$
- TA=RETADR=0030
- TA=(PC)+disp=002D+0003

Program Relocation

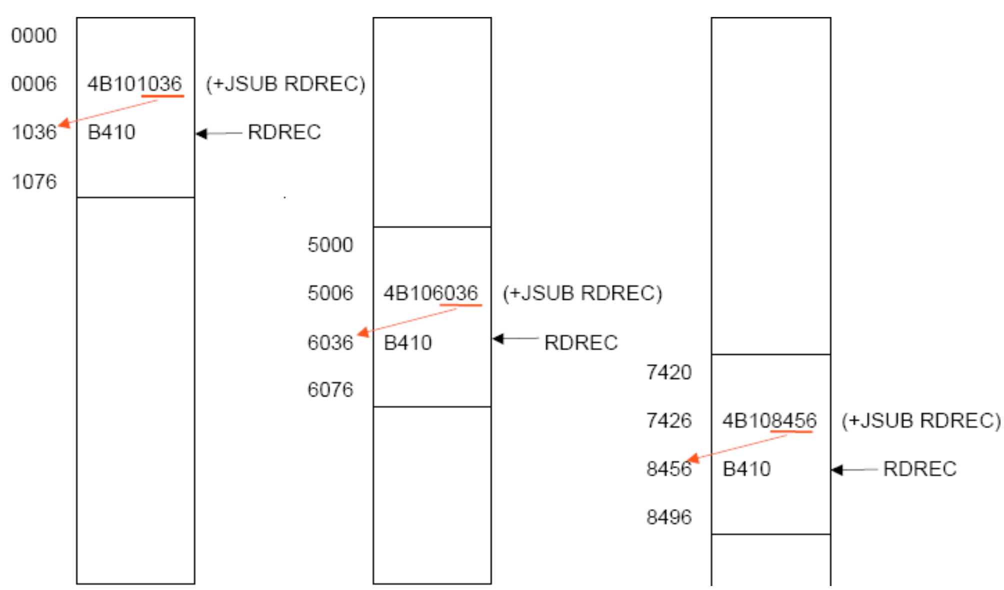
- Sometimes it is required to load and run several programs at the same time. The system must be able to load these programs wherever there is place in the memory. Therefore the exact starting is not known until the load time.

- Absolute Program- In this the address is mentioned during assembling itself. This is called *Absolute Assembly*.

  Eg: Consider the instruction:

  101B LDA THREE 00102D

- This statement says that the register A is loaded with the value stored at location 102D. Suppose it is decided to load and execute the program at location 2000 instead of location 1000.
- Then at address 102D the required value which needs to be loaded in the register A is no more available. The address also gets changed relative to the displacement of the program. Hence we need to make some changes in the address portion of the instruction so that we can load and execute the program at location 2000.
- Apart from the instruction which will undergo a change in their operand address value as the program load address changes. There exist some parts in the program which will remain same regardless of where the program is being loaded.
- Since assembler will not know actual location where the program will get loaded, it cannot make the necessary changes in the addresses used in the program. However, the assembler identifies for the loader those parts of the program which need modification.
- An object program that has the information necessary to perform this kind of modification is called the **relocatable program**.



- The above diagram shows the concept of relocation. Initially the program is loaded at location 0000. The instruction JSUB is loaded at location 0006.
- The address field of this instruction contains 01036, which is the address of the instruction labeled RDREC. The second figure shows that if the program is to be loaded at new location 5000.
- The address of the instruction JSUB gets modified to new location 6036. Likewise the third figure shows that if the program is relocated at location 7420, the JSUB instruction would need to be changed to 4B108456 that correspond to the new address of RDREC.
- The only part of the program that require modification at load time are those that specify direct addresses(format 4 instructions). The rest of the instructions need not be modified. The instructions which doesn't require modification are the ones that is not a memory address (immediate addressing) and PC-relative, Base-relative instructions.
- For an address label, its address is assigned relative to the start of the program (START 0). The assembler produces a *Modification record* to store the starting location and the length of the address field to be modified. The command for the loader must also be a part of the object program. The Modification has the following format:

Modification record

Col. 1 M

Col. 2-7 Starting location of the address field to be modified, relative to the

beginning of the program (Hex)

Col. 8-9 Length of the address field to be modified, in half-bytes (Hex)

One modification record is created for each address to be modified The length is stored in half-bytes (4 bits) The starting location is the location of the byte containing the leftmost bits of the address field to be modified. If the field contains an odd number of half-bytes, the starting location begins in the middle of the first byte.

Eg: Consider the instruction

CLOOP +JSUB RDREC 4B101036

where RDREC is at the address 1036. The modification record for this instruction can be written as

M00000705

- There is one modification record for each address field that needs to be changed when the program is relocated(ie. For each format 4 instructions in the program).

Machine-Independent features:

These are the features which do not depend on the architecture of the machine. Such features are more related to software than to machine architecture. These are:

- Literals
- Symbol defining statements
- Expressions
- Program blocks
- Control sections

Literals:

- It is easy for a programmer to write the value of a constant operand as part of the instruction that uses it.
- This avoids defining the constant elsewhere in the program and making a label for it. Such an operand is called a literal because the value is stated literally in the instruction.
- A literal is defined with a prefix = followed by a specification of the literal value.

Example:

001A ENDFIL LDA =C’EOF’ 032010

-

-

- The example above shows a 3-byte operand whose value is a character string EOF. The object code for the instruction is also mentioned. It shows the relative displacement value of the location where this value is stored. In the example the value is at location (002D) and hence the displacement value is (010). As another example the given statement below shows a 1-byte literal with the hexadecimal value ‘05’.

215 1062 WLOOP TD =X’05’ E32011

- **The difference between a constant defined as a literal and a constant defined as an immediate operand**- In case of literals the assembler generates the specified value as a constant at some other memory location. In immediate mode the operand value is assembled as part of the instruction itself. Example

  0020 LDA #03 010003

- All the literal operands used in a program are gathered together into one or more *literal pool*s. This is usually placed at the end of the program. 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.

  Eg: 1076 * =X'05' 05

- In some cases it is placed at some other location in the object program. An assembler directive LTORG is used. Whenever the LTORG is encountered, it creates a literal pool that contains all the literal operands used since the beginning of the program. The literal pool definition is done after LTORG is encountered. It is better to place the literals close to the instructions.

  LTORG

  002D * =C'EOF' 454F46

- **Recognizing Duplicate literals** – That is the same literal used in more than one place in a program and store only one copy of the data value. For example, the literal =X'05' is used in different instructions in a program, but only one data area with this value is created.
  - Duplicate literals can be identified by comparing character strings. Eg: X'05'
  - Otherwise, generated value can be compared. For eg: the literals =C'EOF' and =X'454F46' are identical operand values.
- The value of some literals depends on their location in the program. Literals referring to the current value of the location counter (denoted by the symbol *) . Such literals are useful for loading base registers.

  Eg: BASE *

  LDB *

  Such literal operands will have different values in different places of the program since they hold the current value of the locaton counter.

- **Handling of literals by the assembler** - A literal table is created for the literals which are used in the program. The literal table contains the literal name, operand value and length and the address assigned to the operand. The literal table is usually created as a hash table using the literal name or value as the key.

- **During Pass-1:**The literal encountered is searched in the literal table. If the literal already exists, no action is taken; if it is not present, the literal is added to the LITTAB (leaving the address unassigned. When Pass 1 encounters a LTORG statement or the end of the program, the assembler makes a scan of the literal table. At this time each literal currently in the table is assigned an address. As addresses are assigned, the location counter is updated to reflect the number of bytes occupied by each literal.
- **During Pass-2:**The assembler searches the LITTAB for each literal encountered in the instruction and replaces it with its equivalent value.

Symbol-Defining Statements:

EQU Statement:

- Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values. The directive used for this **EQU** (Equate). The general form of the statement is

```
Symbol          EQU        value
```

- This statement defines the given symbol (i.e., entering in the SYMTAB) and assigning to it the value specified. The value can be a constant or an expression involving constants. One common usage is to define symbolic names that can be used to improve readability in place of numeric values. For example , instead of +LDT #4096 we can write

```
MAXLEN    EQU       4096
          +LDT      #MAXLEN
```

- When the assembler encounters EQU statement, it enters the symbol MAXLEN along with its value in the symbol table. During the assembly of LDT instruction the assembler searches the SYMTAB for its entry and its equivalent value as the operand in the instruction. The object code generated is the same for both the options discussed, but is easier to understand. If the maximum length is changed from 4096 to 1024, it is difficult to change if it is mentioned as an immediate value wherever required in the instructions. We have to scan the whole program and make changes wherever 4096 is used. If we mention this value in the instruction through the symbol defined by EQU, we may not have to search the whole program but change only the value of MAXLENGTH in the EQU statement (only once).
- Another common usage of EQU statement is **for defining values for the general- purpose registers**. The assembler can use the mnemonics for register usage like a-register A , X – index register and so on. But there are some instructions which requires numbers in place of names in the instructions. For example in the instruction RMO 0,1 instead of RMO A,X. The programmer can assign the numerical values to these registers using EQU directive.

```
A        EQU      0
X        EQU      1 and so on
```

These statements will cause the symbols A, X, L… to be entered into the symbol table with their respective values. An instruction RMO A, X would then be allowed. As another usage if in a machine that has many general purpose registers named as R1, R2,…, some may be used as base register, some may be used as accumulator. Their usage may change from one program to another. In this case we can define these requirement using EQU statements.

```
BASE     EQU      R1
```

```
INDEX     EQU   R2
COUNT     EQU   R3
```

- One restriction with the usage of EQU is whatever symbol occurs in the right hand side of the EQU should be predefined. For example, the following statement is not valid:

```
BETA      EQU      ALPHA
ALPHA     RESW     1
```

As the symbol ALPHA is assigned to BETA before it is defined. The value of ALPHA is not known.

ORG Statement:

- This directive can be used to indirectly assign values to the symbols. This assembler directive changes the value in the location counter. The directive is usually called ORG (for origin). Its general format is:

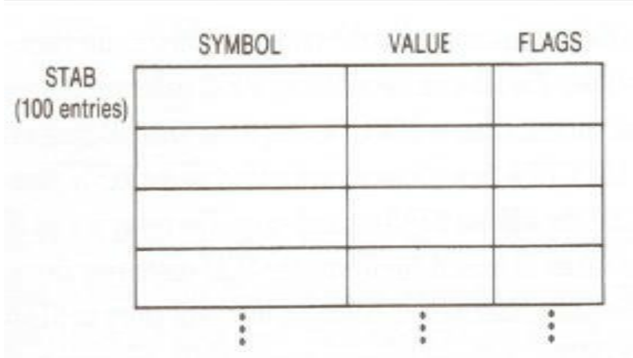
```
ORG       value
```

Where value is a constant or an expression involving constants and previously defined symbols. 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 is encountered. ORG is used to control assignment storage in the object program.

- ORG can be useful in label definition. Suppose we need to define a symbol table with the following structure:

```
SYMBOL    6 Bytes
VALUE     3 Bytes
FLAG      2 Bytes
```

The table looks like the one given below.



- The symbol field contains a 6-byte user-defined symbol; VALUE is a one-word representation of the value assigned to the symbol; FLAG is a 2-byte field specifies symbol type and other information. The space for the ttable can be reserved by the statement:

```
STAB      RESB      1100
```

If we want to refer to the entries of the table using indexed addressing, place the offset value of the desired entry from the beginning of the table in the index register. To refer to the fields SYMBOL, VALUE, and FLAGS individually, we need to assign the values first as shown below:

```
SYMBOL    EQU       STAB
VALUE     EQU       STAB+6
FLAGS     EQU       STAB+9
```

To retrieve the VALUE field from the table indicated by register X, we can write a statement:

```
LDA       VALUE, X
```

The same thing can also be done using ORG statement in the following way:

```
STAB      RESB   1100
          ORG    STAB
SYMBOL    RESB   6
VALUE     RESW   1
FLAG      RESB   2
          ORG    STAB+1100
```

The first statement allocates 1100 bytes of memory assigned to label STAB. In the second statement the ORG statement initializes the location counter to the value of STAB. Now the LOCCTR points to STAB. The next three lines assign appropriate memory storage to each of SYMBOL, VALUE and FLAG symbols. The last ORG statement reinitializes the LOCCTR to a new value after skipping the required number of memory for the table STAB (i.e., STAB+1100).

- While using ORG, the symbol occurring in the statement should be predefined as is required in EQU statement. For example for the sequence of statements below:

```
          ORG    ALPHA
BYTE1     RESB   1
BYTE2     RESB   1
BYTE3     RESB   1
          ORG
ALPHA     RESB   1
```

The sequence could not be processed as the symbol used to assign the new location counter value is not defined. In first pass, as the assembler would not know what value to assign to ALPHA, the other symbol in the next lines also could not be defined in the symbol table. This is a kind of problem of the forward reference.

Expressions:

- Assemblers also allow use of expressions in place of operands in the instruction. Each such expression must be evaluated to generate a single operand value or address. Assemblers generally arithmetic expressions formed according to the normal rules using arithmetic operators +, - *, /. Division is usually defined to produce an integer result.
- Individual terms may be constants, user-defined symbols, or special terms. The only special term used is * (the current value of location counter) which indicates the value of the next unassigned memory location. Thus the statement

```
BUFFEND     EQU          *
```

Assigns a value to BUFFEND, which is the address of the next byte following the buffer area. Some values in the object program are relative to the beginning of the program and some are absolute (independent of the program location, like constants).

- Expressions are classified as either absolute expression or relative expressions , neither absolute nor relative depending on the type of value they produce.

  – **Absolute Expressions:** The expression that uses only absolute terms is absolute expression. Absolute expression may contain relative term provided the relative terms occur in pairs with opposite signs for each pair. None of the relative terms enter into multiplication or division. Example:

```
MAXLEN      EQU          BUFEND-BUFFER
```

In the above instruction the difference in the expression gives a value that does not depend on the location of the program and hence gives an absolute value irrespective of the relocation of the program. The expression can have only absolute terms. Example:

```
MAXLEN      EQU          1000
```

  – **Relative Expressions:** All the relative terms except one can be paired . The remaining unpaired relative term must have a positive sign. None of the relative terms must enter into multiplication or division. A relative term represents some location within the program. Example:

STAB EQU OPTAB + (BUFEND – BUFFER)

– **Neither absolute nor relative:** Expressions that are legal are those expressions whose value remains meaningful when the program is relocated. Expressions that do not meet the conditions for either absolute or relative are neither absolute nor relative. They are considered as errors.

Eg: BUFEND + BUFFER, 100-BUFFER, 3*BUFFER

- **Handling the type of expressions:** to find the type of expression, we must keep track the type of symbols used. This can be achieved by defining the type in the symbol table against each of the symbol as shown in the table below:

Symbol	Type	Value
RETADR	R	0030
BUFFER	R	0036
BUFEND	R	1036
MAXLEN	A	1000

Program Blocks:

- Program blocks allow the generated machine instructions and data to appear in the object program in a different order by Separating blocks for storing code, data, stack, and larger data block.
- Program blocks refer to segments of code that are rearranged within a single object program unit.

- **Assembler Directive USE*:*** indicates which portion of the program belong to the various blocks.

