

## **CST305 – SYSTEM SOFTWARE**

### **UNIT I INTRODUCTION**

**8**

System software and machine architecture – The Simplified Instructional Computer (SIC)  
- Machine architecture - Data and instruction formats - addressing modes - instruction sets - I/O and programming.

### **UNIT II ASSEMBLERS**

**10**

Basic assembler functions - A simple SIC assembler – Assembler algorithm and data structures - Machine dependent assembler features - Instruction formats and addressing modes – Program relocation - Machine independent assembler features - Literals – Symbol-defining statements – Expressions - One pass assemblers and Multi pass assemblers - Implementation example - MASM assembler.

### **UNIT III LOADERS AND LINKERS**

**9**

Basic loader functions - Design of an Absolute Loader – A Simple Bootstrap Loader - Machine dependent loader features - Relocation – Program Linking – Algorithm and Data Structures for Linking Loader - Machine-independent loader features - Automatic Library Search – Loader Options - Loader design options - Linkage Editors – Dynamic Linking – Bootstrap Loaders - Implementation example - MSDOS linker.

### **UNIT IV MACRO PROCESSORS**

**9**

Basic macro processor functions - Macro Definition and Expansion – Macro Processor Algorithm and data structures - Machine-independent macro processor features - Concatenation of Macro Parameters – Generation of Unique Labels – Conditional Macro Expansion – Keyword Macro Parameters-Macro within Macro-Implementation example - MASM Macro Processor – ANSI C Macro language.

### **UNIT V SYSTEM SOFTWARE TOOLS**

**9**

Text editors - Overview of the Editing Process - User Interface – Editor Structure. - Interactive debugging systems - Debugging functions and capabilities – Relationship with other parts of the system – User-Interface Criteria.

**TEXT BOOK** 1. Leland L. Beck, “System Software – An Introduction to Systems Programming”, 3rd Edition, Pearson Education Asia, 2006.

### **REFERENCES**

1. D. M. Dhamdhere, “Systems Programming and Operating Systems”, Second Revised Edition, Tata McGraw-Hill, 2000.
2. John J. Donovan “Systems Programming”, Tata McGraw-Hill Edition, 2000.

## **UNIT I**

# **INTRODUCTION TO SYSTEM SOFTWARE AND MACHINE STRUCTURE**

### **SYSTEM SOFTWARE**

- System software consists of a variety of programs that support the operation of a computer.
- It is a set of programs to perform a variety of system functions as file editing, resource management, I/O management and storage management.
- The characteristic in which system software differs from application software is machine dependency.
- An application program is primarily concerned with the solution of some problem, using the computer as a tool.
- System programs on the other hand are intended to support the operation and use of the computer itself, rather than any particular application.
- For this reason, they are usually related to the architecture of the machine on which they are run.
- For example, assemblers translate mnemonic instructions into machine code. The instruction formats, addressing modes are of direct concern in assembler design.
- There are some aspects of system software that do not directly depend upon the type of computing system being supported. These are known as machine-independent features.
- For example, the general design and logic of an assembler is basically the same on most computers.

### **TYPES OF SYSTEM SOFTWARE:**

1. Operating system
2. Language translators
  - a. Compilers
  - b. Interpreters
  - c. Assemblers
  - d. Preprocessors
3. Loaders
4. Linkers
5. Macro processors

### **OPERATING SYSTEM**

- It is the most important system program that act as an interface between the users and the system. It makes the computer easier to use.

- It provides an interface that is more user-friendly than the underlying hardware.
- The functions of OS are:
  1. Process management
  2. Memory management
  3. Resource management
  4. I/O operations
  5. Data management
  6. Providing security to user's job.

## **LANGUAGE TRANSLATORS**

It is the program that takes an input program in one language and produces an output in another language.



### **Compilers**

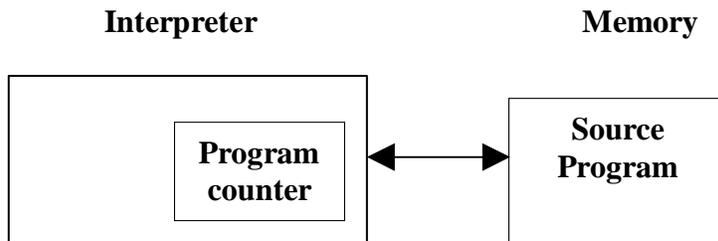
- A compiler is a language program that translates programs written in any high-level language into its equivalent machine language program.
- It bridges the semantic gap between a programming language domain and the execution domain.
- Two aspects of compilation are:
  - Generate code to increment meaning of a source program in the execution domain.
  - Provide diagnostics for violation of programming language, semantics in a source program.
- The program instructions are taken as a whole.



### **Interpreters:**

- It is a translator program that translates a statement of high-level language to machine language and executes it immediately. The program instructions are taken line by line.
- The interpreter reads the source program and stores it in memory.

- During interpretation, it takes a source statement, determines its meaning and performs actions which increments it. This includes computational and I/O actions.
- Program counter (PC) indicates which statement of the source program is to be interpreted next. This statement would be subjected to the interpretation cycle.
- The interpretation cycle consists of the following steps:
  - Fetch the statement.
  - Analyze the statement and determine its meaning.
  - Execute the meaning of the statement.
- The following are the characteristics of interpretation:
  - The source program is retained in the source form itself, no target program exists.
  - A statement is analyzed during the interpretation.



### Assemblers:

- Programmers found it difficult to write or red programs in machine language. In a quest for a convenient language, they began to use a mnemonic (symbol) for each machine instructions which would subsequently be translated into machine language.
- Such a mnemonic language is called Assembly language.
- Programs known as Assemblers are written to automate the translation of assembly language into machine language.



- Fundamental functions:
  1. Translating mnemonic operation codes to their machine language equivalents.
  2. Assigning machine addresses to symbolic tables used by the programmers.

## THE SIMPLIFIED INSTRUCTIONAL COMPUTER (SIC):

It is similar to a typical microcomputer. It comes in two versions:

- The standard model
- XE version

### SIC Machine Structure:

#### Memory:

- It consists of bytes(8 bits) ,words (24 bits which are consecutive 3 bytes) addressed by the location of their lowest numbered byte.
- There are totally 32,768 bytes in memory.

#### Registers:

There are 5 registers namely

1. Accumulator (A)
  2. Index Register(X)
  3. Linkage Register(L)
  4. Program Counter(PC)
  5. Status Word(SW).
- Accumulator is a special purpose register used for arithmetic operations.
  - Index register is used for addressing.
  - Linkage register stores the return address of the jump of subroutine instructions (JSUB).
  - Program counter contains the address of the current instructions being executed.
  - Status word contains a variety of information including the condition code.

#### Data formats:

- Integers are stored as 24-bit binary numbers: 2's complement representation is used for negative values characters are stored using their 8 bit ASCII codes.
- They do not support floating – point data items.

#### Instruction formats:

All machine instructions are of 24-bits wide

<b>Opcode (8)</b>	<b>X (1)</b>	<b>Address (15)</b>
-------------------	--------------	---------------------

- X-flag bit that is used to indicate indexed-addressing mode.

#### Addressing modes:

- Two types of addressing are available namely,
  1. Direct addressing mode
  2. Indexed addressing mode or indirect addressing mode

Mode	Indication	Target Address calculation
Direct	X=0	TA=Address
Indexed	X=1	TA=Address + (X)

- Where(x) represents the contents of the index register(x)

### Instruction set:

It includes instructions like:

1. Data movement instruction  
Ex: LDA, LDX, STA, STX.
2. Arithmetic operating instructions  
Ex: ADD, SUB, MUL, DIV.  
This involves register A and a word in memory, with the result being left in the register.
3. Branching instructions  
Ex: JLT, JEQ, TGT.
4. Subroutine linkage instructions  
Ex: JSUB, RSUB.

### Input and Output:

- I/O is performed by transferring one byte at a time to or from the rightmost 8 bits of register A.
- Each device is assigned a unique 8-bit code.
- There are 3 I/O instructions,
  - 1) The Test Device (TD) instructions tests whether the addressed device is ready to send or receive a byte of data.
  - 2) A program must wait until the device is ready, and then execute a Read Data (RD) or Write Data (WD).
  - 3) The sequence must be repeated for each byte of data to be read or written.

## SIC/XE ARCHITECTURE & SYSTEM SPECIFICATION

**Memory:**

- 1 word = 24 bits (3 8-bit bytes)
- Total (SIC/XE) =  $2^{20}$  (1,048,576) bytes (1Mbyte)

**Registers:**

- 10 x 24 bit registers

MNEMONIC	Register	Purpose
A	0	Accumulator
X	1	Index register
L	2	Linkage register (JSUB/RSUB)
B	3	Base register
S	4	General register
T	5	General register
F	6	Floating Point Accumulator (48 bits)
PC	8	Program Counter (PC)
SW	9	Status Word (includes Condition Code, CC)

**Data Format:**

- Integers are stored in 24 bit, 2's complement format
- Characters are stored in 8-bit ASCII format
- Floating point is stored in 48 bit signed-exponent-fraction format:

s	exponent {11}	fraction {36}
---	---------------	---------------

- The fraction is represented as a 36 bit number and has value between 0 and 1.
- The exponent is represented as a 11 bit unsigned binary number between 0 and 2047.
- The sign of the floating point number is indicated by s : 0=positive, 1=negative.
- Therefore, the absolute floating point number value is:  $f * 2^{(e-1024)}$

**Instruction Format:**

- There are 4 different instruction formats available:

Format 1 (1 byte):

op {8}
--------

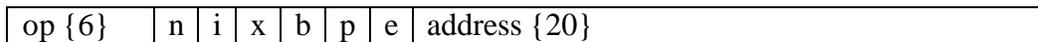
Format 2 (2 bytes):



Format 3 (3 bytes):



Format 4 (4 bytes):



Formats 3 & 4 introduce addressing mode flag bits:

- n=0 & i=1  
Immediate addressing - TA is used as an operand value (no memory reference)
- n=1 & i=0  
Indirect addressing - word at TA (in memory) is fetched & used as an address to fetch the operand from
- n=0 & i=0  
Simple addressing TA is the location of the operand
- n=1 & i=1  
Simple addressing same as n=0 & i=0

Flag x:

x=1 Indexed addressing add contents of X register to TA calculation

Flag b & p (Format 3 only):

- b=0 & p=0  
Direct addressing displacement/address field contains TA (Format 4 always uses direct addressing)
- b=0 & p=1  
PC relative addressing -  $TA = (PC) + disp$  ( $-2048 \leq disp \leq 2047$ )\*
- b=1 & p=0  
Base relative addressing -  $TA = (B) + disp$  ( $0 \leq disp \leq 4095$ )\*\*

Flag e:

e=0 use Format 3  
e=1 use Format 4

## Instructions:

SIC provides 26 instructions, SIC/XE provides an additional 33 instructions (59 total)

SIC/XE has 9 categories of instructions:

- Load/store registers (LDA, LDX, LDCH, STA, STX, STCH, etc.)
- integer arithmetic operations (ADD, SUB, MUL, DIV) these will use register A and a word in memory, results are placed into register A
- compare (COMP) compares contents of register A with a word in memory and sets CC (Condition Code) to <, >, or =
- conditional jumps (JLT, JEQ, JGT) - jumps according to setting of CC
- subroutine linkage (JSUB, RSUB) - jumps into/returns from subroutine using register L
- input & output control (RD, WD, TD) - see next section
- floating point arithmetic operations (ADDF, SUBF, MULF, DIVF)
- register manipulation, operands-from-registers, and register-to-register arithmetics (RMO, RSUB, COMPR, SHIFTR, SHIFTL, ADDR, SUBR, MULR, DIVR, etc)

## Input and Output (I/O):

- $2^8$  (256) I/O devices may be attached, each has its own unique 8-bit address
- 1 byte of data will be transferred to/from the rightmost 8 bits of register A

Three I/O instructions are provided:

- RD Read Data from I/O device into A
- WD Write data to I/O device from A
- TD Test Device determines if addressed I/O device is ready to send/receive a byte of data. The CC (Condition Code) gets set with results from this test:  
*< device is ready to send/receive*  
*= device isn't ready*

SIC/XE Has capability for programmed I/O (I/O device may input/output data while CPU does other work) - 3 additional instructions are provided:

- SIO Start I/O
- HIO Halt I/O
- TIO Test I/O

## SIC, SIC/XE ADDRESSING MODES

Addressing Type	Flag Bits					Notation	Calculation of Target Address	Operand	Notes
	n	i	x	b	p				

Simple	1	1	0	0	0	0	op c	disp	(TA)	Direct-addressing Instruction
	1	1	0	0	0	1	+op m	addr	(TA)	Format 4 & Direct-addressing Instruction
	1	1	0	0	1	0	op m	(PC) + disp	(TA)	Assembler selects either base-relative or program-counter relative mode
	1	1	0	1	0	0	op m	(B) + disp	(TA)	Assembler selects either base-relative or program-counter relative mode
	1	1	1	0	0	0	op c,X	disp + (X)	(TA)	Direct-addressing Instruction
	1	1	1	0	0	1	+op m,X	addr + (X)	(TA)	Format 4 & Direct-addressing Instruction
	1	1	1	0	1	0	op m,X	(PC) + disp + (X)	(TA)	Assembler selects either base-relative or program-counter relative mode
	1	1	1	1	0	0	op m,X	(B) + disp + (X)	(TA)	Assembler selects either base-relative or program-counter relative mode
	0	0	0	-	-	-	op m	b/p/e/disp	(TA)	Direct-addressing Instruction; SIC compatible format.
	0	0	1	-	-	-	op m,X	b/p/e/disp + (X)	(TA)	Direct-addressing Instruction; SIC compatible format.
Indirect	1	0	0	0	0	0	op @c	disp	((TA))	Direct-addressing Instruction
	1	0	0	0	0	1	+op @m	addr	((TA))	Format 4 & Direct-addressing Instruction
	1	0	0	0	1	0	op @m	(PC) + disp	((TA))	Assembler selects either base-relative or program-counter relative mode
	1	0	0	1	0	0	op @m	(B) + disp	((TA))	Assembler selects

												either base-relative or program-counter relative mode
Immediate	0	1	0	0	0	0	op #c	disp	TA			Direct-addressing Instruction
	0	1	0	0	0	1	op #m	addr	TA			Format 4 & Direct-addressing Instruction
	0	1	0	0	1	0	op #m	(PC) + disp	TA			Assembler selects either base-relative or program-counter relative mode
	0	1	0	1	0	0	op #m	(B) + disp	TA			Assembler selects either base-relative or program-counter relative mode

## UNIT II

### ASSEMBLERS

#### BASIC ASSEMBLER FUNCTIONS

Fundamental functions of an assembler:

- Translating mnemonic operation codes to their machine language equivalents.
- Assigning machine addresses to symbolic labels used by the programmer.

**Figure 2.1: Assembler language program for basic SIC version**

Line	Source statement			
5	COPY	START	1000	COPY FILE FROM INPUT TO OUTPUT
10	FIRST	STL	RETADR	SAVE RETURN ADDRESS
15	CLOOP	<u>JSUB</u>	RDREC	READ INPUT RECORD
20		LDA	LENGTH	TEST FOR EOF (LENGTH = 0)
25		COMP	ZERO	
30		<u>JEQ</u>	ENDFIL	EXIT IF EOF FOUND
35		<u>JSUB</u>	WRREC	WRITE OUTPUT RECORD
40		<u>J</u>	CLOOP	LOOP
45	ENDFIL	LDA	EOF	INSERT END OF FILE MARKER
50		STA	BUFFER	
55		LDA	THREE	SET LENGTH = 3
60		STA	LENGTH	
65		<u>JSUB</u>	WRREC	WRITE EOF
70		<u>LDL</u>	RETADR	GET RETURN ADDRESS
75		<u>RSUB</u>		RETURN TO CALLER
80	EOF	BYTE	C'EOF'	
85	THREE	WORD	3	
90	ZERO	WORD	0	
95	RETADR	RESW	1	
100	LENGTH	RESW	1	LENGTH OF RECORD
105	BUFFER	RESB	4096	4096-BYTE BUFFER AREA
110	.			
115	.			SUBROUTINE TO READ RECORD INTO BUFFER
120	.			

