

Linkers



By:

~~Bhargavi Goswami~~

~~Assistant Professor~~

~~Sunshine Group of Institutions~~

~~Rajkot, Gujarat, India.~~

~~Email: bhargavigoswami@gmail.com~~

~~Mob: +918140099018~~

Steps for Execution:

Performed By:

Translator of L:

1. Translation of program

2. Linking of program with other programs needed for its execution.

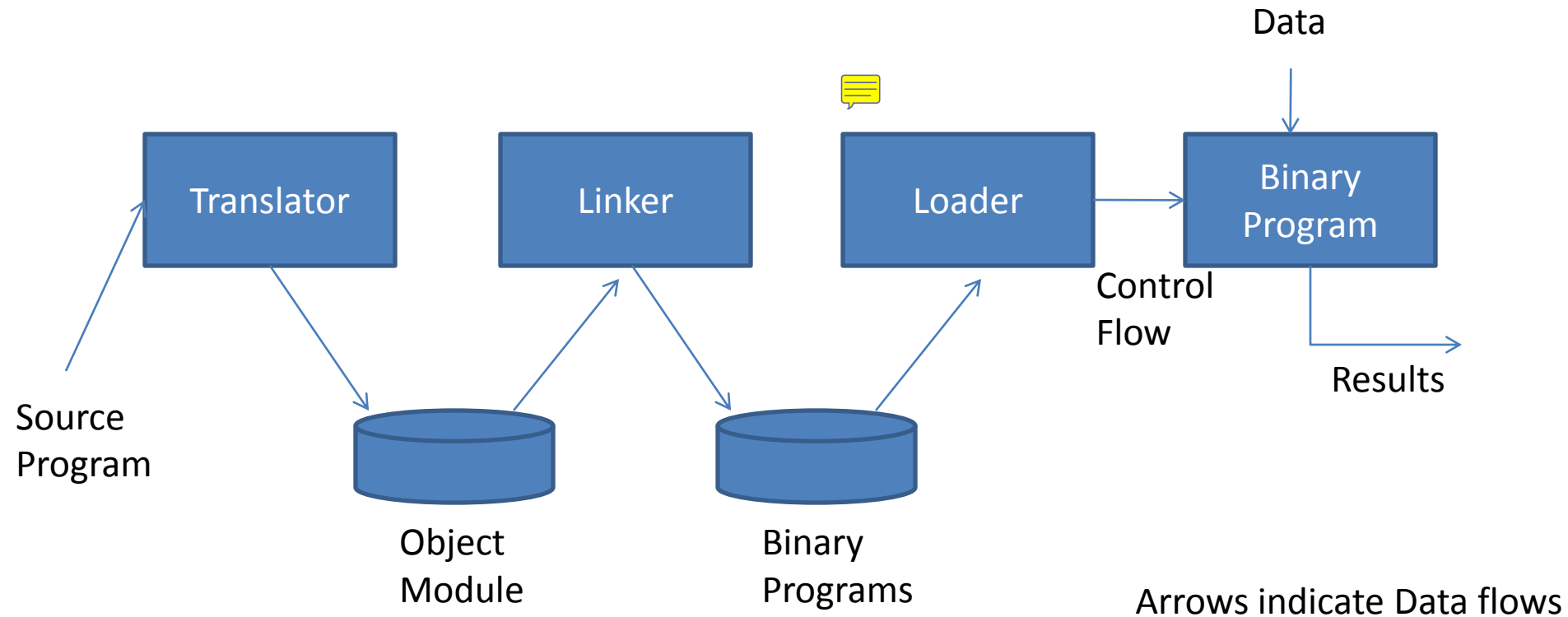
Linker:

3. Relocation of the program to execute from the specific memory area allocated to it.

Loader:

4. Loading of the program in the memory for the purpose of execution.

Schematic of Program Execution:



A Compiler translates lines of code from the programming language into machine language. A Linker creates a link between two programs. A Loader loads the program into memory in the main database, program, etc.

Few Terminologies:

- Translation Time (or Translated) Address: Address assigned by the translator.
- Linked Address: Address assigned by the linker.
- Load Time (or Load) Address: Address assigned by the loader.
- Translated Origin: Address of the origin assumed by the translator. This is the address specified by the programmer in an ORIGIN statement.
- Linked Origin: Address of the origin assigned by the linker while producing a binary program.
- Load Origin: Address of the origin assigned by the loader while loading the program for execution.

