

## MODULE 3

### SYLLABUS

Assembler Design Options: Machine Independent Assembler Features – Program Blocks, Control Sections, Assembler Design Options- Algorithm for Single Pass Assembler, Multi pass Assembler, Implementation Example of MASM Assembler

### 3.1 MACHINE INDEPENDENT ASSEMBLER FEATURES

The features which are NOT closely dependent to machine architecture are called machine independent assembler features. The machine independent assembler features includes:

1. Literals
2. Symbol Defining Statements
3. Expressions
4. Program Blocks
5. Control Sections and Program Linking

#### 3.1.1 LITERALS

- It is convenient for the programmer to be able to write the value of a constant operand as part of the instruction that uses it.
- This avoids having to define the constant elsewhere in the program and make a label for it.
- Such an operand is called a Literal because the value is literally in the instruction.
- A literal is defined with a prefix '=' followed by a specification of the literal value.
- Consider the following example:

```

      .
      .
      LDA      FIVE
      .
      .
FIVE      WORD 5
```

Using the concept of literal we can rewrite the above code as:

```

      .
      .
      LDA      =X'05'
```

**Difference between literal operands and immediate operands**

- For literals prefix is =, and for immediate addressing prefix is #.
- In immediate addressing, the operand value is assembled as part of the machine instruction, ie there is no memory reference.

Line no	Location Counter	
55	0020	LDA #03 010003

In the above example the last 12 bits of the machine code corresponds to 003 which is equal to the immediate value.

- With a literal, the assembler generates the specified value as a constant at some other memory location. The address of this generated constant is used as the target address (TA) for the machine instruction ( using PC-relative or base-relative addressing with memory reference.)

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

**Literal Pool**

- All the literal operands used in a program are gathered together into one or more literal pools. This is usually placed at the end of the program.
- In some cases, it is desirable to place literals into a pool at some other location in the object program. To allow this an assembler directive **LTORG** is used.
- When the assembler encounters a **LTORG** statement, it generates a *literal pool* containing all literal operands used since previous **LTORG** or the beginning of the program
- Literals placed in a pool by **LTORG** will not be repeated in a pool at the end of the program.
- Reason for using **LTORG** is to keep the literal operand close to the instruction (otherwise PC-relative addressing may not be allowed)

**Literal Table (LITTAB)**

- A literal table(**LITTAB**) is created for storing the literals which are used in the program.
- The literal table contains the literal name, operand value and length.
- The literal table is usually created as a hash table on the literal name.

**Duplicate literals**

- The same literal used more than once in the program, then it can be consider as a duplicate literal.
- In such cases, only one copy of the specified value needs to be stored
- To recognize the duplicate literals, two methods are there

1. Compare the character strings defining them

Easier to implement e.g. =X'05'. But not possible to handle the literals like =C'EOF' and =X'454F46'.

Here both literals are same in the form of their data value.

2. Compare the generated data value

Possible to handle the literals like =C'EOF' and =X'454F46'. Here both literals are same in the form of their generated data value. So comparison based on generated data value is needed to identify duplicate literals or not. But this is difficult to implement compared to the first method.

**Implementation of Literals****During Pass-1:**

The literal encountered is searched in the literal table. If the literal already exists, no action is taken; if it is not present, the literal is added to the LITTAB and for the address value it waits till it encounters LTORG or END statement for literal definition.

When Pass 1 encounters a LTORG statement or the end of the program, the assembler makes a scan of the literal table. At this time each literal currently in the table is assigned an address. As addresses are assigned, the location counter is updated to reflect the number of bytes occupied by each literal.

**During Pass-2:**

The assembler searches the LITTAB for each literal encountered in the instruction and replaces it with its equivalent value as if these values are generated by BYTE or WORD. The following figure shows the difference between the SYMTAB and LITTAB

SYMTAB

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

LITTAB

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

### 3.1.2 SYMBOL DEFINING STATEMENTS AND EXPRESSIONS

#### EQU Statement:

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

**Symbol EQU value**

- This statement defines the given symbol (i.e., entering in the SYMTAB) and assigns the value specified to that symbol.
- The value can be a constant or an expression involving constants and any other symbol which is already defined.
- One common usage is to define symbolic names that can be used to improve readability in place of numeric values. For example

LDA #100

This loads the register A with immediate value 100, this does not clearly mention what exactly this value indicates. If a statement is included as:

MAXLEN EQU 100

and then LDA #MAXLEN then it clearly indicates that the value of MAXLEN is some maximum length value and it is to be loaded in A register.