125	RDREC	LDX	ZERO	CLEAR LOOP COUNTER
130		LDA	ZERO	CLEAR A TO ZERO
135	RLOOP	TD	INPUT	TEST INPUT DEVICE
140		<u>JEQ</u>	RLOOP	LOOP UNTIL READY
145		<u>RD</u>	INPUT	READ CHARACTER INTO REGISTER A
150		COMP	ZERO	TEST FOR END OF RECORD (X'00')
155		<u>JEQ</u>	EXIT	EXIT LOOP IF EOR
160		<u>STCH</u>	BUFFER,X	STORE CHARACTER IN BUFFER
165		TIJ	MAXLEN	LOOP UNLESS MAX LENGTH
170		<u>JLT</u>	RLOOP	HAS BEEN REACHED
175	EXIT	STX	LENGTH	SAVE RECORD LENGTH
180		<u>RSUB</u>		RETURN TO CALLER
185	INPUT	BYTE	X'F1'	CODE FOR INPUT DEVICE
190	MAXLEN	WORD	4096	
195	.			
200	.			SUBROUTINE TO WRITE RECORD FROM BUFFER
205	.			
210	WRREC	LDX	ZERO	CLEAR LOOP COUNTER
215	WLOOP	TD	OUTPUT	TEST OUTPUT DEVICE
220		<u>JEQ</u>	WLOOP	LOOP UNTIL READY
225		<u>LDCH</u>	BUFFER,X	GET CHARACTER FROM BUFFER
230		<u>WD</u>	OUTPUT	WRITE CHARACTER
235		TIJ	LENGTH	LOOP UNTIL ALL CHARACTERS
240		<u>JLT</u>	WLOOP	HAVE BEEN WRITTEN
245		<u>RSUB</u>		RETURN TO CALLER
250	OUTPUT	BYTE	X'05'	CODE FOR OUTPUT DEVICE
255		END	FIRST	

Indexed addressing is indicated by adding the modifier “X” following the operand. Lines beginning with “.” contain comments only.

The following assembler directives are used:

- **START:** Specify name and starting address for the program.
- **END :** Indicate the end of the source program and 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.

The program contains a main routine that reads records from an input device( code F1) and copies them to an output device(code 05).

The main routine calls subroutines:

- **RDREC** – To read a record into a buffer.

- **WRREC** – To write the record from the buffer to the output device.

The end of each record is marked with a null character (hexadecimal 00).

## A Simple SIC Assembler

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

1. Convert mnemonic operation codes to their machine language equivalents. Eg: Translate STL to 14 (line 10).
2. Convert symbolic operands to their equivalent machine addresses. Eg: Translate RETADR to 1033 (line 10).
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(line 80).
5. Write the object program and the assembly listing.

All functions except function 2 can be established by sequential processing of source program one line at a time.

Consider the statement

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

This instruction contains a **forward reference** (i.e.) a reference to a label (RETA DR) that is defined later in the program. It is unable to process this line because the address that will be assigned to RETADR is not known. Hence most assemblers make two passes over the source program where the second pass does the actual translation.

The assembler must also process statements called **assembler directives or pseudo instructions** which are not translated into machine instructions. Instead they provide instructions to the assembler itself.

Examples: RESB and RESW instruct the assembler to reserve memory locations without generating data values.

The assembler must write the generated object code onto some output device. This object program will later be loaded into memory for execution.

### Object program format contains three types of records:

- **Header record:** Contains the program name, starting address and length.
- **Text record:** Contains the machine code and data of the program.
- **End record:** Marks the end of the object program and specifies the address in the program where execution is to begin.

**Record format is as follows:**

**Header record:**

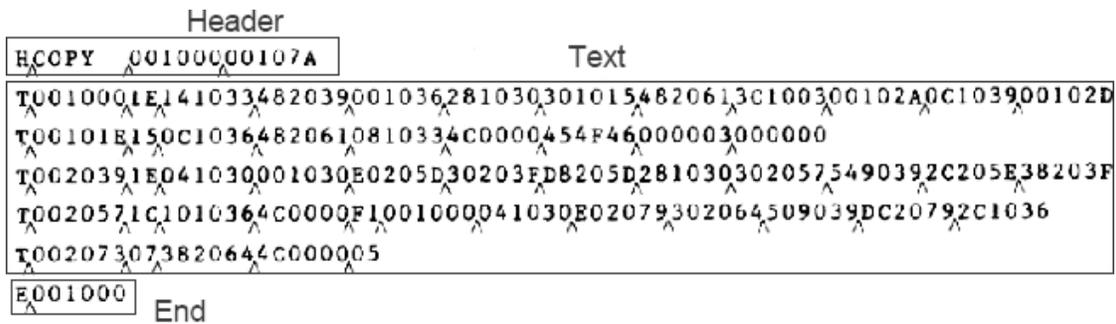
Col. 1            H  
 Col.2-7         Program name  
 Col.8-13        Starting address of object program  
 Col.14-19      Length of object program in bytes

**Text record:**

Col.1            T  
 Col.2-7         Starting address for object code in this record  
 Col.8-9         Length of object code in this record in bytes  
 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.



**Figure 2.3** Object program corresponding to Fig. 2.2.

**Functions of the two passes of assembler:**

**Pass 1 (Define symbols)**

1. Assign addresses to all statements in the program.
2. Save the addresses assigned to all labels for use in Pass 2.
3. Perform some processing of assembler directives.

**Pass 2 (Assemble instructions and generate object programs)**

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

## **Assembler Algorithm and Data Structures**

Assembler uses two major internal data structures:

1. **Operation Code Table (OPTAB)** : Used to lookup mnemonic operation codes and translate them into their machine language equivalents.
2. **Symbol Table (SYMTAB)** : Used to store values(Addresses) assigned to labels.

### **Location Counter (LOCCTR) :**

- Variable used to help in the assignment of addresses.
- It is initialized to the beginning address specified in the START statement.
- After each source statement is processed, the length of the assembled instruction or data area is added to LOCCTR.
- Whenever a label is reached in the source program, the current value of LOCCTR gives the address to be associated with that label.

### **Operation Code Table (OPTAB) :**

- Contains the mnemonic operation and its machine language equivalent.
- Also contains information about instruction format and length.
- In Pass 1, OPTAB is used to lookup and validate operation codes in the source program.
- In Pass 2, it is used to translate the operation codes to machine language program.
- During Pass 2, the information in OPTAB tells which instruction format to use in assembling the instruction and any peculiarities of the object code instruction.

### **Symbol Table (SYMTAB) :**

- Includes the name and value for each label in the source program and flags to indicate error conditions.
- During Pass 1 of the assembler, labels are entered into SYMTAB as they are encountered in the source program along with their assigned addresses.
- During Pass 2, symbols used as operands are looked up in SYMTAB to obtain the addresses to be inserted in the assembled instructions.

Pass 1 usually writes an intermediate file that contains each source statement together with its assigned address, error indicators. This file is used as the input to Pass 2. This copy of the source program can also be used to retain the results of certain operations that

may be performed during Pass 1 such as scanning the operand field for symbols and addressing flags, so these need not be performed again during Pass 2.

### **MACHINE DEPENDENT ASSEMBLER FEATURES**

Consider the design and implementation of an assembler for SIC/XE version.

5	COPY	START	0	COPY FILE FROM INPUT TO OUTPUT
10	FIRST	STL	RETADR	SAVE RETURN ADDRESS
12		LDB	#LENGTH	ESTABLISH BASE REGISTER
13		BASE	LENGTH	
15	CLOOP	+JSUB	RDREC	READ INPUT RECORD
20		LDA	LENGTH	TEST FOR EOF (LENGTH = 0)
25		COMP	#0	
30		JEQ	ENDFIL	EXIT IF EOF FOUND
35		+JSUB	WRREC	WRITE OUTPUT RECORD
40		J	CLOOP	LOOP
45	ENDFIL	LDA	EOF	INSERT END OF FILE MARKER
50		STA	BUFFER	
55		LDA	#3	SET LENGTH = 3
60		STA	LENGTH	
65		+JSUB	WRREC	WRITE EOF
70		J	@RETADR	RETURN TO CALLER
80	EOF	BYTE	C'EOF'	
95	RETADR	RESW	1	
100	LENGTH	RESW	1	LENGTH OF RECORD
105	BUFFER	RESB	4096	4096-BYTE BUFFER AREA
110	.			

```

---
115      .          SUBROUTINE TO READ RECORD INTO BUFFER
120      .
125      RDREC      CLEAR    X          CLEAR LOOP COUNTER
130              CLEAR    A          CLEAR A TO ZERO
132              CLEAR    S          CLEAR S TO ZERO
133              +LDT     #4096
135      RLOOP      TD      INPUT      TEST INPUT DEVICE
140              JEQ     RLOOP      LOOP UNTIL READY
145              RD      INPUT      READ CHARACTER INTO REGISTER A
150              COMPR   A,S        TEST FOR END OF RECORD (X'00')
155              JEQ     EXIT      EXIT LOOP IF EOR
160              STCH    BUFFER,X    STORE CHARACTER IN BUFFER
165              TIXR    T          LOOP UNLESS MAX LENGTH
170              JLT     RLOOP      HAS BEEN REACHED
175      EXIT      STX     LENGTH     SAVE RECORD LENGTH
180              RSUB
185      INPUT     BYTE    X'F1'     CODE FOR INPUT DEVICE
195      .

```

Indirect addressing is indicated by adding the prefix @ to the operand (line70). Immediate operands are denoted with the prefix # (lines 25, 55,133). Instructions that refer to memory are normally assembled using either the program counter relative or base counter relative mode.

The assembler directive BASE (line 13) is used in conjunction with base relative addressing. The four byte extended instruction format is specified with the prefix + added to the operation code in the source statement.

Register-to-register instructions are used wherever possible. For example the statement on line 150 is changed from COMP ZERO to COMPR A,S. Immediate and indirect addressing have also been used as much as possible.

Register-to-register instructions are faster than the corresponding register-to-memory operations because they are shorter and do not require another memory reference. While using immediate addressing, the operand is already present as part of the instruction and need not be fetched from anywhere. The use of indirect addressing often avoids the need for another instruction.

## Instruction Formats and Addressing Modes

- SIC/XE
  - PC-relative or Base-relative addressing: op m
  - Indirect addressing: op @m
  - Immediate addressing: op #c
  - Extended format: +op m
  - Index addressing: op m,x
  - register-to-register instructions
  - larger memory -> multi-programming (program allocation)

### Translation

- Register translation
  - register name (A, X, L, B, S, T, F, PC, SW) and their values (0,1, 2, 3, 4, 5, 6, 8, 9)
  - preloaded in SYMTAB
- Address translation
  - Most register-memory instructions use program counter relative or base relative addressing
  - Format 3: 12-bit address field
    - base-relative: 0~4095
    - pc-relative: -2048~2047
  - Format 4: 20-bit address field

## Program Relocation

The need for program relocation

- It is desirable to load and run several programs at the same time.
- The system must be able to load programs into memory wherever there is room.
- The exact starting address of the program is not known until load time.

Absolute Program

- Program with starting address specified at assembly time
- The address may be invalid if the program is loaded into somewhere else.
- Example:

```

55      101B      LDA      THREE      00102D

```

Calculate based on the starting address 1000

**Reload the program starting at 3000**

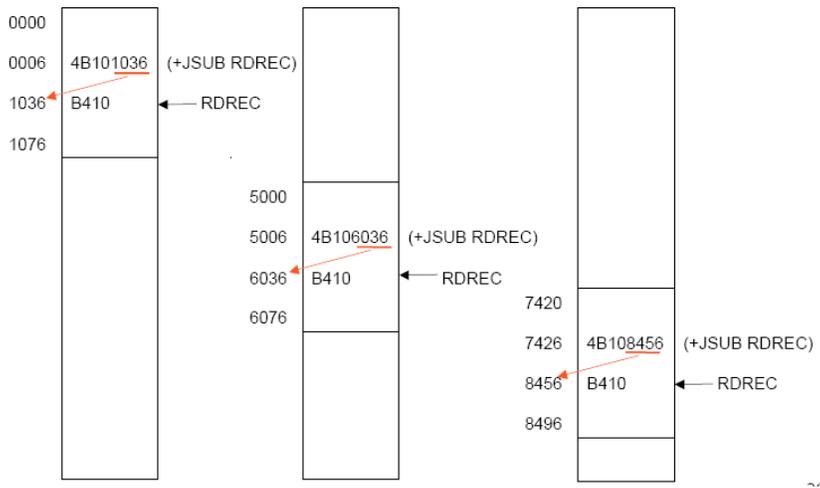
```

55      101B      LDA      THREE      00302D

```

The absolute address should be modified

### Example: Program Relocation



- The only parts of the program that require modification at load time are those that specify direct addresses.
- The rest of the instructions need not be modified.
  - Not a memory address (immediate addressing)
  - PC-relative, Base-relative
- From the object program, it is not possible to distinguish the address and constant.
  - The assembler must keep some information to tell the loader.
  - The object program that contains the modification record is called a relocatable program.

The way to solve the relocation problem

- For an address label, its address is assigned relative to the start of the program (START 0)
- Produce 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.

### Modification record

- One modification record 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.

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)

### Relocatable Object Program

```

HCOPY 00000001077
T0000001D17202D69202D4B1010360320262900003320074B10105D3F2FEC032010
T00001D130F20160100030F200D4B10105D3E2003454F46
T0010361DB410B400B44075101000E32019332FFADB2013A00433200857C003B850
T0010531D3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T001070073B2FEF4E000005
M00000705
M00001405
M00002705
E000000

```

The image shows a disassembly of a program with annotations. A red box highlights the instruction `T0010531D3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850`. A blue arrow points from the text "5 half-bytes" to the first five bytes of this instruction: `1D3B2FEA13`. Another blue arrow points from the text "5 half-bytes" to the next five bytes: `4F0000F1B4`. A third blue arrow points from the text "5 half-bytes" to the final five bytes: `10774000E3`. The instruction is also highlighted with a red box.

## MACHINE INDEPENDENT ASSEMBLER FEATURES

### Literals

- The programmer writes the value of a constant operand as a part of the instruction that uses it. This avoids having to define the constant elsewhere in the program and make a label for it.
- Such an operand is called a Literal because the value is literally in the instruction.

- Consider the following example

```

      :
      LDA      FIVE
      :
FIVE  :
      WORD    5
      :

```

⇒ 

```

      :
      LDA      =X'05'
      :

```

- It is convenient to write the value of a constant operand as a part of instruction.
- A literal is identified with the prefix =, followed by a specification of the literal value.

- Example:

```

45  001A ENDFIL LDA  =C' EOF'          032010
                                     nixbpe disp
                                     000000 110010 010
93                                     LORG
002D *                                =C' EOF'          454F46
215 1062 WLOOP TD   =X' 05'          E32011
230 106B          WD  =X' 05'          DF2008
1076 *                                =X' 05'          05

```

### Literals vs. Immediate Operands

- Literals

The assembler generates the specified value as a constant at some other memory location.

```

45  .. 001A ENDFIL LDA  =C' EOF'          032010

```

- Immediate Operands

```

55  0020          LDA  #3          010003

```

The operand value is assembled as part of the machine instruction

- We can have literals in SIC, but immediate operand is only valid in SIC/XE.

### Literal Pools

- Normally literals are placed into a pool at the end of the program
- In some cases, it is desirable to place literals into a pool at some other location in the object program
- Assembler directive LTORG
  - When the assembler encounters a LTORG statement, it generates a *literal pool* (containing all literal operands used since previous LTORG)
- Reason: keep the literal operand close to the instruction
  - Otherwise PC-relative addressing may not be allowed

### Duplicate literals

- The same literal used more than once in the program
  - Only one copy of the specified value needs to be stored
  - For example, =X'05'
- In order to recognize the duplicate literals
  - Compare the character strings defining them
    - Easier to implement, but has potential problem
    - e.g. =X'05'
  - Compare the generated data value
    - Better, but will increase the complexity of the assembler
    - e.g. =C'EOF' and =X'454F46'

### Problem of duplicate-literal recognition

- '\*' denotes a literal refer to the current value of program counter
  - BUFEND EQU \*
- There may be some literals that have the same name, but different values
  - BASE \*
  - LDB =\* (#LENGTH)
- The literal =\* repeatedly used in the program has the same name, but different values
- The literal "=\*" represents an "address" in the program, so the assembler must generate the appropriate "Modification records".

### Literal table - LITTAB

- Content
  - Literal name
  - Operand value and length
  - Address
- LITTAB is often organized as a hash table, using the literal name or value as the key.

