175 1056 EXIT STX LENGTH 134000

↓

base relative addressing, length = 0033.

(B) = 0033.

disp = 0033 − (B) = 0033 − 0033 = 0000

↓ if PC relative addressing, disp value is too large to fit in 12-bit disp

20 LDA LENGTH

↓

program-counter relative, but also could have need base relative mode. So arbitrarily chosen to attempt program-counter relative assembly first.

<u>Immediate</u> <u>addressing</u>:- no memory reference is involved

55 0020 LDA #3 010003

↓

operand stored in the instr 003 & bit i set to 1 to indicate immediate addressing.

133 103C +LDT #4096 75101000

↓

operand is too large to fit into the 12-bit displacement field, so the extended instr format is called for.

In following ex. immediate operand is the symbol LENGTH.

12 0003 LDB #LENGTH 69202D

Since the value of the symbol is the address assigned to it, this immediate instr has the effect of loading register B with the address of LENGTH, so combining program-counter relative addressing with immediate addressing.

Scanned by CamScanner

TA = operand value. = 002D because LDA stf on lines bits x, b, P & e are all 0, thus TA = disp = 003).

The assembly & machine that specify indirect addressing presents nothing new. The displacement is computed as usual. Then bit n is set to indicate that the contents stored at this location represent the address of the operand, not the operand itself. Line 70 shows a stf that combines program-counter relative & indirect addressing in this way.

Examples :

1) STL RETADR

(PC) = 0003

RETADR = 0030

6bit opcode	n	i	x	b	P	e	12 bit disp
0001 01	1	1	0	0	1	0	0000 0010 1101
1 7					2		0 2 D.

disp = 0030 − 0003 = 002D

= 17202D.

2) LDB #LENGTH.

(PC) = 0006 , LENGTH = 0033 disp = 0033 − 0006 = 002D

6 bit	n	i	x	b	P	e	12 bit
0011 10	0	1	0	0	1	0	0000 0010 1101
6 9					2		0 2 D

69202D

3) +JSUB RDREC RDREC = 1036

6 bit	n	i	x	b	P	e	20 bit
0100 10	1	1	0	0	0	1	0000 0001 0000 0011 0110
4 B					1		0 1 0 3 6

4B101036.

Scanned by CamScanner

4) LDA #3

6 bit	n	i	x	b	P	e	12 bit
0000 00	0	1	0	0	0	0	0000 0000 0011
0 1			0				0 0 3

= 010003

5) STA LENGTH

	n	i	x	b	P	e	
0000 11	1	1	0	0	1	0	0000 0000 1101
0 F			2				0 0 D

(PC) = 0026, LENGTH = 0033, disp = 0033 − 0026 = 000D

= 0F200D

6) J @RETADR

	n	i	x	b	P	e	
0011 11	1	0	0	0	1	0	0000 0000 0011
3 E			2				0 0 3

RETADR = 0030, (PC) = 002D, disp = RETADR − (PC)
= 0030 − 002D = 0003

= 3E2003

7) CLEAR X

8 bit	8 bit
1011 0100	0001 0000
B 4	1 0

= B410

8) +LDT #4096

6 bit	n	i	x	b	P	e	20 bit
0111 01	0	1	0	0	0	1	0000 1001 0000 0000 0000
7 5			1				0 1 0 0 0

= 75101000

Scanned by CamScanner

9) TD INPUT

		n	i	x	b	p	e			
1110	00	1	1	0	0	1	0	0000	0001	1001
E	3			2				0	1	9

INPUT = 105C, (PC) = 1043, disp = 105C − 1043 = 0019

= E32019

10) WD OUTPUT

		n	i	x	b	p	e			
1101	11	1	1	0	0	1	0	0000	0000	1000
D	F			2				0	0	8

= DF2008

11) STCH BUFFER, X

		n	i	x	b	p	e			
0101	01	1	1	1	1	0	0	0000	0000	0003
5	7			C				0	0	3

BUFFER = 0036, Base = 0033, disp = 0036 − 0033 = 0003

= 57C003

12) STX LENGTH

		n	i	x	b	p	e			
0010	00	1	1	0	1	0	0	0000	0000	0000
1		3		4				0	0	0

LENGTH = 0033, (B) = 0033, disp = 0033 − 0033 = 0000

= 134000.

Scanned by CamScanner

Program Relocation:-

It is desirable to have more than one program at a time sharing the memory & other resources of the machine. In such a situation the actual starting address of the program is not known until load time. Previous program is an example of an absolute program. This program must be loaded at address 1000 in order to execute properly.

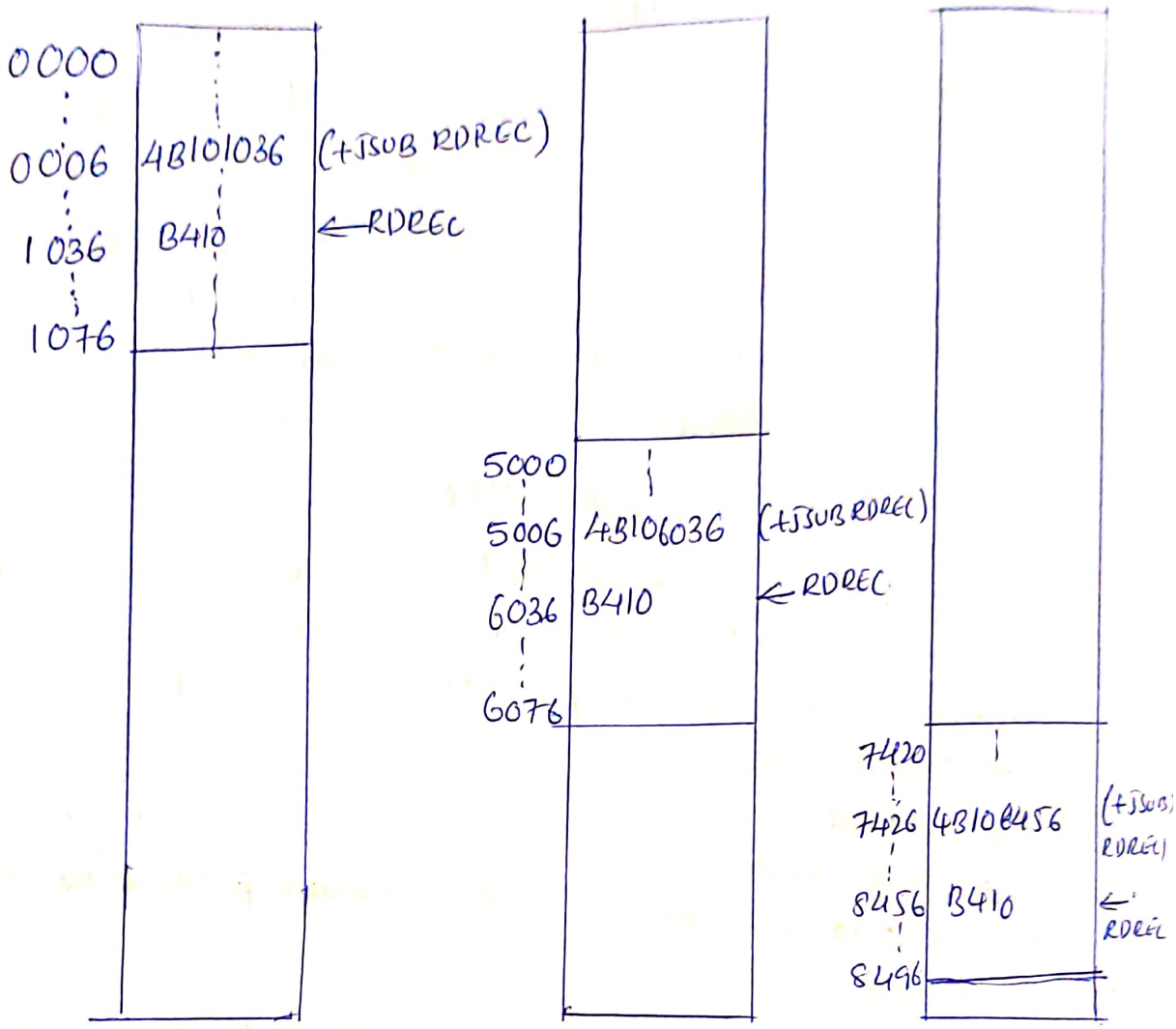
Ex:- 55 101B LDA THREE 00102D

Register A is to be loaded from memory address 102D. Suppose we attempt to load & execute the program at address 2000 instead of address 1000, address 102D will not contain the value. We need to make some change in the address portion of this instn, so we can load & execute our program at address 2000. There are parts of the program that should remain the same regardless of where the program is loaded.

Since the assembler does not know the actual location where the program will be loaded, it cannot make the necessary changes in the addresses need by the program. The assembler can identify for the loader those parts of the object program that need modification. An object program that contains the information necessary to perform this kind of modification is called <u>a relocatable program</u>.

Consider previous program, its started at 1000 & has assembled this using a starting address 0000. The following figure shows this program loaded beginning at address (of the instn labeled RDREC) 0000. The JSUB instn from line 15 is loaded at address 0006. The address field of this instn contains 01036, which is the address of the instn labelled RDREC.

