LOCCTR – Location Counter

2/3/22

Machine Independent Assembler Features

- Literals
- Symbol defining symbols statements
- Expression

(4). Program blocks.

- Entire program is considered as single entity. or treated.
- Instructions and data → different order.
  order is diff from how they appear in source program.
  - Rearranged.
    ↳ object program } Memory loaded version of the program.

→ Assemble Directive : USE

- 3 program blocks.
  ① unnamed block. : which comprises of all Executable instructions
  USE

  ② CDATA
  → All data areas that are correspond to a few words or less in length data

  ③ CBLKS
  - Data areas consisting of larger blocks of memory.

- Makes addressing simple.

for cdata it takes 0000 and continues from there.

Block table

- It has the information about

Block Name	Block No	starting Address	Length
Default	0	0000	0066
CDATA	1	0066	000B.
CBLKS.	2	0071	1000.

just use

ended at 0063.

Pass 1

```
  0066
+ 000B
------
  0071
```

- At the end of pass 1, the length of each block is aware? YES

↳ because for each block there is a seperate location counter

Pass 2 :

Symbol → Address is taken from SYMTAB

\+

Start LOC of block.

```
0000   FIRST   STL   RETADR
0003
```

```
        USE   CDATA
0000    RETADR  RESW   1
```

As it is CDATA

0000

\+ 0066 → length of CDATA.

RETADR → 0066

PC → 0003

TA - PC = 0063.

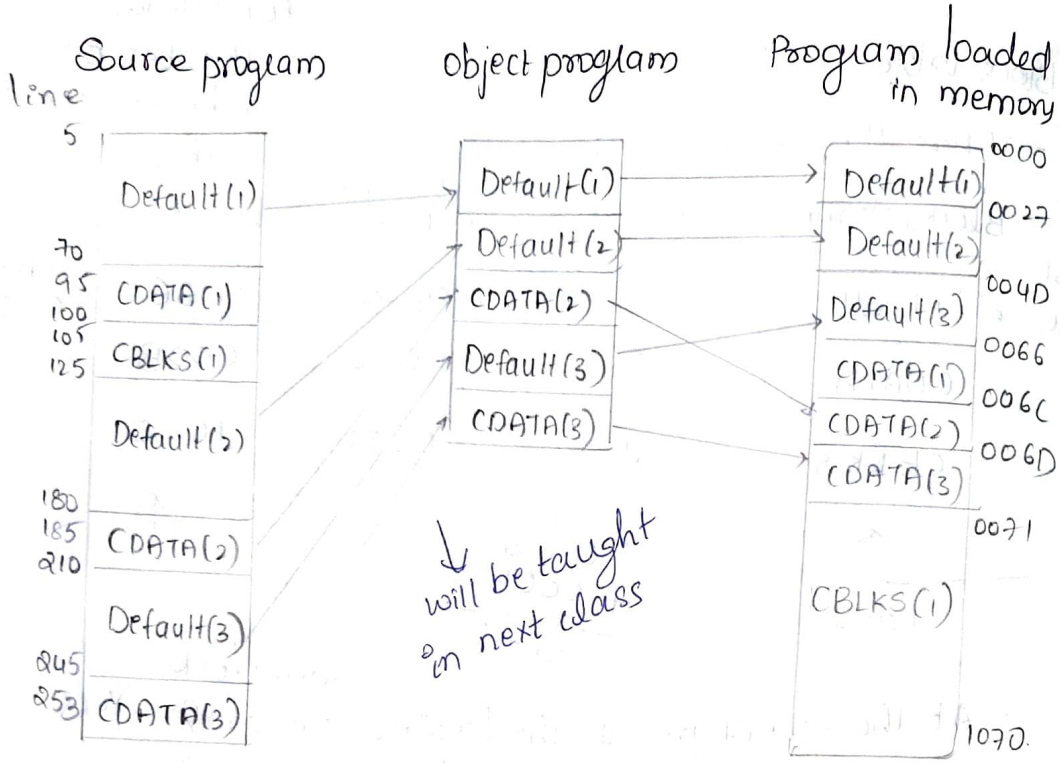
↓

Displacement

* if both PC & TA are in same block directly subtract No need to add length. becoz both PC & TA had to add length

Cacls while writing of object code

⑤ Control sections & Program Linking



--No need of using extended format & Base registers.

A Machine independent Assembly program

0000 → USE

0000	COPY	START	0	
0000	FIRST	STL	RETADR	172063
0003	CLOOP	JSUB	RDREC	4B2021
0006		LDA	LENGTH	032060
0009		COMP	#0	290000 (0010 10|01 0000|000)
000C		JEQ	ENDFIL	332006
000F		JSUB	WRREC	4B203B
0012		J	CLOOP	3F2FEE
0015	ENDFIL	LDA	=C'EOF'	032055
0018		STA	BUFFER	0F2056
001B		LDA	#3	010003
001E		STA	LENGTH	0F2048
0021		JSUB	WRREC	4B2029
0024		J	@RETADR	3E203F

next PC location = 0027
R

```
0066 0000                USE CDATA
0066 0000    RETADR      RESW 1
0069 0003    LENGTH      RESW 1
0071 0000                USE CBLKS
0071 0000    BUFFER      RESB 4096
1070 1000    BUFFEND     EQU *
1000 1000    MAXLEN      EQU BUFFEND-BUFFER
                         length of CBLKS
                         : Subroutine to read record into buffer
0027 0027                USE
0027         RDREC       CLEAR X          B410
0029                     CLEAR A          B400
002B                     CLEAR S          B440
002D                     +LDT #MAXLEN     75101000
0031         RLOOP       TD INPUT         E32038
0034                     JEQ RLOOP        332FFA
0037                     RD INPUT         DB2032
003A                     COMPR A,S        A004
003C                     JEQ EXIT         332008
003F                     STCH BUFFER,X    57A02F
0042                     TIXR T           B850
0044                     JLT RLOOP        3B2FEA
0047         EXIT        STX LENGTH       132017
004A                     RSUB             4F0000
006C 0006                USE CDATA
006C 0006    INPUT       BYTE X'F1'       F1
                         : Subroutine to write record from buffer
004D 004D                USE
004D         WRREC       CLEAR X          B410
004F                     LDT LENGTH       772017
0052         WLOOP       TD =X'05'        E3201B
0055                     JEQ WLOOP        332FFA
0058                     LDCH BUFFER,X    53A016
005B                     WD =X'05'        DF2012
005E                     TIXR T           B850
```

0060	JLT	WLOOP	3B2FEF
0063	RSUB		4F0000
till 0065 — 0007 - 0060	USE	CDATA	
So length = 0066 — 0007 + 0061	LTORG		
000A ← 0070	= C'EOF'		454F46
	= X'05'		05
000B — 0071	END	FIRST	