## Implementation of Literals

### Pass 1

- Build LITTAB with literal name, operand value and length, leaving the address unassigned
- When LORG or END statement is encountered, assign an address to each literal not yet assigned an address
  - updated to reflect the number of bytes occupied by each literal

### Pass 2

- Search LITTAB for each literal operand encountered
- Generate data values using BYTE or WORD statements
- Generate Modification record for literals that represent an address in the program

## SYMTAB & LITTAB

### SYMTAB

Name	Value
COPY	0
FIRST	0
CLOOP	6
ENDFIL	1A
RETADR	30
LENGTH	33
BUFFER	36
BUFEND	1036
MAXLEN	1000
RDREC	1036
RLOOP	1040
EXIT	1056
INPUT	105C
WREC	105D
WLOOP	1062

### LITTAB

Literal	Hex Value	Length	Address
C' EOF'	454F46	3	002D
X' 05'	05	1	1076

## Symbol-Defining Statements

- Most assemblers provide an assembler directive that allows the programmer to define symbols and specify their values.

Assembler directive used is EQU.

- Syntax: symbol EQU value
- Used to improve the program readability, avoid using magic numbers, make it easier to find and change constant values
- Replace +LDT #4096 with  

```
MAXLEN EQU 4096
+LDT      #MAXLEN
```
- Define mnemonic names for registers.  

```
A EQU 0 RMO A,X
X EQU 1
```
- Expression is allowed  

```
MAXLEN EQU BUFEND-BUFFER
```

Assembler directive ORG

- Allow the assembler to reset the PC to values
  - Syntax: ORG value
- When ORG is encountered, the assembler resets its LOCCTR to the specified value.
- ORG will affect the values of all labels defined until the next ORG.
- If the previous value of LOCCTR can be automatically remembered, we can return to the normal use of LOCCTR by simply writing
  - ORG

### Example: using ORG

- If ORG statements are used
 

STAB	RESB	1100	
	ORG	STAB	← Set LOCCTR to STAB
SYMBOL	RESB	6	
VALUE	RESW	1	
FLAGS	RESB	2	← Size of each field
	ORG	STAB+1100	← Restore LOCCTR

- We can fetch the VALUE field by  

```
LDA VALUE,X
X = 0, 11, 22, ... for each entry
```

### Forward-Reference Problem

- Forward reference is not allowed for either EQU or ORG.
- All terms in the value field must have been defined previously in the program.
- The reason is that all symbols must have been defined during Pass 1 in a two-pass assembler.
- Allowed:
 

ALPHA	RESW	1
BETA	EQU	ALPHA
- Not Allowed:
 

BETA	EQU	ALPHA
ALPHA	RESW	1

## Expressions

- The assemblers allow “the use of expressions as operand”
- The assembler evaluates the expressions and produces a single operand address or value.
- Expressions consist of
  - Operator
    - +,-,\*,/ (division is usually defined to produce an integer result)
  - Individual terms
    - Constants
    - User-defined symbols
    - Special terms, e.g., \*, the current value of LOCCTR
- Examples
 

MAXLEN	EQU	BUFEND-BUFFER
STAB	RESB	(6+3+2)*MAXENTRIES

## Relocation Problem in Expressions

- Values of terms can be
  - Absolute (independent of program location)
    - constants
  - Relative (to the beginning of the program)
    - Address labels
    - \* (value of LOCCTR)
- Expressions can be
  - Absolute
    - Only absolute terms.
    - MAXLEN EQU 1000
  - Relative terms in pairs with opposite signs for each pair.
 

MAXLEN	EQU	BUFEND-BUFFER
--------	-----	---------------
  - Relative

All the relative terms except one can be paired as described in “absolute”.  
The remaining unpaired relative term must have a positive sign.

STAB            EQU            OPTAB + (BUFEND – BUFFER)

### Restriction of Relative Expressions

- No relative terms may enter into a multiplication or division operation
  - 3 \* BUFFER
- Expressions that do not meet the conditions of either “absolute” or “relative” should be flagged as errors.
  - BUFEND + BUFFER
  - 100 – BUFFER

### Handling Relative Symbols in SYMTAB

- To determine the type of an expression, we must keep track of the types of all symbols defined in the program.
- We need a “flag” in the SYMTAB for indication.

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

- Absolute value  
BUFEND - BUFFER
- Illegal  
BUFEND + BUFFER  
100 - BUFFER  
3 \* BUFFER

### Program Blocks

- Allow the generated machine instructions and data to appear in the object program in a different order
- Separating blocks for storing code, data, stack, and larger data block
- Program blocks versus. Control sections
  - Program blocks
    - Segments of code that are rearranged within a single object program unit.
  - Control sections
    - Segments of code that are translated into independent object program units.
- Assembler rearranges 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.

### Assembler directive: USE

- 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

### Example

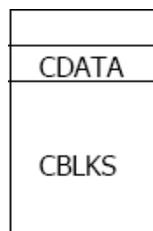
(default) block	Block number				
0000	0	COPY	START	0	
0000	0	FIRST	STL	RETADR	172063
0003	0	CLOOP	JSUB	RDREC	4B2021
0006	0		LDA	LENGTH	032060
0009	0		COMP	#0	290000
000C	0		JEQ	ENDFIL	332006
000F	0		JSUB	WRREC	4B203B
0012	0		J	CLOOP	3F2FEE
0015	0	ENDFIL	LDA	=C'EOF'	032055
0018	0		STA	BUFFER	0F2056
001B	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	2	MAXLEN	EQU	BUFEND-BUFFER	

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

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

### 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.



## Rearrange Codes into Program Blocks

### Pass 1

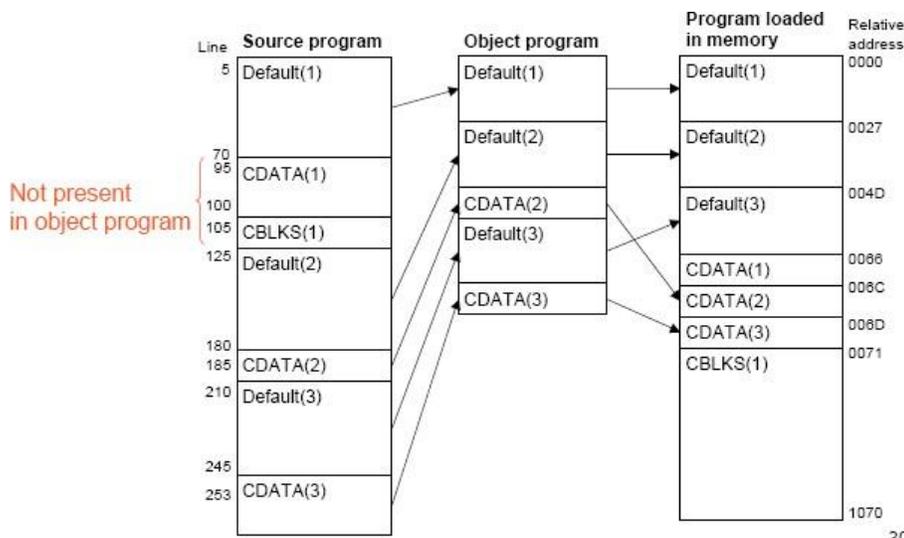
- A separate location counter for each program block
  - Save and restore LOCCTR when switching between blocks
  - At the beginning of a block, LOCCTR is set to 0.
- Assign each label an address relative to the start of the block
- Store the block name or number in the SYMTAB along with the assigned relative address of the label
- Indicate the block length as the latest value of LOCCTR for each block at the end of Pass1
- Assign to each block a starting address in the object program by concatenating the program blocks in a particular order

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

### Pass 2

- Calculate the address for each symbol relative to the start of the object program by adding
  - The location of the symbol relative to the start of its block
  - The starting address of this block

## Program Blocks Loaded in Memory



## Object Program

- It is not necessary to physically rearrange the generated code in the object program
- The assembler just simply inserts the proper load address in each Text record.
- The loader will load these codes into correct place

```
HCOPY 00000001071
T0000001E1720634B20210320602900003320064B203B3F2FEE0320550F2056010003
Default(1)T00001E090F20484B20293E203F
T0000271DB410B400B44075101000E32038332FFADB2032A00433200857A02FB850
Default(2)T000044093B2FEA13201F4F0000
CDATA(4)T00006C01F1
Default(3)T00004D19B410772017E3201B332FFA53A016DF2012B8503B2FEF4F0000
CDATA(5)T00006D04454F4605
E000000
```

## Control Sections and Program Linking

### Control sections

- can be loaded and relocated independently of the other
- are most often used for subroutines or other logical subdivisions of a program
- 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
- assembler directive: CSECT  
secname CSECT
- separate location counter for each control section

### External Definition and Reference

- Instructions in one control section may need to refer to instructions or data located in another section
- External definition
  - EXTDEF name [, name]
  - EXTDEF names symbols that are defined in this control section and may be used by other sections
  - Ex: EXTDEF BUFFER, BUFEND, LENGTH
- External reference
  - EXTREF name [,name]
  - EXTREF names symbols that are used in this control section and are defined elsewhere
  - Ex: EXTREF RDREC, WRREC
- To reference an external symbol, extended format instruction is needed.

Implicitly defined as an external symbol  
first control section

```

COPY      START 0
          EXTDEF BUFFER,BUFEND,LENGTH
          EXTREF RDREC,WRREC
FIRST
CLOOP    STL     RETADR
          +)SUB  RDREC
          LDA     LENGTH
          COMP   #0
          JEQ    ENDFIL
          +)SUB  WRREC
          J      CLOOP
ENDFIL   LDA     =C'EOF'
          STA     BUFFER
          LDA     #3
          STA     LENGTH
          +)SUB  WRREC
          J      @RETADR
RETADR   RESW    1
LENGTH  RESW    1
          LTORG
BUFFER  RESB    4096
BUFEND  EQU     *
MAXLEN  EQU     BUFFEND-BUFFER

```

COPY FILE FROM INPUT TO OUTPUT

SAVE RETURN ADDRESS  
READ INPUT RECORD  
TEST FOR EOF (LENGTH=0)

EXIT IF EOF FOUND  
WRITE OUTPUT RECORD  
LOOP

INSERT END OF FILE MARKER

SET LENGTH = 3

WRITE EOF  
RETURN TO CALLER

LENGTH OF RECORD

4096-BYTE BUFFER AREA

Implicitly defined as an external symbol  
second control section

```

RDREC    CSECT
:
:        SUBROUTINE TO READ RECORD INTO BUFFER
:
          EXTREF BUFFER,LENGTH,BUFFEND
          CLEAR  X
          CLEAR  A
          CLEAR  S
RLOOP    LDT     MAXLEN
          TD     INPUT
          JEQ    RLOOP
          RD     INPUT
          COMPR  A,S
          JEQ    EXIT
          +STCH  BUFFER,X
          TIXR   T
          JLT   RLOOP
EXIT     +STX   LENGTH
          RSUB
INPUT    BYTE   X'F1'
MAXLEN  WORD   BUFFEND-BUFFER

```

CLEAR LOOP COUNTER  
CLEAR A TO ZERO  
CLEAR S TO ZERO

TEST INPUT DEVICE  
LOOP UNTIL READY  
READ CHARACTER INTO REGISTER A  
TEST FOR END OF RECORD (X'00')

EXIT LOOP IF EOR  
STORE CHARACTER IN BUFFER  
LOOP UNLESS MAX LENGTH HAS  
BEEN REACHED

SAVE RECORD LENGTH  
RETURN TO CALLER  
CODE FOR INPUT DEVICE

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
WLOOP    +LDT      LENGTH
:          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

```

## External Reference Handling

### 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 format to provide enough room (that is, relative addressing for external reference is invalid)
- The assembler generates information for each external reference that will allow the loader to perform the required linking.

### Case 2

- 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

### Case 3

- 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

## Records for Object Program

- The assembler must include information in the object program that will cause the loader to insert proper values where they are required.
- 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 (hexiadecimal)
  - 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
- Control section name is automatically an external symbol, i.e. it is available for use in Modification records.

## Object Program

```
COPY
HCOPY .00000001033
DBUFFER000033BUFEND001033LENGTH00002D
RRDREC WRREC
T0000001D1720274B100000320232900003320074B1000003F2FEC0320160F2016
T00001D000100030F200A4B1000003E2000
T00003003454F46
M00000405+RDREC
M00001105+WRREC
M00002405+WRREC
E000000
```

```

RDREC
HRDREC 0000000002B
RBUFFERLENGTHBUFEND
T000001DB410B400B44077201FE3201B332FFADB2015A00433200957900000B850
T00001D0E3B2FE9131000004F0000F1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFEND
M00002806-BUFFER } BUFEND - BUFFER
E
WRREC
HWRREC 0000000001C
RLENGTHBUFFER
T000001CB41077100000E3201232FFA53900000DF2008B8503B2FEE4F000005
M00000305+LENGTH
M00000005+BUFFER
E

```

## Expressions in Multiple Control Sections

- Extended restriction
  - Both terms in each pair of an expression must be within the same control section
  - Legal: BUFEND-BUFFER
  - Illegal: RDREC-COPY
- 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

The assembler design deals with

- Two-pass assembler with overlay structure
- One-pass assemblers
- Multi-pass assemblers

### One-pass assembler

### 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.

### **One-Pass Assemblers**

- Scenario for one-pass assemblers
  - Generate their object code in memory for immediate execution – *load-and-go* assembler
  - External storage for the intermediate file between two passes is slow or is inconvenient to use
- Main problem - Forward references
  - Data items
  - Labels on instructions
- Solution
  - Require that all areas be defined before they are referenced.
  - It is possible, although inconvenient, to do so for data items.
  - Forward jump to instruction items cannot be easily eliminated.
    - Insert (label, *address\_to\_be\_modified*) to SYMTAB
    - Usually, *address\_to\_be\_modified* is stored in a linked-list

### **Sample program for a one-pass assembler**

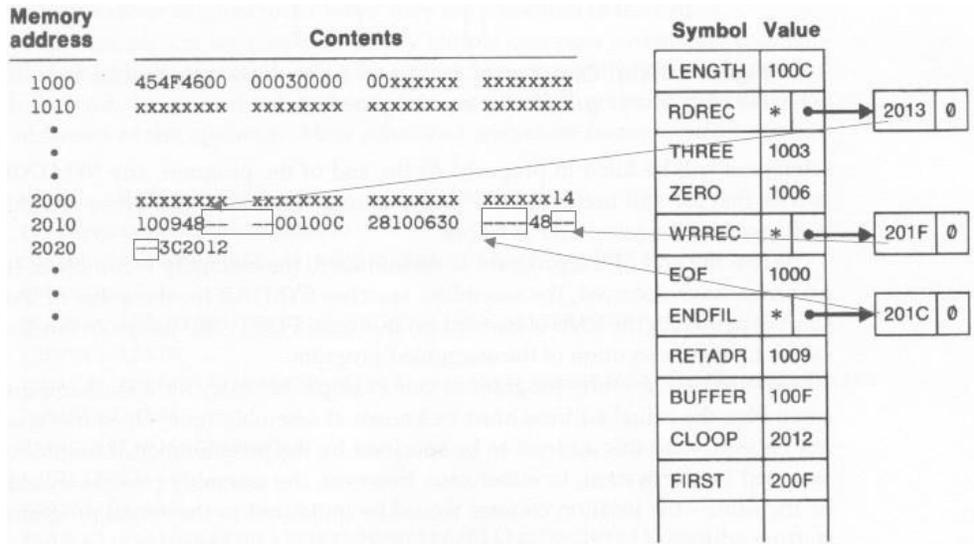
Line	Loc	Source statement			Object code
0	1000	COPY	START	1000	
1	1000	EOF	BYTE	C'EOF'	454F46
2	1003	THREE	WORD	3	000003
3	1006	ZERO	WORD	0	000000
4	1009	RETADR	RESW	1	
5	100C	LENGTH	RESW	1	
6	100F	BUFFER	RESB	4096	
9		.			
10	200F	FIRST	STL	RETADR	141009
15	2012	CLOOP	JSUB	RDREC	48203D
20	2015		LDA	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 Assembler

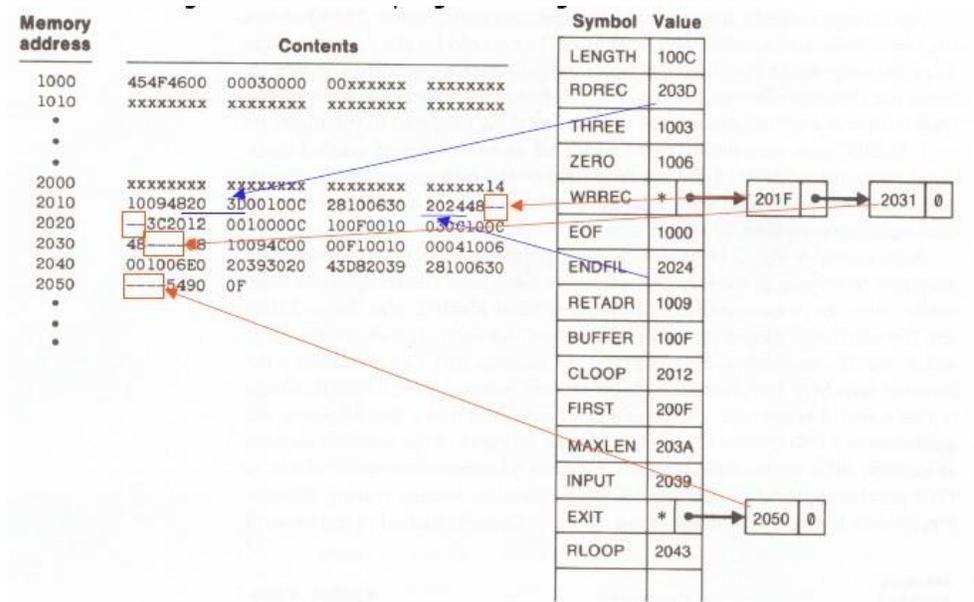
- Omits the operand address if the symbol has not yet been defined.
- Enters this undefined symbol into SYMTAB and indicates that it is undefined.
- Adds the address of this operand address to a list of forward references associated with the SYMTAB entry.
- When the definition for the symbol is encountered, 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.

### Object Code in Memory and SYMTAB

After scanning line 40 of the above program



After scanning line 160 of the above program



If One-Pass Assemblers need to produce object codes

- 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.

### Object code generated by one-pass assembler

