5. LINKER

Execution of a program written in a language L involves following steps:

1. Translation of the program
2. Linking of the program with other program needed for its execution
3. Relocation of the program to execute from the specific memory area allocated to it
4. Loading of the program in the memory for the purpose of execution

These steps are performed by different language processors.

- Step1 by a translator for language L
- Step 2 and 3 by a linker
- Step 4 by a loader

Figure 1 contains a schematic showing step 1-4 in the execution of a program.

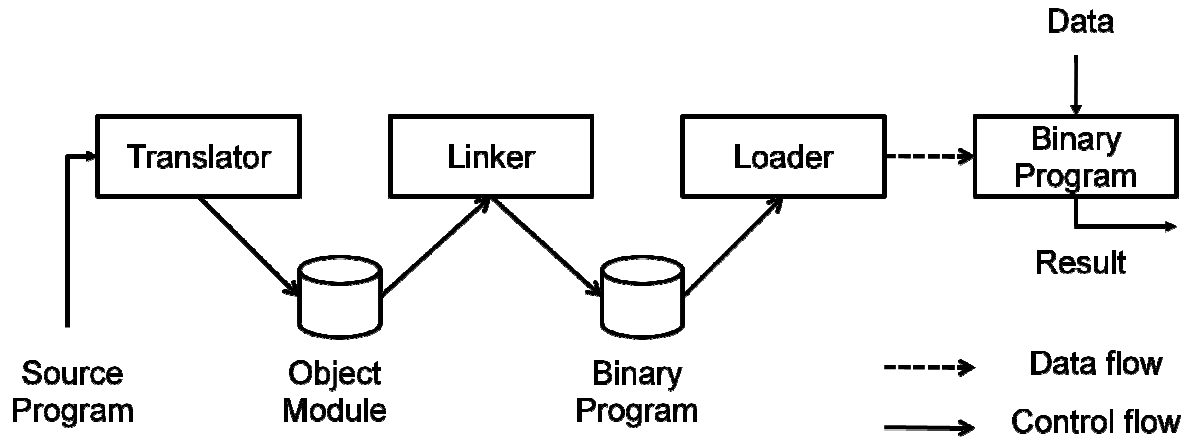


Figure 1: Schematic of program execution

The translator outputs a program called object module for the program. The linker processes a set of object modules to produce a ready to execute program form called binary program. The loader loads this program into the memory for the purpose of execution. The object modules and ready to execute program forms can be stored in the form of files for repeated use.

Translated, linked and load time addresses

While compiling a program P, a translator is given an origin specification for P. This is called translated origin of P. (in assembly program, the programmer can specify the origin in START or ORIGIN statement). Translator uses the value of the translated origin to perform the memory allocation for the symbols declared in P. This results in the assignment of a translation time address t_{symb} to each symbol symb in the program.

- The execution start address or simply the start address of a program is the address of the instruction from which its execution must begin.
- The start address specified by the translator is called translated start address of the program.

The origin of a program may have to be changed by the linker or loader for one of two reasons:

- The same set of the translated addresses may have been used in different object modules constituting the program.
- An operating system may require that a program should execute from a specific area of memory.

The following terminology is used to refer to the address of a program entity at different times:

1. **Translation time (or translated) address**: address assigned by translator
2. **Linked address**: address assigned by the linker
3. **Load time (or load) address:** address assigned by the loader.

The same prefixes translation time (or translated), liked and load time are used with the origin and execution start address of a program. Thus

1. **Translated origin**: address of the origin assumed by translator which is specified by the programmer in an ORIGIN statement.
2. **Linked origin**: address of the origin assigned by the linker while producing a binary program.
3. **Load time (or load) address:** address of the origin assigned by the loader while loading the program for execution.

Example: Consider the following assembly program and its generated code

	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		541)	
	END			

- ✓ Translated origin of the program = 500
- ✓ Translation time address of LOOP = 501
- ✓ Suppose load time origin = 900
- ✓ So load time address of LOOP = 901

5.1 RELOCATION AND LINKING CONCEPTS

5.1.1. Program Relocation:

Let AA be the set of absolute addresses-instruction and data addresses-used in the instructions of a program P. AA≠ ϕ implies that if the program P expects some of its instructions and data to occupy memory words with specific addresses. Such a program is called an address sensitive program. It contains one or more of following:

1. **An address sensitive instruction:** An instruction that uses an address ai included in set AA.

2. **An address constant**: A data word that contains an address ai include in set AA.

An 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. To execute correctly from any other memory area, the address used in each address sensitive instruction of P must be corrected.

Program relocation - Program relocation is the process of modifying the addresses used in the address sensitive instructions of a program such that the program can execute correctly from designated area of memory.

