

MODULE-2

- **Loaders and Linkers: Basic Loader Functions,**
- **Machine Dependent Loader**
- **Features, Machine Independent Loader Features,**
- **Loader Design Options,**
- **Implementation Examples.**

Machine Independent Assembler Features

These are the features which do not depend on the architecture of the machine. These are:

- Literals
- Expressions
- Program blocks
- Control sections

Literals

A literal is defined with a prefix = followed by a specification of the literal value.

Example:

```
45 001A ENDFIL LDA =C"EOF" 032010
```

```
-
```

```
-
```

```
93 002D *      LTORG =C"EOF" 454F46
```

The example above shows a 3-byte operand whose value is a character string EOF. The object code for the instruction is also mentioned. It shows the relative displacement value of the location where this value is stored. In the example the value is at location (002D) and hence the displacement value is (010).

As another example the given statement below shows a 1-byte literal with the hexadecimal value '05'.

```
215 1062 WLOOP TD =X"05" E32011
```

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

```
55 0020 LDA #03 010003
```

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. The assembly listing of a program containing literals usually includes a listing of this literal pool, which shows the assigned addresses and the generated data values. In some cases it is placed at some other location in the object program. An assembler directive LTORG is used. Whenever the LTORG is encountered, it creates a literal pool that contains

all the literal operands used since the beginning of the program. The literal pool definition is done after LORG is encountered. It is better to place the literals close to the instructions.

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

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 LITAB and for the address value, it waits till it encounters LORG for literal definition. When Pass 1 encounters a LORG 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 LITAB for each literal encountered in the instruction and replaces it with its equivalent value as if these values are generated by BYTE or WORD. If a literal represents an address in the program, the assembler must generate a modification relocation for, if it all it gets affected due to relocation. The following figure shows the difference between the SYMTAB and LITAB.

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

LITAB

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

Symbol-Defining Statements:

EQU Statement:

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

Symbol EQU value

This statement defines the given symbol (i.e., entering in the SYMTAB) and assigning to it the value specified. The value can be a constant or an expression involving constants and any

othersymbol 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

```
+LDT  #4096
```

This loads the register T with immediate value 4096, this does not clearly show what exactly this value indicates. If a statement is included as:

```
MAXLEN EQU  4096                                and then
+LDT  #MAXLEN
```

Then it clearly indicates that the value of MAXLEN is some maximum length value. When the assembler encounters EQU statement, it enters the symbol MAXLEN along with its value in the symbol table. During LDT the assembler searches the SYMTAB for its entry and its equivalent value as the operand in the instruction. The object code generated is the same for both the options discussed, but is easier to understand. If the maximum length is changed from 4096 to 1024, it is difficult to change if it is mentioned as an immediate value wherever required in the instructions. We have to scan the whole program and make changes wherever 4096 is used. If we mention this value in the instruction through the symbol defined by EQU, we may not have to search the whole program but change only the value of MAXLENGTH in the EQU statement (only once).

ORG Statement:

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

```
ORG  value
```

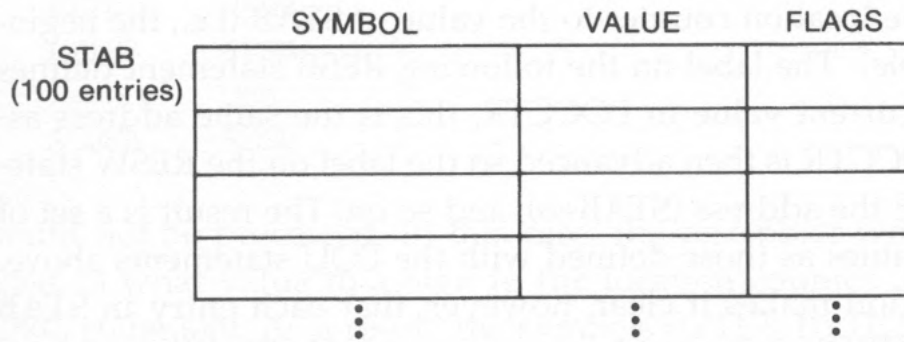
where value is a constant or an expression involving constants and previously defined symbols.