Constants

4/3/22

0071
- 0066
000B → size of USE CDATA

Machine-Independent Assembler Features

- Literals
- Symbol defining statements
- Expressions
- Program blocks

⑤ Control sections & program linking

Object program:-

H^COPY_ _ ^000000^00107 1

T^000000^1E^172063^4B2021^032060^290000^332006^

T^000000^1E^172063^4B2021^032060^290000^332006^
4B203B^3F2FEF^032055^0F2056^010003^A

T^00001E^09^0F2048^4B2029^3F203F

T^000027^1D^B410^B400^B440^75101000^E32038^

T^000027^1D^B410^B400^B440^75101000^E32038^
332FFA^DB2032^A004^332008^57A02F^
B850

T^000044^09^3B2FEA^132017^4F0000

T^00006C^01^F1

T^00004D^19^B410^774017^F32018^332FFA^53A016^
0F2012^B850^3B2FEF^4F0000

T^00006D^04^454F46^05

E^000000

- Control sections → part of the program that maintains its identity after assembly.
- Each control section can be loaded and relocated independently.
  ↳ Resulting flexibility is a major benefit of using control sections

① Define Record → information about external symbols. (EXDEF)

Col 1 : D

Col 2-7 : Name of the external symbol defined in this control section

Col 8-13 : Relative address of symbol written within this control section.

Col 14-73 : Repeat information in col 2-13 for other external symbols

② Refer Record → EXREF.

Col 1 : R

Col 2-7 : Name of the external symbol defi referred to in this control section.

Col 8-73 : Names of other external symbols.

③ Modification record (modified)

Col 1 : M

Col 2-7 : Starting address of the address field to be modified relative to the beginning of the control section (in hexadecimal)

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

Col 10 : Modification flag (+ or -)

Col 11-16 : External symbol whose value is to be added or subtracted from the indicated field.

```
                COPY    START   0           external definitions.
                        EXTDEF  BUFFER, BUFFEN,
                                LENGTH
                        EXTREF  RDREC, WRREC
0000    FIRST   STL-14  RETADR
0003    CLOOP   +JSUB   RDREC   4B100000
0007            LDA     LENGTH  032023
000A            COMP    #0      290000
000D            JEQ     ENDFIL  332007
0010            +JSUB   WRREC   4B100000
0014            J       CLOOP   3F2FEC
0017    ENDFIL  LDA     =C'EOF' 032016
001A            STA     BUFFER  0F2016
001D            LDA     #3      010003
0020            STA     LENGTH  0F200A
0023            +JSUB   WRREC   4B100000
0027            J       @RETADR 3E2000
002A    RETADR  RESW    1
002D    LENGTH  RESW    1
0030    *       LTORG
        =C'EOF'
0033    BUFFER  RESW    4096
1033    BUFFEND EQU     *
1000    MAXLEN  EQU     BUFFEND-BUFFER

0000    RDREC   CSECT
```

: Subroutine to read record into buffer.

```
        EXTREF  BUFFER, LENGTH, BUFFEND
        CLEAR   X
        C

                EXTREF  BUFFER, LENGTH, BUFFEND
0000            CLEAR   X       B410
0002            CLEAR   A       B400
0004            CLEAR   S       B440
0006            LDT     MAXLEN  77201F
```

```
                0009 RLOOP   TID     INPUT      E3201B
                000C         JEQ     RLOOP      332FFA
                000F         RD      INPUT      DB2015
                0012         COMPR   A,S        A004
                0014         JEQ     EXIT       332009
CS2             0017         +STCH   BUFFER,X   57900000
                001B         TIXR    T          B850
                001D         JLT     RLOOP      3B2FEA
                0020 EXIT    +STX    LENGTH     13100000
                0024         RSUB               4F0000
                0027 INPUT   BYTE    X'F1'      F1
                0028 MAXLEN  WORD    BUFFEND-BUFFER.  000000

                0000 WRREC   CSECT
                : Subroutine to write record from buffer

CS3             0000         EXTREF  LENGTH, BUFFER
                0000         CLEAR   X          B410
                0002         +LDT    LENGTH     77100000
                0006 WLOOP   TD      =X'05'     E32012
                0009         JEQ     WLOOP      332FFA
                000C         +LDCH   BUFFER,X   53900000
                0010         WD      =X'05'     DF2008
                0013         TIXR    T          B850
                0015         JLT     WLOOP      3B2FEE
                0018         RSUB               4F0000
                             END     FIRST

                001B         =X'05'             05
```

⇒ The assembler establishes a seperate location counter (begin at 0) for each control section, just as it does for program blocks.

⇒ Control sections differ from program blocks in that they are handled seperately by the assembler.

- The EXDEF statement in a control section names symbols, called external references symbols, that are defined in this control section and may be used by other sections.

Control section names → COPY, RDREC, WRREC do not need to be named in an EXTDEF statement because they are automatically considered to be external symbols.

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

① CLOOP +JSUB RDREC.

RDREC is EXTREF, assembler has no idea where the control section containing RDREC will be loaded, so it cannot assemble the address for this instruction.

Instead assembler inserts an address of zero and passes information to the loader, which will cause the proper address to be inserted at load time.

② 0028 MAXLEN WORD BUFFEND-BUFFER.

Here the value of the data word to be generated is specified by an expression involving two external references: BUFFEND and BUFFER. As before, the assembler stores this value as 0. zero. when the program is loaded, the loader will add to this data area the address of BUFFEND and subtract from it the address of BUFFER, which results in the desired value.

Object program :

H^COPY^ - -^000000^001033

↳ D^BUFFER^000033^BUFFEND^001033^LENGTH^00002D

Define record R^RDREC^ - ^WRREC.

T^000000^1D^172027^4B1000000^032023^290000^332007^4B100000^3F2FEC^032016^0F2016^010003^0F200A

T^ 00001D^0D^010003^0F200A^4B100000^3E2000.

T^000030^03^454F46.

M^000004^05^+RDREC

M^000011^05^+WRREC

M^000024^05^+WRREC.

E^000000.

H^RDREC^000000^00002B

R^BUFFER^LENGTH^BUFFEND

T^000000^1D^B410^B400^B440^77201F^E3201B^
332FFA^DB2015^A004^332009^57900000^B85D.

T^00001D^0E^3B2FE9^13100000^4F0000^F1^000000

M^000018^05^+BUFFER

M^000021^05^+LENGTH

M^000028^06^+BUFEND

M^000028^06^-BUFFER.

} size of word = 24 bytes
6x4 = 24
↓
6 half bytes.

H^WREC_^000000^00001C

R^LENGTH^BUFFER

T^000000^1C^B400^77100000^6E32012^332FFA^539000000^
DF2008^B85D^332FEE^4F0000^05

M^000003^05^+LENGTH

M^00000D^05^+BUFFER.

14/03/22

M/c independent features

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

Design options

- 2-pass Assemblers
  - forward references
    ↳ made for Data & Instruction symbols (labels).
- One pass Assembler

  Can I have a one pass assembler?
  - How do I take care of forward references.
- Multi-pass assembler.