USE [blockname]

- At the beginning, 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. Each program block may actually contain several separate segments of the source program. Assemblers rearrange these segments to gather together the pieces of each block and assign address. Separate the program into blocks in a particular order.Large buffer area is moved to the end of the object program. *Program readability is better*if data areas are placed in the source program close to the statements that reference them. In the example below three blocks are used :

Default: executable instructions

CDATA: all data areas that are less in length

CBLKS: all data areas that consists of larger blocks of memory

Example Code

```
(default) block     Block number
0000   0   COPY     START   0
0000   0   FIRST    STL     RETADR        172063
0003   0   CLOOP    JSUB    RDREC         4B2021
0006   0            LDA     LENGTH        032060
0009   0            COMP    #0            290000
000C   0            JEQ     ENDFIL        332006
000F   0            JSUB    WRREC         4B203B
0012   0            J       CLOOP         3F2FEE
0015   0   ENDFIL   LDA     =C'EOF'       032055
0018   0            STA     BUFFER        0F2056
001B   0            LDA     #3            010003
001E   0            STA     LENGTH        0F2048
0021   0            JSUB    WRREC         4B2029
0024   0            J       @RETADR       3E203F
0000   1            USE     CDATA         <- CDATA block
0000   1   RETADR   RESW    1
0003   1   LENGTH   RESW    1
0000   2            USE     CBLKS         <- CBLKS block
0000   2   BUFFER   RESB    4096
1000   2   BUFEND   EQU     *
1000       MAXLEN   EQU     BUFEND-BUFFER

                                          (default) block
0027   0   RDREC    USE   <-
0027   0            CLEAR   X             B410
0029   0            CLEAR   A             B400
002B   0            CLEAR   S             B440
002D   0            +LDT    #MAXLEN       75101000
0031   0   RLOOP    TD      INPUT         E32038
0034   0            JEQ     RLOOP         332FFA
0037   0            RD      INPUT         DB2032
003A   0            COMPR   A,S           A004
003C   0            JEQ     EXIT          332008
003F   0            STCH    BUFFER,X      57A02F
0042   0            TIXR    T             B850
0044   0            JLT     RLOOP         3B2FEA
0047   0   EXIT     STX     LENGTH        13201F
004A   0            RSUB                  4F0000
0006   1            USE     CDATA         <- CDATA block
0006   1   INPUT    BYTE    X'F1'         F1
```

```
                                                    (default) block
004D   0               USE
004D   0   WRREC       CLEAR     X             B410
004F   0               LDT       LENGTH        772017
0052   0   WLOOP       TD        =X'05'        E3201B
0055   0               JEQ       WLOOP         332FFA
0058   0               LDCH      BUFFER,X      53A016
005B   0               WD        =X'05'        DF2012
005E   0               TIXR      T             B850
0060   0               JLT       WLOOP         3B2FEF
0063   0               RSUB                    4F0000
0007   1               USE       CDATA  <-- CDATA block
                       LTORG
0007   1   *          =C'EOF'                  454F46
000A   1   *          =X'05'                   05
                       END       FIRST
```

- **How the assembler handles program blocks** –

Pass 1

– A separate location counter for each block is maintained.

– The location counter for a block is initialized to zero when the block is first started.

– The current value of the location counter is saved when switching to another block.

– The saved value is continued when resuming previous block.

– After pass 1 the symbol table will be having labels with block no along with address.(For absolute symbol there is no block number.)

– At the end of pass 1 latest value of location counter or each block gives the length of that block.

– Assembler constructs a block table that contains starting addresses and lengths of all blocks

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

Pass 2

– Code generation during pass2 the assembler needs the address relative to the start of the program. (not the start of the individual program block). Assembler adds the label address with its block starting address.

Pass1 algorithm of Program blocks

Pass2 algorithm for program blocks

- **Advantage-** Separation of programs into blocks has reduced the addressing problem. Since the larger buffer are is moved to the end of the object program extended format instructions need not be used. The use of program blocks has achieved the effect of rearranging the source statements without actually rearranging them. The loader will load the object program at the indicated address.

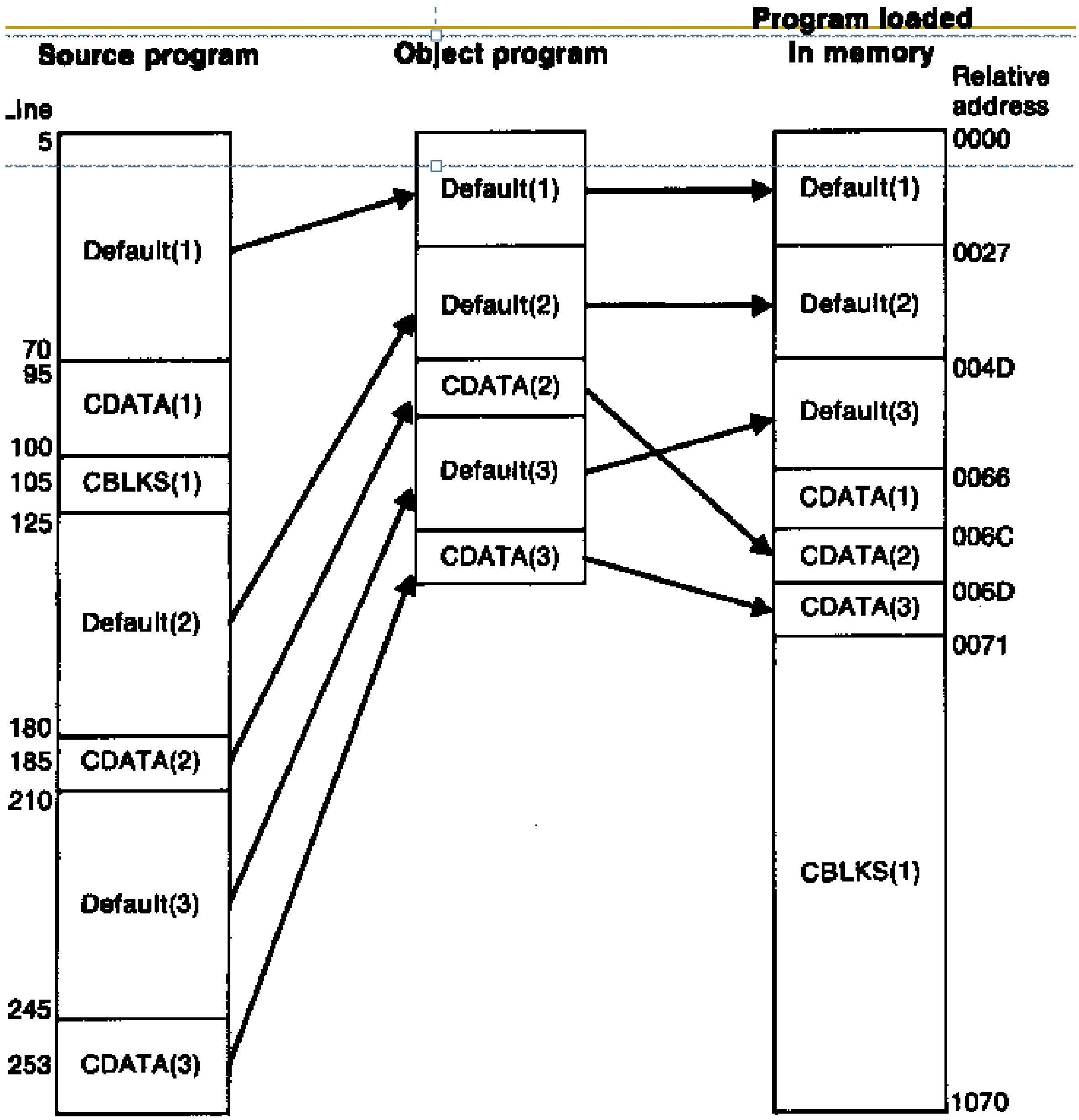


Fig:Program blocks traced through the assembly and loading processes

Pass1 of program blocks

```
begin
  block number = 0 LOCCTR[i] = 0 for all i
  read the first input line .
  if OPCODE = 'START' then
  begin
    write line to intermediate file
    read next input line
  end {if START}
  while OPCODE ≠ 'END' do
  if OPCODE = 'USE'
  begin
    if there is no OPEREND name then
      set block name as default
    else block name as OPERAND name
    if there is no entry for block name then
      insert (block name, block number ++) in block table
    i = block number for block name
    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[i]) into SYMTAB
        end {if symbol}
      Search OPTAB for OPCODE
      if found then
        add 3 instruction length to LOCCTR[i]
      else if OPCODE = 'WORD' then
        add 3 to LOCCTR[i]
      else if OPCODE = 'RESW' then
        add 3 * #[OPERAND] to LOCCTR[i]
      else if OPCODE = 'RESB' then
        add #[OPERAND] to LOCCTR[i]
      else if OPCODE = 'BYTE' then
      begin
        find length of constant in bytes
        add length to LOCCTR[i]
      end {if byte}
else
```

```
    Set error flag
    end {if not a comment}
  write line to intermediate file
  read Text input line
  end {while not END}
write last line to intermediate file
save Length[i] as LOCCTR[i] for all i
Address[o] = starting address
Address[i] = address(i - 1) + Length(i - 1)
              [for i = 1 to max(block number)]
insert(address[i], Length[i]) in block table for all i
end {Pass 1}
```

Pass2 of Program blocks

```
If OPCODE = 'USE' then
  set block number for block name with OPERAND field
  search SYMTAB for OPERAND
  store symbol value + address [block number] as operand address
end {Pass 2}
```

Control Sections:

- A *control section* is a part of the program that maintains its identity after assembly; each control section can be loaded and relocated independently of the others.
- Different control sections are most often used for subroutines or other logical subdivisions. The programmer can assemble, load, and manipulate each of these control sections separately.
- Because of this, there should be some means for linking control sections together. For example, instructions in one control section may refer to the data or instructions of other control sections.
- Since control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. Such references between different control sections are called *external references.*
- The assembler generates the information about each of the external references that will allow the loader to perform the required linking.
- When a program is written using multiple control sections, the beginning of each of the control section is indicated by an assembler directive
  - assembler directive: **CSECT**

    The syntax

    controlsectionname CSECT

  - separate location counter for each control section
- Control sections differ from program blocks in that they are handled separately by the assembler. Symbols that are defined in one control section may not be used directly another control section; they must be identified as external reference for the loader to handle. The external references are indicated by two assembler directives:
  - EXTDEF (external Definition): It is the statement in a control section, names symbols that are defined in this section but may be used by other control sections. Control section names do not need to be named in the EXTREF as they are automatically considered as external symbols.
  - EXTREF (external Reference): It names symbols that are used in this section but are defined in some other control section. The order in which these symbols are listed is not significant. The assembler must include proper information about the external references in the object

program that will cause the loader to insert the proper value where they are required.

Implicitly defined as an external symbol — first control section

```
COPY      START    0                      COPY FILE FROM INPUT TO OUTPUT
          EXTDEF   BUFFER,BUFEND,LENGTH
          EXTREF   RDREC,WRREC
FIRST     STL      RETADR                 SAVE RETURN ADDRESS
CLOOP     +JSUB    RDREC                  READ INPUT RECORD
          LDA      LENGTH                 TEST FOR EOF (LENGTH=0)
          COMP     #0
          JEQ      ENDFIL                 EXIT IF EOF FOUND
          +JSUB    WRREC                  WRITE OUTPUT RECORD
          J        CLOOP                  LOOP
ENDFIL    LDA      =C'EOF'                INSERT END OF FILE MARKER
          STA      BUFFER
          LDA      #3                     SET LENGTH = 3
          STA      LENGTH
          +JSUB    WRREC                  WRITE EOF
          J        @RETADR                RETURN TO CALLER
RETADR    RESW     1
LENGTH    RESW     1                      LENGTH OF RECORD
          LTORG
BUFFER    RESB     4096                   4096-BYTE BUFFER AREA
BUFEND    EQU      *
MAXLEN    EQU      BUFFEND-BUFFER
```

Implicitly defined as an external symbol — second control section

```
RDREC     CSECT
.
.         SUBROUTINE TO READ RECORD INTO BUFFER
          EXTREF   BUFFER,LENGTH,BUFFEND
          CLEAR    X                      CLEAR LOOP COUNTER
          CLEAR    A                      CLEAR A TO ZERO
          CLEAR    S                      CLEAR S TO ZERO
          LDT      MAXLEN
RLOOP     TD       INPUT                  TEST INPUT DEVICE
          JEQ      RLOOP                  LOOP UNTIL READY
          RD       INPUT                  READ CHARACTER INTO REGISTER A
          COMPR    A,S                    TEST FOR END OF RECORD (X'00')
          JEQ      EXIT                   EXIT LOOP IF EOR
          +STCH    BUFFER,X               STORE CHARACTER IN BUFFER
          TIXR     T                      LOOP UNLESS MAX LENGTH HAS
          JLT      RLOOP                    BEEN REACHED
EXIT      +STX     LENGTH                 SAVE RECORD LENGTH
          RSUB                            RETURN TO CALLER
INPUT     BYTE     X'F1'                  CODE FOR INPUT DEVICE
MAXLEN    WORD     BUFFEND-BUFFER
```

Implicitly defined as an external symbol

third control section

```
WRREC      CSECT
.
.          SUBROUTINE TO WRITE RECORD FROM BUFFER
.
            EXTREF    LENGTH,BUFFER
            CLEAR     X                  CLEAR LOOP COUNTER
           +LDT       LENGTH
WLOOP       TD       =X'05'              TEST OUTPUT DEVICE
            JEQ       WLOOP              LOOP UNTIL READY
           +LDCH      BUFFER,X           GET CHARACTER FROM BUFFER
            WD       =X'05'              WRITE CHARACTER
            TIXR      T                  LOOP UNTIL ALL CHARACTERS HAVE
            JLT       WLOOP                 BEEN WRITTEN
            RSUB                         RETURN TO CALLER
            END       FIRST
```

Handling External

Reference Case 1

15 0003 CLOOP +JSUB RDREC 4B100000

- The operand RDREC is an external reference.
  - The assembler has no idea where RDREC is
  - inserts an address of zero
  - can only use extended formatto provide enough room (that is, relative addressing for external reference is invalid)
- The assembler generates information for each external reference that will allow the loaderto perform the required linking.

Case 2

On line 107, BUFEND and BUFFER are defined in the same control section and the expression can be calculated immediately.

107 1000 MAXLEN EQU BUFEND-BUFFER

Case 3

```
190     0028    MAXLEN      WORD        BUFEND-BUFFER               000000
```

- There are two external references in the expression, BUFEND and BUFFER.
- The assembler inserts a value of zero
- passes information to the loader
- Add to this data area the address of BUFEND
- Subtract from this data area the address of BUFFER

Object Code for the example program:

```
0000   COPY     START    0
                EXTDEF   BUFFER,BUFFEND,LENGTH
                EXTREF   RDREC,WRREC
0000   FIRST    STL      RETADR              172027
0003   CLOOP   +JSUB     RDREC               4B100000   Case 1
0007            LDA      LENGTH              032023
000A            COMP    #0                   290000
000D            JEQ      ENDFIL              332007
0010           +JSUB     WRREC               4B100000
0014            J        CLOOP               3F2FEC
0017   ENDFIL   LDA     =C'EOF'              032016
001A            STA      BUFFER              0F2016
001D            LDA     #3                   010003
0020            STA      LENGTH              0F200A
0023           +JSUB     WRREC               4B100000
0027            J       @RETADR              3E2000
002A   RETADR   RESW     1
002D   LENGTH   RESW     1
                LTORG
0030   *       =C'EOF'                       454F46
0033   BUFFER   RESB     4096
1033   BUFEND   EQU      *
1000   MAXLEN   EQU      BUFEND-BUFFER           case 2
```

```
0000      RDREC     CSECT
          .         SUBROUTINE TO READ RECORD INTO BUFFER
          .
                    EXTREF   BUFFER,LENGTH,BUFEND
0000                CLEAR    X                          B410
0002                CLEAR    A                          B400
0004                CLEAR    S                          B440
0006                LDT      MAXLEN                     77201F
0009      RLOOP     TD       INPUT                      E3201B
000C                JEQ      RLOOP                      332FFA
000F                RD       INPUT                      DB2015
0012                COMPR    A,S                        A004
0014                JEQ      EXIT                       332009
0017               +STCH     BUFFER,X                   57900000
001B                TIXR     T                          B850
001D                JLT      RLOOP                      3B2FE9
0020      EXIT     +STX      LENGTH                     13100000
0024                RSUB                                4F0000
0027      INPUT     BYTE     X'F1'                      F1
0028      MAXLEN    WORD     BUFFEND-BUFFER             000000     Case 3
```

```
0000      WRREC     CSECT

          .         SUBROUTINE TO WRITE RECORD FROM BUFFER
          .
                    EXTREF   LENGTH,BUFFER
0000                CLEAR    X                          B410
0002               +LDT      LENGTH                     77100000
0006      WLOOP     TD       =X'05'                     E32012
0009                JEQ      WLOOP                      332FFA
000C               +LDCH     BUFFER,X                   53900000
0010                WD       =X'05'                     DF2008
0013                TIXR     T                          B850
0015                JLT      WLOOP                      3B2FEE
0018                RSUB                                4F0000
                    END      FIRST
001B      *        =X'05'                               05
```

The assembler must also include information in the object program that will cause the loader to insert the proper value where they are required. The assembler maintains two new record in the object code and a changed version of modification record.