- If linked origin ≠ translated origin, relocation must be performed by the linker.
- If load origin ≠ linked origin, relocation must be performed by the loader.

In general, a linker always performs relocation, whereas some loaders do not. If load origin = linked origin then such loaders are called absolute loaders.

Example: The translated origin of the program shown below is 500 .The translation time address of symbol A is 540.The instruction corresponding to statement READ A(existing in translated memory word 500) uses the address 540, hence it is an address sensitive instruction. If the linked origin is 900.A would have the linked address 940.Hence the address in the READ instruction should be corrected to 940.

	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                          541)
            END
```

Performing relocation

Let the translated and linked origins of program p be t_origin_P and l_origin_P respectively. Consider a symbol *symb* in P. Let its translation time address be t_{symb} and linked address be l_{symb}.The relocation factor of P is defined as follows:

$$relocation\_factor_P = l\_origin_P - t\_origin_P \quad \text{---------- (a)}$$

relocation_factor$_P$ can be positive ,negative or zero.

Consider a statement which uses *symb* as operand. The translator puts the address t_{symb} in the instruction generated for it. Now,

$$t_{symb} = t\_origin_P + d_{symb}$$

where d_{symb} is the offset of symb in P. After program has been relocated to the linked origin ,i.e. l_origin_P, we have

$$l_{symb} = l\_origin_P + d_{symb}$$

Using(a),

$$l_{symb} = t\_origin_P + relocation\_factor_P + d_{symb}$$
$$= t\_origin_P + d_{symb} + relocation\_factor_P$$
$$= t_{symb} + relocation\_factor_P \quad \text{---------- (b)}$$

Let IRRp be the set of instruction in program P that relocation. Following (b), relocation of program P can be performed by computing the relocation factor for P and adding it to the translation time address used in every instruction i ϵ IRRp.

Example: Consider the assembly program and its translation

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		541)	
	END			

For this program, relocation factor=900-500

=400

Relocation is performed as follows: IRRp contains the instruction with translated addresses 500and 538. The instruction with translated address 500 contains the address 540 in the operand field. This address is changed to (540+400)=940. The instruction with translated address 538 contains the address 501 in the operand field. Adding 400 to this address make it 901.

5.2.2 Linking

A program unit is any program or routine that is to be linked with another program or routine. Consider an application consist of a set of program units sp = {Pi}. A program unit Pi interact another program unit Pj by using the address of Pj's instruction and data

in its own instructions. To realize such interactions, Pi and Pi must contain public definition and external references as defined in the following:

Public definition: A symbol defined in a program unit that may be referenced in other program units.

External reference: A reference to a symbol which is not defined in the program unit containing the reference.

EXTRN and ENTRY statements:

An ENTRY statement in a program unit lists the public definitions of the program unit. An EXTRN statement lists the symbols to which external references are made in the program unit.

Example: In the following assembly program P, the ENTRY statement indicates that a public definition of TOTAL exists in the program. LOOP and A are not public definitions even though they are defined in the program. The EXTRN statements indicate that the program contains external references to MAX and ALPHA. The assembler does not know the address of an external symbol. If the EXTRN statements did not exist, the assembler would have flagged references to MAX and ALPHA as errors.

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		541)	
	END			

Resolving external references:

Linking: Linking is the process of binding an external reference to the correct link time address.

An external reference is said to be unresolved until linking is performed for it. It is said to be resolved when its linking is completed.

Example: Consider program unit Q contains a public definition of ALPHA which has the translation time address 231. Let Q be linked with the program P which is shown below:

Program P

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		541)	
	END			

Program Q

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

Program unit P contains an external reference to symbol ALPHA which is public definition in Q with the translation time address 231. Let the linked origin of P be 900 and its size be 42 words. Hence the linked origin of Q is 942, and the linked address of ALPHA is 973. Linking is performed by putting the link time address of ALPHA in the instruction of P using ALPHA i.e. putting the address 973 in the instruction with the translation time address 518 in P.

Binary programs:

A binary program is a machine language program comprising a set of program unit SP such that ∀ Pi in SP

1. Pi has been allocated to the memory area whose staring address matches its linked origin, and
2. Each external reference in Pi has been resolved.

To form a binary program from a set of object modules, the programmer invokes the linker by using commands

linker <link origin>, <object module names> [, <execution start address>]

Where <link origin > specifies the memory address to be given to the first words of the binary the binary program. <execution start address > is usually a pair (program unit name, offset in program unit). The linkers convert this into the linked start address, and store it along with the binary program for use when program is to be executed. If specification of <execution start address> is omitted, the execution start address is assumed to be the same as linked origin.

Since we have assumed the linked address = load origin, the loader simply loads the binary program into area of memory staring at its linked origin for execution.

5.2.3 OBJECT MODULE:

The object module of a program contains all the information to relocate and link the program with the other program. The object module of a program P consists of following four components:

1. **Header:** The header contains the translated origin, size and execution start address of P.
2. **Program:** The component contains the machine language program corresponding to P.
3. **Relocation table (RELOCTAB):** this table describes IRRp (Instruction requiring relocation). Each RELOCTAB entry contains single field:
   Translated address: Translated address of an address sensitive instruction.
4. **Linking table(LINKTAB):** This table contains information concerning the public definition and external references in P. Each LINKTAB entry contains following three fields:
   a. **Symbol :** Symbolic name
   b. **Type**: PD/ EXT indicating whether public definition or an external reference
   c. **Translated address**: For public definition, this is the address of the first memory word allocated to the symbol. For an external reference, it is the address of the memory word that is required to contain the address of the symbol.

Example: Consider the program P

Program P

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                          541)
        END
```