Scanned by CamScanner



0006 CLOOP +JSUB RDREC 4B101036.

suppose that we want to load this program beginning at address 5000, RDREC is at 6036, Thus the JSUB instr must be modified as shown to contain this new address. If we loaded the program beginning at address 7420, the JSUB instr would need to be changed to 4B108456 to correspond to the new address of RDREC.

No matter where the program is loaded, RDREC is always 1036 bytes past the starting address of the program. & this problem is solved by

1. When the assembler generates the object code for the JSUB instr we are considering, it will insert the address of RDREC

relative to the start of the program.

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

3. The command for the loader, must also be a part of the object program.

Modification Record :-

Col 1 M

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

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

The length is stored in half bytes b/cos the address field to be modified may not occupy an integral no of bytes. The address field in the JSUB instr. (considered above occupies 20 bits i.e 5 half bytes). The starting location is the location of the byte containing the leftmost bits of the address field to be modified. If this field occupies an odd no of half bytes, it is assumed to begin in the middle of the first byte at the starting location.

JSUB instr, the modification record would be M00000705.

This record specifies that the beginning address of the program is to be added to a field that begins at address 000007 & is 5 half bytes in length. Thus in the assembled instr 4B101036, the first 12 bits will remain unchanged.

Scanned by CamScanner

The program load address will be added to the last 20 bits to produce the correct operand address. Rest of the instr of the program need not be modified when the program is loaded. If instr operand is not a memory address at all, modification is not reqd. No modification is needed b'cos the operand specified using program-counter relative or base relative addressing.

Ex:. STL RETADR — disp = 2D bytes, so will always be 2D bytes away from the STL instr.

So the TA will then produce the correct operand address corresponding to RETADR.

III^ly the distance b/w the LENGTH & BUFFER will always be 3 bytes. Thus the displacement in the base relative instr on line 160 will be correct without modification.

Only parts of the program that require modification at load time are those that specify direct addresses. For this SIC/XE program, the only such that direct addresses are found in extended format (4-byte) instrs.

Machine-INDEPENDENT Assembler Features:

Literals: Define the constant elsewhere in the program (because the value is stated "literally") & make up a label for it. Such an operand is called a literal. A literal is identified with the prefix =, which is followed by a specification of the literal value, using the same notation as in the BYTE stmt.

Ex: 45 001A ENDFIL LDA =C'EOF' 032010

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

Scanned by CamScanner

215 1062 WLOOP TD =X'05' E32011

specifies a 1-byte literal with the hexadecimal value 05.

Notation used for literals varies from assembler to assembler.

Difference b/w a literal & immediate operand is, the operand value is assembled as part of machine instn in immediate addressing & the assembler generates the specified value as a constant at some other memory location in literal. The address of this generated constant is used as the TA for the machine instn. The effect of using a literal is exactly the same as if the programmer had defined the constant explicitly & used the label assigned to the constant as the instruction operand.

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

It is desirable to place literals into a pool at some other location in the object program. LTORG assembler is used for to do this. When the assembler encounters a LTORG stmt, it creates a literal pool that contains all of the literal operands used since the previous LTORG. This literal pool is placed in the object program at the locn where the LTORG directive was encountered. Literals placed in a pool by LTORG will not be repeated in the pool at the end of the program.

Scanned by CamScanner

Line	Loc	Source statement	Description	Object code
5	0000	COPY START 0	copy file from i/p to o/p	
10	0000	FIRST STL RETADR	save return address	172020
13	0003	LDB #LENGTH	Establish Base Register	692020
14		BASE LENGTH		
15	0006	CLOOP +JSUB RDREC	Read INPUT record	4B101036
20	000A	LDA LENGTH	Test for EOF (LENGTH=0)	032026
25	000D	COMP #0		290000
30	0010	JEQ ENDFIL	Exit if EOF found	332007
35	0013	+JSUB WRREC	write output record	4B10105D
40	0017	J CLOOP	Loop	3F2FEC
45	001A	ENDFIL LDA =C'EOF'	Insert End of File marker	032010
50	001D	STA BUFFER		0F2016
55	0020	LDA #3	Set LENGTH = 3	010003
60	0023	STA LENGTH		0F200D
65	0026	+JSUB WRREC	write EOF	4B10105D
70	002A	J @RETADR	Return to caller	3E2003
93		LTORG		
	002D	* =C'EOF'		454F46
95	0030	RETADR RESW 1		
100	0033	LENGTH RESW 1	length of record	
105	0036	BUFFER RESB 4096	4096-byte buffer area	
107	1000	MAXLEN EQU BUFEND-BUFFER		
106	1036	BUFEND EQU *		
110	⋮	Subroutine to Read Record into buffer		
115	⋮			
120	⋮			B410
125	1036	RDREC CLEAR X	clear loop counter	B400
130	1038	CLEAR A	clear A to zero	B440
132	103A	CLEAR S	clear S to zero	75101000
133	103C	+LDT #MAXLEN		E32019
135	1040	RLOOP TD INPUT	Test Input device	332FFA
140	1043	JEQ RLOOP	loop until ready	DB2013
145	1046	RD INPUT	Read character into reg A	A004
150	1049	COMPR A,S	Test for End of record (x'00')	332008
155	104B	JEQ EXIT	Exit loop if for	

Scanned by CamScanner

Line	LOC	Source Statement	Description	object code
160	104E	STCH BUFFER, X	Store character in buffer	57C003
165	1051	TIXR T	Loop unless max length	B850
170	1053	JLT RLOOP	has been reached	3B2FEA
175	1056	EXIT STX LENGTH	Save Record length	134000
180	1059	RSUB	Return to caller	4F0000
185	105C	INPUT BYTE X'F1'	Code for Input device	F1
195	.			
200	.	Subroutine to write	Record from Buffer	
205	.			
210	105D	WRREC CLEAR X	Clear loop counter	B410
212	105F	LDT LENGTH		774000
215	1062	WLOOP TD =X'05'		E32011
220	1065	JEQ WLOOP	Loop until Ready	332FFA
225	1068	LDCH BUFFER,X	Get character from buffer	53C003
230	106B	WD =X'05'	Write character	DF2008
235	106E	TIXR T	Loop until all characters	B850
240	1070	JLT WLOOP	have been written	3B2FEF
245	1073	RSUB	Return to caller	4F0000
255		END FIRST		
258	1076	* =X'05'		05

If we had not used the LTORG stt on line 93 the literal =C'EOF' would be placed in the pool at the end of the program. This literal pool would begin at address 1073. This means that the literal operand would be placed too far away from the instn referencing it to allow program-counter relative addressing. The problem is the large amount of storage reserved for BUFFER. By placing the literal pool before this buffer, we avoid having to use extended format instns when referring to the literals. LTORG is used when it is desirable to keep the literal operand close to the instn that uses it.

Most assemblers recognize duplicate literals i.e, the same literal used in more than one place in the program & store only one copy of the specified data value.

Scanned by CamScanner

Ex:- literal =X '05' is used in programs on lines 215 & 230 only one data area with this value is generated. Both instrs refer to the same address in the literal pool for their operand.

The easier way to recognise duplicate literals is by comparison of the character strings defining them. (String = X'05') Sometimes additional saving is possible if we look at the generated data value instead of the defining expression.

Ex:- literals =C'EOF' & =X'454F46' would specify identical operand values.

If we use the character string defining a literal to recognize duplicates, we must be careful of literals whose value depends upon their location in the program.

We allow literals that refer to the current value of the location counter (*). Such literals are sometimes useful for loading base registers.

Ex:- BASE *
LDB =* as the first lines of a program would load the beginning address of the program into register B. This value would then be available for base relative addressing.

Ex:- If a literal =* appeared on line 13, it would specify an operand value with 0003 & the same literal appeared on line 55, it would specify an operand with value 0020. In such a case, the literal operands have identical names & they have different values & both must appear in the literal pool. The same problem arises

Scanned by CamScanner

if a literal refers to any other item whose value changes b/w one point in the program & another.

The basic data structure needed is a literal table LITTAB. For each literal used this table, contains the literal name, the operand value & length & the address assigned to the operand when it is placed in a literal pool. LITTAB is often organized as a hash table, using the literal name or value as the key.

As each literal operand is recognized during pass 1, the assembler searches LITTAB for the specified literal name. If the literal is already present in the table, no action is needed. If it is not present, the literal is added to LITTAB. When pass 1 encounters a LTORG stmt or the end of the program the assembler makes a scan of the literal table. At this time each literal currently in the table is assigned an address. As these addresses are assigned, the location counter is updated to reflect the no of bytes occupied by each literal.

During pass 2, the operand address for use in generating object code is obtained by searching LITTAB for each literal operand encountered. The data values specified by the literals in each literal pool are inserted at the appropriate places in the object program exactly as if these values had been generated by BYTE or WORD stmts. If a literal value represents an address in the program the assembler must also generate the appropriate Modification record.

Scanned by CamScanner

Symbol-Defining Statements :-

Up to this point the only user-defined symbols were as labels on instns or data areas. The value of such a label is the address assigned to the stt on which it appears. Most assemblers provide an assembler directive that allows the programmer to define symbols & specify their values. The assembler directive generally used is EQU.

Symbol EQU value

This stt defines the given symbol & assign to it the value specified. The value may be given as a constant or as any expression involving constants & previously defined symbols.

Use of EQU is to establish symbolic names that can be used for improved readability in place of numeric values.