Define record (EXTDEF)

Col. 1 D

- Col. 2-7 Name of external symbol defined in this control section
- Col. 8-13 Relative address within this control section (hexadecimal)
- Col.14-73 Repeat information in Col. 2-13 for other external symbols

Refer record (EXTREF)

- Col. 1 R
- Col. 2-7 Name of external symbol referred to in this control section
- Col. 8-73 Name of other external reference symbols

Modification record

- Col. 1 M
- Col. 2-7 Starting address of the field to be modified (hexadecimal)
- Col. 8-9 Length of the field to be modified, in half-bytes (hexadecimal)
- Col.11-16 External symbol whose value is to be added to or subtracted from the indicated field

A define record gives information about the external symbols that are defined in this control section, i.e., symbols named by EXTDEF.A refer record lists the symbols that are used as external references by the control section, i.e., symbols named by EXTREF.

The new items in the modification record specify the modification to be performed: adding or subtracting the value of some external symbol. The symbol used for modification may be defined either in this control section or in another section.

The object program is shown below. There is a separate object program for each of the control sections. In the *Define Record* and *refer record* the symbols named in EXTDEF and EXTREF are included.

COPY

```
HCOPY  000000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC WRREC
T0000001D1720274B1000000320232900003320074B1000003F2FEC0320160F2016
T00001D0D0100030F200A4B1000003E2000
T00003003454F46
M00000405+RDREC
M00001105+WRREC
M00002405+WRREC
E000000
```

RDREC

```
HRDREC 00000000002B
RBUFFERLENGTHBUFEND
T0000001DB410B400B44077201FE3201B332FFADB2015A00433200957900000B850
T00001D0E3B2FE9131000004F0000F1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFEND
M00002806-BUFFER
E
```

BUFEND - BUFFER

WRREC

```
HWRREC 00000000001C
RLENGTHBUFFER
T0000001CB410771000000E3201232FFA53900000DF2008B8503B2FEE4F000005
M00000305+LENGTH
M00000D05+BUFFER
E
```

- In the case of *Define,* the record also indicates the relative address of each external symbol within the control section.For EXTREF symbols, no address information is available. These symbols are simply named in the *Refer record.*
- **Handling Expressions in Multiple Control Sections**: The existence of multiple control sections that can be relocated independently of one another makes the handling of expressions complicated. It is required that in an expression that all the relative terms be paired (for absolute expression), or that all except one be paired (for relative expressions).
- When it comes in a program having multiple control sections then we have an extended restriction that:
    - Both terms in each pair of an expression must be within the same control section

      If two terms represent relative locations within the same control section , their difference is an absolute value (regardless of where the control section is located.

      Legal: BUFEND-BUFFER (both are in the same control section)

    - If the terms are located in different control sections, their difference has a value that is unpredictable.

      Illegal: RDREC-COPY (both are of different control section) it is the difference in the load addresses of the two control sections. This value depends on the way run-time storage is allocated; it is unlikely to be of any use.
- **How to enforce this restriction**
    - When an expression involves external references, the assembler cannot determine whether or not the expression is legal.
    - The assembler evaluates all of the terms it can, combines these to form an initial expression value, and generates Modification records.
    - The loader checks the expression for errors and finishes the evaluation.

Assembler Design Options

- There are two design options or the assembler.
  - One pass assembler: is used when it is necessary to avoid a second pass over the source program.
  - Multipass Assembler: allows an assembler to handle forward references.

One-Pass Assembler

The main problem in designing the assembler using single pass was to resolve forward references. We can avoid to some extent the forward references by:

- Eliminating forward reference to data items, by defining all the storage reservation statements at the beginning of the program rather at the end.
- Unfortunately, forward reference to labels on the instructions cannot be avoided. (forward jumping)
- To provide some provision for handling forward references by prohibiting forward references to data items.

There are two types of one-pass assemblers:

- One that produces object code directly in memory for immediate execution **(Load- and-go assemblers).**
- The other type produces **the usual kind of object code** for later execution.

Load-and-Go Assembler

- Load-and-go assembler generates their object code in memory for immediate execution.
- No object program is written out, no loader is needed.
- It is useful in a system with frequent program development and testing
  - The efficiency of the assembly process is an important consideration.

- Programs are re-assembled nearly every time they are run; efficiency of the assembly process is an important consideration.

Line	Loc	Source statement			Object code
0	1000	COPY	START	1000	
1	1000	EOF	BYTE	C'EOF'	454F46
2	1003	THREE	WORD	3	000003
3	1006	ZERO	WORD	0	000000
4	1009	RETADR	RESW	1	
5	100C	LENGTH	RESW	1	
6	100F	BUFFER	RESB	4096	
9		.			
10	200F	FIRST	STL	RETADR	141009
15	2012	CLOOP	JSUB	RDREC	48203D
20	2015		LDA	LENGTH	00100C
25	2018		COMP	ZERO	281006
30	201B		JEQ	ENDFIL	302024
35	201E		JSUB	WRREC	482062
40	2021		J	CLOOP	302012
45	2024	ENDFIL	LDA	EOF	001000
50	2027		STA	BUFFER	0C100F
55	202A		LDA	THREE	001003
60	202D		STA	LENGTH	0C100C
65	2030		JSUB	WRREC	482062
70	2033		LDL	RETADR	081009
75	2036		RSUB		4C0000
110		.			

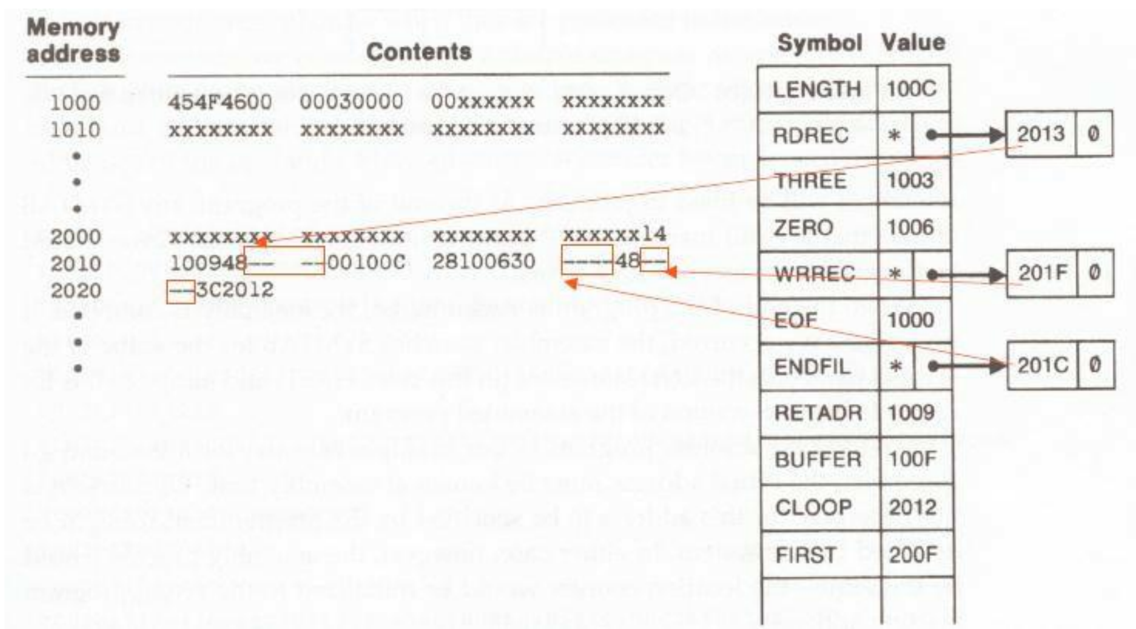
Forward Reference in One-Pass Assemblers: In load-and-Go assemblers when a forward reference is encountered :

- Omits the operand address if the symbol has not yet been defined
- Enters this undefined symbol into SYMTAB and indicates that it is undefined
- Adds the address of this operand address to a list of forward references associated with the SYMTAB entry
- When the definition for the symbol is encountered, scans the reference list and inserts the address.
- At the end of the program, reports the error if there are still SYMTAB entries indicated undefined symbols.
- For Load-and-Go assembler
    - Search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error

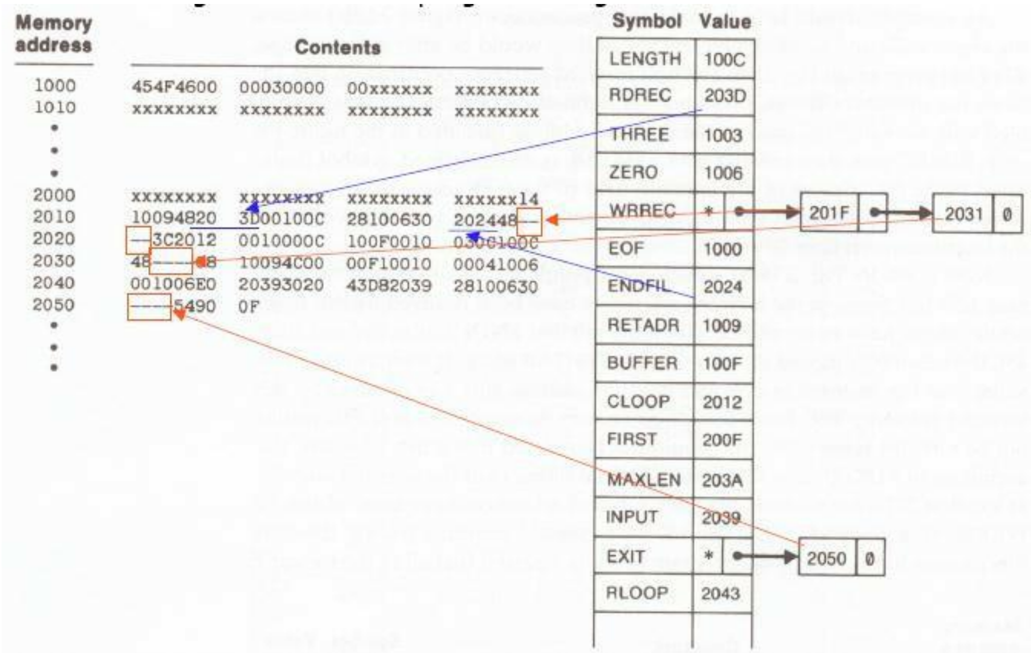
After Scanning line 40 of the program:

40 2021 J` CLOOP 302012

The status is that upto this point the symbol RREC is referred once at location 2013, ENDFIL at 201F and WRREC at location 201C. None of these symbols are defined. The figure shows that how the pending definitions along with their addresses are included in the symbol table.



The status after scanning line 160, which has encountered the definition of RDREC and ENDFIL is as given below:



One-Pass Assembler that generates object code:

- If the operand contains an undefined symbol, use 0 as the address and write the Text record to the object program.
- Forward references are entered into lists as in the load-and-go assembler.
- When the definition of a symbol is encountered, the assembler generates another Text record with the correct operand address of each entry in the reference list.
- When loaded, the incorrect address 0 will be updated by the latter Text record containing the symbol definition.

```
H^COPY  ^001000^00107A
T^001000^09^454F46^000003^000000
T^00200F^15^141009^480000^00100C^281006^300000^480000^3C2012
T^00201C^02^2024
T^002024^19^001000^0C100F^001003^0C100C^480000^081009^4C0000^F1^001000
T^002013^02^203D
T^00203D^1E^041006^001006^E02039^302043^D82039^281006^300000^54900F^2C203A^382043
T^002050^02^205B
T^00205B^07^10100C^4C0000^05
T^00201F^02^2062
T^002031^02^2062
T^002062^18^041006^E02061^302065^50900F^DC2061^2C100C^382065^4C0000
E^00200F
```

Algorithm for one pass assembler

```
begin
  read first input line
  if OPCODE = 'START' then
    begin
      save #[OPERAND] as starting address
      initialize LOCCTR as starting address
      read next input line
    end {if START}
  else
    initialize LOCCTR to 0
while OPCODE ≠ 'END' do
    begin
      if there 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
              begin
                  if symbol value as null
                  set symbol value as LOCCTR and search
                     the linked list with the corresponding
                     operand
                  PTR addresses and generate operand
                     addresses as corresponding symbol
                     values
                  set symbol value as LOCCTR in symbol
                     table and delete the linked list
              end
            else
              insert (LABEL, LOCCTR) into SYMTAB
          end
            search OPTAB for OPCODE
              if found then
                begin
                   search SYMTAB for OPERAND address
              if found then
                if symbol value not equal to null then
                  store symbol value as OPERAND address
                else
                  insert at the end of the linked list
                    with a node with address as LOCCTR
                else
                  insert (symbol name, null)
```

```
          add 3 to LOCCTR
        end
        else if OPCODE = 'WORD' then
          add 3 to LOCCTR & convert comment to
            object code
        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
            convert constant to object code
          end
      if object code will not fit into current
        text record then
        begin
          write text record to object program
          initialize new text record
        end
      add object code to Text record
    end
   write listing line
   read next input line
  end
 write last Text record to object program
 write End record to object program
 write last listing line
end {Pass 1}
```

MultiPass Assembler:

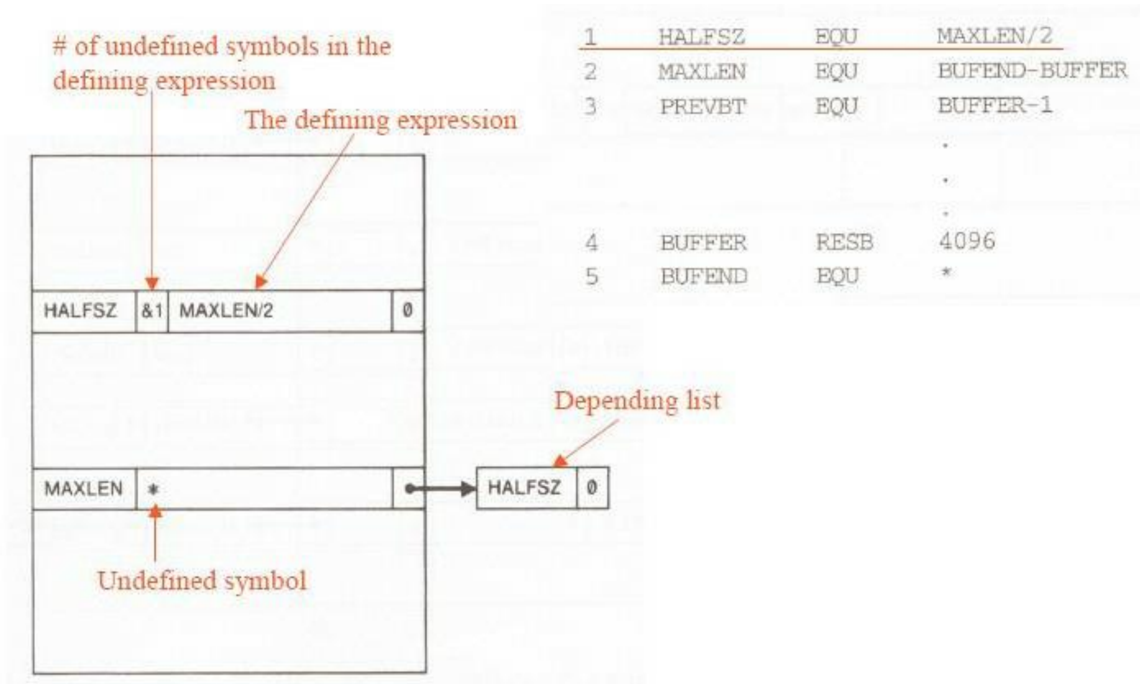
- For a two pass assembler, in EQU assembler directive we required that any symbol on the right hand side be defined previously in the program. This is because o the two pass.If multipass is possible this restriction can be avoided. Eg:

```
ALPHA   EQU   BETA
BETA    EQU   DELTA
DELTA   RESW 1
```

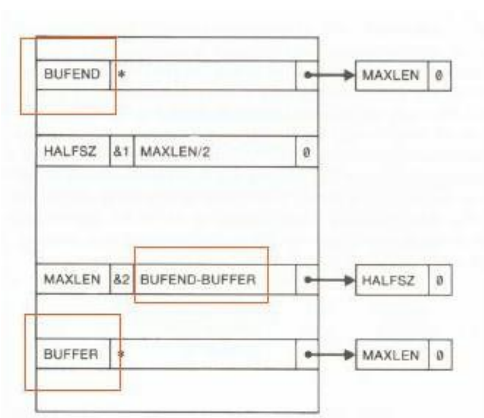
Working of Multipass Assembler:

- A multipass assembler can make as many passes as needed to process the definition of symbols.
- For a forward reference in symbol definition, we store in the SYMTAB:
  - The symbol name
  - The defining expression
  - The number of undefined symbols in the defining expression
- The undefined symbol (marked with a flag *) associated with a list of symbols depend on this undefined symbol.
- When a symbol is defined, we can recursively evaluate the symbol expressions depending on the newly defined symbol.