One-pass assembler :

↳ scan through the program once.

Issues :

- Forward reference.

  Data symbols & Instruction labels.
  - Data
    - forward reference for data can be avoided.
    - By compelling - all the data related declarations

need to be made prior to usage of such symbols.

- Instruction labels
  - No way to avoid this forward reference

These cannot be declared first. - { CMP SIZE / JEQ EXIT } otherwise loop continues

JSUB

EXIT ++++

```
COPY    START   1000
EOF     BYTE    C'EOF'
THREE   WORD    3
ZERO    WORD    0
RETADR  RESW    1
LENGTH  RESW    1
BUFFER  RESB    4096

FIRST   STL     RETADR
CLOOP   JSUB    RDREC
        LDA     LENGTH
        .
        .
        .
RDREC
```

Data symbols declared first.

→ Instruction label

→ This is not resolved

2 types of one pass assembler

① Load-and-go → not generate object program
↳ Directly load into it onto memory after conversion, no seperate object program that is saved on disk.

② which generates the object program.

① Load-and-go one pass assembler

- Line-by line → read source program line-by-line
  ↳ As soon as it read, it translates the same.

Steps :

- Read line-by-line → directly after translation, loaded onto memory
- As soon as, symbol is declared → If make an entry in SYMTAB.
- Assemble the an Instruction to generate object code
  ↳ Searches for the symbol in SYMTAB
  - If found → Use it
  - If not found → make an entry in symbol table as list item.
    ↓
    like RDREC

In 2 pass assembler,
label is entered into SYMTAB, when label is created but not when label is used.

But in one pass assembler,
when label is used and not found in SYMTAB, it enter into SYMTAB as list item

list node

2012 RDREC

2013 is the address field of RDREC

Memory address	Contents				Symbol	Value
1000	454F4600	00030000	00XXXXXX	XXXXXX	LENGTH	100C
1010	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXX	RDREC	* → 2013 | 0
					THREE	1003
					ZERO	1006
2000	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXX14	WRREC	* → 201F | 0 →
2010	100948_20	3R00100C	28100630	2024820	EOF	1000
2020	62_3C2012				ENDFIL	* → 2010 | 0
					RETADR	1009
					BUFFER	100F
					CLOOP	2012
					FIRST	200f

2031 | 0

(2) One-pass-assemblee - that genuates object progeam.

↳ Directly loaded in proper way, not again going back and completing the object codes like last time.
(blanks)

- Object progeam may introduce records, Additional Text records. whereever the adjustments need to made.

-But not modification record ↳ it is risky.

H^COPY_ - ^001000^001071

T^001000^09^454F46^000003^000000

T^00200F^15^141009^480000^00100C^281006^300000^480000

302012 → used at

T^00201C^02^2024 ⟹ label defined & used at 2024

T^002024^19^001000^0C1001F^001003^0C100C^480000^081009^4C0000^F1^001000

⇒ T^002013^02^203D
T^00203D^1E^041006^001006^E02039^302054^D82039^
281006^300000^54900F^2C203A^382043

⇒ T^002059^02^205B
T^00205B^07^101005^4C0000^05

⇒ T^00201F^02^2062 } used two times.
16/3/22 ⇒ T^002031^02^2062 }
T^002062^18^041006^E02061^302065^50900F^D(206)^
2C1005^382065^4C0000
E^002001F

Design options

- One-pass
- Multi-pass

One pass :-

- Forward Reference

↓ Stop the text records when it reach Max (1E) & when label is encountered.

Load-and-go
object programs.

Multi-pass assembler

No issue
```
DELTA  RESW  1
BETA   EQU   DELTA+1   ⇒ no problems
ALPHA  EQU   BETA+2
```

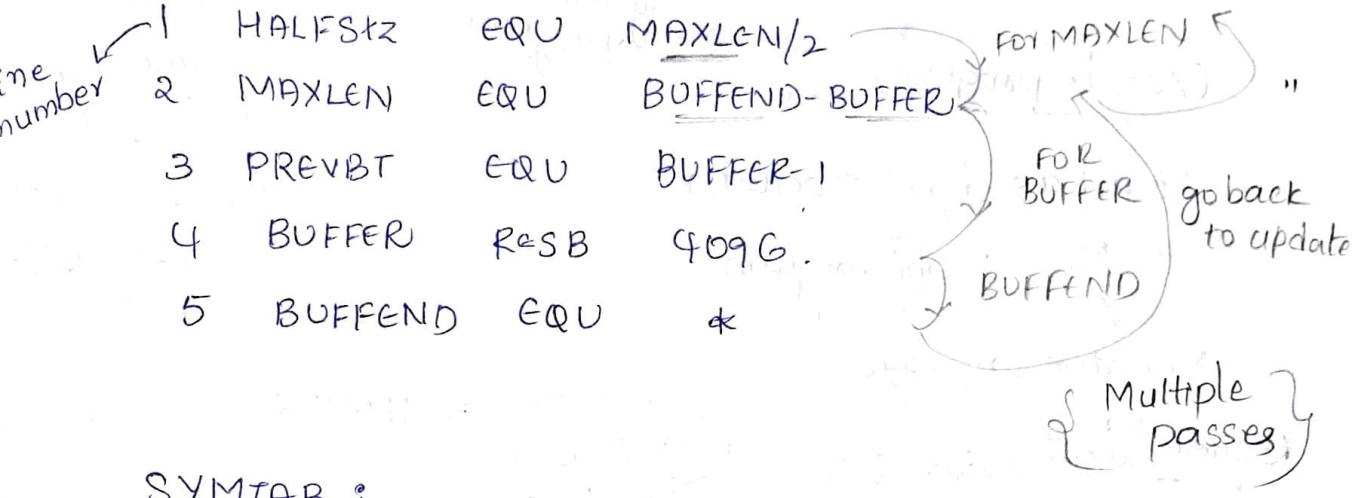
```
ALPHA  EQU   BETA+2     Problem!
BETA   EQU   DELTA+1
DELTA  RESW
```