Ex:- on line 133 of the program, used the stt.

+LDT #4096

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

MAXLEN EQU 4096

We can write line 133, as

+LDT #MAXLEN

When the assembler encounters the EQU stt, it enters MAXLEN into SYMTAB. During assembly of the LDT instn the assembler searches SYMTAB for the symbol MAXLEN using its value as the operand in the instn. The resulting object code is exactly as same as in the original

Scanned by CamScanner

version of the instn. The source stmt is easier to understand. It is also much easier to find & change the value of MAXLEN if this becomes necessary.

EQU is in defining mnemonic names for registers. Our assembler recognizes std mnemonics for registers A, X, L etc. The assembler expected registers numbers instead of names in an instn like RMO. This would require of the programmer to write RMO 0,1 instead of RMO A,X. In

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

These stmts cause the symbols A, X, I, ... to be entered into SYMTAB with their corresponding values 0, 1, 2, ---. An instn like RMO A,X would then be allowed. The assembler would search SYMTAB, finding the values 0 & 1 for the symbols A & X & assemble the instn.

On a machine like SIC, there would be little point in doing this, to have the std register mnemonics built into the assembler. A machine that has general-purpose registers. These registers are designated by 0, 1, 2, ---. Some of these may be used as base registers, some as index registers, some as accumulators

```
BASE   EQU   R1
COUNT  EQU   R2
INDEX  EQU   R3
```

} the programmer can establish such names that reflect the logical function of the registers in the program.

There is another common assembler directive that can be used to indirectly assign values to symbols. The directive called ORG ORG value

Scanned by CamScanner

Where value is a constant or an expression involving constants & previously defined symbols. When this stt is encountered during assembly of a program, the assembler resets its location counter (LOCCTR) to the specified value. Since the values of symbols used as labels are taken from LOCCTR, the ORG stt will affect the values of all labels defined until the next ORG.

The location counter is used to control assignment of storage in the object program; its value would result in an incorrect assembly. ORG, can be useful in label defn

	SYMBOL	VALUE	FLAGS
STAB (100 entries)			
	⋮	⋮	⋮

The SYMBOL field contains a 6-byte user-defined symbol; VALUE is a one-word representation of the value assigned to the symbol: FLAGS is a 2-byte field that specifies symbol type & other information.

```
STAB    RESB    1100
```

To refer to entries in the table using indexed addressing we want to be able to refer to the fields SYMBOL, VALUE and FLAGS individually, so we must also define three labels.

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

Ex:- `LDA  VALUE,X` – to fetch the VALUE field from the table entry indicated by the contents of register X.

Scanned by CamScanner

using ORG,

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

The first ORG resets the location counter to the value of STAB. The label on the following RESB stmt defines SYMBOL to have the current value in LOCCTR; this is the same address assigned to SYMTAB. LOCCTR is then advanced to the label on the RESW stmt assigns to VALUE the address (STAB+6), & so on. The result is a set of labels with the same values as those defined with the EQU stmt above. Each entry in STAB consists of a 6-byte SYMBOL, followed by a one-word VALUE, followed by a 2-byte FLAGS.

The last ORG stmt is very important. It sets the LOCCTR back to its previous value - the address of the next unassigned byte of memory after the table STAB. Any labels on subsequent stmts which do not represent part of STAB are assigned the proper addresses.

ORG — To return to the normal use of LOCCTR.

The descriptions of the EQU & ORG stmts contain restrictions that are common to all symbol-defining assembler directives.

```
ALPHA  RESW 1   } allowed      BETA  EQU ALPHA  } not
BETA   EQU ALPHA               ALPHA RESW 1     } allowed
```

because BETA cannot be assigned a value when it is encountered during pass 1 of the assembly. Two pass assembler design requires that all symbols be defined during pass 1. To ORG, all symbols used to specify the new location counter value must have previously defined. The sequence

Scanned by CamScanner

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

– could not be processed. The assembler would not know what value to assign to the location counter in response to the first ORG stt. The symbols BYTE1, BYTE2, & BYTE3 could not be assigned addresses during pass 1.

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

cannot be resolved by an ordinary two-pass assembler regardless of how the work is divided b/w the passes.

Expressions :-

Most assemblers allow the use of expressions wherever such a single operand is permitted. Each such expn must be evaluated by the assembler to produce a single operand address or value. Assembler generally allow arithmetic expns formed according to the normal rules using the operators +, −, * & /. Division is usually defined to produce an integer result. Individual terms in the expn may be constants, user-defined symbols or special terms. The most common such symbol term the current value of the location counter (*).

```
106 BUFEND  EQU *
```

→ gives BUFEND a value i.e. the address of the next byte after the buffer area.

The values of terms & expns are either relative or absolute. A constant is absolute term. Labels on instns & data areas & references to the location counter value are relative terms. A symbol whose value is given by EQU may be either an absolute term or a relative term depending upon the expn used to define its value.

Scanned by CamScanner

Expressions are classified as either absolute exprns or relative exprns depending upon the type of value they produce. An expression that contains only absolute terms is absolute exprn. Absolute exprns may also contain relative terms provided the relative terms occur in pairs & the terms in each such pair have opposite signs. All relative terms must be capable of being paired in this way. None of the relative terms may enter into a multiplication or division operation.

A relative exprn is one in which all of the relative terms except one can be paired as; the remaining unpaired relative term must have a +ve sign. No relative term may enter into a multiplication or division oprn. Exprns that do not meet the condns given for either absolute or relative exprns should be flagged by the assembler as errors. The expressions include exactly three exprns. Whole value remains meaningful when the program is relocated. A relative term or exprn represents some value that may be written as (s+r) where s is the starting address of the program & r is the value of the term or expression relative to the starting address. Thus a relative term usually represents some location within the program. When relative terms are paired with opposite signs, the dependency on the program starting address is cancelled out, the result is an absolute value.

For the eqt

107 MAXLEN EQU BUFEND-BUFFER

both BUFEND & BUFFER are relative terms, each representing an address within the program. The expression represents an absolute value. The difference b/w the two addresses, which is the length of the buffer area in bytes.

Scanned by CamScanner

The value calculated for this expr. in the loc column. This value does not represent an address as do most of the other entries in that column. It does show the value i.e. associated with the symbol that appears in the source stmt (MAXLEN).

Expressions such as BUFEND + BUFFER, 100-BUFFER, or 3*BUFFER represents neither absolute values nor locations within the program. The values of these expressions depend upon the program starting address in a way that is unrelating to anything within the program itself.

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

With this information the assembler can easily determine the type of each expr. used as an operand & generate Modification records in the object program for relative values.

Program Blocks:

program blocks to refer to segments of code that are arranged within a single object program unit & control sections to refer to segments that are translated into independent object program units.

In program, three blocks are used. The first program block (unnamed) block contains the executable instr. of the program. The second block (CDATA) contains all data areas that (consist of) are a few words or less in length.

Scanned by CamScanner

Line	Loc	Block	Source statement			Description	object code
5	0000	0	COPY	START	0	Copy file from I/P to O/P	172063
10	0000	0	FIRST	STL	RETADR	Save Return address	4B2021
15	0003	0	CLOOP	JSUB	RDREC	Read Input record	032060
20	0006	0		LDA	LENGTH	Test for EOF (length=0)	290000
25	0009	0		COMP	#0		332006
30	000C	0		JEQ	ENDFIL	Exit if EOF found	4B203B
35	000F	0		JSUB	WRREC	write O/P record	3F2FEE
40	0012	0		J	CLOOP	loop	032055
45	0015	0	ENDFIL	LDA	=C'EOF'	Insert End of File Marker	0F2056
50	001B	0		STA	BUFFER		010003
55	001B	0		LDA	#3	Set LENGTH = 3	0F2048
60	001E	0		STA	LENGTH		4B2029
65	0021	0		JSUB	WRREC	write EOF	3E203F
70	0024	0		J	@RETADR	Return to caller	
92	0000	1		USE	CDATA		
95	0000	1	RETADR	RESW	1		
100	0003	1	LENGTH	RESW	1	length of (Blocks) Record	
103	0000	2		USE	CBLKS		
105	0000	2	BUFFER	RESB	4096	4096-Byte Buffer area	
106	1000	2	BUFEND	EQU	*	Next location after buffer	
107	1000		MAXLEN	EQU	BUFEND-BUFFER	Maximum record length	
110			; subroutine to Read Record into Buffer				
115							
120							
123	0027	0		USE			
125	0027	0	RDREC	CLEAR	X	clear loop counter	B410
130	0029	0		CLEAR	A	clear A to zero	B400
132	002B	0		CLEAR	S	clear S to zero	B440
133	002D	0		+LDT	#MAXLEN		75101000
135	0031	0	RLOOP	TD	INPUT	Test I/P device	E32038
140	0034	0		JEQ	RLOOP	loop until ready	332FFA
145	0037	0		RD	INPUT	Read character into Reg A	DB2032
150	003A	0		COMPR	A,S	Test for End of Record (x'00')	A004
155	003C	0		JEQ	EXIT	Exit loop if for	332008
160	003F	0		STCH	BUFFER,X	store character in buffer	57A021