```

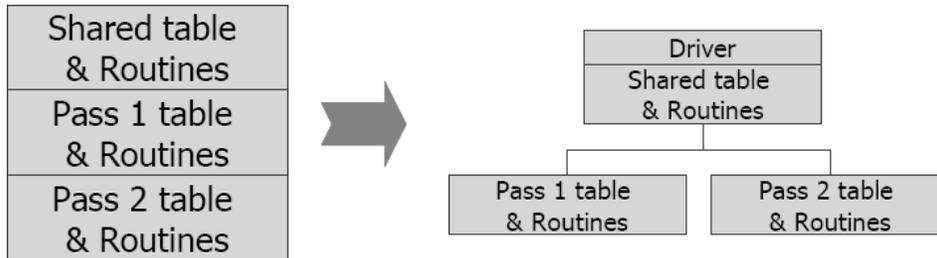
HCOPY  00100000107A
T00100009454F46000003000000
T00200F1514100948000000100C2810063000004800003C2012
T00201C022024
T002024190010000C100F0010030C100C4800000810094C0000F1001000
T00201302203D
T00203D1E041006001006E02039302043D8203928100630000054900F2C203A382043
T00205002205B
T00205B0710100C4C000005
T00201F022062
T002031022062
T00206218041006E0206130206550900FDC20612C100C3820654C0000
E00200F

```

### Two-pass assembler with overlay structure

- Most assemblers divide the processing of the source program into two passes.
- The internal tables and subroutines that are used only during Pass 1 are no longer needed after the first pass is completed.
- The routines and tables for Pass 1 and Pass 2 are never required at the same time.
- There are certain tables (SYMTAB) and certain processing subroutines (searching SYMTAB) that are used by both passes.
- Since Pass 1 and Pass 2 segments are never needed at the same time, they can occupy the same locations in memory during execution of the assembler.
- Initially the Root and Pass 1 segments are loaded into memory.
- The assembler then makes the first pass over the program being assembled.
- At the end of the Pass1, the Pass 2 segment is loaded, replacing the Pass 1 segment.
- The assembler then makes its second pass of the source program and terminates.

- The assembler needs much less memory to run in this way than it would be if both Pass 1 and Pass 2 were loaded at the same time.
- A program that is designed to execute in this way is called an Overlay program because some of its segments overlay others during execution.



## Multi-Pass Assemblers

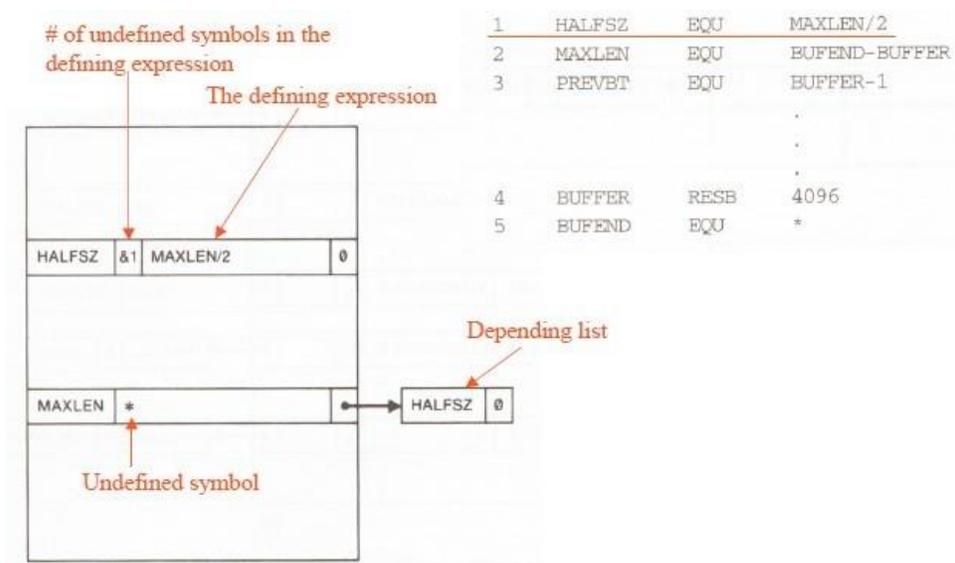
- For a two pass assembler, forward references in symbol definition are not allowed:
 

ALPHA	EQU	BETA
BETA	EQU	DELTA
DELTA	RESW	1
- The symbol BETA cannot be assigned a value when it is encountered during Pass 1 because DELTA has not yet been defined.
- Hence ALPHA cannot be evaluated during Pass 2.
- Symbol definition must be completed in pass 1.
- Prohibiting forward references in symbol definition is not a serious inconvenience.
- Forward references tend to create difficulty for a person reading the program.
- The general solution for forward references is a multi-pass assembler that can make as many passes as are needed to process the definitions of symbols.
- It is not necessary for such an assembler to make more than 2 passes over the entire program.
- The portions of the program that involve forward references in symbol definition are saved during Pass 1.
- Additional passes through these stored definitions are made as the assembly progresses.
- This process is followed by a normal Pass 2.

## Implementation

- 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.

## Example of Multi-pass assembler



Consider the symbol table entries from Pass 1 processing of the statement.

HALFS2 EQU MAXLEN/2

- Since MAXLEN has not yet been defined, no value for HALFS2 can be computed.
- The defining expression for HALFS2 is stored in the symbol table in place of its value.
- The entry &1 indicates that 1 symbol in the defining expression undefined.
- SYMTAB simply contain a pointer to the defining expression.

- The symbol MAXLEN is also entered in the symbol table, with the flag \* identifying it as undefined.
- Associated with this entry is a list of the symbols whose values depend on MAXLEN.

## UNIT III

### LOADERS AND LINKERS

#### INTRODUCTION

- Loader is a system program that performs the loading function.
- Many loaders also support relocation and linking.
- Some systems have a linker (linkage editor) to perform the linking operations and a separate loader to handle relocation and loading.
- One system loader or linker can be used regardless of the original source programming language.
- Loading □ Brings the object program into memory for execution.
- Relocation □ Modifies the object program so that it can be loaded at an address different from the location originally specified.
- Linking □ Combines two or more separate object programs and supplies the information needed to allow references between them.

#### BASIC LOADER FUNCTIONS

Fundamental functions of a loader:

1. Bringing an object program into memory.
2. Starting its execution.

#### Design of an Absolute Loader

For a simple absolute loader, all functions are accomplished in a single pass as follows:

- 1) The Header record of object programs is checked to verify that the correct program has been presented for loading.
- 2) As each Text record is read, the object code it contains is moved to the indicated address in memory.
- 3) When the End record is encountered, the loader jumps to the specified address to begin execution of the loaded program.

An example object program is shown in Fig (a).

```

MCPY 00100000107A
T0010001E1410334820390010362810303010154820613C100300102A0C103900102D
T00101E150C10364820610810334C0000454F46000003000000
T0020391E041030001030E0205D30203FD8205D2810303020575490392C205E38203F
T0020571C101036400000F1001000041030E02079302064509039DC10792C1036
T002073073820644C000005
EO01000
    
```

(a) Object program

Fig (b) shows a representation of the program from Fig (a) after loading.

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 ← COPY
⋮	⋮	⋮	⋮	⋮
2030	XXXXXXXX	XXXXXXXX	xx041030	001030E0
2040	205D3020	3FD8203D	28103030	20575490
2050	392C205E	38203F10	10364C00	00F10010
2060	00041030	E0207930	20645090	39DC2079
2070	2C103638	20644C00	0005XXXX	XXXXXXXX
2080	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
⋮	⋮	⋮	⋮	⋮

(b) Program loaded in memory

## Algorithm for Absolute Loader

```
begin
  read Header record
  verify program name and length
  read first Text record
  while record type ≠ '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
```

- It is very important to realize that in Fig (a), each printed character represents one byte of the object program record.
- In Fig (b), on the other hand, each printed character represents one hexadecimal digit in memory (a half-byte).
- Therefore, to save space and execution time of loaders, most machines store object programs in a **binary form**, with each byte of object code stored as a single byte in the object program.
- In this type of representation a byte may contain any binary value.

## A Simple Bootstrap Loader

When a computer is first turned on or restarted, a special type of absolute loader, called a **bootstrap loader**, is executed. This bootstrap loads the first program to be run by the computer – usually an operating system.

### Working of a simple Bootstrap loader

- The bootstrap begins at address 0 in the memory of the machine.
- It loads the operating system at address 80.
- Each byte of object code to be loaded is represented on device F1 as *two hexadecimal digits* just as it is in a Text record of a SIC object program.

- The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80. The main loop of the bootstrap keeps the address of the next memory location to be loaded in register X.
- After all of the object code from device F1 has been loaded, the bootstrap jumps to address 80, which begins the execution of the program that was loaded.
- Much of the work of the bootstrap loader is performed by the subroutine GETC.
- GETC is used to read and convert a pair of characters from device F1 representing 1 byte of object code to be loaded. For example, two bytes = C "D8"□ '4438'H converting to one byte 'D8'H.
- The resulting byte is stored at the address currently in register X, using STCH instruction that refers to location 0 using indexed addressing.
- The TIXR instruction is then used to add 1 to the value in X.

### Source code for bootstrap loader

```

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
	JBQ	GETC	LOOP UNTIL READY
	RD	INPUT	READ CHARACTER
	COMP	#4	IF CHARACTER IS HEX 04 (END OF FILE),
	JBQ	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	

## MACHINE-DEPENDENT LOADER FEATURES

- The absolute loader has several potential disadvantages. One of the most obvious is the need for the programmer to specify the actual address at which it will be loaded into memory.
- On a simple computer with a small memory the actual address at which the program will be loaded can be specified easily.
- On a larger and more advanced machine, we often like to run several independent programs together, sharing memory between them. We do not know in advance where a program will be loaded. Hence we write relocatable programs instead of absolute ones.
- Writing absolute programs also makes it difficult to use subroutine libraries efficiently. This could not be done effectively if all of the subroutines had pre-assigned absolute addresses.
- The need for *program relocation* is an indirect consequence of the change to larger and more powerful computers. The way relocation is implemented in a loader is also dependent upon machine characteristics.
- Loaders that allow for program relocation are called relocating loaders or relative loaders.

## Relocation

**Two methods for specifying relocation as part of the object program:**

### The first method:

- A Modification is used to describe each part of the object code that must be changed when the program is relocated.

Fig(1) :Consider the program

Line	Loc	Source statement	Object code
5	0000	COPY START 0	
10	0000	FIRST STL RETADR	17202D
12	0003	LDB #LENGTH	69202D
13		BASE LENGTH	
15	0006	CLOOP +JSUB RDREC	4B101036
20	000A	LDA LENGTH	032026
25	000D	COMP #0	290000
30	0010	JEQ ENDFIL	332007
35	0013	+JSUB WRREC	4B10105D
40	0017	J CLOOP	3F2FEC
45	001A	ENDFIL LDA EOF	032010
50	001D	STA BUFFER	0F2016
55	0020	LDA #3	010003
60	0023	STA LENGTH	0F200D
65	0026	-JSUB WRREC	4B10105D
70	002A	J @RETADR	3E2003
80	002D	EOF BYTE C'EOF'	454F46
95	0030	RETADR RESW 1	
100	0033	LENGTH RESW 1	
105	0036	BUFFER RESB 4096	
110		.	
115		SUBROUTINE TO READ RECORD INTO BUFFER	
120		.	
125	1036	RDREC CLEAR X	B410
130	1038	CLEAR A	B400
132	103A	CLEAR S	B440
133	103C	+LDT #4096	75101000
135	1040	RLOOP TD INPUT	E32019
140	1043	JEQ RLOOP	332FFA
145	1046	RD INPUT	DB2013
150	1049	COMPR A, S	A004
155	104B	JEQ EXIT	332008
160	104E	STCH BUFFER, X	57C003
165	1051	TIXR T	B850
170	1053	JLT RLOOP	3B2FEA
175	1056	EXIT STX LENGTH	134000
180	1059	RSUB	4F0000
185	105C	INPUT BYTE X'F1'	F1
195		.	
200		SUBROUTINE TO WRITE RECORD FROM BUFFER	
205		.	
210	105D	WRREC CLEAR X	B410
212	105F	LDT LENGTH	774000
215	1062	WLOOP TD OUTPUT	E32011
220	1065	JEQ WLOOP	332FFA
225	1068	LECH BUFFER, X	53C003
230	106B	WD OUTPUT	DF2008
235	106E	TIXR T	B850
240	1070	JLT WLOOP	3B2FEF
245	1073	RSUB	4F0000
250	1076	OUTPUT BYTE X'05'	05
255		END FIRST	

- Most of the instructions in this program use relative or immediate addressing.
- The only portions of the assembled program that contain actual addresses are the extended format instructions on lines 15, 35, and 65. Thus these are the only items whose values are affected by relocation.

### Object program

```

HCOPY 00000001077
T0000001D17202D69202D4B1010360320262900003320074B10105D3F2FEC032010
T00001D130F20160100030F200D4B10105D3E2003454F46
T0010361DB410B400B44075101000E32019332FFADB2013A00433200857C003B850
T0010531D3B2FEA1340004F0000F1B410774000E32011332FFA53C003DF2008B850
T001070073B2FEFAF000005
M00000705+COPY
M00001405+COPY
M00002705+COPY
E000000

```

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

Fig(2) :Consider a Relocatable program for a Standard SIC machine

Line	Loc	Source statement	Object code
5	0000	COPY START 0	
10	0000	FIRST STL RETADR	140033
15	0003	CLOOP JSUB RDRRC	481039
20	0006	LDA LENGTH	000036
25	0009	COMP ZERO	280030
30	000C	JEQ ENDFIL	300015
35	000F	JSUB WRRRC	481061
40	0012	J CLOOP	3C0003
45	0015	ENDFIL LDA BOP	00002A
50	0018	STA BUFFER	0C0039
55	001B	LDA THREE	00002D
60	001E	STA LENGTH	DC0036
65	0021	JSUB WRRRC	481061

```

200          .          SUBROUTINE TO WRITE RECORD FROM BUFFER
205          *
210      1061      WRREC      LDX      ZERO      040030
215      1064      WLOOP     TD        OUTPUT    E01079
220      1067          JEQ     WLOOP     301064
225      106A          LDCH   BUFFER,X   508039
230      106D          VD     OUTPUT    DC1079
235      1070          TLX   LENGTH    2C0036
240      1073          JLT   LOOP      381064
245      1076          RSUB          4C0000
250      1079      OUTPUT  BYTE      X'05'    05
255          END      FIRST

```

- The Modification record is not well suited for use with all machine architectures. Consider, for example, the program in Fig (2) .This is a relocatable program written for standard version for SIC.
- The important difference between this example and the one in Fig (1) is that the standard SIC machine does not use relative addressing.
- In this program the addresses in all the instructions except RSUB must modified when the program is relocated. This would require 31 Modification records, which results in an object program more than twice as large as the one in Fig (1).

### The second method:

- There are no Modification records.
- The Text records are the same as before except that there is a *relocation bit* associated with each word of object code.
- Since all SIC instructions occupy one word, this means that there is one relocation bit for each possible instruction.