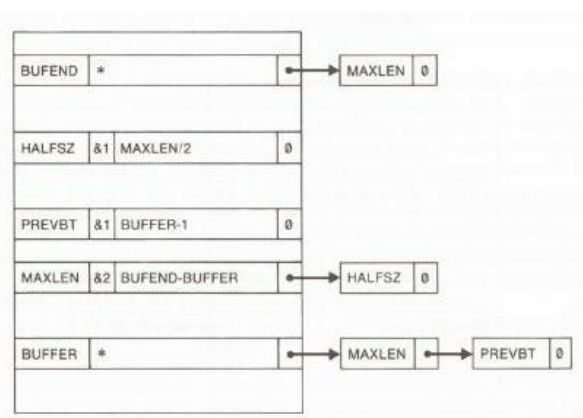
Multi-Pass Assembler Example Program



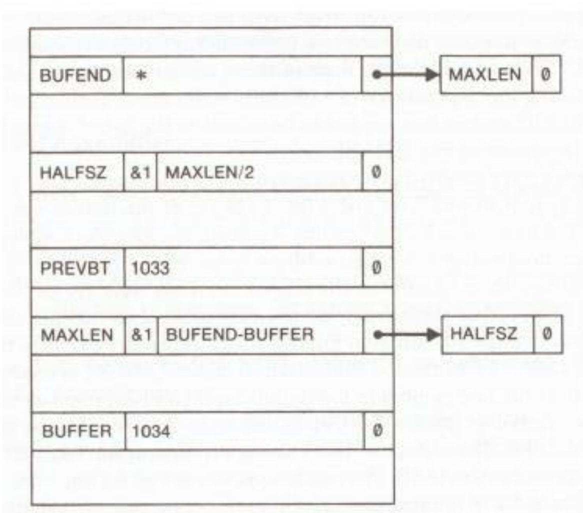
Multi-Pass Assembler : Example for forward reference in Symbol Defining Statements:



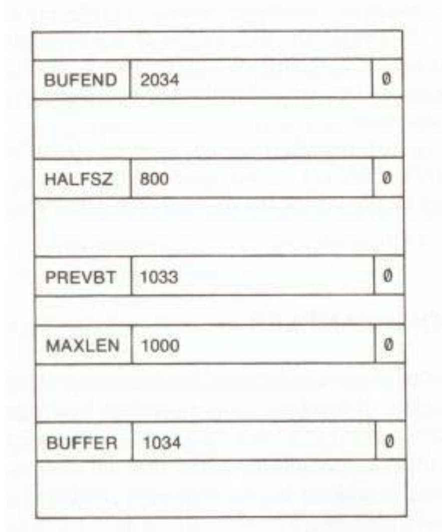
2 MAXLEN EQU BUFEND-BUFFER



3 PREVBT EQU BUFFER-1



4 BUFFER RESB 4096



5 BUFEND EQU *

Implementation Example: MASM ASSEMBLER

- Microsoft MASM assembler works for Petium and other ×86 systems.
- In this system memory is considered as segments.
- An MASM assembly language program is written as collection of segments. Each segment is defined as belonging to a particular class, corresponding to its contents. Commonly used classes are CODE, DATA, CONST and STACK
- During program execution the segments are addressed via the ×86 segment registers. Code segments are addressed using register CS and stack segments are addressed using register SS. These segment registers are 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 statement of the program. Register SS is to indicate the last stack segment processed by the loader.
- Data segments (including constant segments) are normally addressed using DS, ES, or GS.
- By default the assembler assumes that all references to data segments use register DS. This assumption can be changed by the assembler directive ASSUME.
  ASSUME ES: DATASEG2
- Registers DS, ES, FS and GS must be loaded by the program before they can be used to address data segments. Eg:
  MOV AX, DATASEG2

MOV ES, AX

Would set ES to indicae the data segment DATASEG2

- Jump instructions are assembled in two different ways, depending on whether the target of the jump is in the same code segment (near jump) or in a different code segment(far jump).
- The length of the assembled instruction depends on the operands that are used. An operand that specifies a memory location may take varying amounts of space in the instruction depending upon the location o the operand.
- First pass of the ×86 assembler must analyze the operands of an instruction, in addition to looking at the opcode.
- Segments in a MASM source program can be written in more than one place using the assembler directive SEGMENT.
- References between segments that are assembled together are automatically handled by the assembler.
- MASM can also produce an instruction timing listing that shows the number of clock cycles required to execute each machine instruction.

MODULE- 4

LOADERS AND LINKERS

Introduction

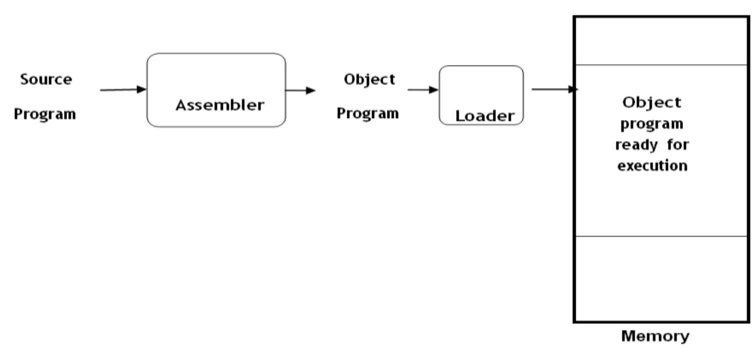
The Source Program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution. This conversion either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies addresses in primary memory where these items are to be loaded for execution.

This contains the following three processes, and they are,

- **Loading** - which allocates memory location and brings the object program into memory for execution - (Loader)
- **Linking**- which combines two or more separate object programs and supplies the information needed to allow references between them - (Linker)
- **Relocation** - which modifies the object program so that it can be loaded at an address different from the location originally specified - (Relocating Loader)

4. 1 Basic Loader Functions:

- A loader is a system software that performs the loading function. It brings object program into memory and starts its execution. The role of loader is as shown in the figure.



Type of Loaders

The different types of loaders are, absolute loader, bootstrap loader, relocating loader (relative loader), and, linking loader. The following sections discuss the functions and design of all these types of loaders.

4.1.1 Design of Absolute Loader:

- The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program. Linking and relocation is not done.
- The algorithm for this type of loader is given here.

Begin

read Header record

verify program name and length

read first Text record

while record type is != 'E' **do**

begin

{if object code is in character form, convert into internal representation}

move object code to specified location in memory

read next object program record

end

jump to address specified in End record

end

Algorithm for Absolute loader

- In this all functions are done in a single pass. The header is checked to veriy that the correct program has been presented for loading. As each text record is read the object code it contains is moved to the indicated address in memory. When the End record is encountered the loader jumps to the specified address to begin execution of the loaded program.

```
H^COPY  ^001000^00107A
T^001000^1E^141033^482039^001036^281030^301015^482061^3C1003^00102A^0C1039^00102D
T^00101E^15^0C1036^482061^081033^4C0000^454F46^000003^000000
T^002039^1E^041030^001030^E0205D^30203F^D8205D^281030^302057^549039^2C205E^38203F
T^002057^1C^101036^4C0000^F1^001000^041030^E02079^302064^509039^DC2079^2C1036
T^002073^07^382064^4C0000^05
E^001000
```

(a) Object program

Memory address	Contents			
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
0010	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮
0FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
1000	14103348	20390010	36281030	30101548
1010	20613C10	0300102A	0C103900	102D0C10
1020	36482061	0810334C	0000454F	46000003
1030	000000xx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮
2030	xxxxxxxx	xxxxxxxx	xx041030	001030E0
2040	205D3020	3FD8205D	28103030	20575490
2050	392C205E	38203F10	10364C00	00F10010
2060	00041030	E0207930	20645090	39DC2079
2070	2C103638	20644C00	0005xxxx	xxxxxxxx
2080	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx
⋮	⋮	⋮	⋮	⋮

← COPY

(b) Program loaded in memory

- The figure (b) shows the representation of program from figure (a) after loading.
- In the object program each byte of assembled code is given using its hexadecimal representation in character form.

- In the object program , each byte of assembled code is given using its hexadecimal representation in character form. For example, the machine opcode for an STL instruction would be represented by the pair of characters "1" and "4". When these are read by the loader , they will occupy two bytes of memory. This opcode must be stored in a single byte with hexa decimal value 14. Thus each pair of bytes from the object program must be packed together into one byte during loading.

4.1.2 A simple bootstrap loader

- When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer-- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 80.

- Working: Consider the bootstrap loader for SIC/XE. The bootstrap loader begins at address 0 in the memory. It loads the OS starting at address 80. Each byte of object code to be loaded is represented on device F1 as two hexa decimal digits(Text record) . Object code is loaded to consecutive memory locations starting at address 80. After all the object code from device F1 has been loaded the bootstrap jumps to the address 80.

- GETC subroutine – This subroutine reads one character from device F1 and converts from ASCII to hex. This is done by subtracing 48 if the character is from 0 to 9. For characters A to F subtract 55. Subroutine jumps to address 80 when end of line is reached.

- Main loop of the bootstrap loader- This 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(represents one byte of object code). These two hexadecimal values are combined to a single byte by shifting the first one left by 4 bit positions and adding the second to it. The resulting byte is stored at address currently in register X

The algorithm for the bootstrap loader is as follows

```
BOOT     START    0        BOOTSTRAP LOADER FOR SIC/XE
.
. THIS BOOTSTRAP READS OBJECT CODE FROM DEVICE F1 AND ENTERS IT
. INTO MEMORY STARTING AT ADDRESS 80 (HEXADECIMAL). AFTER ALL OF
. THE CODE FROM DEVF1 HAS BEEN SEEN ENTERED INTO MEMORY, THE
. BOOTSTRAP EXECUTES A JUMP TO ADDRESS 80 TO BEGIN EXECUTION OF
. THE PROGRAM JUST LOADED.  REGISTER X CONTAINS THE NEXT ADDRESS
. TO BE LOADED.
.
         CLEAR    A        CLEAR REGISTER A TO ZERO
         LDX      #128     INITIALIZE REGISTER X TO HEX 80
LOOP     JSUB     GETC     READ HEX DIGIT FROM PROGRAM BEING LOADED
         RMO      A,S      SAVE IN REGISTER S
         SHIFTL   S,4      MOVE TO HIGH-ORDER 4 BITS OF BYTE
         JSUB     GETC     GET NEXT HEX DIGIT
         ADDR     S,A      COMBINE DIGITS TO FORM ONE BYTE
         STCH     0,X      STORE AT ADDRESS IN REGISTER X
         TIXR     X,X      ADD 1 TO MEMORY ADDRESS BEING LOADED
         J        LOOP     LOOP UNTIL END OF INPUT IS REACHED
```

```
.
. SUBROUTINE TO READ ONE CHARACTER FROM INPUT DEVICE AND
. CONVERT IT FROM ASCII CODE TO HEXADECIMAL DIGIT VALUE. THE
. CONVERTED DIGIT VALUE IS RETURNED IN REGISTER A. WHEN AN
. END-OF-FILE IS READ, CONTROL IS TRANSFERRED TO THE STARTING
. ADDRESS (HEX 80).
.
GETC     TD       INPUT     TEST INPUT DEVICE
         JEQ      GETC      LOOP UNTIL READY
         RD       INPUT     READ CHARACTER
         COMP     #4        IF CHARACTER IS HEX 04 (END OF FILE),
         JEQ      80           JUMP TO START OF PROGRAM JUST LOADED
         COMP     #48       COMPARE TO HEX 30 (CHARACTER '0')
         JLT      GETC      SKIP CHARACTERS LESS THAN '0'
         SUB      #48       SUBTRACT HEX 30 FROM ASCII CODE
         COMP     #10       IF RESULT IS LESS THAN 10, CONVERSION IS
         JLT      RETURN       COMPLETE. OTHERWISE, SUBTRACT 7 MORE
         SUB      #7           (FOR HEX DIGITS 'A' THROUGH 'F')
RETURN   RSUB               RETURN TO CALLER
INPUT    BYTE     X'F1'     CODE FOR INPUT DEVICE
         END      LOOP
```

Figure 3.3 Bootstrap loader for SIC/XE.

4.2Machine-Dependent Loader Features

- Absolute loader is simple and efficient, but the scheme has potential disadvantages. One of the most disadvantage is the programmer has to specify the actual starting address, from where the program to be loaded. This does not create difficulty, if one program to run, but not for several programs. Further it is difficult to use subroutine libraries efficiently.
- This needs the design and implementation of a more complex loader. The loader must provide program relocation and linking, as well as simple loading functions. This depends on machine architecture.

4.2.1Relocation(Relocating loader)

- Loaders that allow program relocation are called relocating loaders.
- There are two methods for providing relocation as part of the object program.
  - Modification record
  - Bit masking

Modification Record

- A modification record is used to describe each part of the object code that must be changed when the program is relocated.
- Consider SIC/XE programs, Most of the instructions in this program uses relative or immediate addressing. So modification not required. Only format 4 instructions require modification
- Each modification record specifies the starting address and length of the field to be modified and what modification to be performed.(adding the start address).

```
H^COPY  ^000000^001077
T^000000^1D^17202D^69202D^4B101036^032026^290000^332007^4B10105D^3F2FEC^032010
T^00001D^13^0F2016^010003^0F200D^4B10105D^3E2003^454F46
T^001036^1D^B410^B400^B440^75101000^E32019^332FFA^DB2013^A004^332008^57C003^B850
T^001053^1D^3B2FEA^134000^4F0000^F1^B410^774000^E32011^332FFA^53C003^DF2008^B850
T^001070^07^3B2FEF^4F0000^05
M^000007^05+COPY
M^000014^05+COPY
M^000027^05+COPY
E^000000
```

Figure 3.5 Object program with relocation by Modification records.

Algorithm for SIC/XE relocation loader

```
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
end
```

Bitmasking

- In SIC program relative addressing is not used. So every instruction needs modification. We can not write modification records for all instructions.
- So relocation bits are used. Each instruction object code is associated with relocation bit.
- Relocation bits for each text record is written together into bitmask after the length using 3 hexadecimal digits.(12 bits)
- Example:

```
H^COPY  ^000000^00107A
T^000000^1E^FFC^140033^481039^000036^280030^300015^481061^3C0003^00002A^0C0039^00002D
T^00001E^15^E00^0C0036^481061^080033^4C0000^454F46^000003^000000
T^001039^1E^FFC^040030^000030^E0105D^30103F^D8105D^280030^301057^548039^2C105E^38103F
T^001057^0A^800^100036^4C0000^F1^001000
T^001061^19^FE0^040030^E01079^301064^508039^DC1079^2C0036^381064^4C0000^05
E^000000
```

Figure 3.7 Object program with relocation by bit mask.

- If the relocation bit is 1 program starting address is to be added to this word.

  FFC= 111111111100

SIC relocation loader algorithm

```
begin
  get PROGADDR from operating system
  while not end of input do
    begin
      read next record
      while record type ≠ 'E' do
      while record type = 'T'
         begin
            get length = second data
            mask bits(M) as third data
               For(i = 0, i < length, i++)
                  if Mi = 1 then
                     add PROGADDR at the location PROGADDR + specified
                        address
                  else
                     move object code from record to location PROGADDR +
                        specified address
            read next record
         end
      end
end
```

4.2.2Program Linking

- Consider the program of control sections. The program is made up of 3 control sections.

    1. Main program
    2. Read subroutine
    3. Write subroutine

- These control sections could be assembled together or they could be assembled independently as separate segments of object code after assembly.
- The programmer thinks the three control sections together as a single program. But loader considers this as separate control sections which are to be linked , relocated and loaded.
- Consider the three separate programs PROGA,PROGB,PROGC. In this example, there are differences in handling the identical expressions within the 3 programs.
- Consider the references and the corresponding modification records.
- The general approach is assembler evaluate as much as of the expression it can. The remaining terms are passed on to the loader through modification records.

```
Loc              Source statement                          Object code

0000    PROGA     START    0
                  EXTDEF   LISTA,ENDA
                  EXTREF   LISTB,ENDB,LISTC,ENDC
                  .
                  .
                  .
0020    REF1      LDA      LISTA                           03201D
0023    REF2     +LDT      LISTB+4                         77100004
0027    REF3      LDX     #ENDA-LISTA                      050014
                  .
                  .
                  .
0040    LISTA     EQU      *
                  .
                  .
0054    ENDA      EQU      *
0054    REF4      WORD     ENDA-LISTA+LISTC                000014
0057    REF5      WORD     ENDC-LISTC-10                   FFFFF6
005A    REF6      WORD     ENDC-LISTC+LISTA-1              00003F
005D    REF7      WORD     ENDA-LISTA-(ENDB-LISTB)         000014
0060    REF8      WORD     LISTB-LISTA                     FFFFC0
                  END      REF1
```

Loc		Source statement		Object code
0000	PROGB	START	0	
		EXTDEF	LISTB,ENDB	
		EXTREF	LISTA,ENDA,LISTC,ENDC	
		.		
		.		
		.		
0036	REF1	+LDA	LISTA	03100000
003A	REF2	LDT	LISTB+4	772027
003D	REF3	+LDX	#ENDA-LISTA	05100000
		.		
		.		
		.		
0060	LISTB	EQU	*	
		.		
		.		
0070	ENDB	EQU	*	
0070	REF4	WORD	ENDA-LISTA+LISTC	000000
0073	REF5	WORD	ENDC-LISTC-10	FFFFF6
0076	REF6	WORD	ENDC-LISTC+LISTA-1	FFFFFF
0079	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	FFFFF0
007C	REF8	WORD	LISTB-LISTA	000060
		END		

Figure 3.8 Sample programs illustrating linking and relocation.