Scanned by CamScanner

Line	Loc/Block	Source Statement	Description	Object code
165	0042 0	TIXR T	Loop unless max length has been reached	B850
170	0044 0	JLT RLOOP		3B2FEA
175	0047 0	EXIT STX LENGTH	Save Record length	13201F
180	004A 0	RSUB	Return to caller	4F0000
183	0006 1	USE CDATA		
185	0006 1	INPUT BYTE X'F1'	Code for I/P device	F1
195		; subroutine to write Record from buffer		
200		;		
205				
208	004D 0	USE		
210	004D 0	WREC CLEAR X	Clear loop counter	B410
212	004F 0	LDT LENGTH		772017
215	0052 0	WLOOP TD =X'05'	Test O/P device	E3201B
220	0055 0	JEQ WLOOP	loop until Ready	332FFA
225	0058 0	LDCH BUFFER,X	Get character from buffer	53A016
230	005B 0	TIXR T	Write character	B850
235	005E 0	JD =X'05'	Loop until all characters have been written.	DF2012
240	0060 0	JLT WLOOP		3B2FEF
245	0063 0	RSUB	Return to caller	4F0000
252	0007 1	USE CDATA		
253		LTORG		
	0007 1	* =C'EOF		454F46
	000A 1	* = X'05'		05
255		END FIRST		

The third contains all data areas that consist of larger blocks of memory. The assembler directive USE indicates which portions of the source program belong to the various blocks. At the beginning of the program, statements are assumed to be part of the unnamed block; if no USE stmts are included, the entire program belongs to this single block. The USE stmt on line 92 signals the beginning of the block named CDATA. Source stmts are associated with this block until the USE stmt on line 103, which

begins the block named CBLKS. The USE stmt may also indicate a continuation of a previously begun block. Thus the stmt. on line 123 resumes the default block, & the stmt. on line 133 resumes the block named CDATA.

Each program block may actually contain several separate segments of the source program. The assembler will rearrange these segments to gather together the pieces of each block. These blocks will then be assigned addresses in the object program, with the blocks appearing in the same order in which they were first begun in the source program.

During pass 1, a separate location counter for each program block. The location counter for a block is initialized to 0 when the block is first begun. The current value of this location counter is saved when switching to another block, & the saved value is restored when resuming a previous block.

During pass 1, each label in the program is assigned an address i.e. relative to the start of the block that contains it. When labels are entered into the symbol table, the block name or number is stored along with the assigned relative address. At the end of pass 1 the latest value of the location counter for each block indicates the length of that block. The assembler can then assign to each block a starting address in the object program.

Scanned by CamScanner

For code generation during pass 2, the assembler needs the address for each symbol relative to the start of the object program. The assembler simply adds the location of the symbol, relative to the start of its block, to the assigned block starting address.

The column headed Loc/Block shows the relative address assigned to each source line & a block number indicating which program block is involved. This is essentially the same information i.e is stored in SYMTAB for each symbol

At the end of Pass 1, the assembler constructs a table that contains the starting addresses & lengths for all blocks.

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

Consider the instruction.

20 0006 0 LDA LENGTH 032060

SYMTAB shows the value of the operand as relative location 0003 within program block 1. The starting address for CDATA is 0066. Thus the desired target address for this instn is 0003 + 0066 = 0069. The instn. is to be assembled using program-counter relative addressing. When the instn is executed the program counter contains the address of the instn (line 25). The address of this instn is relative location 0009 with the default block.

Scanned by CamScanner

Since the default block starts at location 0000, this address of this instr. is relative location 0009. (within the default block) Required displacement 0069 − 0009 = 60.

Pass 1 of program blocks

```
begin
  block number = 0    LOCCTR[i] = 0 for all i
  read the first block input line
  if OPCODE = 'START' then
  begin
    write line to intermediate file
    read next input line
  end (if START)
  while OPCODE ≠ 'END' do
  if OPCODE = 'USE'
  begin
    if there is no operand name then
      set block name as default
    else block name as OPERAND name
    if there is no entry for block name then
      insert (block name, block number++) in block table
    i = block number for block name
    if this is not a comment line then
    begin
      if there is a symbol in the LABEL field then
      begin
        search SYMTAB for LABEL
        if found then
          set error flag (duplicate symbol)
```


Scanned by CamScanner

```
else
    insert (LABEL, LOCCTR[i]) into SYMTAB
end (if symbol)
search OPTAB for OPCODE
if found then
    add 3 instruction length to LOCCTR[i]
else if OPCODE = 'WORD' then
    add 3 to LOCCTR[i]
else if OPCODE = 'RESW' then
    add 3 * #[OPERAND] to LOCCTR[i]
else if OPCODE = 'RESB' then
    add #[OPERAND] to LOCCTR[i]
else if OPCODE = 'BYTE' then
begin
    find length of constant in bytes
    add length to LOCCTR[i]
end (if byte)
else
    set error flag
    end (if not a comment)
write line to intermediate file
read Next input line
end (while not end)
write last line to intermediate file
Save length[i] as LOCCTR[i] for all i
Address[0] = Starting address
Address[i] = address(i-1) * length(i-1)
    [for i = 1 to max(block number)]
```

Scanned by CamScanner

insert (address[i], length[i]) in block table for all i
end (Pass 1)

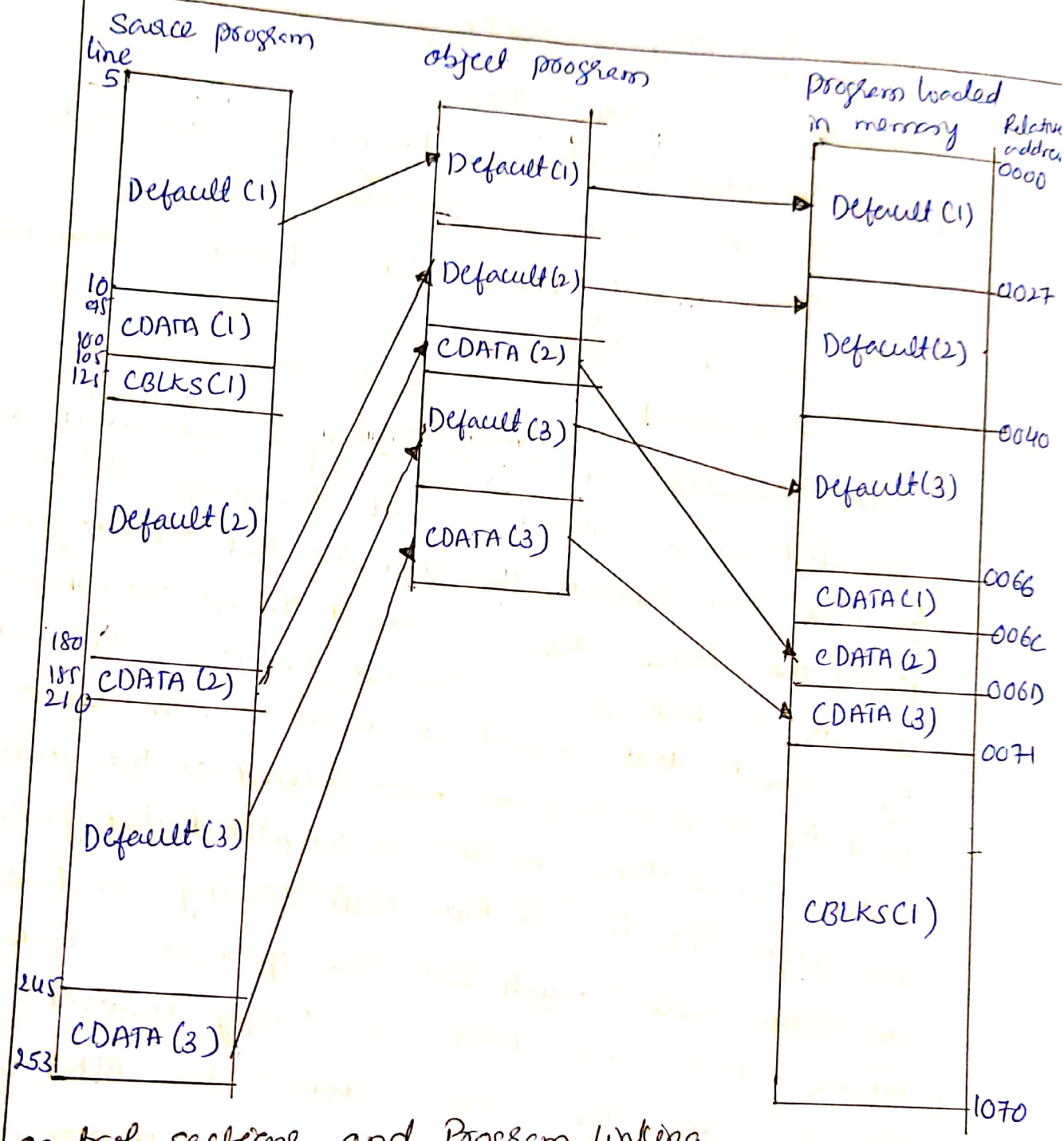
Pass 2 of program blocks

If OPCODE = 'USE' then
Set block number for block name with OPERAND field
Search SYNTAB for OPERAND
Store symbol value + address (block number) as operand address
end (Pass 2)

The assembler can write the object code as it is generated during pass 2 & insert the proper load address in each text record. The first two text records generated from the source program lines 5 through 70. When the USE stt on line 92 is recognized, the assembler writes out the current text record. & so on ... The loader will load the object code from each record at the indicated address. When this loading is completed, the generated code from the default block will occupy relative locations 0000 through 0065; the generated code & reserve storage for CDATA will occupy locations 0066 through 0070; & the storage reserved for CBLKS will occupy locations 0071 through 1070. The following figure traces the blocks of the example program (through) of assembly & loading. Stts are marked CDATA(1) & CBLKS(1) are not actually present in the object program.