Solutions

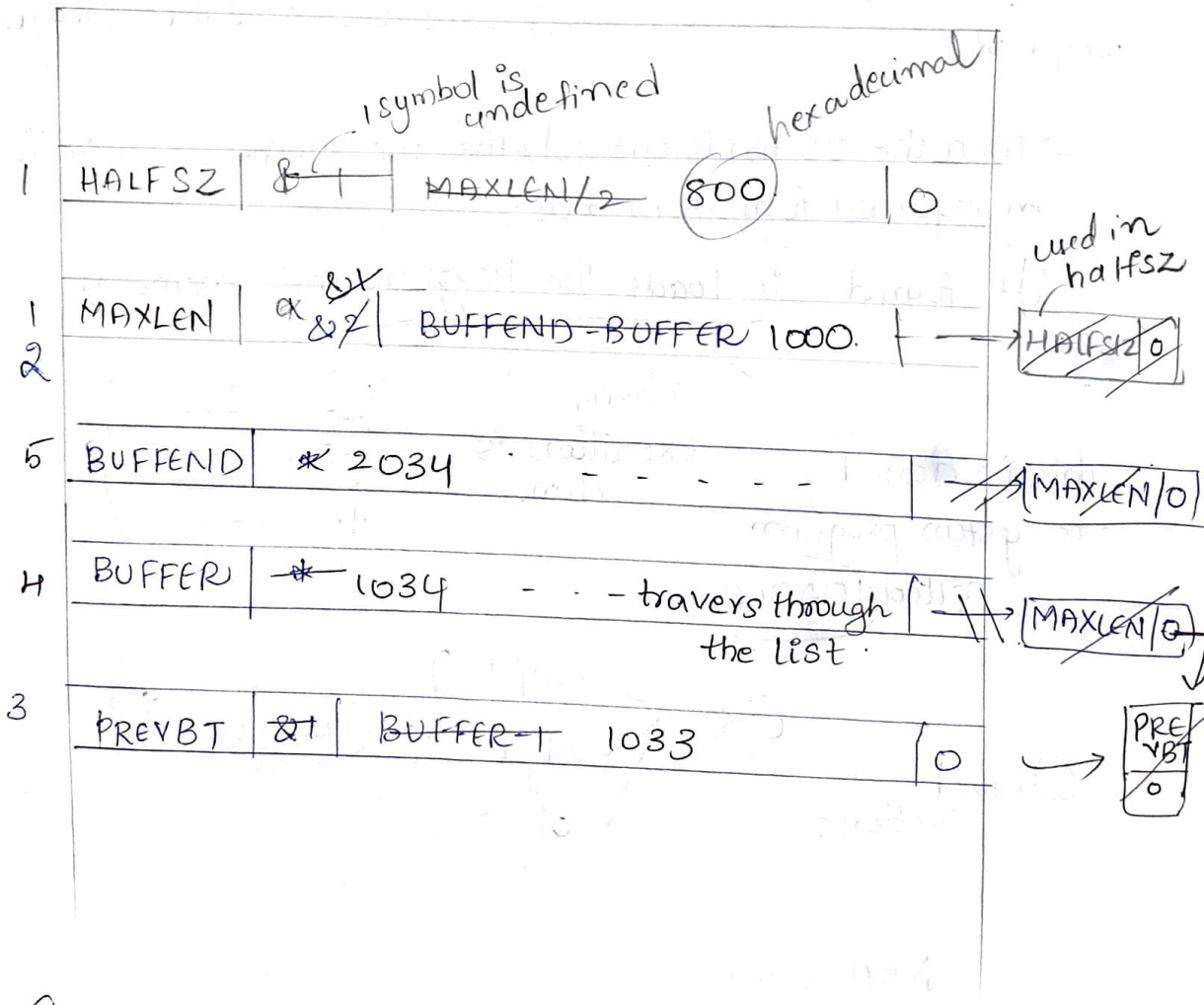
Necessary changes to assembler

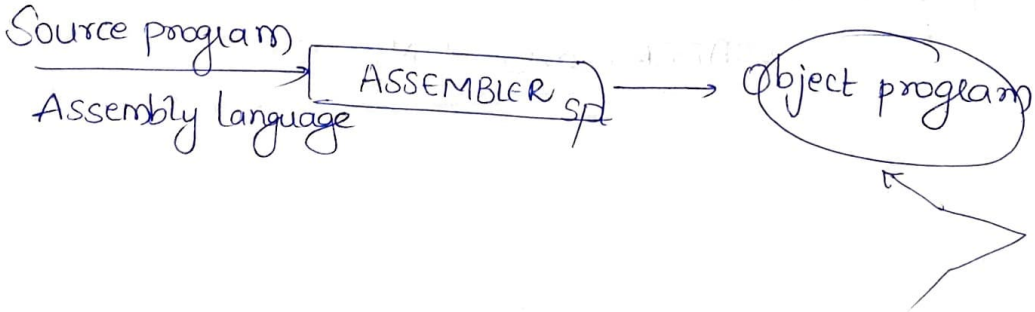
Restrictions on usage of symbols ⇒ declare before using
↓
like in HLLs.

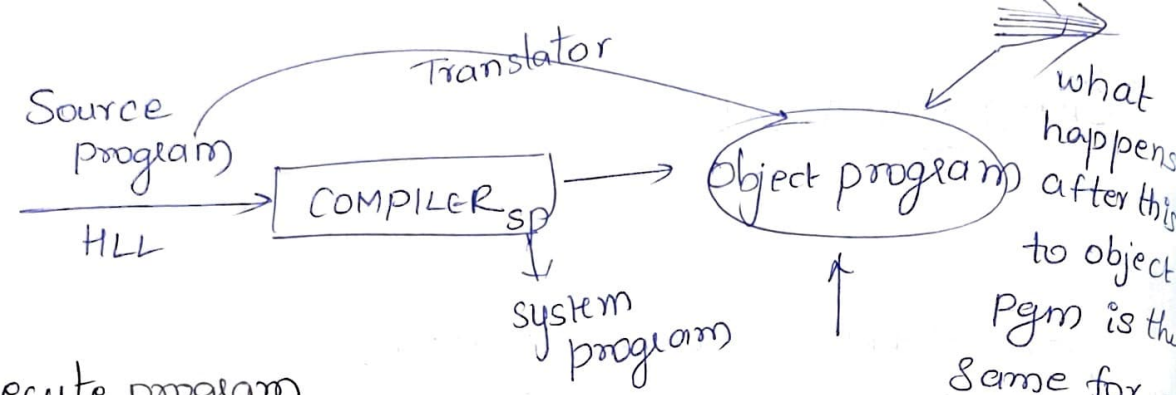
Changes to Assembler :-



SYMTAB :







Execute program

- Stored as object program

ML → obj program

↓

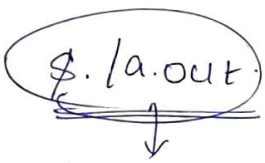
Machine language

Request for Execution of → OS

↓ a program

↓

request.

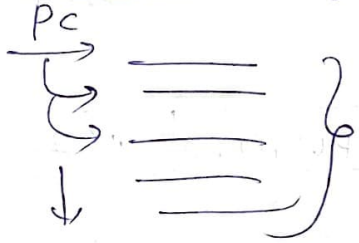


- Then the OS finds out whether the program is available in required format or not.

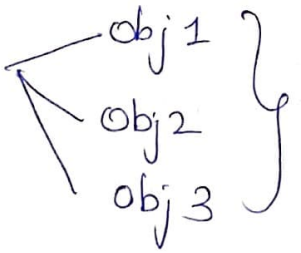
If found, it loads the program onto memory.

This is done by a system program called LOADER

Then, Execution is done.



CSs ↙ Control Sections.



Linker SP.