**Fig (3): Object program with relocation by bit mask**

```

HCOPY 0000000107A
T0000001EFFC1400334810390000362800303000154810613C000300002A0C003900002B
T00001E15E000C00364810610800334C0000434F46000003000000
T0010391E7FC040030000030E0105D30103FD8105D2800303010575480392C105E38103F
T0010570A8001000364C0000F1001000
T00106119FE0040030E01079301064508039DC10792C00363810644C000005
E000000

```

- The relocation bits are gathered together into a **bit mask** following the length indicator in each Text record. In Fig (3) this mask is represented (in character form) as three hexadecimal digits.
- If the relocation bit corresponding to a word of object code is set to **1**, the program's starting address is to be added to this word when the program is relocated. A bit value of **0** indicates that no modification is necessary.
- If a Text record contains fewer than 12 words of object code, the bits corresponding to unused words are set to 0.
- For example, the bit mask FFC (representing the bit string 11111111100) in the first Text record specifies that all 10 words of object code are to be modified during relocation.
- **Example:** Note that the LDX instruction on line 210 (Fig (2)) begins a new Text record. If it were placed in the preceding Text record, it would not be properly aligned to correspond to a relocation bit because of the 1-byte data value generated from line 185.

## Program Linking

Consider the three (separately assembled) programs in the figure, each of which consists of a single control section.

### Program 1 (PROGA):

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	FFFFFF6
005A	REF6	WORD ENDC-LISTC+LISTA-1	00003F
005D	REF7	WORD ENDA-LISTA-(ENDB-LISTB)	000014
0060	REF8	WORD LISTB-LISTA	FFFC0
		END REF1	

### Program 2 (PROGB):

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	FFFFFF6
0076	REF6	WORD ENDC-LISTC+LISTA-1	FFFFFFF
0079	REF7	WORD ENDA-LISTA-(ENDB-LISTB)	FFFFFF0
007C	REF8	WORD LISTB-LISTA	000060
		END	

### Program 3 (PROGC):

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	

### Consider first the reference marked REF1.

For the first program (PROGA),

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

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

- The assembler uses an extended-format instruction with address field set to 00000.
- The 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.

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

### Corresponding object programs

#### PROGA:

```

HPROGA 000000000063
ELISTA 000040ENDA 000054
ELISTE ENDE LISTC ENDC
=
T0000200A03201D77100004050014
=
T0000540E000014FFFFFF600003E000014FFFFC0
M00002405+LISTE
M00005406+LISTC
M00005706+ENDC
M00005706-LISTC
M00005406+ENDC
M00005406-LISTC
M00005406+PROGA
M00005006-ENDE
M00005006+LISTE
M00006006+LISTE
M00006006-PROGA
E000020

```

**PROGB:**

```

HPROGB 00000000007F
ELISTE 000060ENDE 000070
ELISTA ENDA LISTC ENDC
=
T0000360E0310000077202705100000
=
T0000700E000000FFFFFF6FFFFFFF0000060
M00003705+LISTA
M00003805+ENDA
M00003805-LISTA
M00007006+ENDA
M00007006-LISTA
M00007006+LISTC
M00007306+ENDC
M00007306-LISTC
M00007606+ENDC
M00007606-LISTC
M00007606+LISTA
M00007306+ENDA
M00007306-LISTA
M00007006+PROGB
M00007006-LISTA
E

```

**PROGC:**

```

HPROGC 000000000051
L1STC 000030ENDC 000042
L1STA ENDA LISTB ENDB
:
:
T0000180C031000007710000405100000
=
T0000420F00003000000000000110000000000000
M00001805+LISTA
M00001805+LISTB
M00002105+ENDA
M00002105-LISTA
M00004206+ENDA
M00004206-LISTA
M00004206+PROGC
M00004806+LISTA
M00004806+ENDA
M00004806-LISTA
M00004806-ENDB
M00004806+LISTB
M00004806+LISTB
M00004806-LISTA
E

```

- The reference marked REF2 is processed in a similar manner.
- REF3 is an immediate operand whose value is to be the difference between ENDA and LISTA (that is, the length of the list in bytes).
- In PROGA, the assembler has all of the information necessary to compute this value. During the assembly of PROGB (and PROGC), the values of the labels are unknown.
- In these programs, the expression must be assembled as an external reference (*with two Modification records*) even though the final result will be an absolute value independent of the locations at which the programs are loaded.
- **Consider REF4.**
- The assembler for PROGA can evaluate all of the expression in REF4 except for the value of LISTC. This results in an initial value of '000014'H and one Modification record.
- The same expression in PROGB contains no terms that can be evaluated by the assembler. The object code therefore contains an initial value of 000000 and *three* Modification records.
- For PROGC, the assembler can supply the value of LISTC relative to the beginning of the program (but not the actual address, which is not known until the program is loaded).
- The initial value of this data word contains the relative address of LISTC ('000030'H). Modification records instruct the loader to add the beginning address of the program (i.e., the value of PROGC), to add the value of ENDA, and to subtract the value of LISTA.

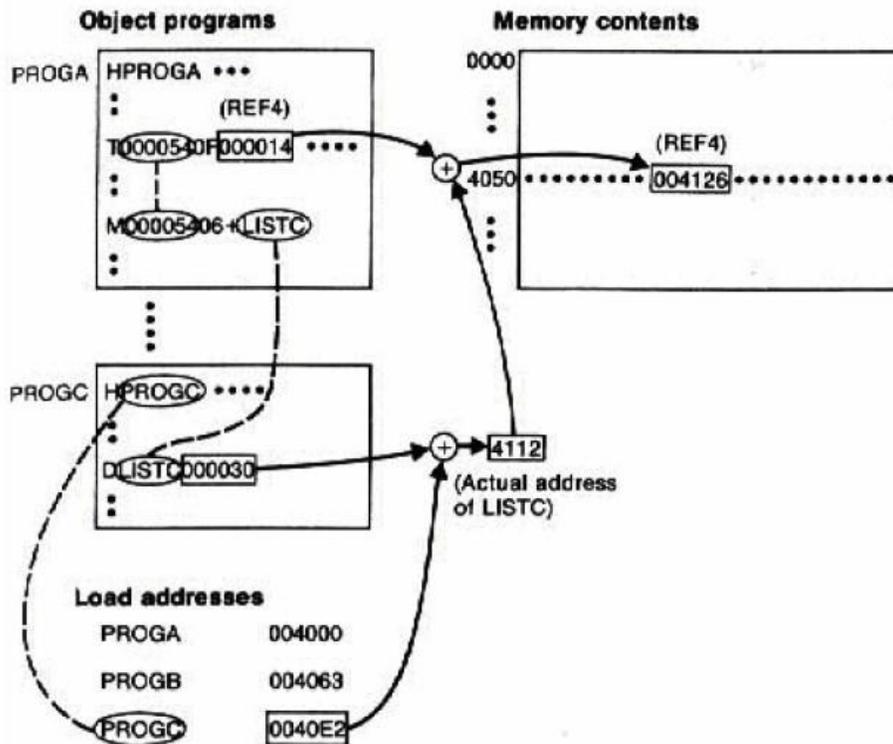
**Fig (4): The three programs as they might appear in memory after loading and linking.**

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 ← PROGC
4100	40C70510	0014.....	.....	.....
4110	.....	.....	.....	.....
4120	.....	00412600	00080040	51000004
4130	000083xx	XXXXXXXX	XXXXXXXX	XXXXXXXX
4140	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
⋮	⋮	⋮	⋮	⋮

PROGA has been loaded starting at address 4000, with PROGB and PROGC immediately following.

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

**Fig (5): Relocation and linking operations performed on REF4 in PROGA**



The initial value (from the Text record) is 000014. To this is added the address assigned to LISTC, which 4112 (the beginning address of PROGC plus 30).

## Algorithm and Data Structures for a Linking Loader

- The algorithm for a *linking loader* is considerably more complicated than the *absolute loader* algorithm.
- A linking loader usually makes *two passes* over its input, just as an assembler does. In terms of general function, the two passes of a linking loader are quite similar to the two passes of an assembler:
- Pass 1 assigns addresses to all external symbols.
- Pass 2 performs the actual loading, relocation, and linking.
- The main data structure needed for our linking loader is an *external symbol table* **ESTAB**.

(1) This table, which is analogous to SYMTAB in our assembler algorithm, is used to store the *name* and *address* of each external symbol in the set of control sections being loaded.

(2) A *hashed organization* is typically used for this table.

- Two other important variables are **PROGADDR** (**program load address**) and **CSADDR** (**control section address**).

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

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

## PASS 1

- During Pass 1, the loader is concerned only with Header and Define record types in the control sections.

### Algorithm for Pass 1 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)
```

1) The beginning load address for the linked program (PROGADDR) is obtained from the OS. This becomes the starting address (CSADDR) for the first control section in the input sequence.

2) The control section name from Header record is entered 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.

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

- At the end of Pass 1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each.
- Many loaders include as an option the ability to print a **load map** that shows these symbols and their addresses.

## PASS 2

- Pass 2 performs the actual *loading*, *relocation*, and *linking* of the program.

### Algorithm for Pass 2 of a Linking loader

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

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

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

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

- The End record for each control section may contain the address of the first instruction in that control section to be executed. Our loader takes this as the transfer point to begin execution. If more than one control section specifies a transfer address, the loader arbitrarily uses the last one encountered.
- If no control section contains a transfer address, the loader uses the beginning of the linked program (i.e., PROGADDR) as the transfer point.
- Normally, a transfer address would be placed in the End record for a main program, but not for a subroutine.

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}
```

This algorithm can be made more efficient. Assign a reference number, which is used (instead of the symbol name) in Modification records, to each external symbol referred to in a control section. Suppose we always assign the reference number 01 to the control section name.

**Fig (6): Object programs using reference numbers for code modification**

HPRGCA 000000000063  
QLISTA 000040ENDB 000054  
R02LISTB 03ENDB 04LISTC 05ENDC

:

T0000200A03201D77100004050014

:

T0000540E00001AFFFFF600003E000014FFFFC0  
M00002405+02  
M00005406+04  
M00005706+05  
M00005706-04  
M00005A06+05  
M00005A06-04  
M00005A06+01  
M00005D06-03  
M00005D06+02  
M00006006+02  
M00006006-01  
E000020

HPRGCB 00000000007F  
QLISTB 000060ENDB 000070  
R02LISTA 03ENDB 04LISTC 05ENDC

:

T0000360E0310000077202705100000

:

T0000700E000000QFFFFF6FFFFFFF0000060  
M00003705+02  
M00003E05+03  
M00003E05-02  
M00007006+03  
M00007006-02  
M00007006+04  
M00007306+03  
M00007306-04  
M00007606+05  
M00007606-04  
M00007606+02  
M00007906+03  
M00007906-02  
M00007C06+01  
M00007C06-02  
E

HPRGCC 000000000051  
QLISTC 000030ENDC 000042  
R02LISTA 03ENDB 04LISTB 05ENDB

:

T0000180C031000007710000405100000

:

T0000420E000030000008000011000000000000  
M00001905+02  
M00001B05+04  
M00002105+03  
M00002105-02  
M00004206+03  
M00004206-02  
M00004206+01  
M00004806+02  
M00004B06+03  
M00004B06-02  
M00004B06-05  
M00004E06+04  
M00004E06+04  
M00004E06-02  
E

## MACHINE-INDEPENDENT LOADER FEATURES

- Loading and linking are often thought of as OS service functions. Therefore, most loaders include fewer different features than are found in a typical assembler.
- They include the use of an automatic library search process for handling external reference and some common options that can be selected at the time of loading and linking.

### Automatic Library Search

- Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded.
- Linking loaders that support *automatic library search* must keep track of external symbols that are referred to, but not defined, in the primary input to the loader.
- At the end of Pass 1, the symbols in ESTAB that remain undefined represent unresolved external references.
- The loader searches the library or libraries specified for routines that contain the definitions of these symbols, and processes the subroutines found by this search exactly as if they had been part of the primary input stream.
- The subroutines fetched from a library in this way may themselves contain external references. It is therefore necessary to repeat the library search process until all references are resolved.
- If unresolved external references remain after the library search is completed, these must be treated as errors.

### Loader Options

- Many loaders allow the user to specify options that modify the standard processing
- **Typical loader option 1:** Allows the selection of alternative sources of input.

**Ex :** INCLUDE program-name (library-name) might direct the loader to read the designated object program from a library and treat it as if it were part of the primary loader input.

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

**Ex :** DELETE csect-name might instruct the loader to delete the named control section(s) from the set of programs being loaded.

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

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

**Ex. :** LIBRARY MYLIB

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

NOCALL STDDEV, PLOT, CORREL

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

## LOADER DESIGN OPTIONS

- Linking loaders perform all linking and relocation at load time.
- There are two alternatives:
  1. **Linkage editors**, which perform linking prior to load time.
  2. **Dynamic linking**, in which the linking function is performed at execution time.
- Precondition: The source program is first assembled or compiled, producing an object program.
- A **linking loader** performs all linking and relocation operations, including automatic library search if specified, and loads the linked program directly into memory for execution.
- A **linkage editor** produces a linked version of the program (load module or executable image), which is written to a file or library for later execution.

## Linkage Editors

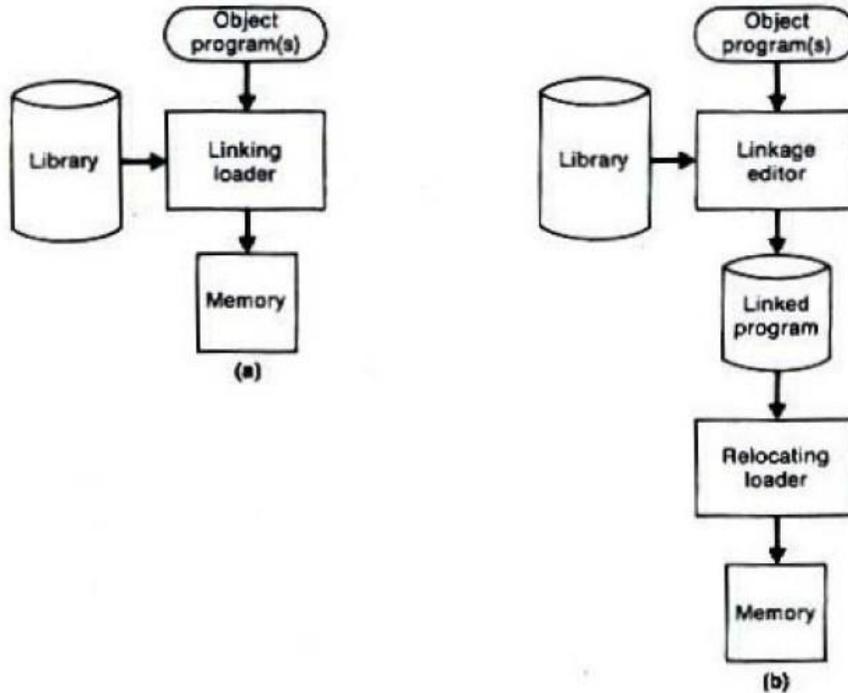
- The linkage editor performs relocation of all control sections relative to the start of the linked program. Thus, all items that need to be modified at load time have values that are relative to the start of the linked program.
- This means that the loading can be accomplished in one pass with no external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

INCLUDE PLANNER (PROGLIB)

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

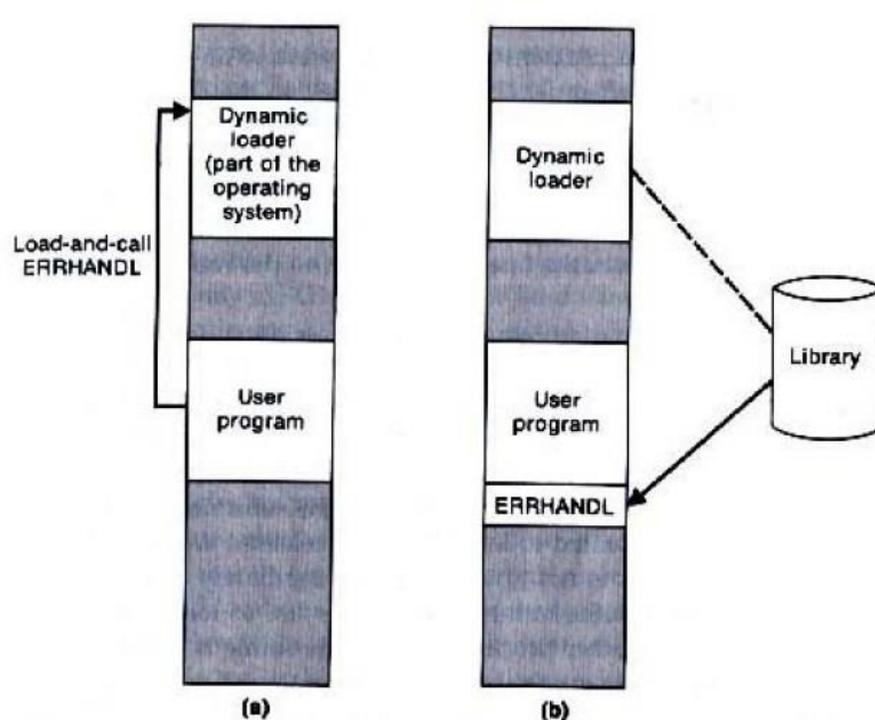
- Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. This can be useful when dealing with subroutine libraries that support high-level programming languages.
- Linkage editors often include a variety of other options and commands like those discussed for linking loaders. Compared to linking loaders, linkage editors in general tend to offer more flexibility and control.