Origin may need to be changed:

- The linked and load origins may differ from translated origin.
- Reason?
 - Same set of translated addresses may have been used by different object modules of the program. This results to conflict in memory allocation.
 - OS may require that a program should execute from specific area of memory. This may require change in its origin, thus changing execution start address and symbol addresses.
- Thus, require changes in origin by linker and loader.

Example:

	Statement	Address	Code
	START 500		
	ENTRY TOTAL		
	EXTRN MAX, ALPHA		
	READ A	500)	+ 09 0 540
LOOP		501)	
	.		
	.		
	MOVER AREG, ALPHA	518)	+ 04 1 000
	BC ANY, MAX	519)	+ 06 6 000
	.		
	.		
	BC LT, LOOP	538)	+ 06 1 501
	STOP	539)	+ 00 0 000
A	DS 1	540)	
TOTAL	DS 1	541)	
	END		

The translated
origin

The Translated Time
Address of LOOP

RELOCATION & LINKING CONCEPTS

- TOPIC LIST:
 - (a) Program Relocation
 - Performing Relocation
 - (b) Linking
 - EXTERN & ENTRY statement
 - Resolving external references
 - Binary Programs
 - (c) Object Module

Program Relocation:

Address Sensitive Program:

- AA – Set of Absolute Addresses, may be instruction address or data address.
- $AA \neq \phi$; means instruction or data occupy memory words with specific address.
- Such a program is called address sensitive program which contains:
 - 1. Address Sensitive Instruction: an instruction which contains address $a_i \in AA$.
 - 2. Address Constant: a data word which contains an address $a_i \in AA$.
- Important: Address Sensitive Program P can execute correctly only if the start address of the memory area allocated to it is the same as its translated origin.

Start Address == Translation Address

- Thus, to execute correctly from any memory area, address used in each address sensitive instruction of P must be 'corrected'.

- Def: Program Relocation: Is the process of modifying the address used in the address sensitive instructions such that the program can execute correctly from the designated area of memory.
- Linker performs relocation if
 - Linked Origin \neq Translated Origin
- Loader performs relocation if
 - Load Origin \neq Linked Origin
- In general: Linker always performs relocation, whereas some loaders do not.
- Absolute loaders do not perform relocation, then
 - Load Origin = Linked Origin
 - Thus, Load Origin and Linked Origin are used interchangeably.

Performing Relocation:

- Relocation Factor (RF) of P is defined as

$$[\text{relocation_factor}_p = l_origin_p - t_origin_p]$$


where, RF is +ve or -ve or 0.
- Symbol is operand

$$[t_{symb} = t_origin_p + d_{symb}]$$

where, d_{symb} = offset
 t_{symb} = translation time address
 l_{symb} = link time address
 t_origin_p = translation origin

$$[l_{symb} = l_origin_p + d_{symb}]$$

where, l_origin_p = linked origin,
- $$L_{symb} = t_{origin} + \text{relocation_factor}_p + d_{symb}$$

$$= t_{symb} + \text{relocation_factor}_p$$


IRR: Instruction Requiring Relocation

- It's a set of instructions that perform relocation in program p.
- Steps:
 - Calculate Relocation Factor(RF)
 - Add it to Translation Time Address(es) for every instruction which is member of IIR.
- Eg: relocation factor = $900 - 500 = 400$

IRR contains translation addresses 540 & 538
(instruction : read A)

Address is changed to $540 + 400 = 940$ and
 $538 + 400 = 938$.

Linking:

- AP is an Application Program consisting of a Set of Program units $SP = \{ P_i \}$.
- A program unit P_i interacts with another program unit P_j using instructions & addresses of P_j .
- For this it must have:
 - Public Definition: a symbol `pub_symb` defined in program unit which may be referenced in other program.
 - External Reference: a reference to a symbol `ext_symb` which is not defined in program unit containing the reference.
- Who will handle the these two things?