Next Unit

LOADERS and LINKERS

LOADERS and LINKERS

Modification records — Absolute addressing
F4 instructions.

16 bytes — 10 Hexadecimal

Absolute loader

141033 — 3 bytes
1 1 1

while loading this, it is considered (14) as 1 byte

Functionalities of Loader :-

1, Brings the program into memory
or loads

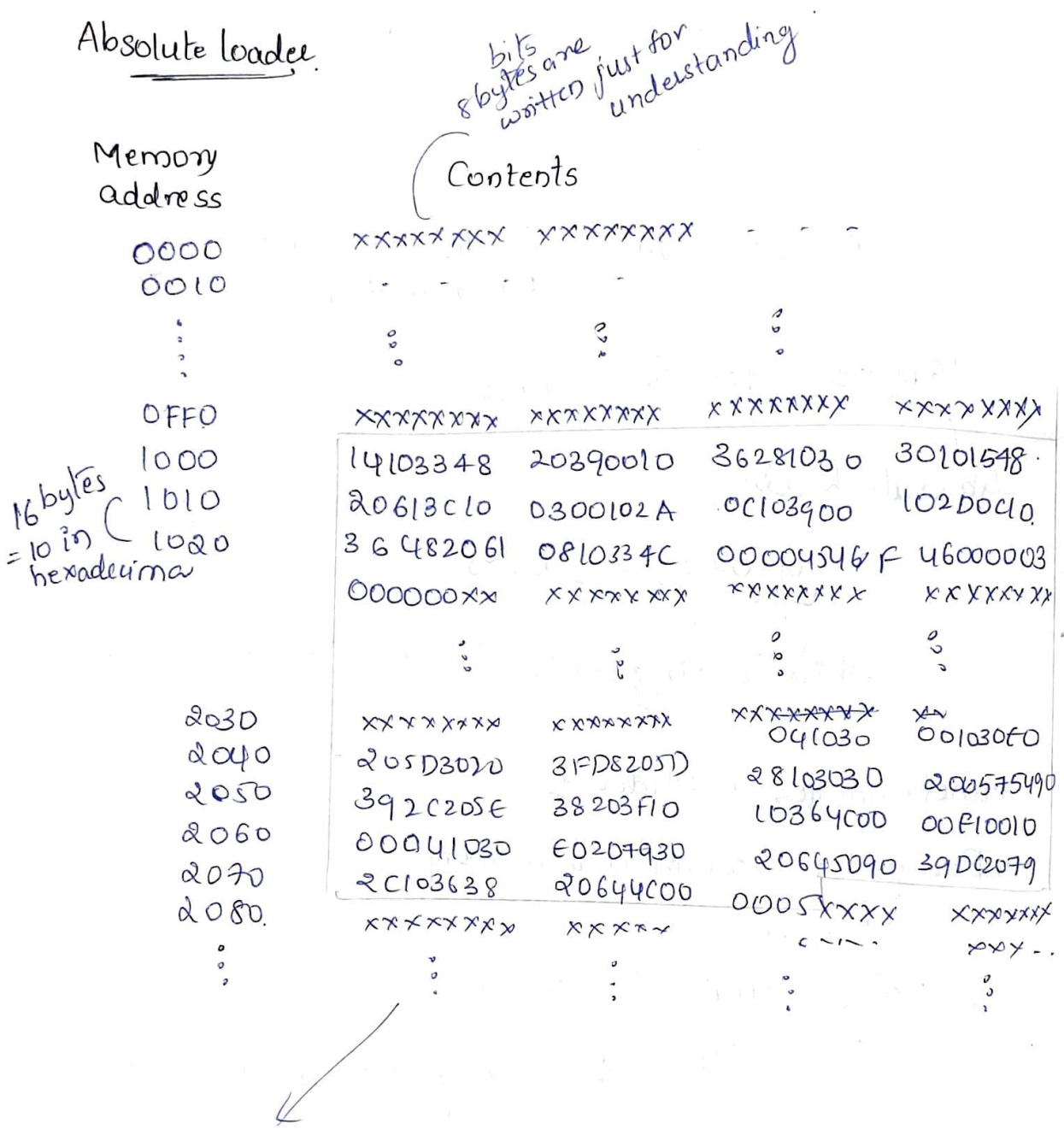
2, It also initiates its execution.
↳ Load the starting address onto PC.

Bootstrap Loader
↓
Absolute Loader

- OS has to load, for any program
↳ loader is taken as part of OS

How entire OS loads into memory → Bootstrap loading

Absolute loader



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 obj code to specified location in memory
            read next object program record
    end
```

jump to address specified in End record

end.

29/3/22

Loaders and linkers

- Absolute loader
  - Bootstrap loader
- Complex loader
  - M/c independent
    - Relocation & linking of control sections.

Done by reloating loader

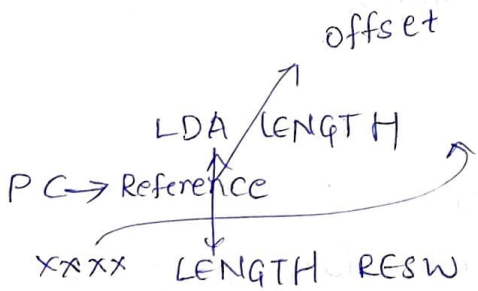
↓

Assembler has to convey that where to relocate the program wrt starting address.

→ Majority of problems can be resolved by using relative addressing

↓

This is used in program relocation because offset is fixed



Directly giving the address

↓

Absolute addressing

- Offset is fixed in relative addressing.

→ Absolute addressing is used in F4 instructions.

↓

20 bit address field.

How to deal...

How to deal with this type of instructions is also conveyed by assembler via Modification records.

1. Modification Record
2. Bitmask

- M^000007^05 ← first version of modification record that we studied.
  - 000007 → Adress of Address field
  - 05 → length in half bytes
  - Loader Adds starting address of the program to this.
    Add field + starting address.

- M^000004^05^+RDREC. — Used in program blocks and control sections.
  - +RDREC → External symbol (ext reference).

How to convert first version as second version?

M^000007^05^+COPY

+COPY → Program name which represents the starting address of the program.

Scenario:

How to relocate the SIC program?
or Can we have a relocatable loader for SIC?

By using Modification record for every instruction writing M^000007^05^+COPY.

If there are 75 instructions, 75 Modification records.

This is a good solution for small (less) no. of instructions like 75.

Better solution for Program containing More no. of instructions is BITMASK

- COPY program (SIC/XE) but write modification records including +COPY.

Relocation by Modification records.

SIC/XE relocation loader algorithm.:

```
begin
  get PROGADDR from operating system
  while not end of i/p do
    begin
      read next record
      while record type ≠ 'E' do
        begin
          read next i/p record
          while record type = 'T' then
            begin
              move obj code from record to location
                ADDR + specified address
            end
          while record type = 'M'
            add progladd R to the location PROGADDR
                PROGADDR + specified
                                address.
        end
    end
end.
```

Bitmask

- used only when more modification records are needed.

Relocation by bit mask.

T^000000^1E^140033^ - - -

T^000000^1E^FFC^140033 - - -

1111 1111 1100 ← no instructions are there

First 10 instructions are modified. need to be

- Constants need not to be modified.

T^00001E^15^E00^0C0036^ - - - -

1110 0000 0000

first 3 instructions need to be modified.

FF · 800

1000 0000 0000

first instruction obj code need to be modified

Relative loader with program Linking

How the control sections linked together.

- LISTB, ENDB

  Buffer defining symbols of program B.

- For external symbols F4 (format 4) is used.
- 3 programs are identical here.

like no problem with these instructions.