The object module of the program P contains the following information:

1. The header contains the information translated origin =500, size=42, execution start address =500.
2. The machine language instruction is shown in above program.
3. The relocation table is follows:

Translated Address
500
538

4. The listing table is as follows:

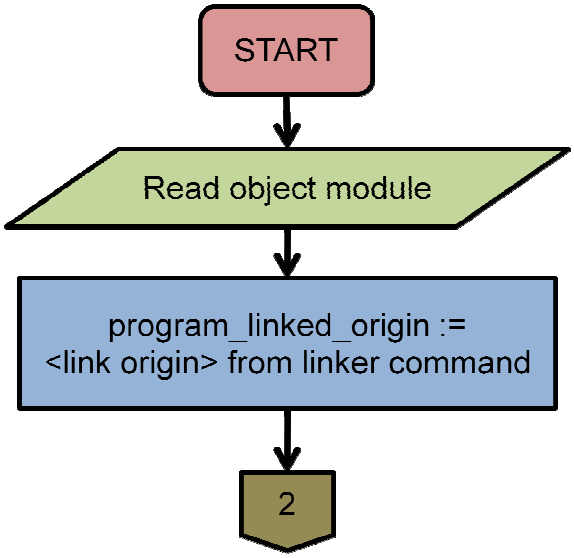
Symbol	Type	Translated address
ALPHA	EXT	518
MAX	EXT	519
A	PD	540

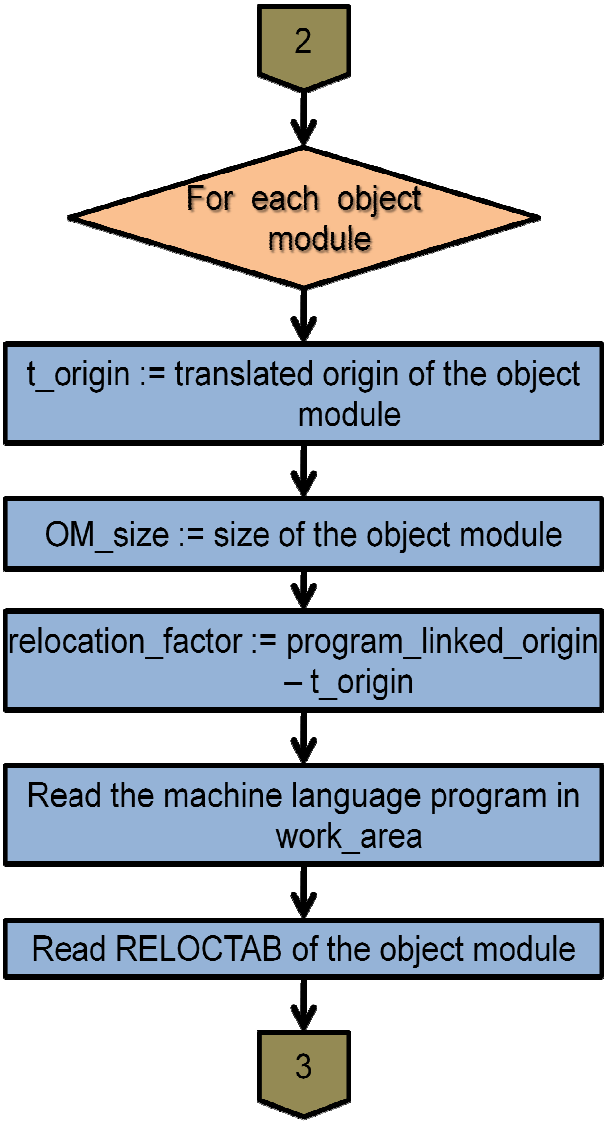
The symbol LOOP does not appear in the linking table. This is because it is not declared as a public definition.