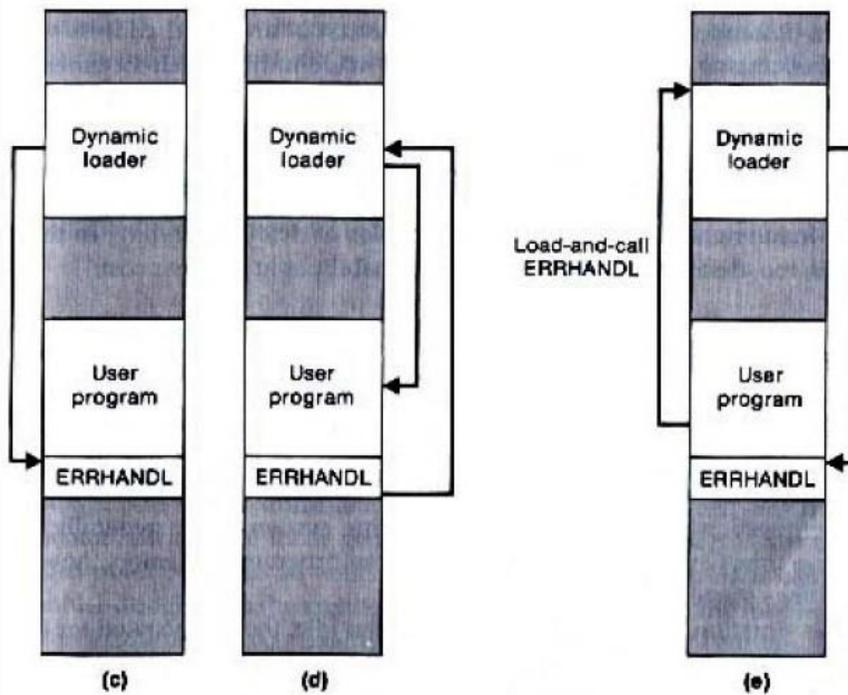
**Fig (7): Processing of an object program using (a) Linking loader and (b) Linkage editor**



## Dynamic Linking

- Linkage editors perform linking operations before the program is loaded for execution.
- Linking loaders perform these same operations at load time.
- Dynamic linking, dynamic loading, or load on call postpones the linking function until execution time: a subroutine is loaded and linked to the rest of the program when it is first called.
- Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library, ex. run-time support routines for a high-level language like C.
- With a program that allows its user to interactively call any of the subroutines of a large mathematical and statistical library, all of the library subroutines could potentially be needed, but only a few will actually be used in any one execution.
- Dynamic linking can avoid the necessity of loading the entire library for each execution except those necessary subroutines.





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

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

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

**Fig (d):** When the called subroutine completes its processing, it returns to its caller (i.e., OS). OS then returns control to the program that issued the request.

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

## Bootstrap Loaders

- With the machine empty and idle there is no need for program relocation.
- We can specify the absolute address for whatever program is first loaded and this will be the OS, which occupies a predefined location in memory.
- We need some means of accomplishing the functions of an absolute loader.
  1. To have the operator enter into memory the object code for an absolute loader, using switches on the computer console.
  2. To have the absolute loader program permanently resident in a ROM.
  3. To have a built –in hardware function that reads a fixed –length record from some device into memory at a fixed location.
- When some hardware signal occurs, the machine begins to execute this ROM program.
- On some computers, the program is executed directly in the ROM: on others, the program is copied from ROM to main memory and executed there.
- The particular device to be used can often be selected via console switches.
- After the read operation is complete, control is automatically transferred to the address in memory where the record was stored, which contains machine instructions that load the absolute program that follow.
- If the loading process requires more instructions that can be read in a single record, this first record causes the reading of others, and these in turn can cause the reading of still more records – boots trap.
- The first record is generally referred to as bootstrap loader:
- Such a loader is added to the beginning of all object programs that are to be loaded into an empty and idle system.
- This includes the OS itself and all stand-alone programs that are to be run without an OS.

## UNIT IV

# MACROPROCESSORS

## INTRODUCTION

### Macro Instructions

- A macro instruction (macro)
  - It is simply a notational convenience for the programmer to write a shorthand version of a program.
  - It represents a commonly used group of statements in the source program.
  - It is replaced by the macro processor with the corresponding group of source language statements. This operation is called “expanding the macro”
- For example:
  - Suppose it is necessary to save the contents of all registers before calling a subroutine.
  - This requires a sequence of instructions.
  - We can define and use a macro, `SAVEREGS`, to represent this sequence of instructions.

### Macro Processor

- A macro processor
  - Its functions essentially involve the substitution of one group of characters or lines for another.
  - Normally, it performs no analysis of the text it handles.
  - It doesn't concern the meaning of the involved statements during macro expansion.
- Therefore, the design of a macro processor generally is machine independent.
- Macro processors are used in
  - assembly language
  - high-level programming languages, e.g., C or C++
  - OS command languages
  - general purpose

### Format of macro definition

A macro can be defined as follows

`MACRO` - `MACRO` pseudo-op shows start of macro definition.

Name [List of Parameters]      –      Macro name with a list of formal parameters.

- .....
- .....
- ..... - Sequence of assembly language instructions.
- MEND - MEND (MACRO-END) Pseudo shows the end of macro definition.

**Example:**

```

MACRO
    SUM X,Y
    LDA X
    MOV BX,X
    LDA Y
    ADD BX
MEND

```

## BASIC MACROPROCESSOR FUNCTIONS

The fundamental functions common to all macro processors are:

1. Macro Definition
2. Macro Invocation
3. Macro Expansion

### Macro Definition and Expansion

- 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 '&'

```

label      op      operands
name  MACRO parameters
      :
      body
      :
      MEND

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

```

5      COPY      START    0          COPY FILE FROM INPUT TO OUTPUT
10     RDBUFF    MACRO    &INDEV ,&BUFADR ,&RECLTH
15     .
20     .          MACRO TO READ RECORD INTO BUFFER
25     .
30     .          CLEAR    X          CLEAR LOOP COUNTER
35     .          CLEAR    A
40     .          CLEAR    S
45     .          +LDT    #4096        SET MAXIMUM RECORD LENGTH
50     .          TD      =X'&INDEV'  TEST INPUT DEVICE
55     .          JEQ     *-3          LOOP UNTIL READY
60     .          RD      =X'&INDEV'  READ CHARACTER INTO REG A
65     .          COMPR   A,S          TEST FOR END OF RECORD
70     .          JEQ     *+11        EXIT LOOP IF EOR
75     .          STCH    &BUFADR,X    STORE CHARACTER IN BUFFER
80     .          TIXR    T           LOOP UNLESS MAXIMUM LENGTH
85     .          JLT     *-19        HAS BEEN REACHED
90     .          STX     &RECLTH     SAVE RECORD LENGTH
95     .          MEND
100    WRBUFF    MACRO    &OUTDEV ,&BUFADR ,&RECLTH
105    .
110    .          MACRO TO WRITE RECORD FROM BUFFER
115    .
120    .          CLEAR    X          CLEAR LOOP COUNTER
125    .          LDT     &RECLTH
130    .          LDCH    &BUFADR,X    GET CHARACTER FROM BUFFER
135    .          TD      =X'&OUTDEV'  TEST OUTPUT DEVICE
140    .          JEQ     *-3          LOOP UNTIL READY
145    .          WD      =X'&OUTDEV'  WRITE CHARACTER
150    .          TIXR    T           LOOP UNTIL ALL CHARACTERS
155    .          JLT     *-14        HAVE BEEN WRITTEN
160    .          MEND
165    .

```

- It shows an example of a SIC/XE program using macro Instructions.
- This program defines and uses two macro instructions, RDBUFF and WRDUFF .
- The functions and logic of RDBUFF macro are similar to those of the RDBUFF subroutine.
- The WRBUFF macro is similar to WRREC subroutine.
- Two Assembler directives (MACRO and MEND) are used in macro definitions.
- The first MACRO statement identifies the beginning of macro definition.
- The Symbol in the label field (RDBUFF) is the name of macro, and entries in the operand field identify the parameters of macro instruction.
- In our macro language, each parameter begins with character &, which facilitates the substitution of parameters during macro expansion.
- The macro name and parameters define the pattern or prototype for the macro instruction used by the programmer. The macro instruction definition has been deleted since they have been no longer needed after macros are expanded.
- Each macro invocation statement has been expanded into the statements that form the body of the macro, with the arguments from macro invocation substituted for the parameters in macro prototype.
- The arguments and parameters are associated with one another according to their positions.

## **Macro Invocation**

- A macro invocation statement (a macro call) gives the name of the macro instruction being invoked and the arguments in expanding the macro.
- The processes of macro invocation and subroutine call are quite different.
  - Statements of the macro body are expanded each time the macro is invoked.
  - Statements of the subroutine appear only one; regardless of how many times the subroutine is called.
- The macro invocation statements treated as comments and the statements generated from macro expansion will be assembled as though they had been written by the programmer.

```

165 .
170 .      MAIN PROGRAM
175 .
180 FIRST  STL      RETADR      SAVE RETURN ADDRESS
190 CLOOP  RDBUFF  F1,BUFFER,LENGTH  READ RECORD INTO BUFFER
195       LDA      LENGTH      TEST FOR END OF FILE
200       COMP    #0
205       JEQ     ENDFIL      EXIT IF EOF FOUND
210       WRBUFF  05,BUFFER,LENGTH  WRITE OUTPUT RECORD
215       J       CLOOP      LOOP
220 ENDFIL  WRBUFF  05,EOF,THREE  INSERT EOF MARKER
225       J       @RETADR
230 EOF    BYTE    C'EOF'
235 THREE  WORD    3
240 RETADR  RESW    1
245 LENGTH  RESW    1          LENGTH OF RECORD
250 BUFFER  RESB   4096       4096-BYTE BUFFER AREA
255       END      FIRST

```

## Macro Expansion

- Each macro invocation statement will be expanded into the statements that form the body of the macro.
- Arguments from the macro invocation are substituted for the parameters in the macro prototype.
  - The arguments and parameters are associated with one another according to their positions.
    - The first argument in the macro invocation corresponds to the first parameter in the macro prototype, etc.
- Comment lines within the macro body have been deleted, but comments on individual statements have been retained.
- Macro invocation statement itself has been included as a comment line.

### Example of a macro expansion

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

- In expanding the macro invocation on line 190, the argument F1 is substituted for the parameter and INDEV wherever it occurs in the body of the macro.
- Similarly BUFFER is substituted for BUFADR and LENGTH is substituted for RECLTH.
- Lines 190a through 190m show the complete expansion of the macro invocation on line 190.
- The label on the macro invocation statement CLOOP has been retained as a label on the first statement generated in the macro expansion.
- This allows the programmer to use a macro instruction in exactly the same way as an assembler language mnemonic.
- After macro processing the expanded file can be used as input to assembler.
- The macro invocation statement will be treated as comments and the statements generated from the macro expansions will be assembled exactly as though they had been written directly by the programmer.

### 4.1.1 Macro Processor Algorithm and Data Structures

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

#### Example 1:

```
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}
```

#### Example 2:

```
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}
```

- Defining MACROS or MACROX does not define RDBUFF and the other macro instructions. These definitions are processed only when an invocation of MACROS or MACROX is expanded.
- A one pass macroprocessor that can alternate between macro definition and macro expansion is able to handle macros like these.
- There are 3 main data structures involved in our macro processor.

### Definition table (DEFTAB)

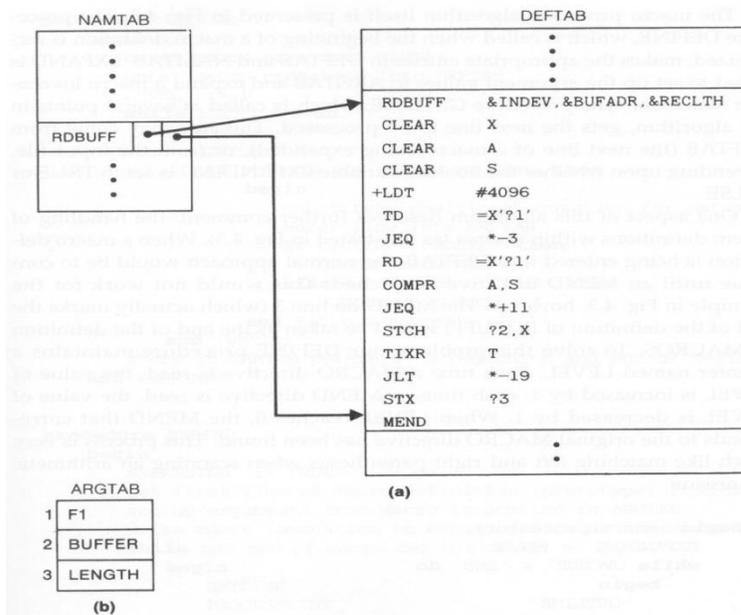
1. The macro definition themselves are stored in definition table (DEFTAB), which contains the macro prototype and statements that make up the macro body.
2. Comment lines from macro definition are not entered into DEFTAB because they will not be a part of macro expansion.

### Name table (NAMTAB)

1. References to macro instruction parameters are converted to a positional entered into NAMTAB, which serves the index to DEFTAB.
2. For each macro instruction defined, NAMTAB contains pointers to beginning and end of definition in DEFTAB.

### Argument table (ARGTAB)

1. The third Data Structure in an argument table (ARGTAB), which is used during expansion of macro invocations.
2. When macro invocation statements are recognized, the arguments are stored in ARGTAB according to their position in argument list.
3. As the macro is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body.



- The position notation is used for the parameters. The parameter &INDEV has been converted to ?1, &BUFADR has been converted to ?2.
- When the ?n notation is recognized in a line from DEFTAB, a simple indexing operation supplies the property argument from ARGTAB.

### Algorithm:

- The 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.
- The procedure GETLINE gets the next line to be processed
- This line may come from DEFTAB or from the input file, depending upon whether the Boolean variable EXPANDING is set to TRUE or FALSE.