Loc		Source statement		Object code
0000	PROGC	START	0	
		EXTDEF	LISTC,ENDC	
		EXTREF	LISTA,ENDA,LISTB,ENDB	
		.		
		.		
		.		
0018	REF1	+LDA	LISTA	03100000
001C	REF2	+LDT	LISTB+4	77100004
0020	REF3	+LDX	#ENDA-LISTA	05100000
		.		
		.		
		.		
0030	LISTC	EQU	*	
		.		
		.		
0042	ENDC	EQU	*	
0042	REF4	WORD	ENDA-LISTA+LISTC	000030
0045	REF5	WORD	ENDC-LISTC-10	000008
0048	REF6	WORD	ENDC-LISTC+LISTA-1	000011
004B	REF7	WORD	ENDA-LISTA-(ENDB-LISTB)	000000
004E	REF8	WORD	LISTB-LISTA	000000
		END		

- Each program contains a list of items(LISTA, LISTB, LISTC). The ends of these lists are marked by ENDA, ENDB, ENDC. Each program contains the same set of references to these external symbols. Three of these are instruction operands(REF1,REF2,REF3). and the others are the values of data words.(REF4 through REF8).
- Consider first reference marked REF1.For PROGA REF1 is simply a reference to a label within the program. It is assembled in the usual way as PC relative instruction.In PROGB the same operand refers to an external symbol. The assembler uses an extended format instruction with addess field set to 00000. Object program for PROGB contains a modification record instructing the loader to add the value of the symbol LISTA to this address field when the program is linked.This reerence is handled exactly in the same way for PROGC.

```
H^PROGA ^000000^000063
D^LISTA ^000040^ENDA  ^000054
R^LISTB ^ENDB  ^LISTC ^ENDC
•
•
T^000020^0A^03201D^77100004^050014
•
•
T^000054^0F^000014^FFFFF6^00003F^000014^FFFFC0
M^000024^05^+LISTB
M^000054^06^+LISTC
M^000057^06^+ENDC
M^000057^06^-LISTC
M^00005A^06^+ENDC
M^00005A^06^-LISTC
M^00005A^06^+PROGA
M^00005D^06^-ENDB
M^00005D^06^+LISTB
M^000060^06^+LISTB
M^000060^06^-PROGA
E^000020
```

Figure 3.9 Object programs corresponding to Fig. 3.8.

```
H^PROGB ^000000^00007F
D^LISTB ^000060^ENDB  ^000070
R^LISTA ^ENDA  ^LISTC ^ENDC
•
•
T^000036^0B^03100000^772027^05100000
•
•
T^000070^0F^000000^FFFFF6^FFFFFF^FFFFF0^000060
M^000037^05^+LISTA
M^00003E^05^+ENDA
M^00003E^05^-LISTA
M^000070^06^+ENDA
M^000070^06^-LISTA
M^000070^06^+LISTC
M^000073^06^+ENDC
M^000073^06^-LISTC
M^000076^06^+ENDC
M^000076^06^-LISTC
M^000076^06^+LISTA
M^000079^06^+ENDA
M^000079^06^-LISTA
M^00007C^06^+PROGB
M^00007C^06^-LISTA
E
```

```
H^PROGC ^000000^000051
D^LISTC ^000030^ENDC  ^000042
R^LISTA ^ENDA  ^LISTB ^ENDB
:
T^000018^0C^03100000^77100004^05100000
:
T^000042^0F^000030^000008^000011^000000^000000
M^000019^05^+LISTA
M^00001D^05^+LISTB
M^000021^05^+ENDA
M^000021^05^-LISTA
M^000042^06^+ENDA
M^000042^06^-LISTA
M^000042^06^+PROGC
M^000048^06^+LISTA
M^00004B^06^+ENDA
M^00004B^06^-LISTA
M^00004B^06^-ENDB
M^00004B^06^+LISTB
M^00004E^06^+LISTB
M^00004E^06^-LISTA
E
```

Figure 3.9 (*cont'd*)

- The figure below shows how the three programs are loaded into memory.

Memory address	Contents				
0000	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	
3FF0	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
4000	....	....	....	....	
4010	....	....	....	....	
4020	03201D77	1040C705	0014....	....	← PROGA
4030	....	....	....	....	
4040	....	....	....	....	
4050	....	00412600	00080040	51000004	
4060	000083..	....	....	....	
4070	....	....	....	....	
4080	....	....	....	....	
4090	....	....	..031040	40772027	← PROGB
40A0	05100014	....	....	....	
40B0	....	....	....	....	
40C0	....	....	....	....	
40D0	00	41260000	08004051	00000400	
40E0	0083....	....	....	....	
40F0	....	....	0310	40407710	
4100	40C70510	0014....	....	....	← PROGC
4110	....	....	....	....	
4120	....	00412600	00080040	51000004	
4130	000083xx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
4140	xxxxxxxx	xxxxxxxx	xxxxxxxx	xxxxxxxx	
⋮	⋮	⋮	⋮	⋮	

Figure 3.10(a) Programs from Fig. 3.8 after linking and loading.

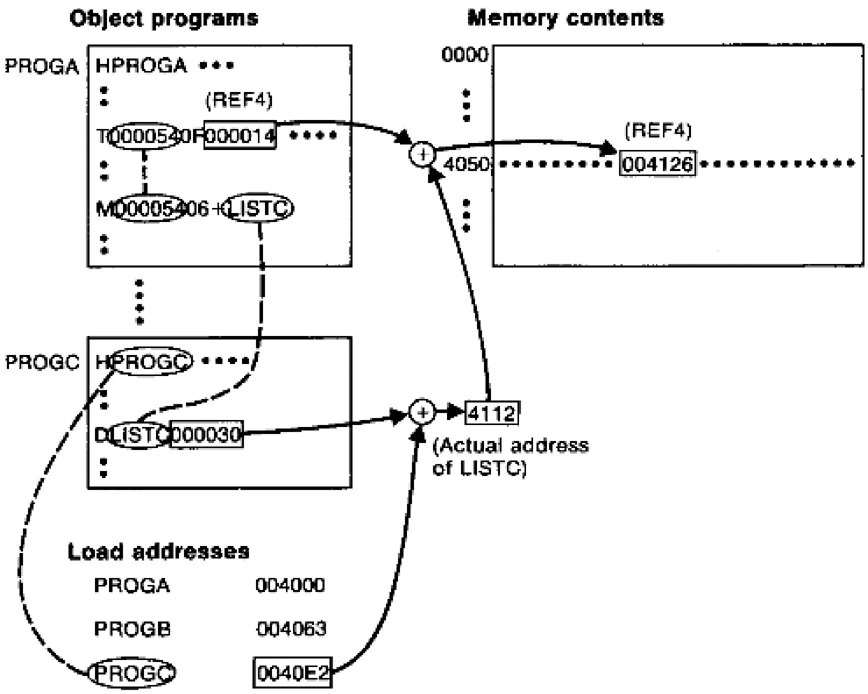


Figure 3.10(b) Relocation and linking operations performed on REF4 from PROGA.

- The values of REF4 through REF8 are same in all the three programs because the same source expression appeared in each program.

4.2.3 Algorithm and data structures for a linking loader

- Consider the algorithm for a linking and relocating loader.
- We use modification records for both relocating and linking
- This type of loader is found on SIC/XE machines whose relative addressing makes relocation unnecessary.
- Input- consists of a set of object programs (control sections) that are to be linked together.
- Control sections or programs contain external references whose definition does not appear in the same program or control section. So linking can not be done until an address is assigned to the external symbol. So it requires two passes.
    - Pass1- Assigns addresses to all external symbols.
    - Pass2- performs the actual loading relocation and linking.
- The **main data structure** for the linking loader is an external symbol table **ESTAB**. It is analogous to SYMTAB. It stores the name and address of each external symbol in the control section. The table also indicates in which control section the symbol is defined.
- Two variables: PROGADDR- Program starting address in memory where the linked program should 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.
- Example: Consider the object programs of PROGA, PROGB, PROGC in fig 3.9 as input to the loader.

Pass1

- During the first pass the loader is concerned only with Header and Define record types in the control sections.
- The beginning load address for the linked program(PROGADDR) is obtained from OS. This becomes the starting address for the first control section(CSADDR).
- The control section name is entered into ESTAB with value given by CSADDR.
- All external symbols appearing in the define record for the control section are also entered into ESTAB. Their addresses are obtained by adding the value specified in the Define record to CSADDR.
- When the END record is read the control section length CSLTH which was saved from the Header record is added to CSADDR. This gives the starting address for the next control section.

- At the end of pass1 , ESTAB contains all external symbols defined in the control sections together with addresses assigned to each.

- Many loaders include the ability to print a **load map** that shows these symbols and their addresses.

Output of pass1

Control section	Symbol name	Address	Length
PROGA		4000	0063
	LISTA	4040	
	ENDA	4054	
PROGB		4063	007F
	LISTB	40C3	
	ENDB	40D3	
PROGC		40E2	0051
	LISTC	4112	
	ENDC	4124	

Algorithm for pass1 of a linking loader

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.

Pass2

- Performs the actual loading, relocation and linking of the program.
- CSADDR holds the starting address of the control section currently being loaded.
- As each Text record is read , the object code is moved to the specified address (plus the current value of the CSADDR).
- When a modification record is encountered , the symbol whose value is to be used for modification is looked up in ESTAB. This value is then added to or subtracted from the indicated location in memory.
- The last step performed by the loader is transferring of control to the loaded program to begin execution.

Pass2 Algorithm

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}
```

Figure 3.11(b) Algorithm for Pass 2 of a linking loader.

- The algorithm can be made more efficient if a slight change is made in the object program format. that is assigning a reference number to each external symbol referred to in a control section. This reference number is used in modification records.

4.3 Machine Independent loader features

4.3.1 Automatic Library search

- This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded. The routines are automatically retrieved from library as they are needed during linking.

- Loader can automatically include routines from a library into the program being loaded.

- The programmer has to only give the subroutine name in the external reference. The routine will be automatically fetched from the library and linked with the main program.

- Working: Enter symbols from Refer record into the symbol table(ESTAB) . When the definition is encountered the address is assigned to the symbol. At the end of pass the symbols in ESTAB remain undefined represent unresolved external references . The loader searches the library for the routines and process the subroutines as if they are part of the input stream.

- The libraries to be searched by the loader contain assembled or compiled versions of the object program(sub program). A special file structure is used for libraries. This is known as directory. This contains the name of the subroutine and a pointer to its address within the file.

4.3.2 Loader Options

- Many loader allow the user to specify options that modify standard processing.
- Loaders have special command language that is used to specify options. Sometimes there is a separate input file to the loader that contains such control statements. The programmer can even include loader control statements in the source program.
  Some of the loader options are:
  1. Selection of alternative sources of input:
     INCLUDE programname(libraryname)
     This command direct the loader to read the designated object program from a library and treat it as if it were primary loader input.
  2. Command to delete external symbols or entire control section
     DELETE csectname
     This instruct the loader to delete the control section from the set of programs being loaded.
  3. CHANGE name1,name2
     This command causes the external symbol name1 to be changed to name2 wherever it appears in the object program.
     Eg: Consider the object program COPY. Here main program is COPY and the two subroutines are RDREC and WRREC. Each of these is a separate control section. Suppose that a set of utility routines are available on the computer system. Two of these READ and WRITE are are designed to perform the same functions as RDREC and WRREC. If we want to use READ and WRITE we can give the loader commands

     INCLUDE READ(UTLIB)
     INCLUDE WRITE(UTLIB)
     DELETE RDREC, WRREC
     CHANGE RDREC,READ
     CHANGE WRREC,WRITE
  4. Another common loader option involves the automatic inclusion of library routines to satisfy external references. Most loaders allow the user to specify alternative libraries to be searched using a statement such as LIBRARY MYLIB . Such user specified libraries are normally searched before the standard libraries. This allows the user to use special

versions of the standard routines.

5. Loaders that perform automatic library search to satisfy external reference allows the user to avoid some references using the command NOCALL. Eg: NOCALL STDDEV, PLOT. This avoids the overhead of loading and linking the unwanted routines
6. Other options:
   - No external reference should be resolved.
   - Specify the output from the loader(load map)
   - Specify the location at which the execution is to begin

4.4 Loader Design

- Loaders do loading , relocation and linking.
- There are 4 types
  - Linkage editor- links the program stores it in a file and later loads.
  - Linking loader- linking during load time
  - Dynamic linking- linking during execurion time
  - Bootstrap loader- loads the first program or OS.

4.4.1Differences between Linkage editor and linking loader

Linking loader	linkage editor
1. Performs all linking and relocation operations and loads the linked program directly into memory for execution	1. Produces a linked version of the program called load module which is written to a file for later execution
2. A linking loader searches the library and resolves external references every time the program is executed.	2. Resolution of external references and library searching are only performed once.
3. More than one pass required.	3. The loading can be accomplished in one pass and no external symbol table required, much less overhead than a linking loader.

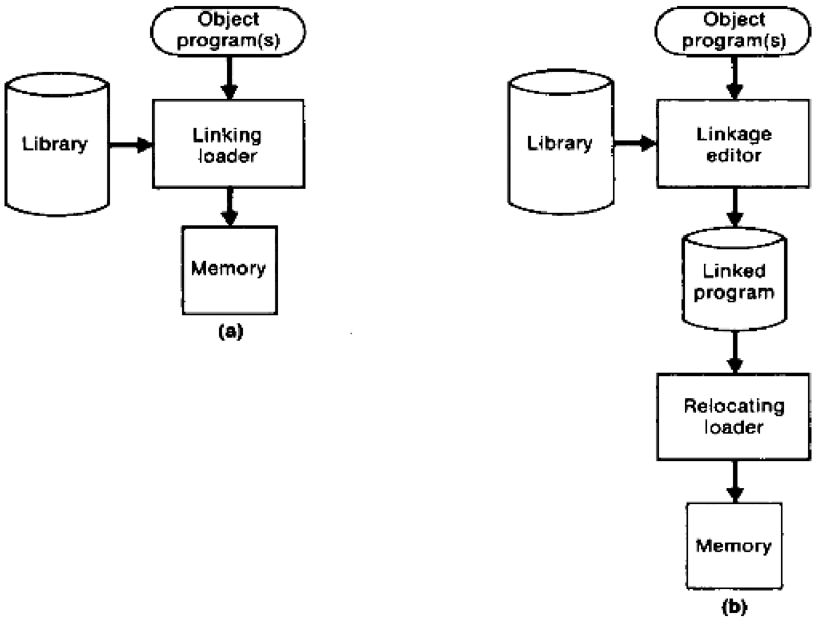


Figure 3.13 Processing of an object program using (a) linking loader and (b) linkage editor.

Advantages of Linkage editors

- Linkage editors can perform many useful functions besides simply preparing an object program for execution. Consider the example, a program PLANNER that uses a large number of subroutines. Suppose that one subroutine called PROJECT is changed. After new version of PROJECT is assembled the linkage editor can be used to replace this subroutine in the linked version of PLANNER.
  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. Eg: For FORTRAN programs there are a number of subroutines that are used for input and output. They are read and write datablocks, encode and decode data items etc. Linkage editor can be used to combine these subroutines into a package with the following commands.