```
0000  PROGA  START   0
            EXTDEF  LISTA,ENDA
            EXTREF  LISTB,ENDB,LISTC,ENDC
            :
002D0 REF1  LDA     LISTA       03201D
0023  REF2  +LDT    LISTB+4     77200004
0027  REF3  LDX     #ENDA-LISTA 050014
            :
0040  LISTA EQU     *
            :
0054  ENDA  EQU     *
0054  REF4  WORD    ENDA-LISTA+LISTC    000014
0057  REF5  WORD    ENDC-LISTC-10       FFFFF6
005A  REF6  WORD    ENDC-LISTC+LISTA-1  00003F
005D  REF7  "       ENDA-LISTA-(ENDB-LISTB) 000014
0060  REF8  "       LISTB-LISTA         FFFFC0
            END     REF1
```

these instructions need to not to change

LISTB - unknown

evaluated of know things

here LISTA is wrt starting address of program, so we have to add the Address which relocating

object program :

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

05 → 5X4 = 20 Address field starts at 0024.

Here no need to take Address field, these are just definitions

M^000005A^06^-LISTC
M^000005A^06^+PROGA
M^000005AD^06^-ENDB
M^000005D^06^+LISTB
M^000005D^06^+LISTB
M^000005B^06^+LISTB.
60^
M^000060^06^-PROGA
E^000020

- PROGB, PROGC are same, just interchange of F3 & F4 instruction
(external symbols of that program are written F4.

30/3/22

— OS tells the loader to load progA at 4000, B&C should immediately follow

→ T^000020^ →
while encountering this text record, Relocating loader will automatically add 1000 to 0020.

→ Linking loader should have values for external symbols.
(Similar problem seen at forward reference in assembler.
↓
Resolved by making SYMTAB in pass 1

Similarly Here also external symbols are stored → EXTAB
External symbol table

It should scans upto Refer record.

→ While scanning PROGB, LISTBA is encountered, then it sees EXTAB, if it is there it leaves else it enters in EXTAB

↓

But in assembler, it throws error, when symbol come again in diff

- After linking and loading of CS's, WORDS → constants are same for all 3 programs.

But rest diff → becoz in PROGA it is FY
but PROGB, C it may F3
if it is same format, then that obj code will also be same.

Load address

PROGA	004000	~size = 0063
PROGB	004063	
PROGC	0040E2	

all symbols are stored.
↑
like in assembler

→ No need to store all other symbols in linker, only external reference symbols need to store.

4/4/22

Linking Loader

① Which are the object programs need to be loaded?

② PROGA
PROGB
PROGC

↓
How Linking loader know this?
- Standard library.

When PROGA has to be loaded → external symbols are stored in EXTAB.

Linker also going to make "2 passes".

ESTAB → External symbol table.

- External symbols.

CSName	SymbolName	Address	Length
PROGA		4000	0063.
	LISTA	4040	
	ENDA	4054	
PROGB		4063	007F
	LISTB	40C3	
	ENDB	40D3	
PROGC	LISTC	40E2	0057
	ENDC	4112	
		4124	

4063 + 7F = 40E2

This table is generated after pass 1

PROGADDR - program address → Starting address of the linked program

CSADDR - Control section address → Starting address of control section

CSLTH - Control section length → length of this control section

EXECADDR - Execution (exec) address → to start execution of loaded program

Pass 1:

ESTAB creation