```

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}

```

```

procedure EXPAND
  begin
    EXPANDING := TRUE
    get first line of macro definition {prototype} from DEFTAB
    set up arguments from macro invocation in ARG TAB
    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 ARG TAB for positional notation
      end {if}
    else
      read next line from input file
    end {GETLINE}

```

Figure 4.5 (cont'd)

## MACHINE INDEPENDENT MACRO PROCESSOR FEATURES

Machine independent macro processor features are extended features that are not directly related to architecture of computer for which the macro processor is written.

### Concatenation of Macro Parameter

- Most Macro Processor allows parameters to be concatenated with other character strings.
- A program contains a set of series of variables:
  - XA1, XA2, XA3,...

- XB1, XB2, XB3,...
- If similar processing is to be performed on each series of variables, the programmer might want to incorporate this processing into a macro instruction.
- The parameter to such a macro instruction could specify the series of variables to be operated on (A, B, C ...).
- The macro processor constructs the symbols by concatenating X, (A, B, ...), and (1,2,3,...) in the macro expansion.
- Suppose such parameter is named &ID, the macro body may contain a statement: LDA X&ID1, in which &ID is concatenated after the string "X" and before the string "1".
  - ◊ LDA XA1 (&ID=A)
  - ◊ LDA XB1 (&ID=B)
- Ambiguity problem:
  - E.g., X&ID1 may mean
  - "X" + &ID + "1"
  - "X" + &ID1
  - This problem occurs because the end of the parameter is not marked.
- Solution to this ambiguity problem:
  - Use a special concatenation operator "□" to specify the end of the parameter
  - LDA X&ID □1
  - So that the end of parameter &ID is clearly identified.

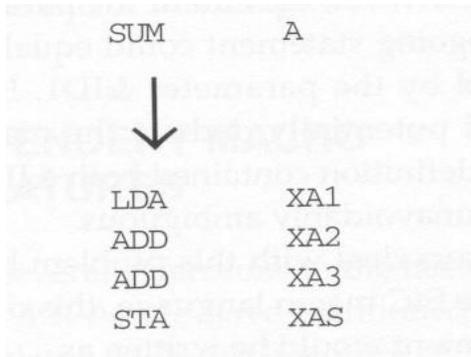
### Macro definition

```

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

```

### Macro invocation statements



- The macroprocessor deletes all occurrences of the concatenation operator immediately after performing parameter substitution, so the character □ will not appear in the macro expansion.

## Generation of Unique Labels

- Labels in the macro body may cause “duplicate labels” problem if the macro is invoked and expanded multiple times.
- Use of relative addressing at the source statement level is very inconvenient, error-prone, and difficult to read.
- It is highly desirable to
  1. Let the programmer use label in the macro body
    - Labels used within the macro body begin with \$.
  2. Let the macro processor generate unique labels for each macro invocation and expansion.
    - During macro expansion, the \$ will be replaced with \$xx, where xx is a two-character alphanumeric counter of the number of macro instructions expanded.
    - XX=AA, AB, AC .....

Consider the definition of WRBUFF

5	COPY	START	0
	:		
	:		
135	TD		=X '&OUTDEV'
	:		
140	JEQ		*-3
	:		
155	JLT		*-14
	:		
255	END		FIRST

- If a label was placed on the TD instruction on line 135, this label would be defined twice, once for each invocation of WRBUFF.
- This duplicate definition would prevent correct assembly of the resulting expanded program.
- The jump instructions on line 140 and 155 are written using the relative operands \*-3 and \*-14, because it is not possible to place a label on line 135 of the macro definition.
- This relative addressing may be acceptable for short jumps such as “JEQ \*-3”
- For longer jumps spanning several instructions, such notation is very inconvenient, error-prone and difficult to read.
- Many macroprocessors avoid these problems by allowing the creation of special types of labels within macro instructions.

### RDBUFF definition

25	RDBUFF	MACRO	&INDEV, &BUFADR, &RECLTH	
30		CLEAR	X	CLEAR LOOP COUNTER
35		CLEAR	A	
40		CLEAR	S	
45		+LDT	#4096	SET MAXIMUM RECORD LENGTH
50	<u>\$LOOP</u>	TD	=X'&INDEV'	TEST INPUT DEVICE
55		JEQ	<u>\$LOOP</u>	LOOP UNTIL READY
60		RD	=X'&INDEV'	READ CHARACTER INTO REG A
65		COMPR	A, S	TEST FOR END OF RECORD
70		JEQ	<u>\$EXIT</u>	EXIT LOOP IF EOR
75		STCH	&BUFADR, X	STORE CHARACTER IN BUFFER
80		TIXR	T	LOOP UNLESS MAXIMUM LENGTH
85		JLT	<u>\$LOOP</u>	HAS BEEN REACHED
90	<u>\$EXIT</u>	STX	&RECLTH	SAVE RECORD LENGTH
95		MEND		

- Labels within the macro body begin with the special character \$.

### Macro expansion

```

        .      RDBUFF  F1, BUFFER, LENGTH

30          CLEAR  X          CLEAR LOOP COUNTER
35          CLEAR  A
40          CLEAR  S
45          +LDT   #4096      SET MAXIMUM RECORD LENGTH
50  $AALoop  TD     =X'F1'    TEST INPUT DEVICE
55          JEQ    $AALoop    LOOP UNTIL READY
60          RD     =X'F1'    READ CHARACTER INTO REG A
65          COMPR A, S      TEST FOR END OF RECORD
70          JEQ    $AAEXIT    EXIT LOOP IF EOR
75          STCH  BUFFER, X  STORE CHARACTER IN BUFFER
80          TIXR  T          LOOP UNLESS MAXIMUM LENGTH
85          JLT   $AALoop    HAS BEEN REACHED
90  $AAEXIT  STX    LENGTH   SAVE RECORD LENGTH

```

- Unique labels are generated within macro expansion.
- Each symbol beginning with \$ has been modified by replacing \$ with \$AA.
- The character \$ will be replaced by \$xx, where xx is a two-character alphanumeric counter of the number of macro instructions expanded.
- For the first macro expansion in a program, xx will have the value AA. For succeeding macro expansions, xx will be set to AB, AC etc.

## Conditional Macro Expansion

- Arguments in macro invocation can be used to:
  - Substitute the parameters in the macro body without changing the sequence of statements expanded.
  - Modify the sequence of statements for **conditional macro expansion** (or conditional assembly when related to assembler).
    - This capability adds greatly to the power and flexibility of a macro language.

**Consider the example**

```

25  RDBUFF  MACRO  &INDEV, &BUFADR, &RECLTH, &EOR, &MAXLTH
26          IF    (&EOR NE '')
27  &EORCK  SET    1
28          ENDIF
30          CLEAR X          CLEAR LOOP COUNTER
35          CLEAR A
38          IF    (&EORCK EQ 1)
40  LDCH   =X'&EOR'        SET EOR CHARACTER
42  RMO    A, S
43          ENDIF
44          IF    (&MAXLTH EQ '')
45  +LDT   #4096            SET MAX LENGTH = 4096
46          ELSE
47  +LDT   #&MAXLTH        SET MAXIMUM RECORD LENGTH
48          ENDIF
50  $LOOP  TD    =X'&INDEV'  TEST INPUT DEVICE
55          JEQ   $LOOP      LOOP UNTIL READY
60          RD    =X'&INDEV'  READ CHARACTER INTO REG A
63          IF    (&EORCK EQ 1)
65  COMPR  A, S            TEST FOR END OF RECORD
70          JEQ   $EXIT      EXIT LOOP IF EOR
73          ENDIF
75          STCH  &BUFADR, X  STORE CHARACTER IN BUFFER
80          TIXR  T          LOOP UNLESS MAXIMUM LENGTH
85          JLT   $LOOP      HAS BEEN REACHED
90  $EXIT  STX   &RECLTH    SAVE RECORD LENGTH
95          MEND

```

Diagram annotations:

- A box labeled "Macro Time variable" has an arrow pointing to the `&EORCK` symbol on line 27.
- A box labeled "Boolean Expression" has an arrow pointing to the `(&EORCK EQ 1)` expression on line 38.

- Two additional parameters used in the example of conditional macro expansion
  - `&EOR`: specifies a hexadecimal character code that marks the end of a record
  - `&MAXLTH`: specifies the maximum length of a record
- Macro-time variable (SET symbol)
  - can be used to
    - store working values during the macro expansion
    - store the evaluation result of Boolean expression
    - control the macro-time conditional structures
  - begins with “&” and that is not a macro instruction parameter
  - be initialized to a value of 0
  - be set by a macro processor directive, SET
- Macro-time conditional structure
  - IF-ELSE-ENDIF
  - WHILE-ENDW

## **Implementation of Conditional Macro Expansion (IF-ELSE-ENDIF Structure)**

- A symbol table is maintained by the macroprocessor.
  - This table contains the values of all macro-time variables used.
  - Entries in this table are made or modified when SET statements are processed.
  - This table is used to look up the current value of a macro-time variable whenever it is required.
- The testing of the condition and looping are done while the macro is being expanded.
- When an IF statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated. If value is
  - TRUE
    - The macro processor continues to process lines from DEFTAB until it encounters the next ELSE or ENDIF statement.
    - If ELSE is encountered, then skips to ENDIF
  - FALSE
    - The macro processor skips ahead in DEFTAB until it finds the next ELSE or ENDLF statement.

## **Implementation of Conditional Macro Expansion (WHILE-ENDW Structure)**

- When an WHILE statement is encountered during the expansion of a macro, the specified Boolean expression is evaluated. If value is
  - TRUE
    - The macro processor continues to process lines from DEFTAB until it encounters the next ENDW statement.
    - When ENDW is encountered, the macro processor returns to the preceding WHILE, re-evaluates the Boolean expression, and takes action again.
  - FALSE
    - The macro processor skips ahead in DEFTAB until it finds the next ENDW statement and then resumes normal macro expansion.

## **Keyword Macro Parameters**

- **Positional parameters**

- Parameters and arguments are associated according to their positions in the macro prototype and invocation. The programmer must specify the arguments in proper order.
- If an argument is to be omitted, a null argument should be used to maintain the proper order in macro invocation statement.
- For example: Suppose a macro instruction GENER has 10 possible parameters, but in a particular invocation of the macro only the 3<sup>rd</sup> and 9<sup>th</sup> parameters are to be specified.
- The statement is GENER „DIRECT,,,,,,3.
- It is not suitable if a macro has a large number of parameters, and only a few of these are given values in a typical invocation.

- **Keyword 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.
- If the 3<sup>rd</sup> parameter is named &TYPE and 9<sup>th</sup> parameter is named &CHANNEL, the macro invocation would be

GENER TYPE=DIRECT,CHANNEL=3.

- It is easier to read and much less error-prone than the positional method.

### **Consider the example**

- Here each parameter name is followed by equal sign, which identifies a keyword parameter and a default value is specified for some of the parameters.

```

25  RDBUFF  MACRO    &INDEV=F1, &BUFADR=, &RECLTH=, &EOR=04, &MAXLTH=4096
26          IF      (&EOR NE '')
27  &EORCK  SET      1
28          ENDIF
30          CLEAR   X          CLEAR LOOP COUNTER
35          CLEAR   A
38          IF      (&EORCK EQ 1)
40          LDCH   =X'&EOR'    SET EOR CHARACTER
42          RMO    A,S
43          ENDIF
47          +LDT   #&MAXLTH    SET MAXIMUM RECORD LENGTH
50  $LOOP   TD      =X'&INDEV'  TEST INPUT DEVICE
55          JEQ    $LOOP      LOOP UNTIL READY
60          RD     =X'&INDEV'  READ CHARACTER INTO REG A
63          IF      (&EORCK EQ 1)
65          COMPR  A,S        TEST FOR END OF RECORD
70          JEQ    $EXIT      EXIT LOOP IF EOR
73          ENDIF
75          STCH   &BUFADR,X   STORE CHARACTER IN BUFFER
80          TIXR   T          LOOP UNLESS MAXIMUM LENGTH
85          JLT    $LOOP      HAS BEEN REACHED
90  $EXIT   STX     &RECLTH    SAVE RECORD LENGTH
95          MEND

```

```

      RDBUFF  BUFADR=BUFFER, RECLTH=LENGTH


---


30          CLEAR   X          CLEAR LOOP COUNTER
35          CLEAR   A
40          LDCH   =X'04'     SET EOR CHARACTER
42          RMO    A,S
47          +LDT   #4096     SET MAXIMUM RECORD LENGTH
50  $AALoop  TD      =X'F1'   TEST INPUT DEVICE
55          JEQ    $AALoop   LOOP UNTIL READY
60          RD     =X'F1'   READ CHARACTER INTO REG A
65          COMPR  A,S        TEST FOR END OF RECORD
70          JEQ    $AAEXIT   EXIT LOOP IF EOR
75          STCH   BUFFER,X   STORE CHARACTER IN BUFFER
80          TIXR   T          LOOP UNLESS MAXIMUM LENGTH
85          JLT    $AALoop   HAS BEEN REACHED
90  $AAEXIT  STX     LENGTH   SAVE RECORD LENGTH

```

Here the value if &INDEV is specified as F3 and the value of &EOR is specified as null.

## MACROPROCESSOR DESIGN OPTIONS

### Recursive Macro Expansion

```
10  RDBUFF  MACRO  &BUFADR, &RECLTH, &INDEV
15  .
20  .      MACRO TO READ RECORD INTO BUFFER
25  .
30          CLEAR  X          CLEAR LOOP COUNTER
35          CLEAR  A
40          CLEAR  S
45          +LDT   #4096      SET MAXIMUM RECORD LENGTH
50  $LOOP  RDCHAR  &INDEV    READ CHARACTER INTO REG A
65          COMPR  A, S      TEST FOR END OF RECORD
70          JEQ    $EXIT     EXIT LOOP IF EOR
75          STCH   &BUFADR, X STORE CHARACTER IN BUFFER
80          TIXR   T         LOOP UNLESS MAXIMUM LENGTH
85          JLT    $LOOP     HAS BEEN REACHED
90  $EXIT  STX     &RECLTH   SAVE RECORD LENGTH
95          MEND
```

```
5  RDCHAR  MACRO  &IN
10  .
15  .      MACRO TO READ CHARACTER INTO REGISTER A
20  .
25          TD     =X'&IN'   TEST INPUT DEVICE
30          JEQ    *-3       LOOP UNTIL READY
35          RD     =X'&IN'   READ CHARACTER
40          MEND
```

- RDCHAR:
  - read one character from a specified device into register A
  - should be defined beforehand (i.e., before RDBUFF)

## Implementation of Recursive Macro Expansion

- Previous macro processor design cannot handle such kind of recursive macro invocation and expansion, e.g., RDBUFF BUFFER, LENGTH, F1
- Reasons:
  - 1) The procedure EXPAND would be called recursively, thus the invocation arguments in the ARGTAB will be overwritten.
  - 2) The Boolean variable EXPANDING would be set to FALSE when the “inner” macro expansion is finished, that is, the macro process would forget that it had been in the middle of expanding an “outer” macro.
  - 3) A similar problem would occur with PROCESSLINE since this procedure too would be called recursively.
- Solutions:
  - 1) Write the macro processor in a programming language that allows recursive calls, thus local variables will be retained.
  - 2) Use a stack to take care of pushing and popping local variables and return addresses.
- Another problem: can a macro invoke itself recursively?

## One-Pass Macro Processor

- A one-pass macro processor that alternate between macro definition and macro expansion in a recursive way is able to handle recursive macro definition.
- Because of the one-pass structure, the definition of a macro must appear in the source program before any statements that invoke that macro.

## Handling Recursive Macro Definition

- In DEFINE procedure
  - When a macro definition is being entered into DEFTAB, the normal approach is to continue until an MEND directive is reached.
  - This would not work for recursive macro definition because the first MEND encountered in the inner macro will terminate the whole macro definition process.
  - To solve this problem, a counter LEVEL is used to keep track of the level of macro definitions.
    - Increase LEVEL by 1 each time a MACRO directive is read.
    - Decrease LEVEL by 1 each time a MEND directive is read.
    - A MEND can terminate the whole macro definition process only when LEVEL reaches 0.

- This process is very much like matching left and right parentheses when scanning an arithmetic expression.

## **Two-Pass Macro Processor**

- Two-pass macro processor
  - Pass 1:
    - Process macro definition
  - Pass 2:
    - Expand all macro invocation statements
- Problem
  - This kind of macro processor cannot allow recursive macro definition, that is, the body of a macro contains definitions of other macros (because all macros would have to be defined during the first pass before any macro invocations were expanded).

## **Example of Recursive Macro Definition**

- MACROS (for SIC)
  - Contains the definitions of RDBUFF and WRBUFF written in SIC instructions.
- MACROX (for SIC/XE)
  - Contains the definitions of RDBUFF and WRBUFF written in SIC/XE instructions.
- A program that is to be run on SIC system could invoke MACROS whereas a program to be run on SIC/XE can invoke MACROX.
- Defining MACROS or MACROX does not define RDBUFF and WRBUFF. These definitions are processed only when an invocation of MACROS or MACROX is expanded.