Scanned by CamScanner

Program blocks traced through the assembly & loading processes.



Control sections and Program linking

A control section is a part of the program that maintains its identity after assembly each such control section can be loaded & relocated independently of the others. Different control sections are used for subroutines or other logical

logical subdivisions of a program. The programmer can assemble, load & manipulate each of these control sections separately.

Control sections form logically related parts of a program. Ex:- entry in one control section need to refer to instns or data located in another section. Because control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. The assembler has no idea where each other control section will be located at execution time. Such references b/w control sections are called external references. The assembler generates information for each external reference that will allow the loader to perform the reqd linking.

Line	Loc	Source Statement	Description	Object Code
5	0000	COPY START 0	Copy file from I/P to O/P	
6		EXTDEF BUFFER, BUFEND, LENGTH		
7		EXTREF RDREC, WRREC		
10	0000	FIRST STL RETADR	Save Return address	17 2027
15	0003	CLOOP +JSUB RDREC	Read I/P Record	4B100000
20	0007	LDA LENGTH	Test for EOF (length=0)	032023
25	000A	COMP #0		290000
30	000D	JEQ ENDFIL	Exit if EOF found	332007
35	0010	+JSUB WRREC	Write O/P record	4B10D000
40	0014	J CLOOP	Loop	3F2FEC
45	0017	ENDFIL LDA =C'EOF'	Insert End of file marker	032016
50	001A	STA BUFFER	Buffer	0F2016
55	001D	LDA #3	Set length =3	010003
60	0020	STA LENGTH		0F200A
65	0023	+JSUB WRREC	Write EOF	4B100000

Scanned by CamScanner

Line	Loc	Source statement	Description	Object code
70	0027	J@ RETADR	Return to caller	3F2000
95	002A	RETADR RESW 1		
100	002D	LENGTH RESW 1	length of Record	
103		LTORG		
	0030	* =C'EOF'		454F46
105	0033	BUFFER RESB 4096	4096 - Byte buffer area	
106	1033	BUFEND EQU *		
107	1000	MAXLEN EQU BUFEND-BUFFER		
109	0000	RDREC CSECT		
110		.		
115		. subroutine to read record into buffer		
120		.		
122		EXTREF BUFFER, LENGTH, BUFEND		
125	0000	CLEAR X	clear loop counter	B410
130	0002	CLEAR A	clear A to zero	B400
132	0004	CLEAR S	clear S to zero	B440
133	0006	LDT MAXLEN		77201F
135	0009	RLOOP TD INPUT	Test I/P device	E32018
140	000C	JEQ RLOOP	loop until ready	332FFA
145	000F	RD INPUT	Read character into Register A	DB2015
150	0012	COMPR A, S	Test for end of Record	A004
155	0014	JEQ EXIT	Exit loop if for	332009
160	0017	+STCH BUFFER, X	Store character in buffer	57900000
165	001B	TIXR T	loop unless Max length	B850
170	001D	JLT LROOP	has been reached	3B2FE9
175	0020	EXIT +STX LENGTH	save record length	13100000
180	0024	RSUB	Return to caller	4F0000
185	0027	INPUT BYTE X'F1'	code for I/P device.	F1
190	0028	MAXLEN WORD BUFEND-BUFFER		000000

Scanned by CamScanner

Line	LOC	Source of statement	Description	object code
193	0000	WRREC CSECT		
195		:		
200		: subroutine to write record from buffer		
205		:		
207		EXTREF LENGTH, BUFFER		
210	0000	CLEAR X	clear loop counter	B410
212	0002	+LDT LENGTH		77100000
215	0006	WLOOP TD =x'05'		E32012
220	0009	JEQ WLOOP	loop until Ready	332FFA
225	000C	+LDCH BUFFER, X	get character from buffer	53900000
230	0010	WD =x'05'	write character	DF2008
235	0013	TIXR T		B850
240	0015	JLT WLOOP	loop until all characters have been written	3B2FEE
245	0018	RSUB	Return to Caller	4F0000
255		END FIRST		
	001B	* =x'05'		05

In above program, there are 3 control sections. one for main program & one for each subroutine. The START stmt identifies the beginning of the assembly & gives a name (COPY) to the first assembler directive signals the start of a new control section named RDREC. The CSECT stmt begins the control section named WRREC. The assembler establishes a separate location counter for each control section.

Symbols that are defined in one control section may not be used directly by another control section. They must be identified as external references for the loader to handle. Two assembler directives to identify such references: EXTDEF (external definition) EXTREF (external reference)

Scanned by CamScanner

The EXTDEF stmt in a control section names symbols called external symbols that are defined in this control section & may be used by other sections. Control section names do not need to be named in an EXTDEF stmt bcoz they are automatically considered to be external symbols.

The EXTREF stmt names symbols that are used in this control section & are defined elsewhere.

Ex: Symbols BUFFER, BUFFEND and LENGTH – COPY & made available to other sections by the EXTDEF stmt.

WRREC control section uses two of these symbols as specified in the EXTREF stmt. The order in which symbols are listed in the EXTDEF & EXTREF stmt is not significant.

Consider the instn,

15 0003 CLOOP +JSUB RDREC 4B100000

↓

external reference, assembler has no idea where the control section containing RDREC will be loaded, so it cannot assemble the address for this instn. Relative addressing is not possible. Thus an extended format instn. must be used to provide room for the actual address to be inserted.

111hy 160 0017 +STCH BUFFER, X 57900000 makes an external reference to BUFFER. The instn. is assembled using extended format with an address of zero. The x bit is set to 1 to indicate indexed addressing as specified by the instn.

Scanned by CamScanner

190 0028 MAXLEN WORD BUFEND-BUFFER 000000

Here the value of the data word to be generated is specified by an expression involving two external references BUFEND & BUFFER. The assembler stores this value as zero. When the program is loaded, the loader will add to this data area the address of BUFEND & subtract from it the address of BUFFER.

The value of the expression can be calculated immediately by the assembler (line 107). Same thing cannot be done for line 190, BUFEND & BUFFER are defined in another control section, so their values are unknown at assembly time.

Any attempt to refer to a symbol in another control section must be flagged as an error unless the symbol is identified as an external reference. The assembler must also allow the same symbol to be used in different control sections.

The two new record types are Define & Refer.

A Define Record gives information about external symbols that are defined in this control section, symbols EXTDEF.

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

Define Record

Col. 1 D

Scanned by CamScanner

Col. 2–7 Name of external symbol defined in this control section.

Col. 8–13 Relative address of symbol within this control section.

Col. 14–73 Repeat information in Col. 2–13 for other external symbols.

Refer Record:

Col. 1 R

Col. 2–7 Name of external symbol referred to in this control section

Col. 8–73 Names of other external reference symbols

The other inform. needed for program linking is added to the Modification record type.

Modification record:

Col. 1 M

Col. 2–7 Starting address of the field to be modified, relative to the beginning of the control section

Col. 8–9 length of the field to be modified, in half-bytes.

Col. 10 Modification flag (+ or −)

Col. 11–16 External symbol whose value is to be added to or subtracted from the indication field

H^COPY ^000000^001033

D^BUFFER^000033^BUFEND^001033^LENGTH^000002D

R^RDREC ^WRREC

T^000000^1D^172027^4B100000^032023^290000^332007^4B100000^
^3F2FEC^032016^0F2016

Scanned by CamScanner

T^000000^0D^010003^0F200A^4B100000^3E2000

T^000030^03^454F46

M^000004^05^+RDREC

M^000007^05^+WRREC

M^000014^05^+WRREC

E^000000

H^RDREC^000000^000028

R^BUFFER^LENGTH^BUFEND

T^ - - - - - - - - -

T^ - - - - - - - - -

M^000018^05^+BUFFER

M^000021^05^+LENGTH

M^000028^06^+BUFEND

M^000028^06^-BUFFER

E

H^WRREC^000000^0000IC

R^LENGTH^BUFFER

T^ - - - - - - - - -

M^000003^05^+LENGTH

M^00000D^05^+BUFFER

E

The Define and Refer records for each control section include the symbols named in the EXTDEF & EXTREF stmts. For the case of Define, the record also indicates the relative addresses of each external symbol within the control section. For EXTREF stmts, no address information is available. These symbols are simply named in the Refer Record.

The address field for the JSUB instn on line 15 begins at relative address 0004. Its initial value in the object program is zero. The Modification record

Scanned by CamScanner

M00000405+RDREC in control section COPY specifies that the address of RDREC is to be added to this field, thus producing the correct machine instn for execution. The other two modification records in COPY perform similar fns for the instns on lines 35465. The first modification record in control section RDREC fills in the proper address for the external reference on line 160.

The handling of the data word generated by line 190 is only slightly different. The value of this word is to be BUFEND-BUFFER, where both BUFEND & BUFFER are defined in another control section. The assembler generates an initial value of zero for this word. The last two Modification records in RDREC direct that the address of BUFEND be added to this field, & the address of BUFFER be subtracted from it. This computation, performed at load time, results in the desired value for the data word.