```
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}
```

instructor

i/p given to linking loader = which are the control sections need to link.

read one define record and exit but if many define records are there then? if Refer record encountered then exit

for next control section

indicated address = offset within the control section

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}

5/4/22

Algorithm for linking loader

- ESTAB (Data structure) is hash Table.
- PROGADDR
- CSADDR
- CSLTH
- EXECADR ⌝ address from which execution starts

mylinkingloader ⌜ input

$./mylinkingloader (PROGA.O PROGB.O PROGC.O).

3 possibilities.

① One cs mentions the address, other cs won't mention ⌜ like one cs is considered as main. Main contains that address.
then execution address will be that address.

↓

E, (000020)
+
E xxxxxx
E yyyyyy

② No cs specify ⌝ Assembler directive will tell the main. where it starts execution.

③ User tells this is fully linked program.

Who convey the execution address —

3 cs specify, then all those are considered individu

If standard system library is used, then how linker works?

Automatic library search / Automatic library call :-

- This is going to be taken care by the linking loader.

#include <stdio.h>

Search and find out what is required in this Particular program

Linking loader ⇒ goes through the library

Hello World example program.

- If certain symbols are not resolved, then there may be a error.
- When seach will finish? Till all symbols resolve.

It's not just one scan

(they may have some external symbols in each of them.

Libraries are automatically linked, & there is no involvement of user, so it is called as Automatic Library seach

It stops, when all symbols resolve (all symbols enter into & ESTAB)

→ It requires some better way of seaching

Special Data structures

↓

Inverted file structure

Used for directory searches

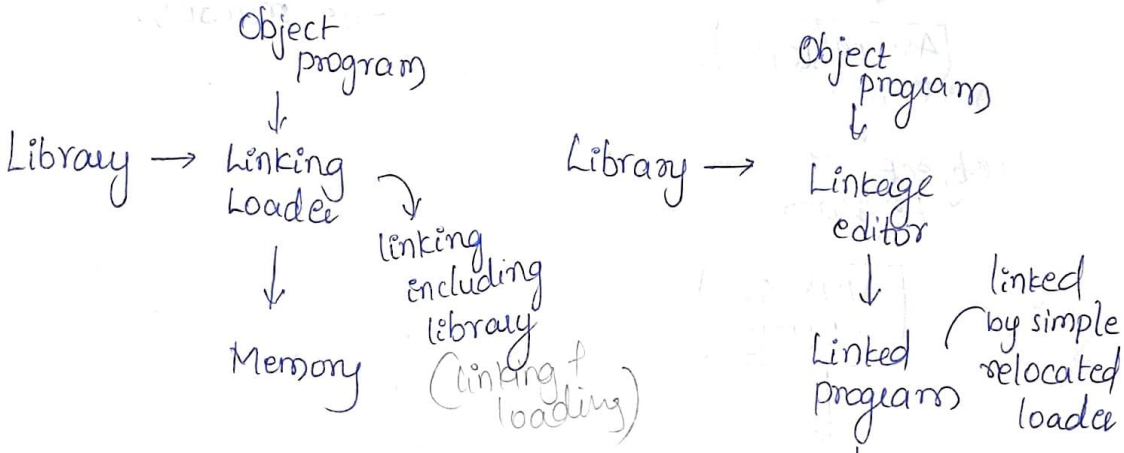
Loader options :

INCLUDE prog.name(library)

DELETE → if you don't want older edition of progeam (control section) then you can DELETE that

CHANGE → it will change that particular CS to new one

LINKAGE EDITOR :



Programs :-

① rarely executed → linking loader → program changing everytime

② Frequently executed → linkage editor → We can run everytime which include library search and all
- instead of that we will store linked version of that and execute whenever we want by using linkage editor

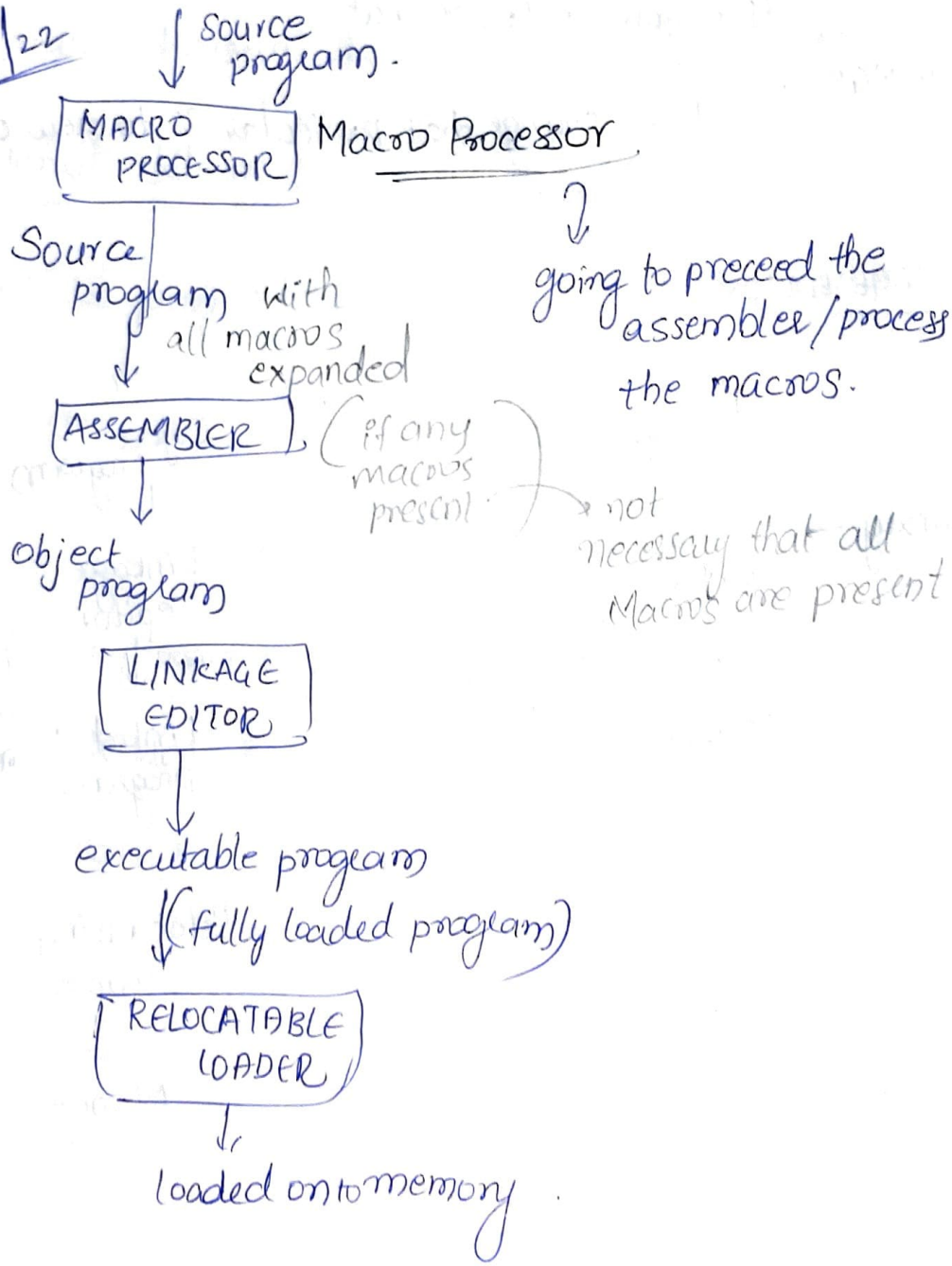
③ if we want to include some extra in the same program ⇒ then also Linkage editor

DLL → Dynamic Linking Library.

↳ At run time linking takes place.

Multiple programs use system library

6/4/22



For high level language PREPROCESSOR

MACRO

or

MACRO INSTRUCTION

(short term notation of some group of instructions in the program.

```
RLOOP   TD   INDEV
        JEQ  RLOOP
        RD   INDEV
```

if device is not ready EQ flag is set.

RDMACRO : short cut notation

Macro invocation

- instead of writing these three lines again and again, just define a macro.

- Macro processor, just replaces the macro with corresponding instructions.

- Whatever source program statements btw MACRO & MEND is one macro definition