```

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}

```

## General-Purpose Macro Processors

### Goal

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

### Advantages

- 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.

### **Disadvantages**

- 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
- Tokens, e.g., identifiers, constants, operators, keywords
- Syntax

## **Macro Processing within Language Translators**

Macro processors can be

### **1) 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.

### **2) 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.

### **3) Integrated macro processor**

## **Line-by-Line Macro Processor**

### **Benefits**

- It avoids 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.
- As an example in FORTRAN  
DO 100 I = 1,20
  - a DO statement:
    - DO: keyword
    - 100: statement number
    - I: variable name
- 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.

### **Drawbacks of Line-by-line or Integrated Macro Processor**

- They must be specially designed and written to work with a particular implementation of an assembler or compiler.
- The cost of macro processor development is added to the costs of the language translator, which results in a more expensive software.
- The assembler or compiler will be considerably larger and more complex.

## **UNIT V**

### **TEXT- EDITORS**

#### **OVERVIEW OF THE EDITING PROCESS.**

An interactive editor is a computer program that allows a user to create and revise a target document. The term document includes objects such as computer programs,

texts, equations, tables, diagrams, line art and photographs-anything that one might find on a printed page. Text editor is one in which the primary elements being edited are character strings of the target text. The document editing process is an interactive user-computer dialogue designed to accomplish 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.

**Traveling** – Selection of the part of the document to be viewed and edited. It involves first **traveling** through the document to locate the area of interest such as “next screenful”, ”bottom”,and “find pattern”. Traveling specifies where the area of interest is;

**Filtering** - The selection of what is to be viewed and manipulated is controlled by filtering. Filtering extracts the relevant subset of the target document at the point of interest such as next screenful of text or next statement.

**Formatting:** Formatting determines how the result of filtering will be seen as a visible representation (the view) on a display screen or other device.

**Editing:** In the actual editing phase, the target document is created or altered with a set of operations such as insert, delete, replace, move or copy.

**Manuscript oriented editors** operate on elements such as single characters, words, lines, sentences and paragraphs; **Program-oriented editors** operates on elements such as identifiers, keywords and statements

## **THE USER-INTERFACE OF AN EDITOR.**

The user of an interactive editor is presented with a conceptual model of the editing system. The model is an abstract framework on which the editor and the world on which the operations are based. The **line editors** simulated the world of the keypunch they allowed operations on numbered sequence of 80-character card image lines.

The **Screen-editors** define a world in which a document is represented as a quarter-plane of text lines, unbounded both down and to the right. The user sees, through a cutout, only a rectangular subset of this plane on a multi line display terminal. The cutout can be moved left or right, and up or down, to display other portions of the document. The user interface is also concerned with the input devices, the output devices, and the interaction language of the system.

**INPUT DEVICES:** The input devices are used to enter elements of text being edited, to enter commands, and to designate editable elements. Input devices are categorized as: 1) Text devices 2) Button devices 3) Locator devices

1) **Text or string devices** are typically typewriter like keyboards on which user presses and release keys, sending unique code for each key. Virtually all computer key boards are of the QWERTY type.

2) **Button or Choice devices** generate an interrupt or set a system flag, usually causing an invocation of an associated application program. Also special function keys are also available on the key board. Alternatively, buttons can be simulated in software by displaying text strings or symbols on the screen. The user chooses a string or symbol instead of pressing a button.

3) **Locator devices:** They are two-dimensional analog-to-digital converters that position a cursor symbol on the screen by observing the user's movement of the device. The most common such devices are the **mouse** and the **tablet**.

**The Data Tablet** is a flat, rectangular, electromagnetically sensitive panel. Either the ballpoint pen like stylus or a puck, a small device similar to a mouse is moved over the surface. The tablet returns to a system program the co-ordinates of the position on the data tablet at which the stylus or puck is currently located. The program can then map these data-tablet coordinates to screen coordinates and move the cursor to the corresponding screen position. Text devices with arrow (Cursor) keys can be used to simulate locator devices. Each of these keys shows an arrow that point up, down, left or right. Pressing an arrow key typically generates an appropriate character sequence; the program interprets this sequence and moves the cursor in the direction of the arrow on the key pressed.

**VOICE-INPUT DEVICES:** which translate spoken words to their textual equivalents, may prove to be the text input devices of the future. Voice recognizers are currently available for command input on some systems.

**OUTPUT DEVICES** The output devices let the user view the elements being edited and the result of the editing operations.

- The first output devices were **teletypewriters** and other character-printing terminals that generated output on paper.
- Next "**glass teletypes**" based on Cathode Ray Tube (CRT) technology which uses CRT screen essentially to simulate the hard-copy teletypewriter.
- Today's **advanced CRT terminals** use hardware assistance for such features as moving the cursor, inserting and deleting characters and lines, and scrolling lines and pages.
- The modern **professional workstations** are based on personal computers with high resolution displays; support multiple proportionally spaced character fonts to produce realistic facsimiles of hard copy documents.

**INTERACTION LANGUAGE:**

The interaction language of the text editor is generally one of several common types.

**The typing oriented or text command-oriented method** It is the oldest of the major editing interfaces. 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 are usually echoed to the output device. Typed specification often requires the user to remember the exact form of all commands, or at least their abbreviations. If the command language is complex, the user must continually refer to a manual or an on-line Help function. The typing required can be time consuming for in-experienced users.

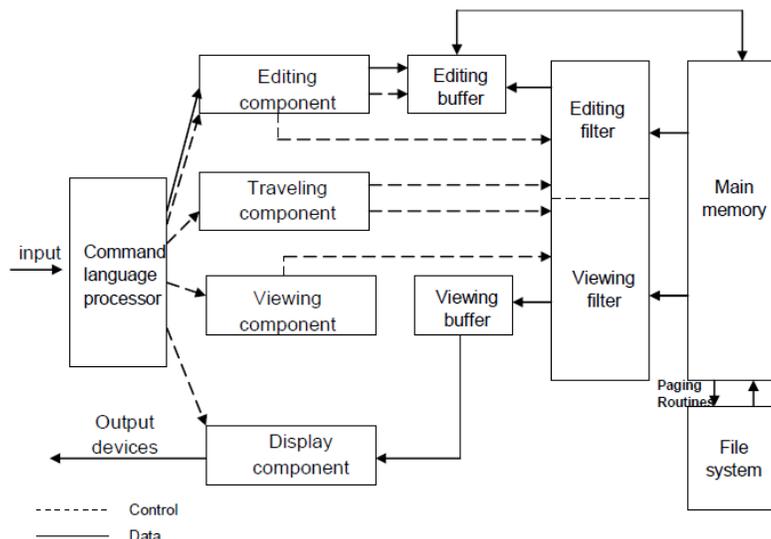
**Function key interfaces:** Each command is associated with marked key on the key board. This eliminates much typing. E.g.: Insert key, Shift key, Control key

**Disadvantages:**

- Have too many unique keys
- Multiple key stroke commands

**Menu oriented interface** A menu is a multiple choice set of text strings or icons which are graphical symbols that represent objects or operations. The user can perform actions by selecting items for the menus. The editor prompts the user with a menu. One problem with menu oriented system can arise when there are many possible actions and several choices are required to complete an action. The display area of the menu is rather limited

**EDITOR STRUCTURE**



Most Text editors have a structure similar to that shown above.

The command Language Processor It accepts input from the user's input devices, and analyzes the tokens and syntactic structure of the commands. It functions much like the lexical and syntactic phases of a compiler. The command language processor may invoke the semantic routines directly. In a text editor, these semantic routines perform functions such as editing and viewing. The semantic routines involve traveling, editing, viewing and display functions. Editing operations are always specified by the user and display operations are specified implicitly by the other three categories of operations. Traveling and viewing operations may be invoked either explicitly by the user or implicitly by the editing operations

#### 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, which is the collection of modules dealing with editing tasks. The current editing pointer can be set or reset explicitly by the user using travelling commands, such as next paragraph and next screen, or implicitly as a side effect of the previous editing operation such as delete paragraph.

#### Traveling Component

The traveling component of the editor actually performs the setting of the current editing and viewing pointers, and thus determines the point at which the viewing and /or editing filtering begins.

#### Viewing Component

The start of the area to be viewed is determined by the current viewing pointer. This pointer is maintained by the viewing component of the editor, which is a collection of modules responsible for determining the next view. The current viewing pointer can be set or reset explicitly by the user or implicitly by system as a result of previous editing operation. The viewing component formulates an ideal view, often expressed in a device independent intermediate representation. This view may be a very simple one consisting of a window's worth of text arranged so that lines are not broken in the middle of the words.

#### Display Component

It takes the idealized view from the viewing component and maps it to a physical output device in the most efficient manner. The display component produces a display by mapping the buffer to a rectangular subset of the screen, usually a window

#### Editing Filter

Filtering consists of the selection of contiguous characters beginning at the current point. The editing filter filters the document to generate a new editing buffer based on the current editing pointer as well as on the editing filter parameters

#### Editing Buffer

It contains the subset of the document filtered by the editing filter based on the editing pointer and editing filter parameters

#### Viewing Filter

When the display needs to be updated, the viewing component invokes the viewing filter. This component filters the document to generate a new viewing buffer based on the current viewing pointer as well as on the viewing filter parameters.

### Viewing Buffer

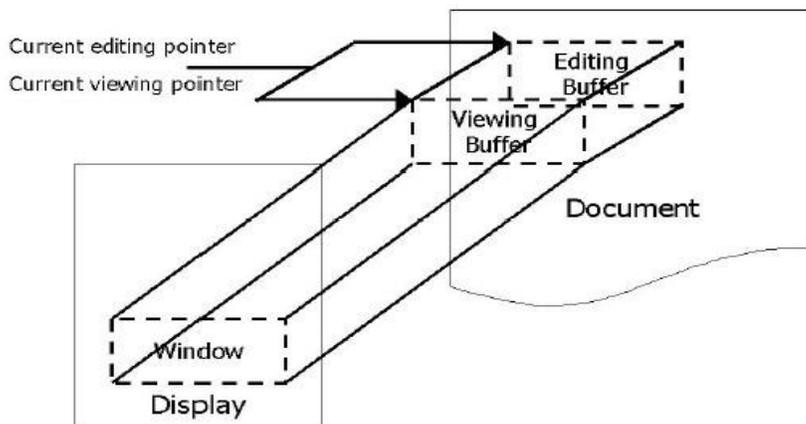
It contains the subset of the document filtered by the viewing filter based on the viewing pointer and viewing filter parameters. E.g. The user of a certain editor might travel to line 75, and after viewing it, decide to change all occurrences of “ugly duckling” to “swan” in lines 1 through 50 of the file by using a change command such as

```
[1,50] c/ugly duckling/swan/
```

As a part of the editing command there is implicit travel to the first line of the file. Lines 1 through 50 are then filtered from the document to become the editing buffer. Successive substitutions take place in this editing buffer without corresponding updates of the view

In *Line editors*, the viewing buffer may contain the current line; in *screen editors*, this buffer may contain rectangular cut out of the quarter-plane of text. This viewing buffer is then passed to the display component of the editor, which produces a display by mapping the buffer to a rectangular subset of the screen, usually called a *window*.

The editing and viewing buffers, while independent, can be related in many ways. In a simplest case, they are identical: the user edits the material directly on the screen. On the other hand, the editing and viewing buffers may be completely disjoint.



Simple relationship between editing and viewing buffers

**Windows** typically cover the entire screen or rectangular portion of it. Mapping viewing buffers to windows that cover only part of the screen is especially useful for editors on modern graphics based workstations. Such systems can support multiple windows, simultaneously showing different portions of the same file or portions of different file.

This approach allows the user to perform inter-file editing operations much more effectively than with a system only a single window.

The mapping of the viewing buffer to a window is accomplished by two components of the system.

(i) First, the viewing component formulates an ideal view often expressed in a device **independent intermediate representation**. This view may be a very simple one consisting of a windows worth of text arranged so that lines are not broken in the middle of words. At the other extreme, the idealized view may be a facsimile of a page of fully formatted and typeset text with equations, tables and figures.

(ii) Second the display component takes these idealized views from the viewing component and maps it to a physical output device the most efficient manner possible.

The components of the editor deal with a user document on two levels:

- (i) **In main memory and**
- (ii) **In the disk file system.**

Loading an entire document into main memory may be infeasible. However if only part of a document is loaded and if many user specified operations require a disk read by the editor to locate the affected portions, editing might be unacceptably slow. In some systems this problem is solved by the mapping the entire file into **virtual memory** and letting the operating system perform efficient demand paging.

An alternative is to provide is the editor paging routines which read one or more logical portions of a document into memory as needed. Such portions are often termed **pages**, although there is usually no relationship between these pages and the hard copy document pages or virtual memory pages. These pages remain resident in main memory until a user operation requires that another portion of the document be loaded.

Editors function in three basic types of computing environment:

- (i) **Time-sharing environment**
- (ii) **Stand-alone environment and**
- (iii) **Distributed environment.**

Each type of environment imposes some constraint on the design of an editor. The Time-Sharing Environment The time sharing editor must function swiftly within the context of the load on the computer's processor, central memory and I/O devices.

The Stand alone Environment The editor on a stand-alone system must have access to the functions that the time sharing editors obtain from its host operating system. This may be provided in pare by a small local operating system or they may be built into the editor itself if the stand alone system is dedicated to editing. Distributed Environment The editor operating in a distributed resource sharing local network must, like a standalone editor, run independently on each user's machine and must, like a time sharing editor, content 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 interactively.

### **DEBUGGING FUNCTIONS AND CAPABILITIES**

**Execution sequencing:** It is the observation and control of the flow of program execution. For example, the program may be halted after a fixed number of instructions are executed.

**Breakpoints** – The programmer may define break points which cause execution to be suspended, when a specified point in the program is reached. After execution is suspended, the debugging command is used to analyze the progress of the program and to diagnose errors detected. Execution of the program can then be removed.

**Conditional Expressions** – Programmers can define some conditional expressions, evaluated during the debugging session, program execution is suspended, when conditions are met, analysis is made, later execution is resumed

**Gaits-** Given a good graphical representation of program progress may even be useful in running the program in various speeds called gaits. A Debugging system should also provide functions such as tracing and traceback. Tracing can be used to track the flow of execution logic and data modifications. The control flow can be traced at different levels of detail – procedure, branch, individual instruction, and so on...

**Traceback** can show the path by which the current statement in the program was reached. It can also show which statements have modified a given variable or parameter. The statements are displayed rather than as hexadecimal displacements. **Program-display Capabilities** It is also important for a debugging system to have good program display capabilities. It must be possible to display the program being debugged, complete with statement numbers. **Multilingual Capability** A debugging system should consider the language in which the program being debugged is written. Most user environments and many applications systems involve the use of different programming languages. A single debugging tool should be available to multilingual situations.

#### **Context Effects**

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

COBOL - MOVE 6.5 TO X  
FORTRAN - X = 6.5

Likewise conditional statements should use the notation of the source language

COBOL - IF A NOT EQUAL TO B  
FORTRAN - IF (A .NE. B)

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

#### **Display of source code**

The language translator may provide the source code or source listing tagged in some standard way so that the debugger has a uniform method of navigating about it.

**Optimization:**

It is also important that a debugging system be able to deal with optimized code. Many optimizations involve the rearrangement of segments of code in the program

For eg. - invariant expressions can be removed from loop - separate loops can be combined into a single loop - redundant expression may be eliminated - elimination of unnecessary branch instructions The debugging of optimized code requires a substantial amount of cooperation from the optimizing compiler.

**Relationship with Other Parts of the System**

An interactive debugger must be related to other parts of the system in many different ways. Availability Interactive debugger must appear to be a part of the run-time environment and an integral part of the system. When an error is discovered, immediate debugging must be possible because it may be difficult or impossible to reproduce the program failure in some other environment or at some other times. Consistency with security and integrity components User need to be able to debug in a production environment. When an application fails during a production run, work dependent on that application stops. Since the production environment is often quite different from the test environment, many program failures cannot be repeated outside the production environment. Debugger must also exist in a way that is consistent with the security and integrity components of the system. Use of debugger must be subjected to the normal authorization mechanism and must leave the usual audit trails. Someone (unauthorized user) must not access any data or code. It must not be possible to use the debuggers to interface with any aspect of system integrity. Coordination with existing and future systems The debugger must coordinate its activities with those of existing and future language compilers and interpreters. It is assumed that debugging facilities in existing language will continue to exist and be maintained. The requirement of cross-language debugger assumes that such a facility would be installed as an alternative to the individual language debuggers.

**USER- INTERFACE CRITERIA**

The interactive debugging system should be user friendly. The facilities of debugging system should be organized into few basic categories of functions which should closely reflect common user tasks.

**Full – screen displays and windowing systems**

- The user interaction should make use of full-screen display and windowing systems. The advantage of such interface is that the information can be should displayed and changed easily and quickly.

**Menus:**

- With menus and full screen editors, the user has far less information to enter and remember
- It should be possible to go directly to the menus without having to retrace an entire hierarchy.
- When a full-screen terminal device is not available, user should have an equivalent action in a linear debugging language by providing commands.

**Command language:**

- The command language should have a clear, logical, simple syntax. Parameters names should be consistent across set of commands

- Parameters should automatically be checked for errors for type and range values.
- Defaults should be provided for parameters.
- Command language should minimize punctuations such as parenthesis, slashes, and special characters.

**On Line HELP facility**

- Good interactive system should have an on-line HELP facility that should provide help for all options of menu
- Help should be available from any state of the debugging system.