EXTRN & ENTRY statements:

- The ENTRY statement lists the public definition of program unit.
 - i.e it list those symbols defined in program unit which may be referenced in other program units.
- The EXTRN statement list the symbols to which external references are made in the program unit.
- Eg: see the pg 223.
 - TOTAL is ENTRY statement.(public defination)
 - MAX and ALPHA are EXTRN statements (external reference)
 - Assembler don't know address of EXTRN symb and so it puts 0's address field of instruction wherever these symb are found.
 - What if we don't refer these var with EXTRN stmts?
 - Assembler gives errors.
 - Thus, requirement arises for resolving external references.

Resolving External Reference:

- Before AP (application program) executes, every external reference should be bound to correct link time address.
- Who will do this binding?
 - Here comes Linker into the picture.
- **Def: Linking**: Linking is the process of binding an external reference to correct link time address.
- External reference is said to be unresolved until linking is performed and resolved once linking is completed.
- Linking is performed for the eg defined in pg. no 223 and the Q program unit defined as follows:

Example: Linked with Program Unit Q

	Statement	Address	Code
	START 200		
	ENTRY ALPHA		
	- -		
ALPHA	DS 25	231)	+ 00 0 025
	END		

ALPHA is external reference in that example which is linked with above given unit.

Binary Programs:

- Is a machine language program comprising set of program units SP such that for all $P_i \in SP$,
 - 1. P_i relocated at link origin.
 - 2. Linking is performed for each external reference in P_i .
- Linker Command:
linker <link origin> <object module names>
[, <execution start address>]
- Linker converts object module into-> the set of program units (SP) into-> a binary program.
- If link address = load address, loader simply loads binary program into appropriate memory area for execution.

Object Module:

- Object Module(OM) contains all the information necessary to relocate and link the program with other programs.
- OM consist of four components:
 - **1. Header**: Contains translated origin, size and execution start address of P.
 - **2. Program**: Contains machine language program corresponding P.
 - **3. Relocation Table**: (RELOCTAB) describing IRRp (Instruction Requiring Relocation). Each entry contains field: 'Translated Address' - of address sensitive instruction.
 - **4. Linking Table**: (LINKTAB) contains information concerning public definition and external reference. Has three fields:
 - Symbol: symbolic name
 - Type: PD/EXT i.e **public definition or external reference**
 - Translated Address: If PD, address of first memory word allocated. If EXT, address of memory word containing reference of symbol.

Example:

- For previously given assembly program, object module contains following information:
- 1. Translated Origin: 500, Size:42, Exe. Strt. Add=500.
- 2. Machine Language Instructions at pg 223.
- 3. Relocation table:
- 4. Linking table:
- LOOP is not here as it is not PD.

500
538

ALPHA	EXT	518
MAX	EXT	519
A	PD	540



DESIGN OF LINKER:

- What influence relocation requirement of a program?
 - Ans: Addressing Structure of Computer System.
- How to reduce relocation requirement of a program?
 - Ans: By using segmented addressing structure.
- JUMP avoids use of absolute address, hence instruction is no more address sensitive.
- Thus, no relocation is needed.
- Effective Operand Address would be calculated considering starting address as 2000 would be
$$\langle CS \rangle + 0196 = 2196 \text{ (which is corrected address)}$$
- Lets take example to understand it more clearly.

Sr. No	Statement			Offset
0001	DATA_HERE	SEGMENT		
0002	ABC	DW	25	0000
0003	B	DW	?	0002
.	.			
.	.			
0012	SAMPLE	SEGMENT		
0013		SEGMENT	CS:SAMPLE,	
			DS:DATA_HERE	
0014		MOV	AX, DATA_HERE	0000
0015		MOV	DS, AX	0003
0016		JMP	A	0005
0017		MOV	AL,B	0008
.	.			
0027	A	MOV	AX,BX	0196
.	.			
0043	SAMPLE	ENDS		
0044		END		

Example:

- Code written in assembly language for Intel 8088.
 - **ASSUME** statements declares segment register CS and DS for memory addressing.
 - So, all memory addressing is performed using suitable displacement of their contents.
 - Translation time address of A is 0196.
 - In statement 16, reference of A due to JMP statement makes displacement of 196 from the content of CS register.
 - Hence avoids usage of absolute addressing. Thus, instruction is not address sensitive.
- Displacement -> avoids usage of absolute address -> not address sensitive instruction.**
- As DS is loaded with execution time address of DATA_HERE, reference to B would be automatically relocated to correct address.
 - Thus, Use of segment register reduces relocation requirement but doesn't eliminate it.
 - Inter Segment Calls & Jumps are handled in similar way.
 - Relocation is more involved in case of intra segment jumps assembled in FAR format.
 - Linker computes both:
 - Segment Base Address
 - Offset of External Symbol
 - Thus, no reduction in linking requirements.
-