- When the assembler encounters EQU statement, it enters the symbol MAXLEN along with its value in the symbol table. During LDA the assembler searches the SYMTAB for its entry and its equivalent value as the operand in the instruction.

- The object code generated is the same for both the options discussed, but is easier to understand.
- If the maximum length is changed from 100 to 500, it is difficult to change if it is mentioned as an immediate value wherever required in the instructions. We have to scan the whole program and make changes wherever 100 is used.
- If we mention this value in the instruction through the symbol defined by EQU, we may not have to search the whole program but change only the value of MAXLENGTH in the EQU statement.

### **ORG Statement:**

- This directive can be used to indirectly assign values to the symbols. The directive is usually called ORG (means origin).
- Its general format is:

#### **ORG value**

where value is a constant or an expression involving constants and previously defined symbols.

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

- Eg: ORG ALPHA

When this statement is encountered during assembly of a program, the assembler resets its location counter (LOCCTR) to the value of ALPHA.

### **3.1.3 EXPRESSIONS**

- The assemblers allow **the use of expressions as operand**
- The assembler evaluates the expressions and produces a single operand address or value.
- Assemblers generally allow arithmetic expressions as operands formed according to the normal rules using arithmetic operators +, -, \*, /. (Division is usually defined to produce an integer result.)
- Individual terms may be constants, user-defined symbols, or special terms.
- The only special term used is \* ( the current value of location counter) which indicates the value of the next unassigned memory location.

Thus the statement

    BUFFEND EQU \*

Assigns the value of LOCCTR to BUFFEND, which is the address of the next byte following the buffer area.

- Some values in the object program are relative to the beginning of the program and some are absolute (independent of the program location, like constants). Hence, expressions are classified as either **absolute expression or relative expressions** depending on the type of value they produce.

- **Absolute Expressions:**

- The expression that uses only absolute terms is absolute expression. Absolute expression may contain relative term provided the relative terms occur in pairs with opposite signs for each pair.

- Example:

    MAXLEN EQU BUFEND-BUFFER

In the above instruction the difference in the expression BUFEND-BUFFER gives a value that does not depend on the location of the program and hence gives an absolute value

- **Relative Expressions:**

- The expression that uses the values relative to the program are called relative expression.
- Absolute expression may contain relative term provided the relative terms occur in pairs with opposite signs for each pair.

- Example:

    MAXLEN EQU ALPHA + BUFEND-BUFFER

In the above instruction the difference in the expression BUFEND-BUFFER gives a value that does not depend on the location of the program but it is added to the value of ALPHA which is program relative. Hence this expression is relative.

### 3.1.4 PROGRAM BLOCKS

- Program blocks allow the generated machine instructions and data to appear in the object program in a different order by **separating blocks for storing code, data, stack, and larger data block**.
- To implement the program block the Assembler Directive used is **USE**

- Syntax is

**USE [block name]**

- At the beginning, statements are assumed to be part of the unnamed (or default) block.
- Whenever a USE CDATA statement is encountered, statements upto next USE belongs to the program block named CDATA.
- If no USE statements are included, the entire program belongs to this single block.
- Each program block may actually contain several separate segments of the source program. Assemblers rearrange these segments to gather together the pieces of each block and assign address.
- Consider the following example:

```

COPY          START 0
              LDA      LENGTH
              .....
              .....
              USE       CDATA
MAX           RESW     1
LENGTH       RESW     1
              USE       CBLOCKS
BUFFER       RESB     00
              .....

//Subroutine to read record into buffer
              USE
RDREC        CLEAR    X
              LDA      INPUT
              .....
              .....
              USE       CDATA
INPUT        BYTE     X'F1'
              .....

//Subroutine to write record from buffer
              USE
WRREC        STA      MAX
              .....
              USE       CDATA
MIN          RESW     1
BUFEND       RESW     1
```

- In the example give above three program blocks are used :

DEFAULT: executable instructions.

CDATA: all data areas that are less in length.

CBLOCKS: all data areas that consists of larger blocks of memory.

DEFAULT
CDATA
CBLOCKS

### Arranging code into program blocks:

**During Pass 1 assembler performs the following operations:**

- A separate location counter for each program block is maintained.
  - At the beginning of a block, LOCCTR is set to 0.
  - Save and restore LOCCTR when switching between blocks.
- Assign each label an address relative to the start of the block.
- Store the block name or number in the SYMTAB along with the assigned relative address of the label
- Indicate the block length as the latest value of LOCCTR for each block at the end of Pass1
- Assign to each block a starting address in the object program by concatenating the program blocks in a particular order
- At the end of pass 1 a block table is generated.