When this statement is encountered during assembly of a program, the assembler resets its location counter (LOCCTR) to the specified value. Since the values of symbols used as labels are taken from LOCCTR, the ORG statement will affect the values of all labels defined until the next ORG is encountered. ORG is used to control assignment storage in the object program. Sometimes altering the values may result in incorrect assembly.

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

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

The table looks like the one given below.



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

```
STAB      RESB      1100
```

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

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

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

```
LDA      VALUE, X
```

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

```
STAB      RESB      1100

          ORG      STAB

SYMBOL    RESB      6
VALUE     RESW      1
FLAG      RESB      2

          ORG      STAB+1100
```

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

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

```
ORG      ALPHA
```

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

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

EXPRESSIONS:

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

```
BUFFEND EQU *
```

Assigns a value to BUFFEND, which is the address of the next byte following the buffer area. Some values in the object program are relative to the beginning of the program and some are absolute (independent of the program location, like constants). 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 gives a value that does not depend on the location of the program and hence gives an absolute immaterial o the relocation of the program. The expression can have only absolute terms. Example:

```
MAXLEN EQU 1000
```

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

```
STAB EQU OPTAB + (BUFEND – BUFFER)
```

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

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

Program Blocks:

Program blocks allow the generated machine instructions and data to appear in the object program in a different order by Separating blocks for storing code, data, stack, and larger data block.

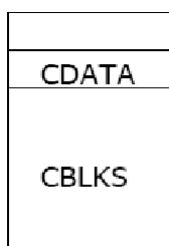
Assembler Directive USE:

USE [blockname]

At the beginning, statements are assumed to be part of the unnamed (default) block. If no USE statements are included, the entire program belongs to this single block. Each program block may actually contain several separate segments of the source program. Assemblers rearrange these segments to gather together the pieces of each block and assign address. Separate the program into blocks in a particular order. Large buffer area is moved to the end of the object program. Program readability is better if data areas are placed in the source program close to the statements that reference them.

In the example below three blocks are used :

- Default: executable instructions
- CDATA: all data areas that are less in length
- CBLKS: all data areas that consists of larger blocks of memory



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

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

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

Arranging code into program blocks:Pass 1

A separate location counter for each program block is maintained.

Save and restore LOCCTR when switching between blocks.

At the beginning of a block, LOCCTR is set to 0.

Assign each label an address relative to the start of the block.

Store the block name or number in the SYMTAB along with the assigned relative address of the label

Indicate the block length as the latest value of LOCCTR for each block at the end of Pass1

Assign to each block a starting address in the object program by concatenating the program blocks in a particular order

Pass 2

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

The starting address of this block

Control Sections:

A control section is a part of the program that maintains its identity after assembly; each control section can be loaded and relocated independently of the others. Different control sections are most often used for subroutines or other logical subdivisions. The programmer can assemble, load, and manipulate each of these control sections separately.

Because of this, there should be some means for linking control sections together. For example, instructions in one control section may refer to the data or instructions of other control sections. Since control sections are independently loaded and relocated, the assembler is unable to process these references in the usual way. Such references between different control sections are called external references.

The assembler generates the information about each of the external references that will allow the loader to perform the required linking. When a program is written using multiple control sections, the beginning of each of the control section is indicated by an assembler directive
assembler directive: CSECT

The syntax :

secname CSECT

separate location counter for each control section

Control sections differ from program blocks in that they are handled separately by the assembler. Symbols that are defined in one control section may not be used directly another control section; they must be identified as external reference for the loader to handle. The external references are indicated by two assembler directives:

EXTDEF (external Definition):

It is the statement in a control section, names symbols that are defined in this section but may be used by other control sections. Control section names do not need to be named in the EXTREF as they are automatically considered as external symbols.

EXTREF (external Reference):

It names symbols that are used in this section but are defined in some other control section.

The order in which these symbols are listed is not significant. The assembler must include proper information about the external references in the object program that will cause the loader to insert the proper value where they are required.