```
INCLUDE   PLANNER(PROGLIB)
DELETE    PROJECT
INCLUDE   PROJECT(NEWLIB)
REPLACE   PLANNER(PROGLIB)

INCLUDE   READR(FTNLIB)
INCLUDE   WRITER(FTNLIB)

INCLUDE   BLOCK(FTNLIB)
INCLUDE   DEBLOCK(FTNLIB)
INCLUDE   ENCODE(FTNLIB)
INCLUDE   DECODE(FTNLIB)
.
.
.
SAVE      FTNIO(SUBLIB)
```

- Linkage editors can also allow the user to specify that external references are not to be resolved by automatic library search.

4.4.2 Dynamic Linking

- In dynamic linking the linking function is done at execution time. That is 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. For eg: in C such fuctions are stored in dynamic linking library.. A single copy of the routines in this library could be loaded into memory and all programs share this.
- In object oriented program dynamic linking is often used for references to software objects.
- Advantage:- Dynamic linking provide the ability to load the routines only when they are required. For eg: consider the subroutine which diagnose the error in input data during execution. If such errors are rare these subroutines need not be used.
- Consider the following example of dynamic linking. Here the routines that are to be dynamically loaded must be called via an OS service request.

Loading and calling a subroutine via dynamic linking

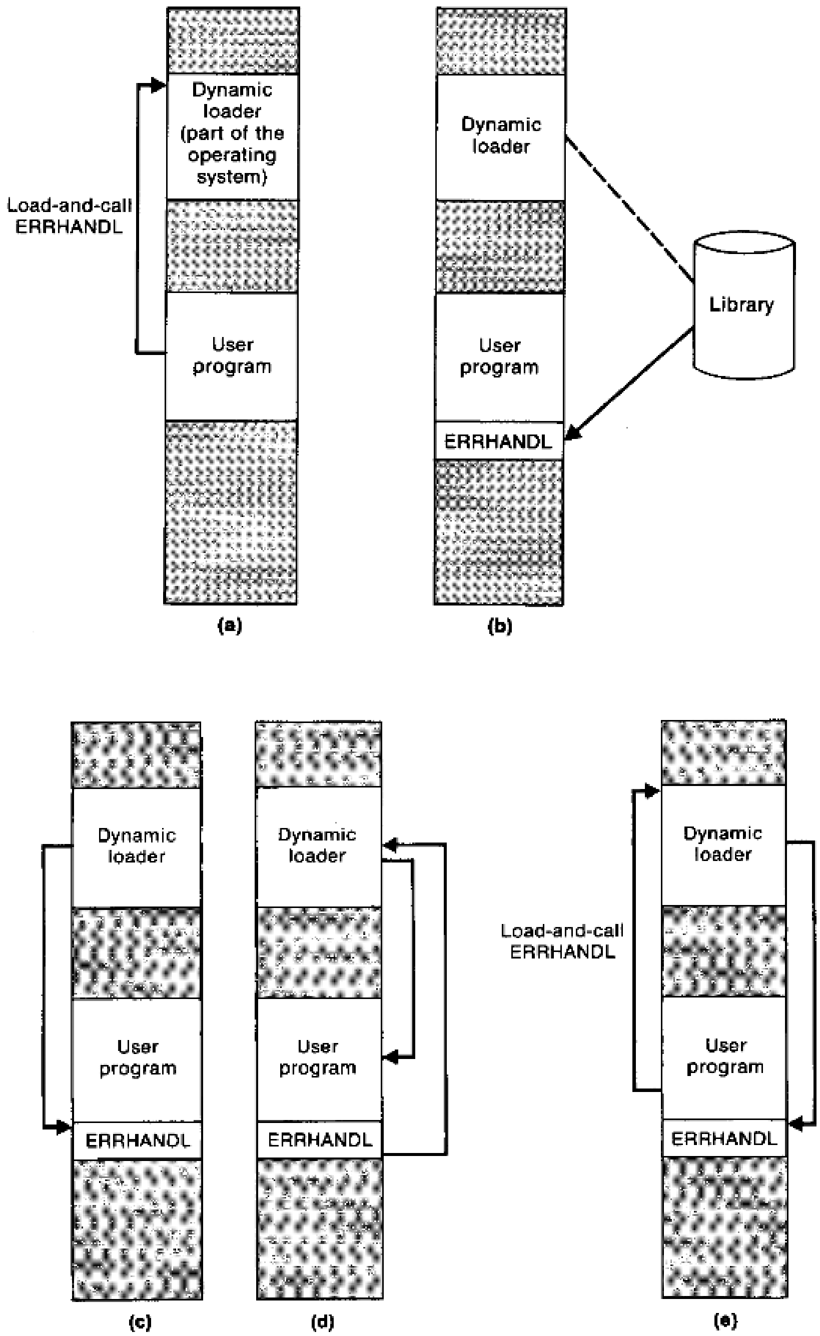


Figure 3.14 Loading and calling of a subroutine using dynamic linking.

- When the dynamic linking is used the association of an actual address with the symbolic name of the called routine is done at execution time.. This is known as **dynamic binding**.

4.3.3 Bootstrap loaders

- Consider how the loader itself is loaded into memory. OS loads the loader. How the OS gets loaded.
- In an idle system if we specify the absolute address the program can be loaded at that location. that is a mechanism of absolute loader is required.
- One solution to this is to have a built in hardware function that reads a fixed length record from some device into memory at some fixed location. This device can be selected via console switches. After the read operation is complete the control is automatically transferred to the address in memory where the record was stored. This record contains machine instructions that load the absolute program that follows.
- If the loading process requires more instructions than can be read in a single record this first record causes the reading of others and in turn other records . Hence the name **Bootstrap.**

MODULE 5

MACRO PROCESSOR

A *Macro* represents a commonly used group of statements in the source programming language.

- A macro instruction (macro) is a notational convenience for the programmer
    - It allows the programmer to write shorthand version of a program (module programming)
- The macro processor replaces each macro instruction with the corresponding group of source language statements (*expanding*)
    - Normally, it performs no analysis of the text it handles.
    - It does not concern the meaning of the involved statements during macro expansion.
- The design of a macro processor generally is *machine independent!*
- Two new assembler directives are used in macro definition
    - **MACRO:** identify the beginning of a macro definition
    - **MEND:** identify the end of a macro definition
- Prototype for the macro
    - Each parameter begins with '&'
        - name MACRO parameters

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

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

5.1 <u>Basic Macro Processor Functions:</u>

- Macro Definition and Expansion
- Macro Processor Algorithms and Data structures

5.1.1 Macro Definition and Expansion:

- Consider the example of an SIC/XE program using macro instructions. This program defines and uses two macro instructions , RDBUFF and WRBUFF.

- The functions and logic of RDBUFF macro are similar to RDREC subroutine.

```
COPY      START   0                 COPY FILE FROM INPUT TO OUTPUT
RDBUFF    MACRO   &INDEV,&BUFADR,&RECLTH
.
.         MACRO TO READ RECORD INTO BUFFER
.
          CLEAR   X                 CLEAR LOOP COUNTER
          CLEAR   A
          CLEAR   S
         +LDT     #4096             SET MAXIMUM RECORD LENGTH
          TD      =X'&INDEV'        TEST INPUT DEVICE
          JEQ     *-3               LOOP UNTIL READY
          RD      =X'&INDEV'        READ CHARACTER INTO REG A
          COMPR   A,S               TEST FOR END OF RECORD
          JEQ     *+11              EXIT LOOP IF EOR
          STCH    &BUFADR,X         STORE CHARACTER IN BUFFER
          TIXR    T                 LOOP UNLESS MAXIMUM LENGTH
          JLT     *-19                HAS BEEN REACHED
          STX     &RECLTH           SAVE RECORD LENGTH
          MEND
```

```
WRBUFF    MACRO     &OUTDEV,&BUFADR,&RECLTH
.
.         MACRO TO WRITE RECORD FROM BUFFER
.
          CLEAR     X               CLEAR LOOP COUNTER
          LDT       &RECLTH
          LDCH      &BUFADR,X       GET CHARACTER FROM BUFFER
          TD       =X'&OUTDEV'      TEST OUTPUT DEVICE
          JEQ       *-3             LOOP UNTIL READY
          WD       =X'&OUTDEV'      WRITE CHARACTER
          TIXR      T               LOOP UNTIL ALL CHARACTERS
          JLT       *-14              HAVE BEEN WRITTEN
          MEND
.
.         MAIN PROGRAM
.


FIRST     STL       RETADR          SAVE RETURN ADDRESS
CLOOP     RDBUFF    F1,BUFFER,LENGTH  READ RECORD INTO BUFFER
          LDA       LENGTH          TEST FOR END OF FILE
          COMP     #0
          JEQ       ENDFIL          EXIT IF EOF FOUND
          WRBUFF    05,BUFFER,LENGTH  WRITE OUTPUT RECORD
          J         CLOOP           LOOP
ENDFIL    WRBUFF    05,EOF,THREE    INSERT EOF MARKER
          J        @RETADR
EOF       BYTE      C'EOF'
THREE     WORD      3
RETADR    RESW      1
LENGTH    RESW      1               LENGTH OF RECORD
BUFFER    RESB      4096            4096-BYTE BUFFER AREA
          END       FIRST
```

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

- Two new assembler directives (Macro and MEND) are used in macro definitions. The keyword macro identifies the beginning of the macro definition. The symbol in the label field (RDBUFF) is the name of the macro and entries in the operand field identify the parameters of the macro. Each parameter begins with the character & which helps in the substitution of parameters during macro expansion. Following the macro directive are the statements that make up the body of the macro definition. These are the statements that will be generated as the expansion of the macro. The MEND directive marks the end of the macro.
- Macro invocation or call is written in the main program. In macro invocation the name of the macro is followed by the arguments. Output of the macroprocessor is the expanded program.

```
5      COPY     START    0                  COPY FILE FROM INPUT TO OUTPUT
180    FIRST    STL      RETADR             SAVE RETURN ADDRESS
190    .CLOOP   RDBUFF   F1,BUFFER,LENGTH   READ RECORD INTO BUFFER
190a   CLOOP    CLEAR    X                  CLEAR LOOP COUNTER
190b            CLEAR    A
190c            CLEAR    S
190d            +LDT     #4096              SET MAXIMUM RECORD LENGTH
190e            TD       =X'F1'             TEST INPUT DEVICE
190f            JEQ      *-3                LOOP UNTIL READY
190g            RD       =X'F1'             READ CHARACTER INTO REG A
190h            COMPR    A,S                TEST FOR END OF RECORD
190i            JEQ      *+11               EXIT LOOP IF EOR
190j            STCH     BUFFER,X           STORE CHARACTER IN BUFFER
190k            TIXR     T                  LOOP UNLESS MAXIMUM LENGTH
190l            JLT      *-19                 HAS BEEN REACHED
190m            STX      LENGTH             SAVE RECORD LENGTH
195             LDA      LENGTH             TEST FOR END OF FILE
200             COMP     #0
205             JEQ      ENDFIL             EXIT IF EOF FOUND
```

Expanded Program

- Another simple example is given below:
- Program with macro

```
EX1       MACRO     &A,&B
          LDA       &A
          STA       &B
          MEND

SAMPLE    START     1000
          EX1       N1,N2
N1        RESW      1
N2        RESW      1

          END
```

Expanded program

```
SAMPLE   START   1000

.        EX1     N1,N2
         LDA     N1
         STA     N2
N1       RESW    1
N2       RESW    1
```

Macro expansion

- Macro definition statements have been deleted since they are no longer required after the macros are expanded. Each macro invocation statement has been expanded into the statements that form the body of the macro with the arguments from the macro invocation is substituted for the parameters in the macro definition. Macro invocation statement is included as a comment line in the expanded program.

- After macroprocessing the expanded file can be used as input to the assembler.

- Differences between macro and subroutine: The statements that form the expansion of a macro are generated and (assembled) each time the macro is invoked. Statements in a subroutine appear only once, regardless of how many time the subroutine is called.

5.1.2 *Macro Processor Algorithm and Data Structure:*

- It is easy to design a two pass macro processor in which all macro definitions are processed during the first pass and all macro invocation statements are expanded during the second pass.

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

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

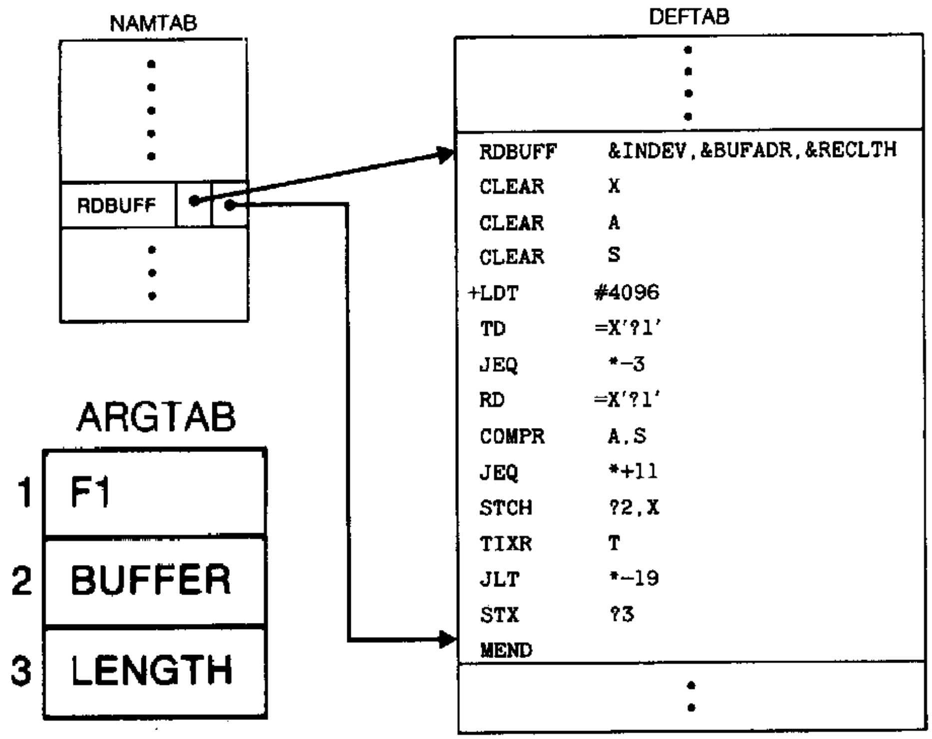
```
1   MACROX    MACRO       {Defines  SIC/XE macros}
2   RDBUFF    MACRO       &INDEV,&BUFADR,&RECLTH
              .
              .           {SIC/XE version}
              .
3             MEND        {End of RDBUFF}
4   WRBUFF    MACRO       &OUTDEV,&BUFADR,&RECLTH
              .
             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

(b)

Figure 4.3 Example of the definition of macros within a macro body.

- Here defining MACROS does not define RDBUFF and WRBUFF. These definitions are processed only when an invocation of MACROS is expanded.
- A one pass macro processor that can alternate between macro definition and macro expansion is able to handle these type of macros.

- There are 3 main data structures:-
  - DEFTAB- The macro definitions are stored in a definition table(DEFTAB) which contain the macro definition and the statements that form the macro body. References to the macro instruction parameters are converted to positional notation.
  - NAMTAB- Macro names are entered into 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.
  - ARGTAB- is used during the expansion of the macro invocation. When a macro invocation statement is recognized the arguments are stored in argument table. As the macro is expanded arguments from ARGTAB are substituted for the corresponding parameters in the macro body.
  - Eg



Macro processor algorithm

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


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

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}

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)*

- Procedure DEFINE which is called when the beginning of a macro definition is recognized makes the appropriate entries in DEFTAB and NAMTAB.
- EXPAND is called to set up the argument values in ARGTAB and expand a *Macro Invocation* statement.
- Procedure GETLINE is called to get the next line to be processed either from the DEFTAB or from the input file .
- Handling of macro definition within macro:- When a macro definition is encountered it is entered in the DEFTAB. The normal approach is to continue entering till MEND is encountered. If there is a program having a Macro defined within another Macro.While defining in the DEFTAB the very first MEND is taken as the end of the Macro definition. This does not complete the definition as there is another outer Macro which completes the definition of Macro as a whole. Therefore the DEFINE procedure keeps a counter variable LEVEL.Every time a Macro directive is encountered this counter is incremented by 1. The moment the innermost Macro ends indicated by the directive MEND it starts decreasing the value of the counter variable by one. The last MEND should make the counter value set to zero. So when LEVEL becomes zero, the MEND corresponds to the original MACRO directive.

5.3Machine-independent Macro-Processor Features.