In case of relocation, the modification reqd is adding the beginning address of the control section to certain fields in the object program. The symbol used as the name of the control section has its value the reqd address. Since the control section name is automatically an external symbol, it is available for use in modification records.

M00000705	to	M00000705+COPY
M00001405		M00001405+COPY
M00002705		M00002705+COPY

If two terms represent relative locations in the same control section, their difference is an absolute value

Scanned by CamScanner

If they are in different control sections, their difference has a value i.e unpredictable.

The expn, BUFEND-BUFFER has its value the length of BUFFER in bytes. The value of the expn, RDREC-COPY is the difference in the load addresses of the two control sections. This value depends on the way run-time storage is allocated.

Assembler design options:

Main problem is to assemble a program in one pass involves forward references, because some operands are symbols that have not yet been defined in the source program. Thus the assembler does not know what address to insert in the translated instn. Eliminating forward references to data items requires that all such areas be defined in the source program, before they are referenced. ∴ the assembler must make some special provision for handling forward references.

There are two main types of one-pass assembler. One type produces object code directly in memory for immediate execution; the other type produces the usual kind of object program, for later execution.

One-pass assemblers that generate their object code in memory for immediate execution. No object program is written out & no loader is needed. This kind of load-and-go assembler is useful in a system i.e is oriented toward program development & testing. A university computing system for student use is a example of an environment. The assembler generates object code instns as it scans the

Scanned by CamScanner

source program. If an mnem operand is a symbol that has not yet been defined, the operand address is omitted when the mnem is assembled. The symbol used as an operand is entered into the symbol table to indicate that the symbol is undefined. The address of the operand field of the mnem that refers to the undefined symbol is added to a list of forward references associated with the symbol table entry. When the definition for a symbol is encountered, the forward references associated with the symbol table entry. When the defn for a symbol is encountered, the forward reference list for that symbol is scanned & the proper address is inserted into any mnems previously generated.

Since the operand (RDREC) was not yet defined, the the mnem was assembled with no value assigned as the operand address. RDREC was then entered into SYMTAB as an undefined symbol; the address of the operand field of the mnem was inserted in a list associated with RDREC.

Memory address	Contents	
1000	454F4600	00030000 00XXXX XXXXXX
1010	XXXXXXXX	XXXXXXXX XXXXXXXX---
⋮		
2000	XXXXXXXX	XXXXXXXX XXXXXXXX---
2010	100948--	--00100C 28100630
2020		---48--
⋮		

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

Scanned by CamScanner

At the end of the program, any SYMTAB entries that are still marked with * indicate undefined symbols.

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

One-pass assemblers that produce object programs as O/P are often used on systems where external working-storage devices, are not available. One-pass assemblers that produce object programs follow a different procedure.

Forward references are entered into lists. When the defn of a symbol is encountered, instns that made forward references to that symbol may no longer be available in memory for modification.

<u>Algorithm for one pass assembler</u>

```
begin
  read first input line
  if OPCODE = 'START' then
    begin
      Save #[OPERAND] as starting address
      initialize LOCCTR as starting address
      read next input line
    end (if START)
  else
    initialize LOCCTR to 0
While OPCODE ≠ 'END' do
  begin
    if there is not a comment line then
      begin
```


Scanned by CamScanner

```
if there is a symbol in the LABEL field then
  begin
    search SYMTAB for LABEL
      if found then
      begin
        if symbol value as null
        set symbol value as LOCCTR & search the
        linked list with the corresponding operand
        PTR addresses & generate operand addresses
        addresses as corresponding symbol values
        set symbol value as LOCCTR in symbol
        table & delete the linked list
    end
      else
        insert (LABEL, LOCCTR) into SYMTAB
  end
    search OPTAB for OPCODE
      if found then
        begin
          search SYMTAB for OPERAND address
      if found then
        if symbol value not equal to null then
        store symbol value as OPERAND address
      else
        insert at the end of the linked list
        with a node with address as LOCCTR
      else
        insert (Symbol, null)
        add 3 to LOCCTR
```

Scanned by CamScanner

```
end
  else if OPCODE = 'WORD' then
    add 3 to LOCCTR & convert comment to object code
  else if OPCODE = 'RESW' then
    add 3 #[OPERAND] to LOCCTR
  else if OPCODE = 'RESB' then
    add #[OPERAND] to LOCCTR
  else if OPCODE = "BYTE" then
  begin
      find length of constant in bytes
      add length to LOCCTR
      convert constant to object code
  end
  if object will not fit into current text record then
  begin
      write text record to object program initialize
                                  new text record
  end
  add object code to Text record
end
write listing line
read next input line
end
write last Text record to object program
write End record to object program
write last listing line
end {pass 1}
```


Scanned by CamScanner

Multi-Pass Assemblers :-

EQU assembler directive, required that any symbol used on the right-hand side be defined in the source program. A similar requirement was imposed for ORG. The symbol defn process in a two-pass assembler. The sequence

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

The symbol BETA cannot be assigned a value when it is encountered during the first pass because DELTA has not been defined. ALPHA cannot be evaluated during the second pass. Any assembler that makes only two sequential passes over the source program cannot resolve such a sequence of defns.

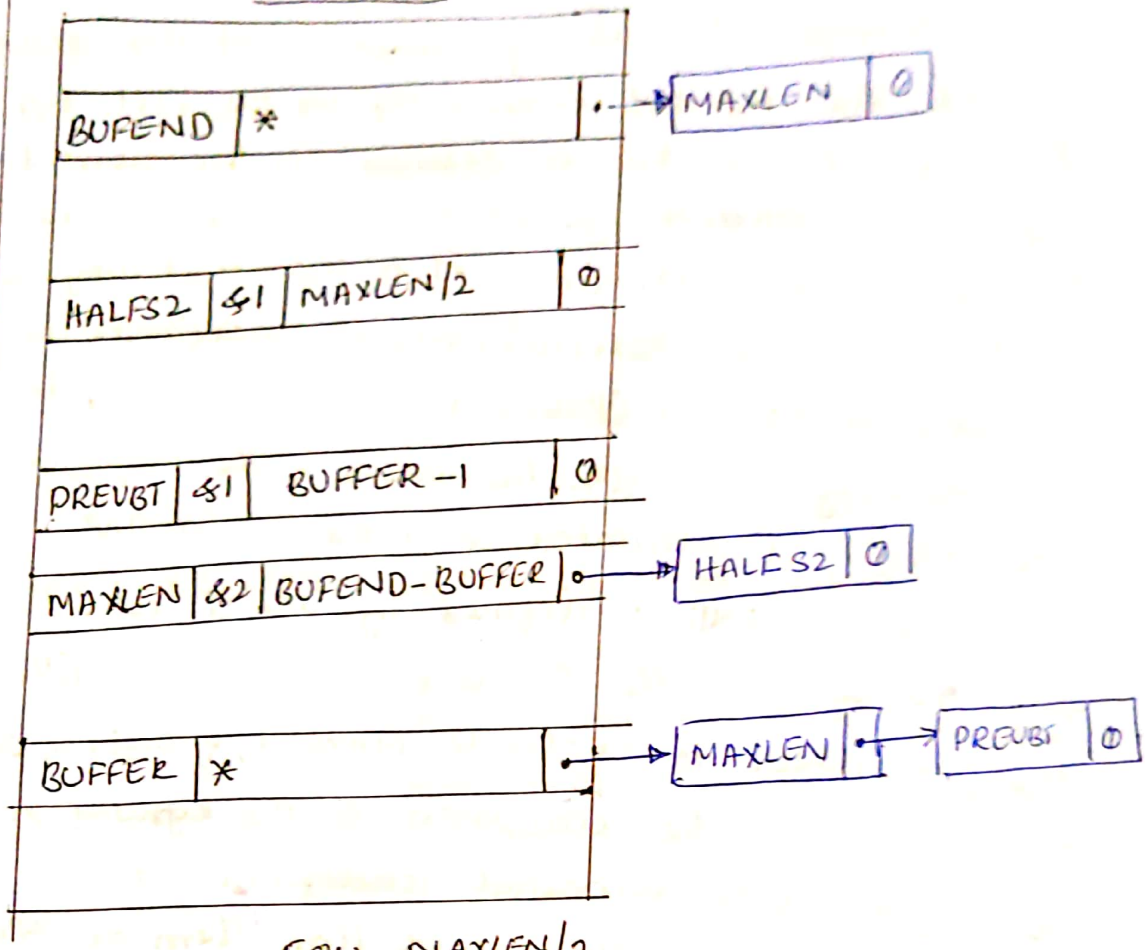
Restrictions such as prohibiting forward references in symbol defn are inconvenience. So multi-pass assembler can make as many passes as are needed to process the defns. of symbol. The portions of the program that involves forward references in symbol defn are saved during pass 1. Additional passes through these stored defns are made as the assembly progresses. This process is followed by pass 2.