		Implicitly defined as an external symbol	
COPY	START	0	first control section
		EXTDEF	BUFFER,BUFEND,LENGTH
		EXTREF	RDREC,WRREC
FIRST CLOOP	STL	RETADR	SAVE RETURN ADDRESS
		+JSUB	RDREC
		LDA	LENGTH
		COMP	#0
		JEQ	ENDFIL
		+JSUB	WRREC
		J	CLOOP
ENDFIL	LDA	=C'EOF'	EXIT IF EOF FOUND
		STA	BUFFER
		LDA	#3
		STA	LENGTH
		+JSUB	WRREC
		J	@RETADR
RETADR	RESW	1	WRITE OUTPUT RECORD
LENGTH	RESW	1	LOOP
		LTORG	INSERT END OF FILE MARKER
BUFFER	RESB	4096	SET LENGTH = 3
BUFEND	EQU	*	WRITE EOF
MAXLEN	EQU	BUFFEND-BUFFER	RETURN TO CALLER
			LENGTH OF RECORD
			4096-BYTE BUFFER AREA

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

Implicitly defined as an external symbol
third control section

WRREC CSECT

```

SUBROUTINE TO WRITE RECORD FROM BUFFER

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

```

Object Code for the example program:

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

Case 1

0000	RDREC	CSECT		
	*		SUBROUTINE TO READ RECORD INTO BUFFER	
		EXTREF	BUFFER,LENGTH,BUFEND	
0000		CLEAR	X	B410
0002		CLEAR	A	B400
0004		CLEAR	S	B440
0006		LDT	MAXLEN	77201F
0009	RLOOP	TD	INPUT	E3201B
000C		JEQ	RLOOP	332FFA
000F		RD	INPUT	DB2015
0012		COMPR	A,S	A004
0014		JEQ	EXIT	332009
0017		+STCH	BUFFER,X	57900000
0018		TIXR	T	B850
001D		JLT	RLOOP	3B2FE9
0020	EXIT	+STX	LENGTH	13100000
0024		RSUB		4F0000
0027	INPUT	BYTE	X'F1'	F1
0028	MAXLEN	WORD	BUFEND-BUFFER	000000 Case 2
0000	WRREC	CSECT		
	*		SUBROUTINE TO WRITE RECORD FROM BUFFER	
		EXTREF	LENGTH,BUFFER	
0000		CLEAR	X	B410
0002		+LDT	LENGTH	77100000
0006	WLOOP	TD	=X'05'	E32012
0009		JEQ	WLOOP	332FFA
000C		+LDCH	BUFFER,X	53900000
0010		WD	=X'05'	DF2008
0013		TIXR	T	B850
0015		JLT	WLOOP	3B2FEE
0018		RSUB		4F0000
		END	FIRST	
001B	*	=X'05'		05

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

Define record (EXTDEF)

Col. 1 D

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

Col. 8-13 Relative address within this control section (hexadecimal)

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

Refer record (EXTREF)

Col. 1 R

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

Col. 8-73 Name of other external reference symbols

Modification record

Col. 1 M

Col. 2-7 Starting address of the field to be modified (hexadecimal)

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

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

A define record gives information about the external symbols that are defined in this control section, i.e., symbols named by EXTDEF.

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

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

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

In the case of Define, the record also indicates the relative address of each external symbol within the control section.

For EXTREF symbols, no address information is available. These symbols are simply named in the Refer record.

COPY

HCOPY 00000001033

DBUFFER000033BUFEND001033LENGTH00002D

RRDREC WRREC

T0000001D1720274B100000320232900003320074B1000003F2FEC0320160F2016

T00001D000100030F200A4B1000003E2000

T00003003454F46

M00000405+RDREC

M00001105+WRREC

M00002405+WRREC

E000000

```

RDREC
HRDREC 00000000002B
RBUFFERLENGTHBUFEND
T0000001DB410B400B44077201FE3201B332FFADB2015A00433200957900000B850
T00001D0E3B2FE9131000004F000QF1000000
M00001805+BUFFER
M00002105+LENGTH
M00002806+BUFEND
M00002806-BUFFER } BUFEND - BUFFER
E

WRREC
HWRREC 00000000001C
RLENGTHBUFFER
T0000001CB41077100000E3201232FFA53900000DF2008B8503B2FEE4F000005
M00000305+LENGTH
M00000D05+BUFFER
E

```

Assembler Design Options

One and Multi-Pass Assembler

- So far, we have presented the design and implementation of a two-pass assembler.
- Here, we will present the design and implementation of
 - One-pass assembler
 - If avoiding a second pass over the source program is necessary or desirable.
 - Multi-pass assembler
 - Allow forward references during symbol definition.