#### **Block Table**

Block Name	Block Number	Starting Address	Ending Address	Length of Block
Default	0	0000	0065	0066
CDATA	1	0066	0070	000B
CBLKS	2	0071	1070	1000

**During Pass 2 assembler performs the following operations:**

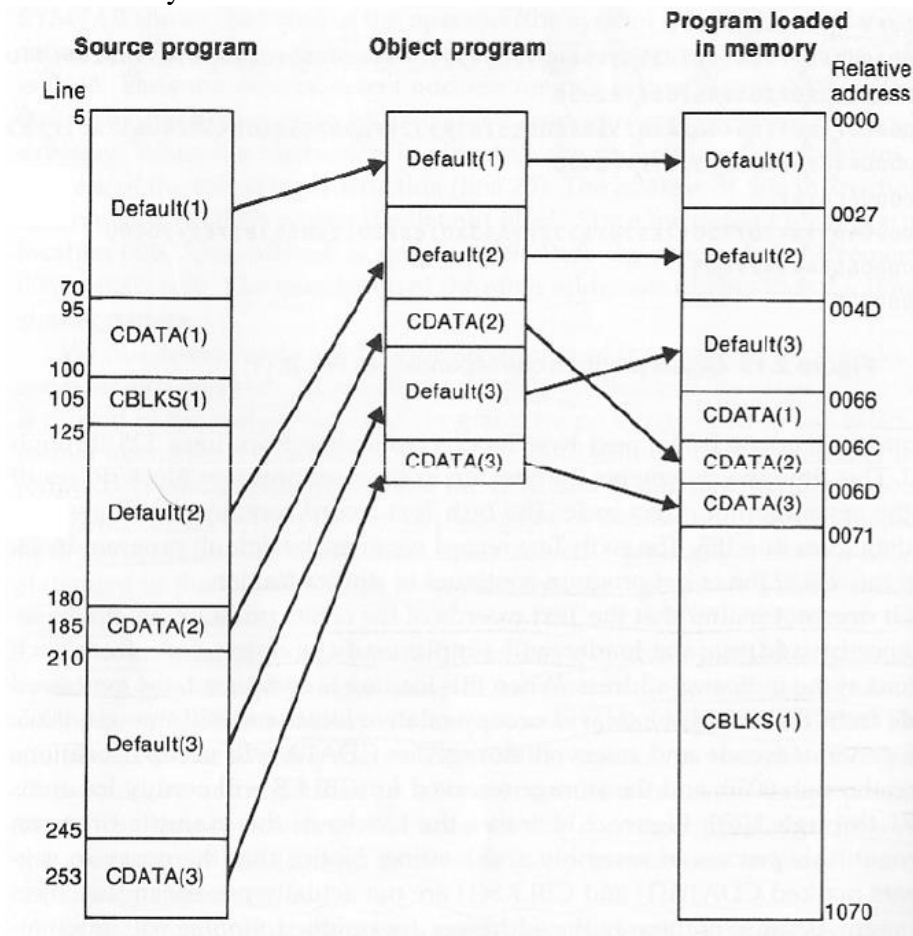
- Calculate the address for each symbol relative to the start of the object program by adding
  - The location of the symbol relative to the start of its block
  - The starting address of this block

### **Program Blocks Loaded in Memory**

Separation of program into blocks results in the movement of the large buffer (CBLKS) to the end of the object program. As a result extended format, base register



addressing etc are no longer needed. Modification records are also not needed. This improves program readability.



### 3.1.5 CONTROL SECTIONS

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

using multiple control sections, the beginning of each of the control section is indicated by an assembler directive: **CSECT**

- The syntax

**secname CSECT**

- The assembler maintain separate LOCCTR beginning at 0 for each control sections.
- Control sections differ from program blocks in that they are handled separately by the assembler.

### Handling of External References

Instructions in one control section may need to refer to instructions or data located in another section. This is called as **external references**. The external references are indicated by two assembler directives: **EXTDEF** and **EXTREF**

#### **EXTDEF (External Definition)**

- It defines the symbols that are defined in this control section and may be used by other sections
- Syntax - **EXTDEF** name [, name]
- Ex: **EXTDEF** BUFFER, BUFEND, LENGTH which means the symbols BUFFER, BUFEND and LENGTH are defined in this control section and may be used by some other control sections.