Ex:

```
1  HALFS2   EQU   MAXLEN/2
2  MAXLEN   EQU   BUFEND - BUFFER
3  PREVBT   EQU   BUFFER - 1
                  :
                  :
4  BUFFER   RESB  4096
5  BUFEND   EQU   *
```

Scanned by CamScanner

Symbol table entries from pass 1



HALFS2 EQU MAXLEN/2

MAXLEN has not yet been defined, so no value for HALFS2 can be computed. HALFS2 is stored in the symbol in place of its value. Then entry &1 indicates that one symbol in the defining expression is undefined. Associated with this entry is a list of symbols whose depend on MAXLEN

MAXLEN EQU BUFEND-BUFFER

both are undefined symbols involved in the defn. Both of these are entered into SYMTAB with list indicating the dependence of MAXLEN upon them. Similarly, the definition of PREVBT causes this symbol to be added to the list of dependencies on BUFFER. When line 4 is read, the location counter contains the hexadecimal value 1034. This

Scanned by CamScanner

address is stored as the value of BUFFER. The assembler then examines the list of symbols that are dependent on BUFFER. The symbol table entry for the first symbol in this list shows that it depends on two currently undefined symbols; ∴ MAXLEN cannot be evaluated immediately. Instead, the &2 is changed to &1 to show that only one symbol in the definition (BUFEND) remains undefined. The other symbol in the list (PREVBT) can be evaluated because it depends only on buffer. The value of the defining expression for PREVBT is calculated & stored in SYMTAB.

When BUFEND is defined by line 5, its value is entered into the symbol table. The list associated with then directs the assembler to evaluate MAXLEN, & entering a value for MAXLEN causes the evaluation of the symbol in its list. If any symbols remained undefined at the end of the program, the assembler would flag them as errors.

BUFEND	2034	0
HALFS2	500	0
PREVBT	1033	0
MAXLEN	1000	0
BUFFER	1034	0

Scanned by CamScanner

Macroprocessors: Basic macro processor functions:

A macro defn represents a commonly used group of statements in the source programming language. The macro processor replaces each macro defn with the corresponding group of source language stmts. This is called expanding the macros. The functions of a macro processor involve the substitution of one group of characters or lines for another. Use of macro processors is in assembly language programming & also be used with high-level programming languages, operating system command languages. These are general-purpose macro processors that are not tied to any particular language.

Macro program:

Line	Source Statement		
5	COPY	START 0	COPY file from i/p to o/p
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
65		COMPR A,S	Test for end of Record
70		JEQ *+11	Exit loop if for

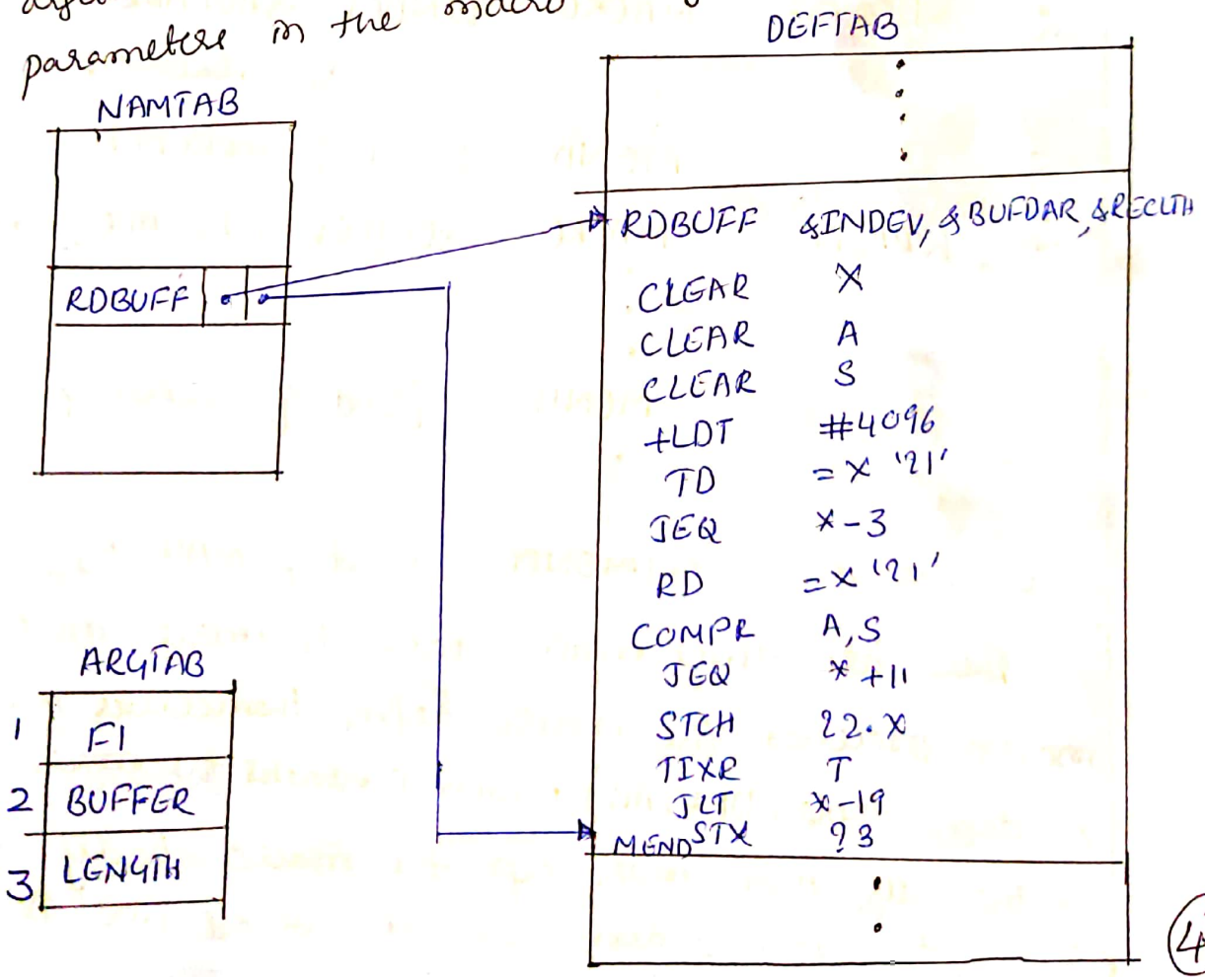
Scanned by CamScanner

```
        STCH  &BUFADR, X      Store character in buffer
        TIXR  T               loop unless maximum length
        JLT   *-19            has been reached
        STX   &RECLTH         save Record length
        MEND
WRBUFF  MACRO &OUTDEV, &BUFADR, &RECLTH
.
.       Macro to write record from buffer
.
        CLEAR X               clear loop counter
        LDT &RECLTH
        LDCH &BUFADR, X       GET character from buffer
        TD    =X '&OUTDEV'    Test output device
        JEQ   *-3             Loop until ready
        WD    =X'&OUTDEV'     write character
        TIXR  T               Loop until all characters
        JLT   *-14            have been written.
        MEND
.
.       Main program
.
FIRST   STL   RETADR          Save Return address
CLOOP   RDBUFF F1, BUFFER, LENGTH   Read record into buffer
        LDA   LENGTH          Test for end of file
        COMP  #0
        JEQ   ENDFIL          exit if EOF found
        WRBUFF 05, BUFFER, LENGTH   write o/p record
        J ~~@RETADR~~ CLOOP   Loop
ENDFIL  WRBUFF 05, EOF, THREE Insert EOF marker
        J @RETADR
```

Scanned by CamScanner

because they will not be part of the macro expansion. References to the macro instr. parameters are converted to a positional notation for efficiency in substituting arguments. The macro names are also entered into NAMTAB, which serves as an index to DEFTAB. For each macro instr defined, NAMTAB contains pointers to the beginning & end of defn in DEFTAB.

The 3rd data structure is an argument table (ARGTAB) used during the expansion of macro invocations. When a macro invocation stmt is recognized the arguments are stored in ARGTAB according to their position in the argument list. As the macro is recognized the arguments are stored in ARGTAB according to their position in the argument list. As the macro is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body.



Scanned by CamScanner

```
1   MACROS    MACRO   (Defines SIC std. version no.)
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}
```