One-Pass Assembler

- The main problem is about forward reference.
- Eliminating forward reference to data items can be easily done.
 - Simply ask the programmer to define variables before using them.
- However, eliminating forward reference to instruction cannot be easily done.
 - Sometimes your program needs a forward jump.
 - Asking your program to use only backward jumps is too restrictive.

Line	Loc	Source statement			Object code
0	1000	COPY	START	1000	
1	1000	EOF	BYTE	C'EOF'	454F46
2	1003	THREE	WORD	3	000003
3	1006	ZERO	WORD	0	000000
4	1009	RETADR	RESW	1	
5	100C	LENGTH	RESW	1	
6	100F	BUFFER	RESB	4096	
9		.			
10	200F	FIRST	STL	RETADR	141009
15	2012	CLOOP	JSUB	RDREC	48203D
20	2015		LDA	LENGTH	00100C
25	2018				
110		.			
115		.			
120			SUBROUTINE TO READ RECORD		
121	2039	INPUT	BYTE	X'F1'	F
122	203A	MAXLEN	WORD	4096	0
124		.			
125	203D	RDREC	LDX	ZERO	0
130	2040		LDA	ZERO	0
135	2043	RLOOP	TD	INPUT	E
140	2046		JEQ	RLOOP	3
145	2049		RD	INPUT	D
150	204C		COMP	ZERO	2
155	204F		JEQ	EXIT	3
160	2052		STCH	BUFFER, X	5
165	2055		TIX	MAXLEN	2
170	2058		JLT	RLOOP	3
175	205B	EXIT	STX	LENGTH	1
180	205E		RSUB		4
185					

- There are two types of one-pass assembler:

– Produce object code directly in memory for immediate execution

- No loader is needed
- Load-and-go for program development and testing
- Good for computing center where most students reassemble their programs each time.
- Can save time for scanning the source code again

– Produce the usual kind of object program for later execution

Internal Implementation

- The assembler generate object code instructions as it scans the source program.
- If an instruction operand is a symbol that has not yet been defined, the operand address is omitted when the instruction is assembled.
- The symbol used as an operand is entered into the symbol table.
- This entry is flagged to indicate that the symbol is undefined yet.
- The address of the operand field of the instruction that refers to the undefined symbol is added to a list of forward references associated with the symbol table entry.
- When the definition of the symbol is encountered, the forward reference list for that symbol is scanned, and the proper address is inserted into any instruction previously generated.

Memory address	Contents				Symbol	V
1000	454F4600	00030000	00xxxxxx	xxxxxx	LENGTH	1
1010	xxxxxx	xxxxxx	xxxxxx	xxxxxx	RDREC	*
•					THREE	1
•					ZERO	1
•					WRREC	*
2000	xxxxxx	xxxxxx	xxxxxx	xxxxxx14	EOF	1
2010	100948--	--00100C	28100630	--48--	ENDFIL	*
2020	--3C2012				RETADR	1
•					BUFFER	1
•					CLOOP	2
•					FIRST	2

-
- | Memory address | Contents | | | | Symbol | Value |
|----------------|----------|----------|----------|----------|--------|-------|
| 1000 | 454F4600 | 00030000 | 00xxxxxx | xxxxxx | LENGTH | 100C |
| 1010 | xxxxxx | xxxxxx | xxxxxx | xxxxxx | RDREC | 203D |
| ... | ... | ... | ... | ... | THREE | 1003 |
| ... | ... | ... | ... | ... | ZERO | 1006 |
| 2000 | xxxxxx | xxxxxx | xxxxxx | xxxxxx | WRREC | * |
| 2010 | 10094820 | 3D00100C | 28100630 | 202448-- | EOF | 1000 |
| 2020 | 5C2012 | 0010000C | 100F0010 | 050C100C | ENDFIL | 2024 |
| 2030 | 48----08 | 10094C00 | 00F10010 | 00041006 | RETADR | 1009 |
| 2040 | 001006E0 | 20393020 | 43D82039 | 28100630 | BUFFER | 100F |
| 2050 | ----5490 | 0F | | | CLOOP | 2012 |
| ... | ... | ... | ... | ... | FIRST | 200F |
| ... | ... | ... | ... | ... | MAXLEN | 203A |
| | | | | | INPUT | 2039 |
| | | | | | EXIT | * |
| | | | | | RLOOP | 2043 |
- Between scanning line 40 and 160:
 - On line 45, when the symbol ENDFIL is defined, the assembler places its value in the SYMTAB entry.
 - The assembler then inserts this value into the instruction operand field (at address 201C).
 - From this point on, any references to ENDFIL would not be forward references and would not be entered into a list.
 - At the end of the processing of the program, any SYMTAB entries that are still marked with * indicate undefined symbols.
 - These should be flagged by the assembler as errors.