Relocation Algorithm:

1. `program_linked_origin := <link origin>` from Linker command;
2. For each object module
 - (a) `t_origin := translated origin of object module;`
`OM_size := size of object module;`
 - (b) `relocation_factor := program_linked_origin – t_origin;`
 - (c) Read the machine language program in `work_area`;
 - (d) Read RELOCTAB of object module.
 - (e) For each entry in RELOCTAB
 - (i) `Translated_addr := address in RELOCTAB entry;`
 - (ii) `address_in_work_area := address of work_area +`
`translated_address – t_origin;`
 - (iii) Add `relocation_factor` to the operand address in the word with the address '`address_in_work_area`'.
 - (f) `Program_linked_origin := program_linked_origin + OM_Size.`

Example:

- Let address of work_area = 300.
- While relocating OM, relocation factor = 400.
- For the first RELOCTAB entry,
 $\text{address_in_work_area} = 300 + 500 - 500 = 300$.
- This word contains the instruction for READ A.
- It is reallocated by adding 400 to operand address in it.
- For the second RELOCTAB entry,
 $\text{address_in_work_area} = 300 + 538 - 500 = 338$.
- The instruction in this word is similarly relocated by adding 400 to the operand address in it.

Linking Requirements:

- In Fortran all program units are translated separately.
- Hence, all sub program calls and common variable reference require linking.
- Programs are nested in main program.
- Procedure reference do not require linking, they can be handled using relocation.
- Build in functions require linking.
- Linker processes all object modules being linked & builds a table of all public definition & load time address.
- Name Table (NTAB)
 - (a) Symbol: Symbol Name for external reference or object module.
 - (b) Linked Address: For public definition contains link_address. For object module contains link_origin.
- Most information in NTAB is derived from LINKTAB entries with type=PD.

Algorithm: Program Linking

1. `program_linked_origin` := <link origin> from linker command.
2. For each object module
 - (a) `t_origin` := translated origin of object module. `OM_size` := size of the object module;
 - (b) `Relocation_factor` := `program_linked_origin` – `t_origin`.
 - (c) Read the machine language program in `work_area`.
 - (d) Read LINKTAB of the object module.
 - (e) For each LINKTAB entry with type = PD
 - `name`:= symbol;
 - `linked_address` := `translated_address` + `relocation_factor`;
 - `enter(name,linked_address)` in NTAB.
 - (f) Enter (object module name, `program_linked_origin`) in NTAB.
 - (g) `program_linked_origin` := `program_linked_origin` + `OM_size`;

3. for each object module

(a) $t_origin :=$ translated origin of the object module.

(b) For each LINKTAB entry with type=EXT

(i) $address_in_work_area :=$

$address_of_work_area +$
 $program_linked_origin -$

$\langle link_origin \rangle + translated\ address -$
 $t_origin;$

(ii) Search symbol in NTAB and copy its linked address. Add the linked address to the operand address in the word with the address $address_in_work_area$.

Example:

- While linking program P and program Q with `linked_origin=900`, NTAB contains following information:
- `Work_area = 300`.
- When LINKTAB of alpha is processed during linking, `address_in_work_area := 300 + 900 - 900 + 518 - 500`. i.e 318.
- Hence linked address of ALPHA is 973, is copied from NTAB entry of ALPHA and added to word in address 318.

Symbol	Linked Address
P	900
A	940
Q	942
ALPHA	973

SELF RELOCATING PROGRAMS

- Types:
 - Non Re-locatable Programs
 - Re-locatable Programs
 - Self Re-locating Programs
- 1. Non-Re-locatable Programs:
 - It cannot be executed in any memory area other than area starting on its translated origin.
 - Has address sensitive programs.
 - Lack information concerning address sensitive instruction
 - Eg. Hand coded machine language program.

- 2. Re-locatable Programs:
 - It can be processed to relocate it into desired data area of memory.
 - Eg: Object Module.
- 3. Self Relocating Programs:
 - It can perform relocation of its own address sensitive instruction.
 - Consist of:
 - A table of Information (for address sensitive prg)
 - Relocating logic
 - Self re-locating program can execute in any area of memory.
 - As load address is different for each execution, good for time sharing systems.