The design of macro processor doesn't depend on the architecture of the machine. We will be studying some extended feature for this macro processor. These features are:

- Concatenation of Macro Parameters
- Generation of unique labels
- Conditional Macro Expansion
- Keyword Macro Parameters

5.3.1Concatenation of Macro parameters:

- Most macro processor allows parameters to be concatenated with other character strings. Suppose that a program contains a series of variables named by the symbols XA1, XA2, XA3,…, another series of variables named XB1, XB2, XB3,…, etc. If similar processing is to be performed on each series of labels, the programmer might put this as a macro instruction.
- The parameter to such a macro instruction could specify the series of variables to be operated on (A, B, etc.). The macro processor would use this parameter to construct the symbols required in the macro expansion (XA1, XB1, etc.).

- Suppose that the parameter to such a macro instruction is named &ID. The body of the macro definition might contain a statement like
    - LDA X&ID1
- & is the starting character of the macro instruction; but the end of the parameter is not marked. So in the case of &ID1, the macro processor could deduce the meaning that was intended.

- If the macro definition contains &ID and &ID1 as parameters, the situation would be unavoidably ambiguous.
- Most of the macro processors deal with this problem by providing a special **concatenation operator.** In the SIC macro language, this operator is the character →. Thus the statement LDA X&ID1 can be written as

  LDA X&ID→1

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

```
SUM    A                    SUM    BETA
 ↓                           ↓
LDA    XA1                  LDA    XBEATA1
ADD    XA2                  ADD    XBEATA2
ADD    XA3                  ADD    XBEATA3
STA    XAS                  STA    XBEATAS
```

- The above figure shows a macro definition that uses the concatenation operator as previously described. The statement SUM A and SUM BETA shows the invocation statements and the corresponding macro expansion.

5.3.2Generation of Unique Labels

- it is not possible to use labels for the instructions in the macro definition, since every expansion of macro would include the label repeatedly which is not allowed by the assembler.
- We can use the technique of generating unique labels for every macro invocation and expansion.
- During macro expansion each $ will be replaced with $XX, where xx is a two- character alphanumeric counter of the number of macro instructions expansion.

For example,

XX = AA, AB, AC…

This allows 1296 macro expansions in a single program.

The following program shows the macro definition with labels to the instruction.

```
RDBUFF   MACRO    &INDEV, &BUFADR, &RECLTH
         CLEAR    X                      CLEAR LOOP COUNTER
         CLEAR    A
         CLEAR    S
        +LDT      #4096                  SET MAXIMUM RECORD LENGTH
$LOOP    TD       =X'&INDEV'             TEST INPUT DEVICE
         JEQ      $LOOP                  LOOP UNTIL READY
         RD       =X'&INDEV'             READ CHARACTER INTI REG A
         COMPR    A, S                   TEST FOR END OF RECORD
         JEQ      $EXIT                  EXIT LOOP IF EOR
         STCH     &BUFADR, X             STORE CHARACTER IN BUFFER
         TIXR     $LOOP                  HAS BEEN REACHED
$EXIT    STX      &RECLTH                SAVE RECORD LENGTH
         MEND
```

The following figure shows the macro invocation and expansion first time.

```
.               RDBUFF    F1, BUFFER, LENGTH

                CLEAR     X                  CLEAR LOOP COUNTER
                CLEAR     A
                CLEAR     S
               +LDT       #4096              SET MAXIMUM RECORD LENGTH
      $AALOOP   TD        =X'F1'             TEST INPUT DEVICE
                JEQ       $AALOOP            LOOP UNTIL READY
                RD        =X'F1'             READ CHARACTER INTI REG A
                COMPR     A, S               TEST FOR END OF RECORD
                JEQ       $AAEXIT            EXIT LOOP IF EOR
                STCH      BUFFER, X          STORE CHARACTER IN BUFFER
                TIXR      T                  LOOP UNLESS MAXIMUM LENGTH
                JLT       $AALOOP            HAS BEEN REACHED
      $AAEXIT   STX       LENGTH             SAVE RECORD LENGTH
```

- If the macro is invoked second time the labels may be expanded as $ABLOOP $ABEXIT.

5.3.3Conditional Macro Expansion

  - IF ELSE
  - WHILE loop
- We can modify the sequence of statements generated for a macro expansion depending on conditions.

IF ELSE ENDIF structure

- Consider the following example.

```
RDBUFF   MACRO    &INDEV,&BUFADR,&RECLTH,&EOR,&MAXLTH
         IF       (&EOR NE '')
&EORCK   SET      1
         ENDIF
         CLEAR    X              CLEAR LOOP COUNTER
         CLEAR    A
         IF       (&EORCK EQ 1)
         LDCH     =X'&EOR'       SET EOR CHARACTER
         RMO      A,S
         ENDIF
         IF       (&MAXLTH EQ '')
        +LDT      #4096          SET MAX LENGTH = 4096
         ELSE
        +LDT      #&MAXLTH       SET MAXIMUM RECORD LENGTH
         ENDIF
$LOOP    TD       =X'&INDEV'     TEST INPUT DEVICE
         JEQ      $LOOP          LOOP UNTIL READY
         RD       =X'&INDEV'     READ CHARACTER INTO REG A
         IF       (&EORCK EQ 1)
         COMPR    A,S            TEST FOR END OF RECORD
         JEQ      $EXIT          EXIT LOOP IF EOR
         ENDIF
         STCH     &BUFADR,X      STORE CHARACTER IN BUFFER
         TIXR     T              LOOP UNLESS MAXIMUM LENGTH
         JLT      $LOOP            HAS BEEN REACHED
$EXIT    STX      &RECLTH        SAVE RECORD LENGTH
         MEND
```

(a)

```
.        RDBUFF   F3,BUF,RECL,04,2048
```

```
         CLEAR    X              CLEAR LOOP COUNTER
         CLEAR    A
         LDCH     =X'04'         SET EOR CHARACTER
         RMO      A,S
        +LDT      #2048          SET MAXIMUM RECORD LENGTH
$AALOOP  TD       =X'F3'         TEST INPUT DEVICE
         JEQ      $AALOOP        LOOP UNTIL READY
         RD       =X'F3'         READ CHARACTER INTO REG A
         COMPR    A,S            TEST FOR END OF RECORD
         JEQ      $AAEXIT        EXIT LOOP IF EOR
         STCH     BUF,X          STORE CHARACTER IN BUFFER
         TIXR     T              LOOP UNLESS MAXIMUM LENGTH
         JLT      $AALOOP          HAS BEEN REACHED
$AAEXIT  STX      RECL           SAVE RECORD LENGTH
```

(b)

Figure 4.8 Use of macro-time conditional statements.

- Here the definition of RDBUFF has two additional parameters. &EOR(end of record) &MAXLTH(maximum length of the record that can be read)
- The macro processor directive SET – The statement assigns a value 1 to &EORCK and &EORCK is known as macrotime variable. A **macrotime variable** is used to store working values during the macro expansion. Any symbol that begins with & and that is not a macro instruction parameter is assumed to be a macro time variable. All such variables are initialized to a value 0.
- Implementation of Conditional macro expansion- Macro processor maintains a symbol table that contains the values of all macrotime variables used. Entries in this table are made when SET

statements are processed. The table is used to look up the current value of the variable.

- Testing of Boolean expression in IF statement occurs at the time macros are expanded. By the time the program is assembled all such decisions are made and conditional macro instruction directives are removed.
- IF statements are different from COMPR which test data values during program expansion.

Looping-WHILE

- Consider the following example.

```
 25  RDBUFF   MACRO   &INDEV,&BUFADR,&RECLTH,&EOR
 27  &EORCT   SET     %NITEMS(&EOR)
 30           CLEAR   X                  CLEAR LOOP COUNTER
 35           CLEAR   A
 45          +LDT     #4096              SET MAX LENGTH = 4096
 50  $LOOP    TD      =X'&INDEV'         TEST INPUT DEVICE
 55           JEQ      $LOOP             LOOP UNTIL READY
 60           RD      =X'&INDEV'         READ CHARACTER INTO REG A
 63  &CTR     SET     1
 64           WHILE    (&CTR LE &EORCT)
 65           COMP    =X'0000&EOR[&CTR]'
 70           JEQ      $EXIT
 71  &CTR     SET      &CTR+1
 73           ENDW
 75           STCH     &BUFADR,X         STORE CHARACTER IN BUFFER
 80           TIXR     T                 LOOP UNLESS MAXIMUM LENGTH
 85           JLT      $LOOP               HAS BEEN REACHED
 90  $EXIT    STX      &RECLTH           SAVE RECORD LENGTH
100           MEND
```

(a)

```
 .          RDBUFF   F2,BUFFER,LENGTH,(00,03,04)

30              CLEAR    X              CLEAR LOOP COUNTER
35              CLEAR    A
45             +LDT     #4096           SET MAX LENGTH = 4096
50   $AALOOP    TD      =X'F2'          TEST INPUT DEVICE
55              JEQ      $AALOOP        LOOP UNTIL READY
60              RD      =X'F2'          READ CHARACTER INTO REG A
65              COMP    =X'000000'
70              JEQ      $AAEXIT
65              COMP    =X'000003'
70              JEQ      $AAEXIT
65              COMP    =X'000004'
70              JEQ      $AAEXIT
75              STCH     BUFFER,X       STORE CHARACTER IN BUFFER
80              TIXR     T              LOOP UNLESS MAXIMUM LENGTH
85              JLT      $AALOOP          HAS BEEN REACHED
90   $AAEXIT    STX      LENGTH         SAVE RECORD LENGTH
```

(b)

- Here the programmer can specify a list of end of record characters.
- In the macro invocation statement there is a list(00,03,04) corresponding to the parameter &EOR. Any one of these characters is to be considered as end of record.
- The WHILE statement specifies that the following lines until the next ENDW are to be generated repeatedly as long as the condition is true.
- The testing of these condition and the looping are done while the macro is being expanded.The conditions do not contain any runtime values.
- %NITEMS is a macroprocessor function that returns as its value the number of members in an argument list. Here it has the value 3. The value of &CTR is used as a subscript to select the proper member of the list for each iteration of the loop. &EOR[&CTR] takes the values 00,03,04 .
- Implementation- When a WHILE statement is encountered during a macro expansion the specified Boolean expression is evaluated , if the value is false the macroprocessor skips ahead in DEFTAB until it finds the ENDW and then resumes normal macro expansion(not at run time).

5.3.4Keyword Macro Parameters

- All the macro instruction definitions used positional parameters. Parameters and arguments are matched according to their positions in the macro prototype and the macro invocation statement.
- The programmer needs to be careful while specifying the arguments. If an argument is to be omitted the macro invocation statement must contain a null argument mentioned with two commas.
- Positional parameters are suitable for the macro invocation. But if the macro invocation has large number of parameters, and if only few of the values need to be used in a typical invocation, a different type of parameter specification is required.
- Eg: Consider the macro GENER which has 10 parameters, but in a particular invocation of a macro only the third and nineth parameters are to be specified. If positional parameters are used the macro invocation will look like

GENER , , DIRECT, , , , , , 3,

- But using keyword parameters this problem can be solved. We can write

GENER TYPE=DIRECT, CHANNEL=3

```
RDBUFF   MACRO   &INDEV=F1,&BUFADR=,&RECLTH=,&EOR=04,&MAXLTH=4096
         IF      (&EOR NE '')
&EORCK   SET     1
         ENDIF
         CLEAR   X               CLEAR LOOP COUNTER
         CLEAR   A
         IF      (&EORCK EQ 1)
         LDCH    =X'&EOR'        SET EOR CHARACTER
         RMO     A,S
         ENDIF
        +LDT     #&MAXLTH        SET MAXIMUM RECORD LENGTH
$LOOP    TD      =X'&INDEV'      TEST INPUT DEVICE
         JEQ     $LOOP           LOOP UNTIL READY
         RD      =X'&INDEV'      READ CHARACTER INTO REG A
         IF      (&EORCK EQ 1)
         COMPR   A,S             TEST FOR END OF RECORD
         JEQ     $EXIT           EXIT LOOP IF EOR
         ENDIF
         STCH    &BUFADR,X       STORE CHARACTER IN BUFFER
         TIXR    T               LOOP UNLESS MAXIMUM LENGTH
         JLT     $LOOP             HAS BEEN REACHED
$EXIT    STX     &RECLTH         SAVE RECORD LENGTH
         MEND
```

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

```
          CLEAR   X               CLEAR LOOP COUNTER
          CLEAR   A
          LDCH    =X'04'          SET EOR CHARACTER
          RMO     A,S
         +LDT     #4096           SET MAXIMUM RECORD LENGTH
$AALOOP   TD      =X'F1'          TEST INPUT DEVICE
          JEQ     $AALOOP         LOOP UNTIL READY
          RD      =X'F1'          READ CHARACTER INTO REG A
          COMPR   A,S             TEST FOR END OF RECORD
          JEQ     $AAEXIT         EXIT LOOP IF EOR
          STCH    BUFFER,X        STORE CHARACTER IN BUFFER
          TIXR    T               LOOP UNLESS MAXIMUM LENGTH
          JLT     $AALOOP           HAS BEEN REACHED
$AAEXIT   STX     LENGTH          SAVE RECORD LENGTH
```

(b)

Figure 4.10 Use of keyword parameters in macro instructions.

Keyword parameters

- Each argument value is written with a keyword that names the corresponding parameter.
- Arguments may appear in any order.
- Null arguments no longer need to be used.
- It is easier to read and much less error-prone than the positional method.

5.4 Macro Processor Design Options

5.4.1Recursive Macro Expansion

- We have seen an example of the *definition* of one macro instruction by another. But we have not dealt with the *invocation* of one macro by another. The following example shows the invocation of one macro by another macro.

```
RDBUFF   MACRO    &BUFADR, &RECLTH, &INDEV
.
.        MACRO TO READ RECORD INTO BUFFER
.
         CLEAR    X              CLEAR LOOP COUNTER
         CLEAR    A
         CLEAR    S
        +LDT      #4096          SET MAXIMUN RECORD LENGTH
$LOOP    RDCHAR   &INDEV         READ CHARACTER INTO REG A
         COMPR    A, S           TEST FOR END OF RECORD
         JEQ      &EXIT          EXIT LOOP IF EOR
         STCH     &BUFADR, X     STORE CHARACTER IN BUFFER
         TIXR     T              LOOP UNLESS MAXIMUN LENGTH
         JLT      $LOOP          HAS BEEN REACHED
$EXIT    STX      &RECLTH        SAVE RECORD LENGTH
         MEND
```

```
RDCHAR        MACRO   &IN
.
.     MACROTO READ CHARACTER INTO REGISTER A
.
              TD      =X'&IN'              TEST INPUT DEVICE
              JEQ      *-3                 LOOP UNTIL READY
              RD      =X'&IN'              READ CHARACTER
              MEND
```

Problem of Recursive Expansion

- Previous macro processor design cannot handle such kind of recursive macro invocation and expansion
  - The procedure EXPAND would be called recursively, thus the invocation arguments in the ARGTAB will be overwritten.
  - The Boolean variable EXPANDING would be set to FALSE when the “inner” macro expansion is finished, *i.e.*, the macro process would forget that it had been in the middle of expanding an “outer” macro.

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

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

The Boolean variable EXPANDING would be set to TRUE, and expansion of the macro invocation statement would begin. The processing would proceed normally until statement invoking RDCHAR is processed. This time, ARGTAB would look like

Paramet	Value

er	
1	F1
2	(Unuse d)
--	--

At the expansion, when the end of RDCHAR is recognized, EXPANDING would be set to FALSE. Thus the macro processor would 'forget' that it had been in the middle of expanding a macro when it encountered the RDCHAR statement. In addition, the arguments from the original macro invocation (RDBUFF) would be lost because the value in ARGTAB was overwritten with the arguments from the invocation of RDCHAR.

- Solutions
  - Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained.
  - If you are writing in a language without recursion support, use a stack to take care of pushing and popping local variables and return addresses.

5.4.2General-Purpose Macro Processors

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

- *Pros*
  - Programmers do not need to learn many macro languages.
  - Although its development costs are somewhat greater than those for a language specific macro processor, this expense does not need to be repeated for each language, thus save substantial overall cost.

- *Cons*
  - Large number of details must be dealt with in a real programming language
    - Situations in which normal macro parameter substitution should not occur, e.g., comments.
    - Facilities for grouping together terms, expressions, or statements. Eg: some languages use begin and end . Some use { and }
    - Tokens, e.g., identifiers, constants, operators, keywords

    - Syntax used for macro definition and macro invocation statement is different.