Multi-Pass Assembler

- If we use a two-pass assembler, the following symbol definition cannot be allowed.


```

ALPHA EQU BETA

BETA EQU DELTA

DELTA RESW 1
      
```
- This is because ALPHA and BETA cannot be defined in pass 1. Actually, if we allow multi-pass processing, DELTA is defined in pass 1, BETA is defined in pass 2, and ALPHA is defined in pass 3, and the above definitions can be allowed.
- This is the motivation for using a multi-pass assembler.

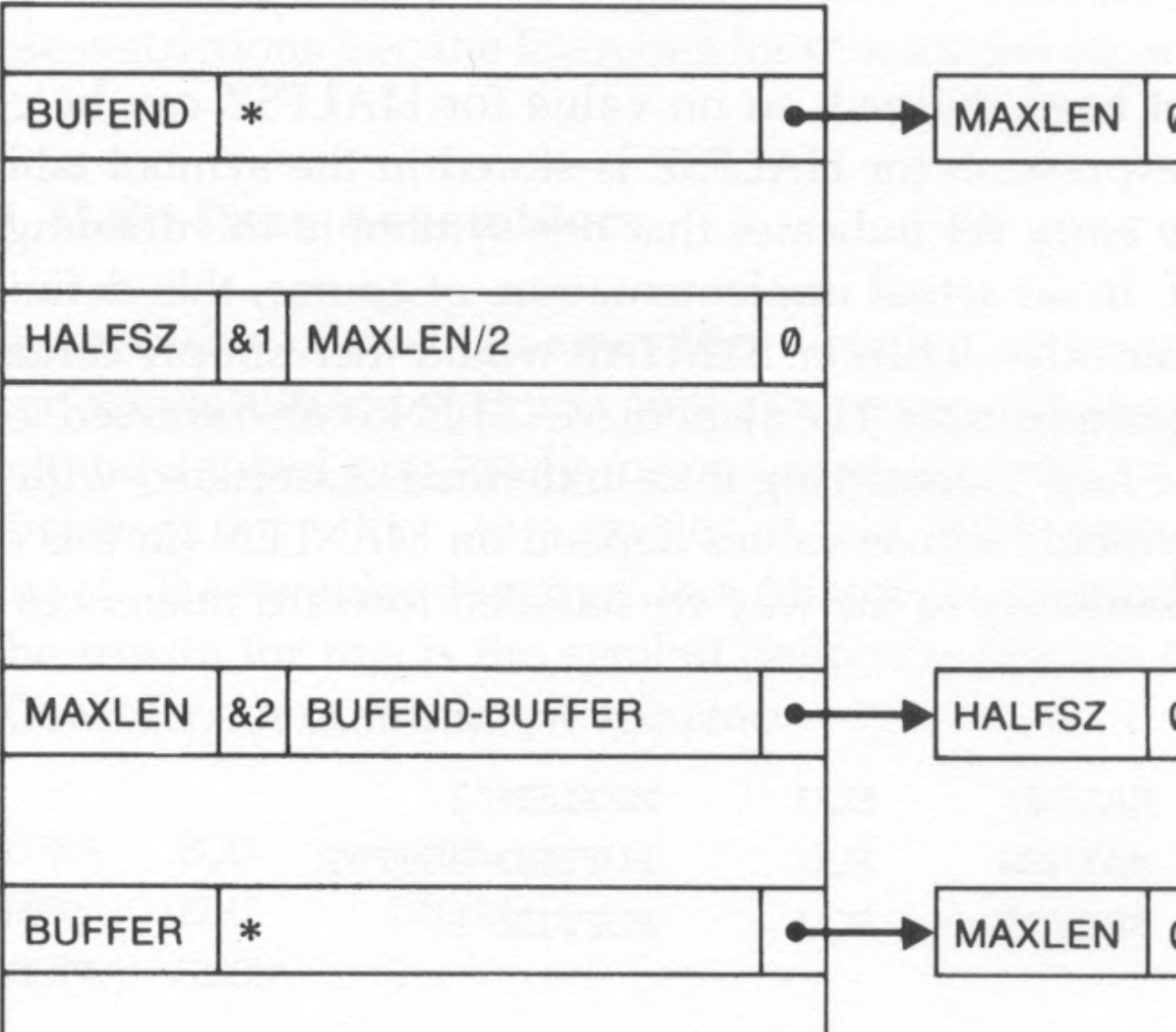
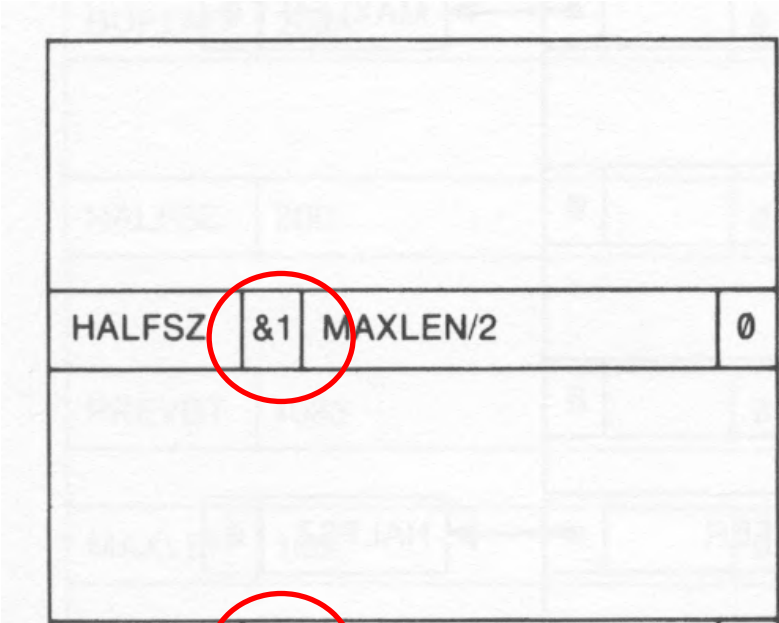
- It is unnecessary for a multi-pass assembler to make more than two passes over the entire program.
- Instead, only the parts of the program involving forward references need to be processed in multiple passes.
- The method presented here can be used to process any kind of forward references.