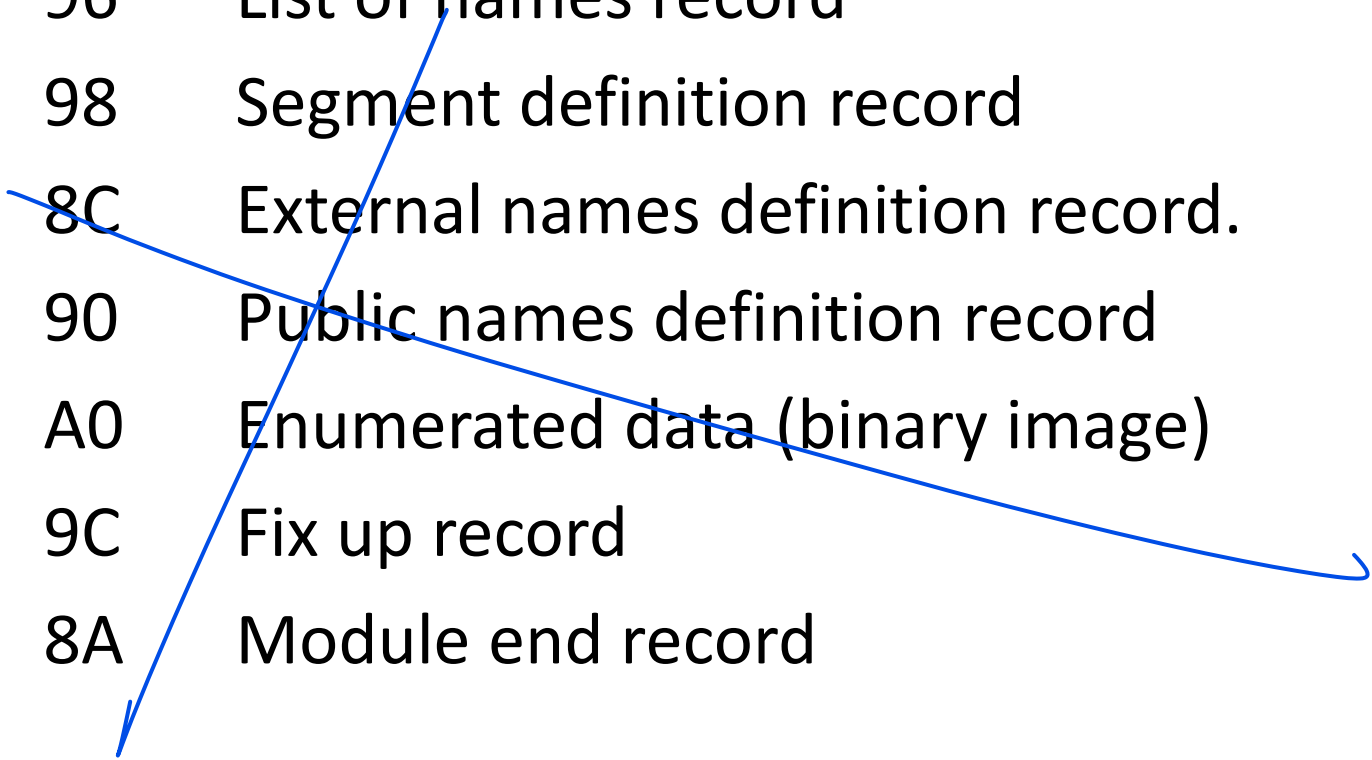
A LINKER FOR MS DOS

- Let us discuss the design of a linker for Intel 8088/80/x86 processors which resembles LINK of MS DOS.
- Object Module Format:
 - An Intel 8088 object module is a sequence of object records.
 - Each object record describes specific aspect of program in object module.
 - There are 5 categories comprising total 14 types of object records.
 - Five Categories:
 - 1. Binary image(ie code generated by translator)
 - 2. External reference
 - 3. Public definition
 - 4. Debugging information (e.g. Line numbers)
 - 5. Miscellaneous information (e.g. comments)
 - Our syllabus includes only first 3 categories comprising of total 8 object record types.
 - Each object record contains variable length information and may refer to contents of previous object records.