#### **EXTREF (External Reference)**

- It names symbols that are used in this section but are defined in some other control section.
- Syntax - **EXTREF** name [,name]
- Ex: **EXTREF** A,B which means the symbols A and B are used in this control section but are defined in some other control section.

The assembler must include information in the object program that will cause the loader to handle external references properly. For this three types of records are used in object program: **Define, Refer and Modification Record**.

Define Record	
Col. 1	D
Col. 2-7	Name of external symbol defined in this control section
Col. 8-13	Relative address of symbol within this control section (hex)
Col. 14-73	Repeat information in Col 2-13 for other external symbols

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

Modification Record (revised)	
Col. 1	M
Col. 2-7	Starting location of the target address to be modified, relative to the beginning of the program (not relative to the first text record)
Col. 8-9	Length of this record in half-byte
Col. 10	Modification flag (+ or -)
Col. 11-16	External symbol whose value is to be added to or subtracted from the indicated field

The format of modification record which we studied in Module 2 is revised to support the handling of external references.

Consider the following code segments:

```

COPY          START 0
              EXTDEF BUFFER, BUFFEND, LENGTH
              EXTREF A,B
              LDA          ALPHA
              .....
              .....
              .....
BUFFER        WORD          3
BUFFEND       EQU           *
LENGTH        EQU           BUFFEND-BUFFER

              RDREC         CSECT
              EXTREF        BUFFER, BUFFEND, LENGTH
              .....
              .....
              .....
              LDA          BUFFER
              .....
              .....
              .....

              END

```

The object program generated for the above code segment is:

```
H^ COPY ^ 000000^001033
D^BUFFER^000033^BUFEND^001033^LENGTH^00002D
R^A  ^B
T^.....
T^.....
.....
.....
M^000004^05^+RDREC
.....
E^000000
```

## 3.2 ASSEMBLER DESIGN OPTIONS

In this section, two alternatives to the standard two-pass assembler logic is discussed. They are:

Single Pass Assembler

Multipass Assembler

### 3.2.1 SINGLE PASS ASSEMBLER

These assemblers are used when it is necessary or desirable to avoid a second pass over the source program. The main problem in designing the assembler using single pass was to resolve forward references.

One-pass assemblers could produce object codes either in memory or to external storage. One-pass assemblers usually need to modify object code already generated, so whether object code is stored in memory or external storage imposes different considerations on assembler design. Based on this one-pass assemblers can be classified into two types:

1. One that produces object code directly in memory for immediate execution (**Load-and-go assemblers**).
2. One pass assembler generating object code for later execution.

#### 1. Load-and-Go Assembler

Load-and-go assembler generates their object code in memory for immediate execution. Since no object program is written out, no loader is needed. It is useful in a system with frequent program development and testing. Since the object program is produced in memory, the handling of forward references becomes less difficult.

**Working of One pass assembler (Load and Go Assembler)**

In load-and-Go assemblers when a forward reference is encountered :

- Omits the operand address if the symbol has not yet been defined(placess 000 at the operand addresses position)
- Enters this undefined symbol into SYMTAB and indicates that it is undefined
- Adds the location at which the operand is referenced to a list of forward references associated with the SYMTAB entry
- When the definition for the symbol is encountered, scans the reference list and inserts the address.
- At the end of the program, reports the error if there are still SYMTAB entries indicated undefined symbols(\* indicates undefined).
- When the END statement is encountered, search SYMTAB for the symbol named in the END statement and jumps to this location to begin execution if there is no error.

**In short, whenever any undefined symbol is encountered it will insert into SYMTAB as a new entry and indicate that it is undefined and also adds the location at which the operand is referenced as a linked list associated with that SYMTAB entry. When the definition for the symbol is encountered, scans the reference list and inserts the address in proper location.**