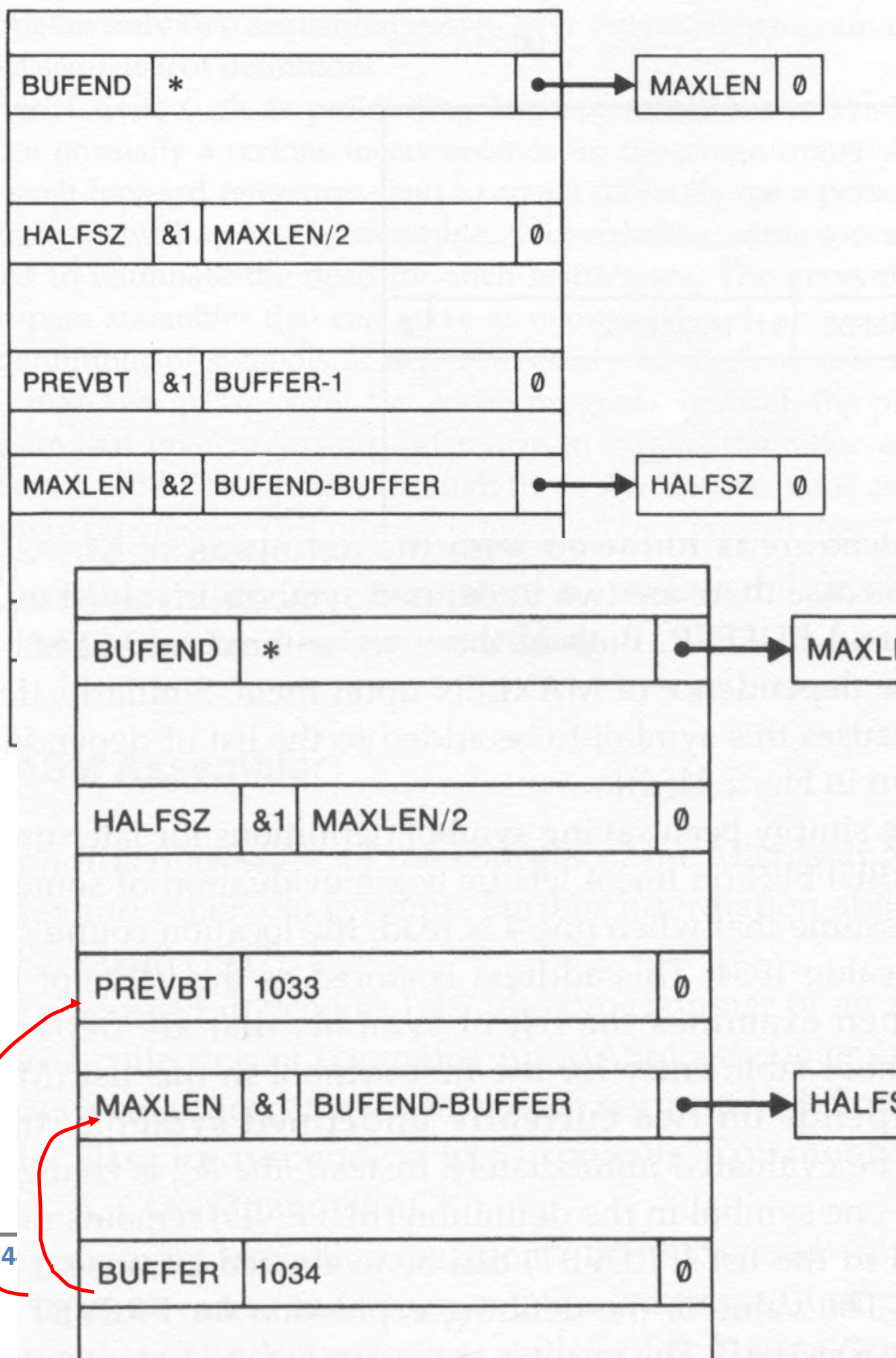
Multi-Pass Assembler Implementation

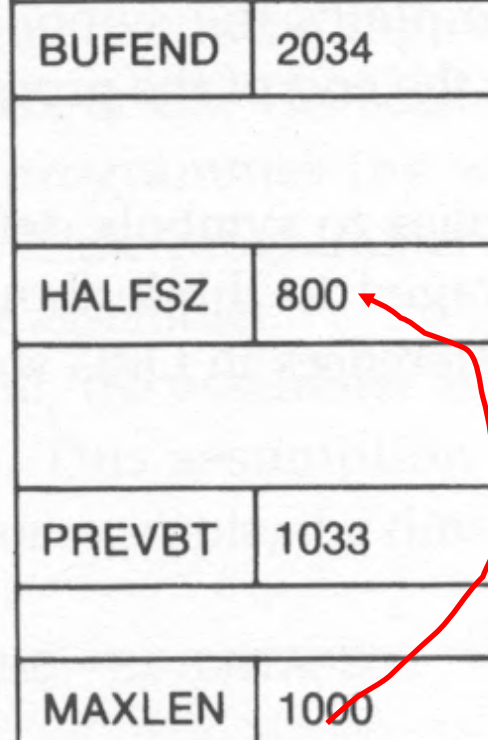
Steps:

- Use a symbol table to store symbols that are not totally defined yet.
- For a undefined symbol, in its entry,
 - We store the names and the number of undefined symbols which contribute to the calculation of its value.
 - We also keep a list of symbols whose values depend on the defined value of this symbol.
- When a symbol becomes defined, we use its value to reevaluate the values of all of the symbols that are kept in this list.
- The above step is performed recursively.

1	HALFSZ	EQU	MAXLEN/2
2	MAXLEN	EQU	BUFEND-B
3	PREVBT	EQU	BUFFER-1
			.
			.
			.
4	BUFFER	RESB	4096
5	BUFEND	EQU	*







BUFEND	2034	0
HALFSZ	800	0
PREVBT	1033	0
MAXLEN	1000	0
BUFFER	1034	0