Object Records of Intel 8088

(order is important)

THEADR	80	Translator header record
LNAMES	96	List of names record
SEGDEF	98	Segment definition record
EXTDEF	8C	External names definition record.
PUBDEF	90	Public names definition record
LEDATA	A0	Enumerated data (binary image)
FIXUPP	9C	Fix up record
MODEND	8A	Module end record



Object Record Formats:

- THEADR record:
 - Translator Header Record
 - Typically derived by the translator from the source file name.
 - This file name is used by the linker to report errors.
 - An assembly programmer can specify the module name in the NAME directive.

THEADR:

80H	Length	T-module name	Check Sum
-----	--------	---------------	-----------

96H	Length	Name List	Check Sum
-----	--------	-----------	-----------

- LNAMEs:
 - record lists the names for use by SEGDEF records.
- SEGDEF:
 - Designates a segment name
 - Uses index into in to the list
 - Attribute field indicates whether segment is re-locatable or absolute.
 - Also indicates the manner in which it is combined with other segments.
 - Also indicates alignment requirement of its base address.
 - Attribute field also contains origin specification for absolute segment.
 - Stack segments with same names are concatenated and common segments with same names are overlapped.


98H	Length	Attributes (1-4)	Segment Length (2)	Name Index (1)	Check Sum
-----	--------	---------------------	-----------------------	----------------	-----------

EXTDEF record

8CH	Length	External Reference List	Check Sum
-----	--------	----------------------------	-----------

PUBDEF record

90H	Length	base (2-4)	Name	offset (2)	...	Check Sum
-----	--------	------------	------	------------	-----	-----------



- **EXTDEF:**
 - The EXTDEF record contains a list of external references used by this module.
- **PUBDEF:**
 - Contains list of public names declared.
 - Base specification identifies the segment.
 - (name, offset) pair defines one public name.

FIXUPP record

9CH	Length	Locat (1)	Fix dat (2)	Frame datam (1)	Target datum(1)	Target offset(2)	...	Check Sum
-----	--------	-----------	-------------	-----------------	-----------------	------------------	-----	-----------

- **FIXUPP:**
 - Designates external symbol name
 - Uses index to the list
- Locate contains offset of fix up location in previous LEDATA.
- Frame datum refers to SEGDEF record.
- Target datum and target offset specify relocation and linking information.
- Target Datum contains segment index or an external index.
- Target offset contains offset from name indicated in target datum.
- Fix data field indicates manner in which target datum and fix datum fields are to be interpreted.

FIXUPP codes

Loc Code	Meaning:
0	Low order byte to be fixed
1	Offset is to be fixed
2	Segment is to be fixed
3	Pointer (segment offset) to be fixed.

Fixdat field codes

Loc Code	Meaning:
0	Segment index & displacement
2	External index & target displcmt
4	Segment index (without offset)
6	External index (without offset)

MODEND record

8AH	Length	Type (1)	Start addr (5)	Check sum
-----	--------	----------	-------------------	--------------

- ~~MODEND record:~~
 - Signifies the end of module.
 - Type field indicates whether it is the main program.
 - Has 2 components: (a) segment, (b) offset
- LEDATA records:
 - Contains binary image of code generated by language translator.
 - Segment index identifies the segment to which the code belongs.
 - Offset specifies the location of the code within the segment.

LEDATA record

AOH	Length	Segment Index (1-2)	Data Offset (2)	data	Check sum
-----	--------	------------------------	--------------------	------	--------------

Algorithm 7.3 (**First Pass of LINKER**)

1. program_linked_origin := <load origin>;
2. Repeat step 3 for each object module to be linked.
3. Select an object module and process its object records.
 - (a) If an LNAMEs record, enter the names in NAMELIST.
 - (b) If a SEGDEF record
 - (i) i := name index; segment_name := NAMELIST[i];
segment_addr := start address in attributes;
 - (ii) If an absolute segment, enter (segment_name, segment_addr) in NTAB.
 - (iii) If the segment is re-locatable and cannot be combined with other segments
 - Align the address containing in program_linked_origin on the next word or paragraph as indicated in attribute list.
 - Enter (segment_name, program_linked_origin) in NTAB.
 - program_linked_origin := program load origin + segment length;

(c) For each PUBDEF record

(i) $i := \text{base}$; $\text{segment_name} := \text{NAMELIST}[i]$;

(ii) $\text{segment_addr} := \text{load address of}$
 $\text{segment_name in NTAB}$;

(iii) $\text{sym_addr} := \text{segment_addr} + \text{offset}$;

(iv) Enter (symbol, sym_addr) in NTAB.