SIC/XE machine

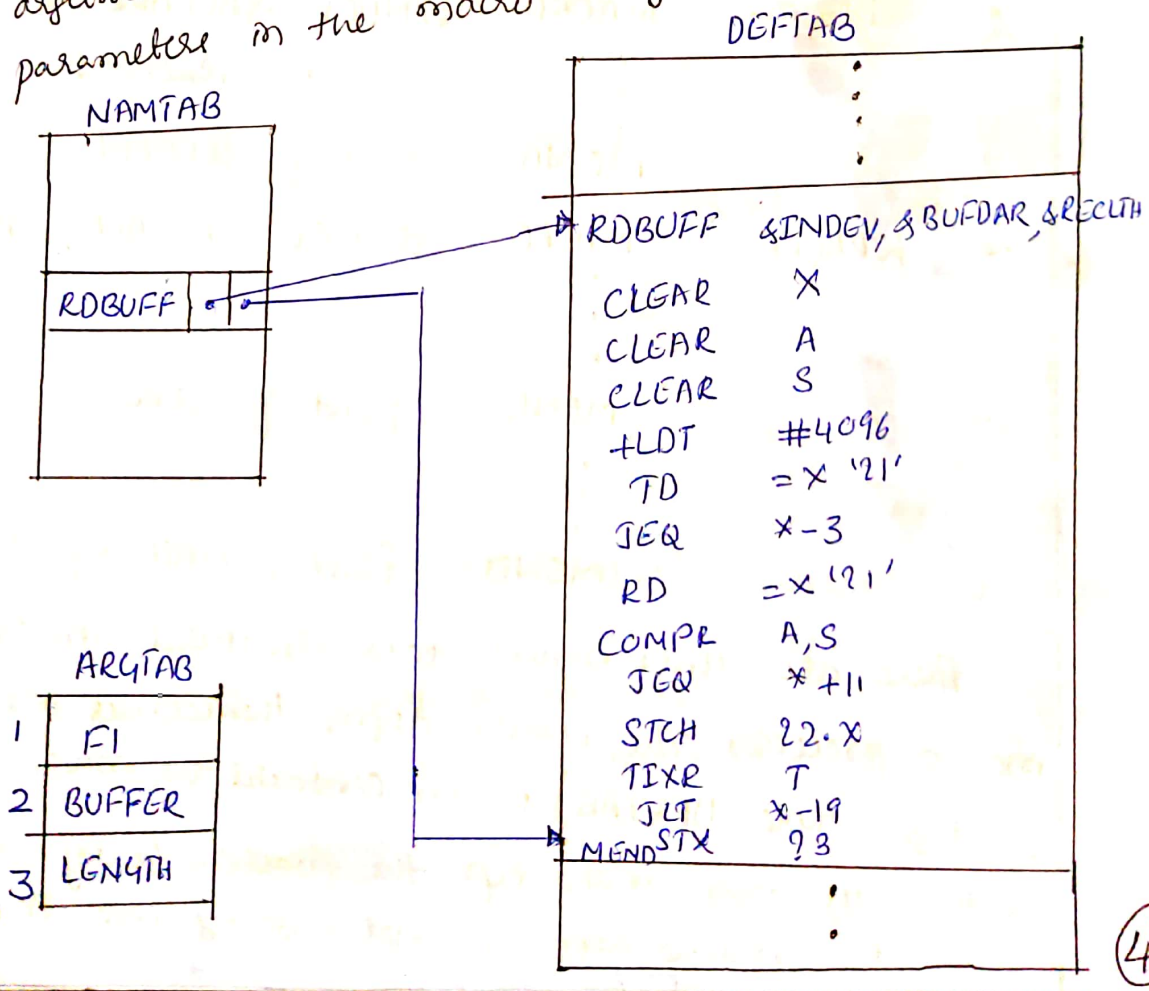
```
1   MACROX    MACRO   {Defines SIC/XE macro}
2   RDBUFF    MACRO   &INDEV, &BUFADR, &RECLTH
                  .
                  .   [SIC/XE version}
                  .
3             MEND    {End of WRBUFF}
4   WRBUFF    MACRO   &OUTDEV, &BUFADR, &RECLTH
                  .
                  .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

There are three main data structures involved in a macro processor. The macro defns themselves are stored in a defn table (DEFTAB) which contains the macro prototype & the stmts that make up the macro body. Comment lines from the macro defn are not entered into DEFTAB

Scanned by CamScanner

because they need not be part of the macro expansion. References to the macro instr. parameters are converted to a positional notation for efficiency in substituting arguments. The macro names are also entered into NAMTAB, which serves as an index to DEFTAB. For each macro instr defined, NAMTAB contains pointers to the beginning & end of defn in DEFTAB.

The 3rd data structure is an argument table (ARGTAB) used during the expansion of macro invocations. When a macro invocation stmt is recognized the arguments are stored in ARGTAB according to their position in the argument list. As the macro is recognized the arguments are stored in ARGTAB according to their position in the argument list. As the macro is expanded, arguments from ARGTAB are substituted for the corresponding parameters in the macro body.



Scanned by CamScanner

230	EOF	BYTE	C'EOF'	
235	THREE	WORD	3	
240	RETADR	RESW	1	
245	LENGTH	RESW	1	length of EOF found
250	BUFFER	RESB	4096	4096-Byte Buffer Area
255		END	FIRST2	

Above program defines & uses two macro instns, RDBUFF & WRBUFF. The fns. & logic of the RDBUFF macro are similar to those of the RDREC subroutine. The WRBUFF macro is similar to the WRREC subroutine. The defns of these macro instns appear in the source program following the START stt.

Two new assembler directives (MACRO & MEND) & used in macro defns. The first MACRO stt identifies the beginning of a macro definition. The symbol in the label field is the name of the macro & the entries in the operand field identify the parameters of the macro instn. In our macro language, each parameter begins with the character &, which facilitates the substitution of parameters during macro expansion. The macro name & parameters define a pattern or prototype for the macro instns used by the programmer. Following the MACRO directive are the stts that make up the body of the macro defn. These are the stts that will generated as the expansion of the macro. The MEND assembler directive marks the end of the macro definition. The definition of the WRBUFF macro follows a similar pattern.

Macro invocation gives the name of the macro instn being invoked & the arguments to be used in expanding the macro.

Scanned by CamScanner

The macro instn defns have been deleted since they are no longer needed after the macros are expanded. Each macro invocation stmt has been expanded into the the stmts that form the body of the macro, with the arguments from the macro invocation substituted for the parameters in the macro prototype & so on. In expanding the macro invocation on line 190, the argument F1 is substituted for the parameter &INDEV wherever it occurs in the body of the macro. BUFFER is substitute for &BUFADR & LENGTH is substituted for &RECLTH. The comment lines within the macro body have been deleted, but comments on individual stmts have been retained. The macro invocation stmt itself has been included as a comment line. The serves as documentation of the stmt written by the programmer. The label on the macro invocation stmt (CLOOP) has been retained as a label on the first stmt generated in the macro expansion. This allows the ~~stmt~~ programmer to use a macro instn in exactly the same way as an assembler language mnemonic. The two invocations of WRBUFF specify different arguments, so they produce different expansions.

Macro Processor Algorithm and Data Structures

All macro defns are processed during the first pass, & all macro invocation stmts are expanded during the 2nd pass. macro MACROS contains stmts that define RDBUFF, WRBUFF & other macro instns for a SIC System & MACROX defines these same macros for a SIC/XE system.

Scanned by CamScanner

Figure shows the definition of RDBUFF stored in DEFTAB with an entry in NAMTAB identifying the beginning & end of the defn. The positional notation that has been used for the parameters: the parameter &INDEV has been converted to ?1, &BUFADR has been converted ?2, & so on. ARGTAB as it would appear during expansion of the RDBUFF stmt on line 190. For this invocation, the first argument is F1, the second is BUFFER etc. This scheme makes substitution of macro arguments much more efficient. When the ?n notation is recognized in a line from DEFTAB, a simple indexing operation supplies the proper argument from ARGTAB.

Algorithm for a one-pass macro processor

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

procedure PROCESSLINE
  begin
     search NAMTAB for OPCODE
     if found then
        EXPAND
     else if OPCODE = 'MACRO' then
        DEFINE
```


Scanned by CamScanner

```
    else write source line to expanded file
end {PROCESSLINE}

procedure DEFINE
  begin
      enter macro name into NAMTAB
      enter macro prototype into DEFTAB
      LEVEL := 1
      while LEVEL > 0 do
        begin
          GETLINE
          if this is not a comment line then
            begin
              substitute positional notation for parameters
              enter line into DEFTAB
              if OPCODE = 'MACRO' then
                LEVEL := LEVEL + 1
              else if OPCODE = 'MEND' then
                LEVEL := LEVEL - 1
              end {if not comment}
            end {while}
          Store in NAMTAB pointers to beginning & end of
                                          definition.
      end (DEFINE)

procedure EXPAND
    begin
        EXPANDED := TRUE
        get first line of macro definition (prototype)
                                        from DEFTAB
        Set up arguments from macro invocation in ARGTAB
        write macro invocation to expanded line as a comm.
```

Scanned by CamScanner

```
[procedure EXPAND
  begin
    EXPANDING := TRUE
    Get first line of macro definition (prototype) from DEFTAB]
  While not end of macro definition do
    begin
      GETLINE
      PROCESSLINE
    end {while}
    EXPANDING := FALSE
  end {EXPAND}
Procedure GETLINE
  begin
    if EXPANDING then
      begin
        get next line of macro definition from DEFTAB
        substitute arguments from ARGTAB for positional notation
      end{if}
    else
      read next line from input file
  end{GETLINE}.
```

The procedure DEFINE is called when the beginning of a macro defn. is recognized, makes the appropriate entries in NAMTAB & DEFTAB. EXPAND is called to set up argument values in ARGTAB & expand a macro invocation stmt. The procedure GETLINE is called at several points in the algm, gets the next line to be processed. This line may come from DEFTAB or from the i/p file, depending upon whether

Scanned by CamScanner

the Boolean variable EXPANDING is set to TRUE or FALSE when a macro definition is being entered into DEFTAB, the normal approach would be to continue until an MEND directive is reached. This would not work for the example in fig. The MEND on line 3 would be taken as the end of the defn of maintains a counter named LEVEL. Each time a MACRO directive is read, the value of LEVEL is increased by 1: each time an MEND directive is read, the value of LEVEL is decreased by 1. When LEVEL reaches 0, the MEND that corresponds to the original MACRO directive has been found. This process is very much like matching left & right parantheses when scanning an arithmetic expression.

KSR

Signature of HOD

Signature of Faculty

Scanned by CamScanner