```
NAME    MACRO   <PARAMETERS>
        |
        |  body of MACRO.
        |
        MEND.
```

MEND - Macro End

- There are no labels used. (like RLOOP)
- We cannot have labels to instructions — problematic !!

This problem is not there in case of usage of labels.

(if someone introduced an instruction in middle then we have to change that *+11 / *-3.

CLOOP RDBUFF F1, BUFFER, LENGTH ← macro invocation

(

After this line macro expansion will be done.

↙

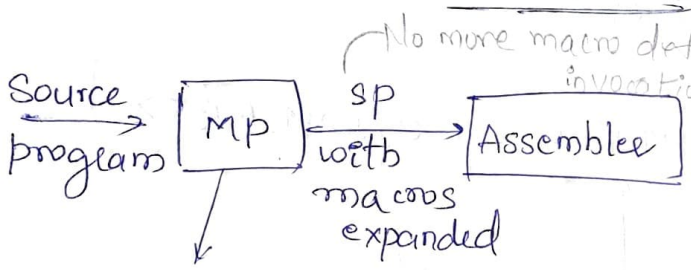
This is what done by MACROPROCESSOR.

- Parameters during invocations are diff (even for same buffers) macros

- Macro processor have to check whether macro keyword is there in opcode / whether it is directive / normal opcode
- If macro keyword, it checks NAMTAB whether the symbol is already entered or not. If not then make entry of that definition.
  macro.

19/4/22

MACRO PROCESSORS



- Macro definition
- Macro Invocation
- Macro Expansion

} → programmer role

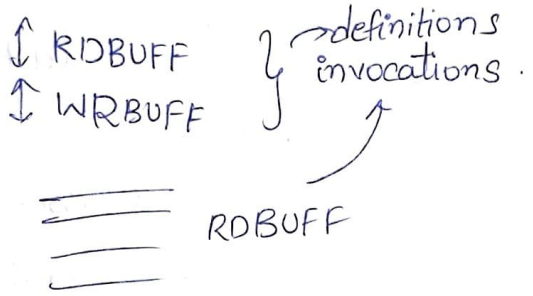
use of macro is known as macro invocation

How is Macro defined?
- 2 keywords (kind of directives) MACRO, MEND

MACRO – where MACRO starts
MEND - where macro ends.

<NAME> MACRO <Params> – macro definition

Invocation → <NAME> Parameters.



Data structures used by Macro Processor

3 data structures

① DEF TAB (store the definition of macro definition)
② NAM TAB (Name table - name of the macro and pointers to the start & end pair of definition)
③ ARG TAB (Argument table - parameters)

invocation → values given to the parameters.

DEFTAB

First line - MACRO prototype without the keyword MACRO in opcode field.
<NAME> parameters.

- Parameters are considered as positional parameters
  - '?1' → 1st parameter
  - '?2' - 2nd parameter.
  - '?3' - 3rd parameter
  - ↓ Question mark followed by position.

ARGTAB :-

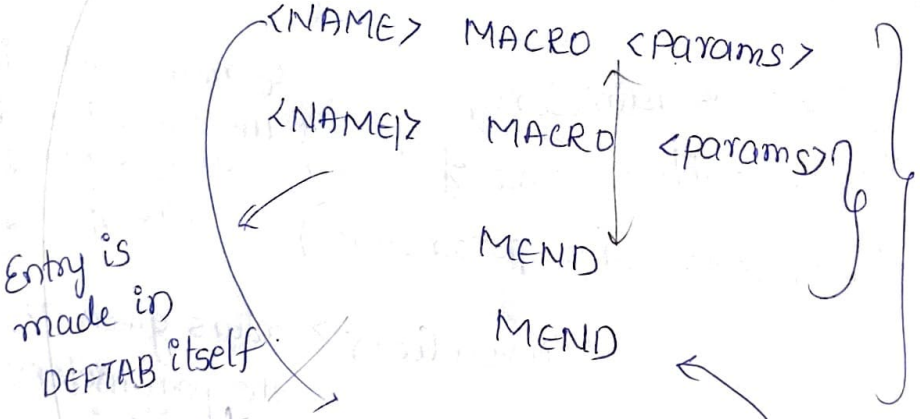
Values used in place of particular parameters are stored/listed in ARGTAB.

- What if MACRO definition is in middle of program
  - Again problem!
    - "Frwd Reference"
  - Solution
    - 2 passes.

      Pass-1 → fill those tables

      Pass-2 → Expansion.

- What if MACRO defined in Another macro.



- it is not that much smart that MEND of that particular MACRO is here.

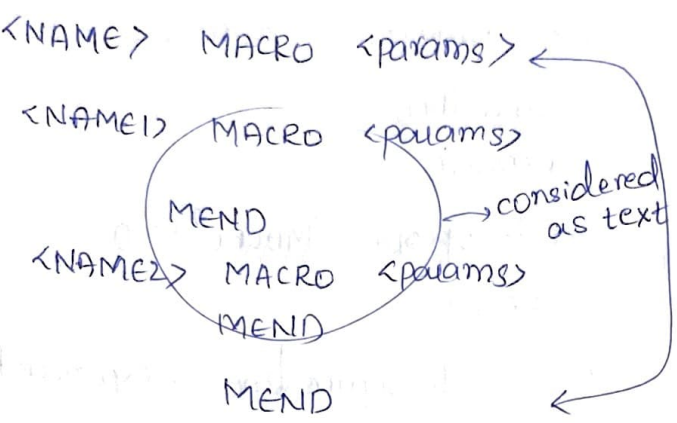
Scenario:

1 MACRO is for SIC
1 MACRO for SIC/XE
- Use appropriately

- Suppose, this situation arise, we need to do some changes.

SIC/XE → MACRO2

SIC → MACRO1



- When MACRO1 is expanded, MACRO2 has to be defined.

Pass 1 - Definition
Pass 2 - Expansion
} But in this case we have to switch btw these two passes.

- When MACRO1 is invoked, MACRO1 is defined and the macros inside that macro are considered as text.

Clearly explained by the example.

- MACRO within a MACRO
- MACRO defined for SIC, MACRO defined for SIC/XE.

Algorithm for Macro Processor {one-pass}

```
begin (macro processor)
    EXPANDING := FALSE        (flag set to false)
    while OPCODE ≠ 'END' do   (this is the field that macro processor have to look into; 'END' of source program)
        begin
            GETLINE           (function calls)
            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 DEFINE
    begin
        enter macro name into NAMTAB
        enter macro prototype into DEFTAB        Name followed by
        LEVEL := 1                                parameter
        while LEVEL>0 do
            begin
                GETLINE
                if this is not a comment line then
                    begin
                        substitute positional notation for
                                                  Parameters
                        enter line into DEFTAB
                        if OPCODE = 'MACRO' then
                            LEVEL := LEVEL + 1
                        else if OPCODE = 'MEND' then
                            LEVEL := LEVEL - 1
                    end {if not comment}
            end {while}
        Store in NAMTAB pointers to beginning and end of
                                                  definition.
```

```
    end {DEFINE}

procedure EXPAND
  begin
    EXPANDING := TRUE
    get first line of macro definition {prototype} from
     DEFTAB
    Set up arguments from macro invocation in ARGTAB
    write macro invocation to expanded file as a comment
    while not end of macro definition do
       begin
          GETLINE
          PROCESSLINE
       end {while}
     EXPANDING := FALSE
  end {EXPAND}

procedure GETLINE
  begin
    if EXPANDING then
      begin
        get next line of macro definition from DEFTAB
        Substitute augurments from ARGTAB for
            positional notation.
      end {if}
    else
        read next line from input file
  end {GETLINE}.
```

- Generally A program contains macros definition first and then main program starts {which includes macros usage}

- Macro processor, first define the macros and then when enters the main program, during the usaage of macros it expands the macros there

→ Now, what happens when there is one macro inside another macro ?.

```
1  MACROS   MACRO    {Defines SIC standard version
                      compiler 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}
```

(a)

```
1. MACROX   MACRO    {Defines SIC/XE macros}
Rest all
same                 {SIC/XE version}.
```

(b).

- While defining MACROS, new definitions for RDBUFF & WRBUFF will not happen. Just they will include as text.

- During the expansion of MACROS (when used in main program) ⇒ definitions of RDBUFF & WRBUFF made will.

Note: If WRBUFF is inside RDBUFF then there will be a clash, so make sure don't make that mistake while writing the program.