-
- In first pass, linker only processes the object records relevant for building NTAB.
 - Second pass performs relocation and linking.

Algorithm 7.4 (**Second Pass of LINKER**)

1. List_of_object_module := object modules named in LINKER command;
2. Repeat step 3 until list_of_object_modules is empty.
3. Select an object module and process its object records.
 - (a) if an LNAMEs record
Enter the names in NAMELIST.
 - (b) if a SEGDEF record
i := name index; segment_name := NAMELIST[i];
 - (c) if an EXTDEF record
 - (i) external_name := name from EXTDEF record;
 - (ii) if external_name is not found in NTAB, then
 - Locate object module in library which contains external_name as a segment or public definition.
 - Add name of object module to list_of_object_module.
 - Perform first pass of LINKER, for new object module.
 - (iii) Enter (external_name, load address from NTAB) in EXTTAB.

(d) if an LEDATA record

- (i) $i := \text{segment index}$; $d := \text{data offset}$;
- (ii) $\text{program_load_origin} := \text{SEGTAB}[i].\text{load address}$;
- (iii) $\text{address_in_work_area} := \text{address of work_area} + \text{program_load_origin} - \langle \text{load origin} \rangle + d$;
- (iv) move data from LEDATA into the memory area starting at the address $\text{address_in_work_area}$.

(e) if a FIXUPP record, for each FIXUPP specification

- (i) $f := \text{offset from locat field}$;
- (ii) $\text{fix_up_address} := \text{address_in_work_area} + f$;
- (iii) Perform required fix up using a load address from SEGTAB or EXTTAB and the value of code in locat and fix dat.

~~(f) if MODEND record~~

~~if start address is specified, compute the corresponding load address and record it in the executable file being generated.~~

Linking for OVERLAYS:

- **Def: (Overlay)** An overlay is a part of a program (or software package) which has the same load origin as some other parts of the program.
- Overlays are used to reduce the main memory requirement of a program.
- Overlay structured programs:
 - We refer to program containing overlays as an overlay structured program.
 - Such program consist of:
 1. A permanently resident portion, called the root.
 2. A set of overlays.
- Execution of overlay structured program:
 - 1. root is loaded in the memory
 - 2. other overlays are loaded as and when needed.
 - 3. loading of an overlay overwrites a previously loaded overlay with the same load origin.
- This reduces the memory requirement of the program.
- Hence, facilitate execution of those programs who's size exceeds memory size.
- How to design overlay structure?
 - By identifying mutually exclusive module.
 - Modules that do not call each other.
 - These modules do not need to reside simultaneously in the memory.
 - Hence keep those modules in different overlays with same load origin.

- See example 7.15 on pg. no. 245.
- See Overlay Tree on pg. no. 246.
- See Example 7.16 which is MS-DOS LINK command to implement overlay structure.
- See Example 7.17 is the IBM mainframe linker command for overlay structures.
- Execution of an overlay structured program:
 - **Overlay manager module** is included in executable file which is responsible for overlay load as and when required.
 - **Interrupt producing instruction** replaces all cross overlay boundary calls.
- Changes in Linker Algorithm?
 - Assignment of load address to segments after execution of root.
 - Mutually exclusive modules are assigned same `program_load_origin`.
 - Also algorithm has to identify inter-overlay call and determine destination overlay.

Loaders

- Two Types:
 1. Absolute Loader:
 - Can load only programs with load origin = linked origin.
 - This is inconvenient when load address differ from the one specified for execution.
 2. Relocating loader:
 - Performs relocation while loading a program for execution.
 - Permits program to be executed in different parts of memory.
- MS DOS support 2 object program forms:
 1. .COM:
 - Contains non re-locatable object program.
 - Absolute loaders are invoked.
 2. .EXE:
 - Contains re-locatable object program.
 - Relocating loaders are invoked.
- At the end of loading, execution starts.

7th Chapter Ends Here

THANK YOU