**Algorithm for Single Pass Assembler (Load and Go Assembler)**

```
begin
    read first input line
    if OP CODE = 'START' then
    {
        save #[OPERAND] as starting address
        initialize LOCCTR as starting address
    } //end of if OP CODE = 'START'
    else
        initialize LOCCTR to 0
    write Header record to object program
    read next input line
    while OP CODE ≠ 'END'
    {
        if this is not a comment line
        {
            if there is a symbol in the LABEL field
            {
                search SYMTAB for LABEL
```

```
    if found
    {
        if symbol value as null
        {
            set symbol value as LOCCTR

            search the attached forward reference list(if exist) and the address
            of the symbol is inserted into any instructions previously generated

            delete the forward reference list attached to that symbol

        }
    }
    else

        insert (LABEL, LOCCTR) into SYMTAB
} //end of if there is a symbol in the LABEL field

search OPTAB for OPCODE

if found

    search SYMTAB for OPERAND ADDRESS

    if found
    {
        if symbol value not equal to null

            store symbol value as operand address

        else

            insert a node with address LOCCTR at the end of the
            forward reference list of that symbol

    }
    else
    {
        insert (symbol name, null)

        insert a node with address LOCCTR at the end of the forward
        reference list of that symbol

    }

    add 3 to LOCCTR

else if OPCODE ='WORD'

    add 3 to LOCCTR

else if OPCODE ='RESW'

    add 3 #[OPERAND] to LOCCTR

else if OPCODE ='RESB'

    add #[OPERAND] to LOCCTR

else if OPCODE = 'BYTE'
```

```

        {
            find length of constant in bytes

            add length to LOCCTR

            convert constant to object code

        }

    if object code will not fit into current text record

    {
        write Text record to object program

        initialize new text record

    }

    add object code to Text record

    read next input line

}

} //end of while OP CODE ≠ 'END'

write last Text record to object program

write End record to object program

end

```

### Example:

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

Memory address	Contents	Symbol	Value
1000	454F4600 00030000 00xxxxxx xxxxxxxx	LENGTH	100C
1010	xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx	RDREC	* → 2013 0
...	...	THREE	1003
...	...	ZERO	1006
2000	xxxxxxxx xxxxxxxx xxxxxxxx xxxxxx14	WRREC	* → 201F 0
2010	100948--- --00100C 28100630 ----48---	EOF	1000
2020	---3C2012	ENDFIL	* → 201C 0
...	...	RETADR	1009
...	...	BUFFER	100F
...	...	CLOOP	2012
...	...	FIRST	200F

Line	Loc	Source statement	Object code
15	2012	CLOOP JSUB RDREC	48203D
30	201B	JEQ ENDFIL	302024
35	201E	JSUB WRREC	482062

When the definition for the symbols RDREC and ENDFILL are encountered, the reference list associated with the symbols is scanned and the address is inserted at proper location. It is given in following figure:

Memory address	Contents				Symbol	Value
1000	454F4600	00030000	00xxxxxx	xxxxxxx	LENGTH	100C
1010	xxxxxxx	xxxxxxx	xxxxxxx	xxxxxxx	RDREC	203D
...					THREE	1003
...					ZERO	1006
2000	xxxxxxx	xxxxxxx	xxxxxxx	xxxxxxx14	WRREC	* → 201F → 2031 0
2010	10094820	3000100C	28100630	202448	EOF	1000
2020	3C2012	0010000C	100F0010	030C100C	ENDFIL	2024
2030	480008	10094C00	00F10010	00041006	RETADR	1009
2040	001006E0	20393020	43D82039	28100630	BUFFER	100F
2050	5490	0F			CLOOP	2012
...					FIRST	200F
...					MAXLEN	203A
					INPUT	2039
					EXIT	* → 2050 0
					RLOOP	2043

Line	Loc	Source statement	Object code
35	201E	JSUB WRREC	482062
65	2030	JSUB WRREC	482062
160	204F	JEQ EXIT	30205B

## 2. One pass assembler generating object code for later execution.

In this type of one pass assembler, the generated object program is stored in external storage (e.g., files on disks). So random updates to operands target addresses (as in load-and-go load-and-go assemblers do) are not permitted.

For any symbol involved in forward references, once the target address of the symbol is identified, additional text records must be generated to overwrite those previously omitted target addresses. Records must be loaded in the same order as they appear in the object program. Actually, the handling of forward references are jointly done by the assembler and the linking loader.

One pass assembler which generates object code unlike load and go assembler operates in the following fashion:

- If the operand contains an undefined symbol, use 0 as the address and write the Text record to the object program.
- Forward references are entered into lists as in the load-and-go assembler.



- When the definition of a symbol is encountered, the assembler generates another Text record with the correct operand address of each entry in the reference list.
- When loaded, the incorrect address 0 will be updated by the latter Text record containing the symbol definition.