5.4.3Macro Processing within Language Translators

- The macro processors we discussed are called “Preprocessors”.
    - Process macro definitions
    - Expand macro invocations
    - Produce an expanded version of the source program, which is then used as input to an assembler or compiler
- You may also combine the macro processing functions with the language translator:
    - Line-by-line macro processor
    - Integrated macro processor

Line-by-Line Macro Processor

- Used as a sort of input routine for the assembler or compiler
    - Read source program
    - Process macro definitions and expand macro invocations
    - Pass output lines to the assembler or compiler
- Benefits
    - Avoid making an extra pass over the source program.
    - Data structures required by the macro processor and the language translator can be combined (e.g., OPTAB and NAMTAB)
    - Utility subroutines can be used by both macro processor and the language translator.
        - Scanning input lines
        - Searching tables
        - Data format conversion
    - It is easier to give diagnostic messages related to the source statements

Integrated Macro Processor

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

        - Ex (blanks are not significant in FORTRAN)
            - DO 100 I = 1,20
                - a DO statement
            - DO 100 I = 1
                - An assignment statement
                - DO100I: variable (blanks are not significant in FORTRAN)
- An integrated macro processor can support macro instructions that depend upon the context in which they occur.

- Disadvantages- They must be specially designed and written to work with a particular implementation of an assembler or compiler.. Cost of development is high.

EDITORS AND DEBUGGING SYSTEMS

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.

5.1 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).

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

- Document-editing process in 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 and how to display it
  3) Specify and execute operations that modify the target document
  4) Update the view appropriately

- The above task involves traveling, filtering and formatting.

  - Traveling – To locate the area of interest. This is done by operations such as next screenful, bottom and find pattern.

  - Filtering- extracts the relevant subset of the target document.

  - Formatting- How the result of filtering will be seen as a visible representation(the view) on a display screen.

  - Editing- The target document is created or altered with a set of operations such as insert, delete, replace, move and copy.
- 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.
- So in overall the user might travel to the end of the document. A screenful of text would be filtered, this segment would be formatted, and the view would be displayed on an output device. The user could then edit the view.

5.1.2User 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:

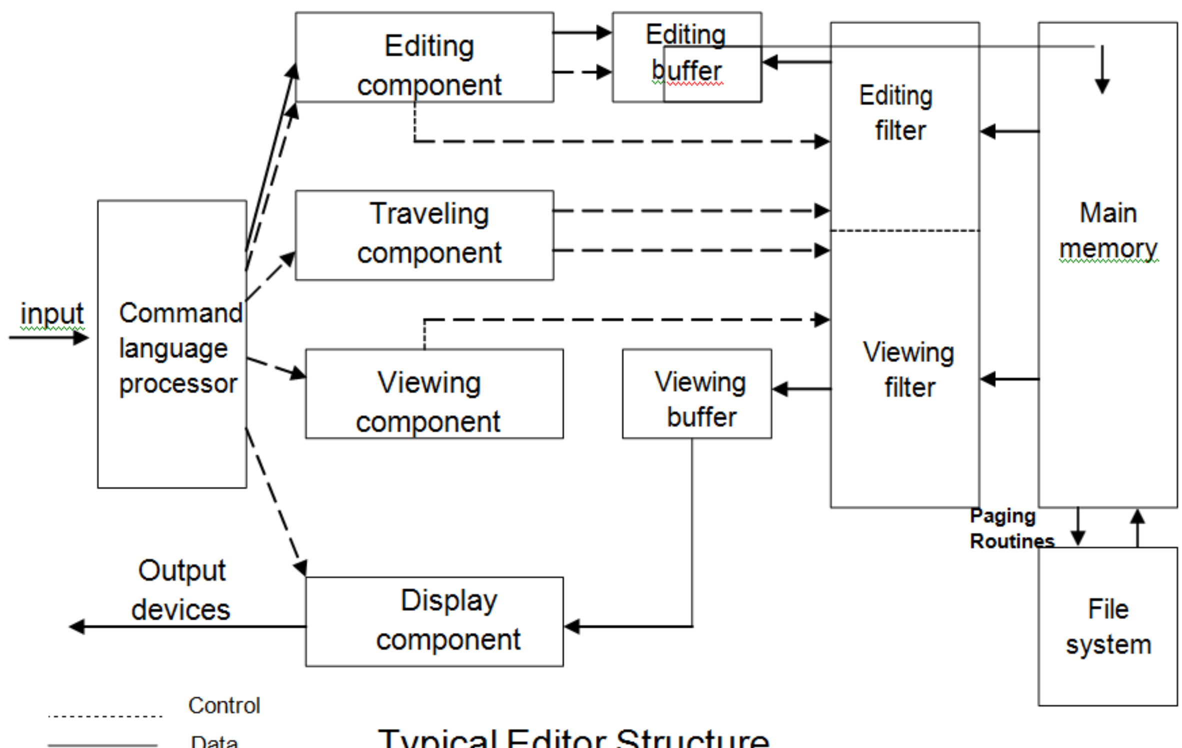
    - text devices- are type writer like key boards on which a user presses and releases keys sending a unique code for each key.
    - button or choice devices- generate an interrupt causing an invocation of an associated application program action. They include a set of function keys. Buttons can be simulated in software.
    - Locator devices – are two dimensional analog to digital converters that position a cursor symbol on the screen by observing the user's movement of the device. Eg: mouse, data tablet. Returns the coordinates of the position of the device. Text devices with arrow keys can be used as locator devices . Arrow shows left, right , up or down.
    - Voice input devices- Translates spoken words to their textual equivalent.

- **Output Devices** lets the user view the elements being edited and the results of the editing operations. CRT terminals use hardware assistance for such features as moving the cursor , inserting and deleting characters and lines etc.
- **The interaction language** is one of the common types.
    - **Typing or text command oriented-** the user communicates with the editor by typing text strings both for command names and for operands.These strings are sent to the editor and echoed to the output device.This requires the user to remember the commands.
    - **Function key oriented-** In this each command is associated with a marked key on the user's keyboard.
    - **Menu oriented systems-** A menu is a multiple choice set of text strings or icons which are graphic symbols that represent object or operations. The user can perform actions by selecting items from the menu. Some systems have the most used functions on a main command menu and have secondary menus to handle the less frequently used functions.

5.1.3Editor 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

- The **command language processor** accepts input from the user's input devices and analyses the tokens and syntactic structure of the commands. That is, it function like lexical and syntactic phases of a compiler. It invokes the semantic routines directly. The command language processor also produces an intermediate representation of the desired editing operations. This representation is decoded by an interpreter that invokes the appropriate semantic routines.

- **Editing Component -** 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..,.

  - **Travelling component** – performs the setting of the current editing and viewing pointers and thus determines the point at which the viewing/editing filtering begins.
  - **Editing filter-** When the user issues an editing command the editing component invokes the editing filter. This component filters the document to generate a new **editing buffer** based on the current editing pointer as well as on the editing filter parameters.

  - **Filtering** consists of selection of continuous characters beginning at the current point.

  - **Viewing component**- thee start of the area to be viewed is determined by the viewing pointer. This pointer is maintained by the viewing component. When the display need to be updated the viewing component invokes the **viewing filter**. This component filters the document to generate a new **viewing buffer**.

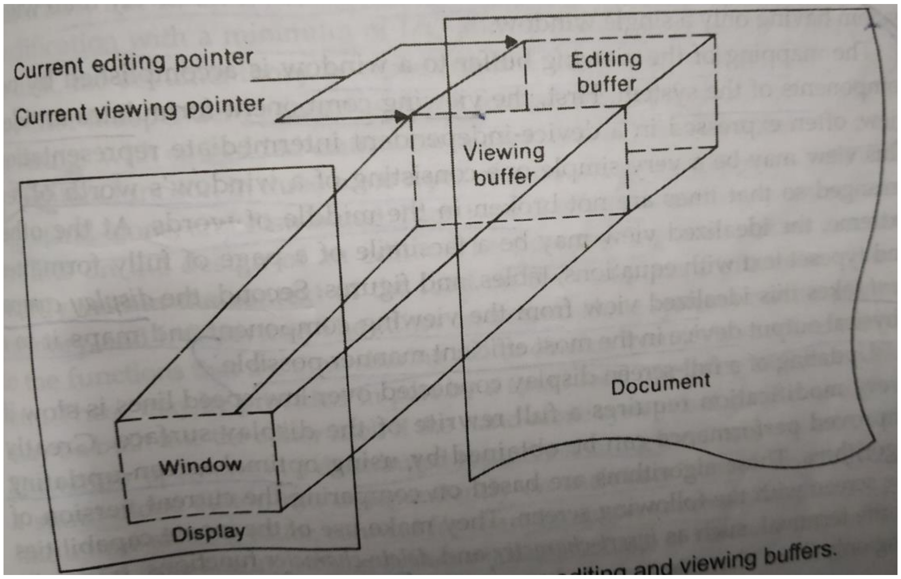
  - **Display component**- The viewing buffer is then passed to the display component which produces a display by mapping the buffer to a rectangular subset of the screen called window.

  - The editing and viewing buffers can be independent or overlapped.

  - The mapping of viewing buffer to window is accomplished by two components.

    1. Viewing component- formulates an ideal view

    2. Display component – takes this ideal view from viewing component and maps it to the output device.



Simple relationship between editing and viewing buffers

- 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.
- Many editors use terminal control database. They can call terminal independent library routines such as scroll down, or read cursor positions.
- Types of editors based on computing environment: Editors function in three basic types of computing environments:

1. Time sharing
2. Stand-alone
3. 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.

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

- Debugging methods

Debugging Functions and Capabilities:

- One important requirement of any IDS is unit test functions specified by the programmer. Such functions deal with execution sequencing , which is the observation and control of the flow of program execution.Eg: The program may be suspended after a fixed number of instructions are executed. The programmer can define break points. After the program is suspended debugging commands can be used to diagnose errors.
- A Debugging system should also provide functions such as tracing and trace back

- 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.
- Trace back 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.
- 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.

- A debugging system 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 debugging system should be able to deal with optimized code. Many optimizations involve rearrangement of code in the program.Eg: Separate loops can be combined into single loop.
- Storage of variables- When a program is translated the compiler assigns a home location in memory for each variables. Variable values can be temporarily held in registers to improve speed of access. If a user changes the value of a variable in home location while debugging the modified value might not be used by the program.
- The debugging of optimized code requires cooperation from optimized compiler.

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.

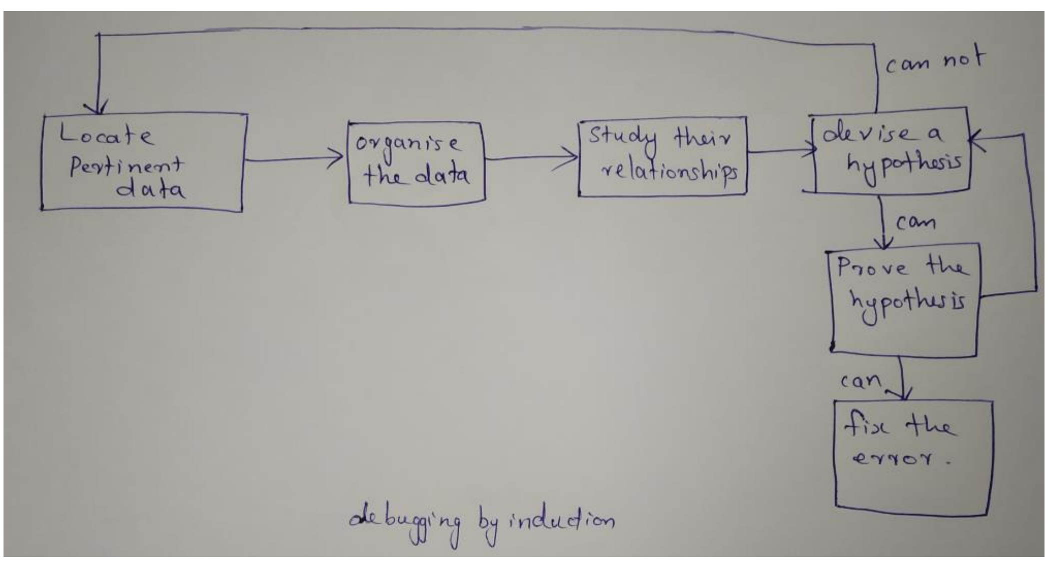
- Debugging is more important at production time than it is at application-development time. When an application fails during a production run, work dependent on that application stops.
- The debugger must also exist in a way that is consistent with the security and integrity components of the system.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

Debugging Methods

1. Debugging by Induction
2. Debugging by Deduction
3. Debugging by Backtracking

Debugging by Induction

- In induction one proceeds from the particulars to the whole.ie, By starting with the symptoms of the error in the result of one or more test cases and looking for relationships among the symptoms.



1. Locate the pertinent data: Consider all the available data or symptoms about the problems

2. Organise the data: Pertinent data is structured to allow one to observe patterns of particular importance and search for contradictions. One such organization structure can be a table.

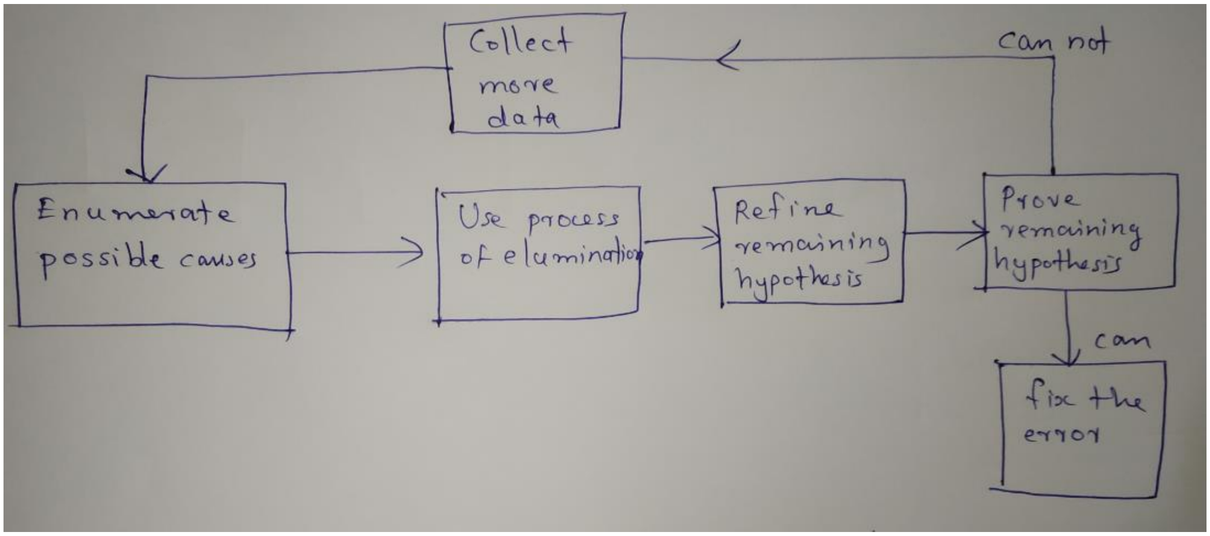
3. Devise a hypothesis: In this step study the relationship between the clues and devise using patterns, one or more hypothesis about the cause o the error.

4. Prove the hypothesis: Prove the reasonableness of the hypothesis before proceeding. A failure to this, results in the fixing of only one symptom of the problem.

Debugging by Deduction

- Is a process of proceeding from general theories or premises to arrive at a conclusion.
    1. Enumerate all possible cases- The first step is to develop all causes of the error.
    2. Use the data to eliminate possible causes- By careful analysis of data particularly by looking for contradictions attempt to eliminate all possible causes except one.
    3. Refine the remaining hypothesis- The possible causes at this point may be correct. But refine it to be more specific.

    4. Prove the remaining hypothesis.



Debugging by Back Tracking

- For small programs the method of backtracking is more effective to locate errors.
- To use this method start at the place in the program where an incorrect result was produced and go backwards in the program one step at a time. That is executing the program in reverse order to derive the values of all variables in the previous step. Then the error can be localized.

Device Driver

A device driver is a particular form of software application that allows one hardware device (such as a personal computer) to interact with another hardware device (such as a printer). A device driver may also be called a *software driver*.

Drivers facilitate communication between an operating system and a peripheral hardware device. Each driver contains knowledge about a particular hardware device or software interface that other programs -- including the underlying operating system (OS) -- does not have.

In the past, device drivers were written for specific operating systems and specific hardware peripherals. If a peripheral device was not recognized by their computer's OS, the end user had to locate and manually install the right driver.

Today, most operating systems include a library of plug-n-play drivers that allows peripheral hardware to connect automatically with an operating system. This approach also has the advantage of allowing programmers to write high-level application code without needing to know what hardware their code will run on