**Example:**

```

HCOPY  00100000107A
T00100009454F46000003000000
T00200F1514100948000000100C2810063000004800003C2012
T00201C022024
T002024190010000C100F0010030C100C4800000810094C0000F1001000
T00201302203D
T00203D1E041006001006E02039302043D8203928100630000054900F2C203A382043
T00205002205B
T00205B0710100C4C000005
T00201F022062
T002031022062
T00206218041006E0206130206550900FDC20612C100C3820654C0000
EO0200F

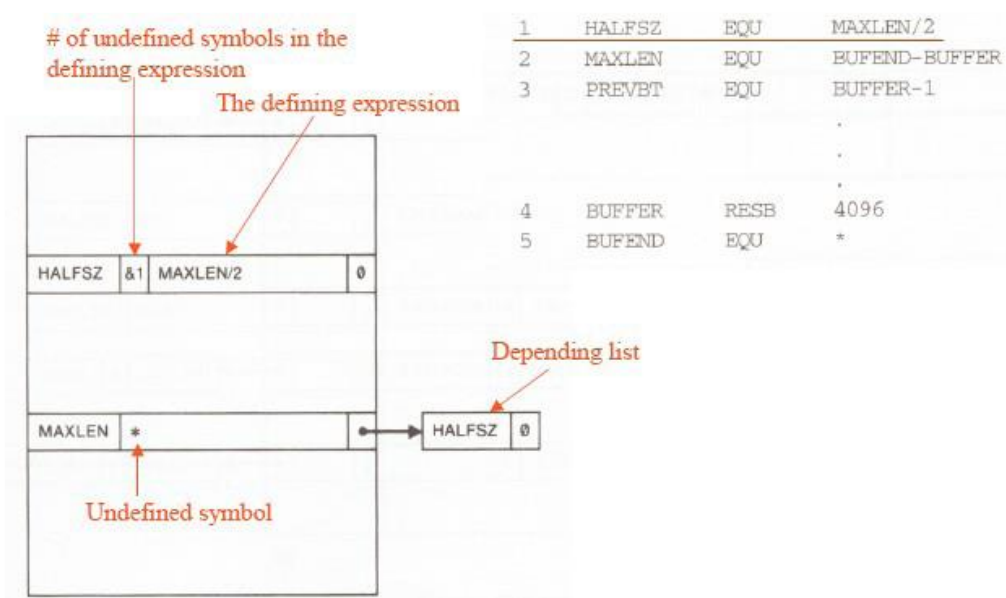
```

**3.2.1 MULTI PASS ASSEMBLER**

- For a two pass assembler, forward references in symbol definition are not allowed:  
 ALPHA EQU BETA  
 BETA EQU DELTA  
 DELTA RESW 1
- Here the problem is, the symbol BETA cannot be assigned a value when it is encountered during Pass 1 because DELTA has not yet been defined. Hence ALPHA cannot be evaluated during Pass 2. So that the symbol definition must be completed in pass 1.
- The general solution for this type of forward references is to use a multi-pass assembler that can make as many passes as are needed to process the definitions of symbols.
- It is not necessary for such an assembler to make more than 2 passes over the entire program.
- The portions of the program that involve forward references in symbol definition are saved during Pass 1. Additional passes through these stored definitions are made as the assembly progresses. This process is followed by a normal Pass 2.

**Implementation of Multipass Assembler**

- For a forward reference in symbol definition, we store in the SYMTAB:
  - The symbol name
  - The defining expression
  - The number of undefined symbols in the defining expression
- The undefined symbol (marked as \*) associated with a list of symbols depend on this undefined symbol.
- When a symbol is defined, we can recursively evaluate the symbol expressions depending on the newly defined symbol.
- The portions of the program that involve forward references in symbol definition are saved during Pass 1. Additional passes through these stored definitions are made as the assembly progresses. This process is followed by a normal Pass 2.

**Example:**

- Consider the symbol table entries from Pass 1 processing of the statement.  
HALFS2 EQU MAXLEN/2
- Since MAXLEN has not yet been defined, no value for HALFS2 can be computed. The defining expression for HALFS2 is stored in the symbol table in place of its value.
- The entry &1 indicates that 1 symbol in the defining expression undefined.
- SYMTAB simply contain a pointer to the defining expression.
- The symbol MAXLEN is also entered in the symbol table, with the flag \* identifying it as undefined. Associated with this entry is a list of the symbols whose values depend on MAXLEN.

If possible study the portion given below