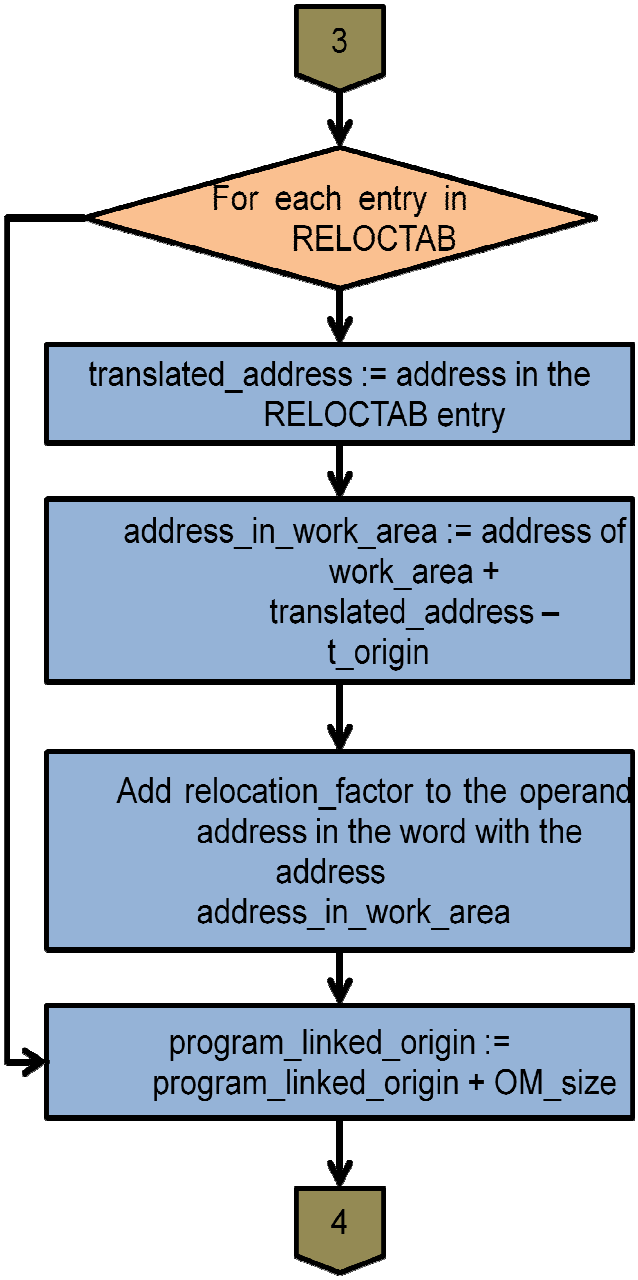
5.3 DESIGN OF A LINKER:

Algorithm (Program Relocation)

1. program_linked_origin := <link origin> from linker command;
2. For each object module
    a. t_origin := translated origin of the object module;
        i. 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 RELOCTAB of the object module.
    e. For each entry in RELOCTAB
        i. translated_address := address in the RELOCTAB entry;
        ii. address_in_work_area := address of work_area +
            a. translated_address –
            b. 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;


START
Read object module
program_linked_origin :=
<link origin> from linker command
2


2
For each object module
t_origin := translated origin of the object module
OM_size := size of the object module
relocation_factor := program_linked_origin – t_origin
Read the machine language program in work_area
Read RELOCTAB of the object module
3


3
For each entry in RELOCTAB
translated_address := address in the RELOCTAB entry
address_in_work_area := address of work_area + translated_address – t_origin
Add relocation_factor to the operand address in the word with the address address_in_work_area
program_linked_origin := program_linked_origin + OM_size
4

Example: Consider program P and Q

Program P

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		541)	
	END			

Program Q

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

Let the address of work area be 300. While relocating the object module, relocation factor = 400. Hence for the first RELOCTAB entry, address_in_work_area=300+500-

500=300. This word contains the instruction for READ A. It is relocated by adding 400 to the operand address in it. For the second RELOCTAB entry, 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.

5.3.2 Linking requirements

An external reference to a symbol ALPHA can be resolved only if ALPHA is declared as a public definition in some object module. Using this observation as the basis, program linking can be performed as follows: The linker processes all object modules being linked and builds a table of all public definitions and their load time addresses.

A name table (NTAB) is defined for use in program linking. Each entry of the table contains the following fields:

1. **Symbol:** Symbolic name of an external reference or an object module.
2. **Linked_address**: For public definition, this field contains linked address of symbol. For an object module it contains the linked origin of object module.

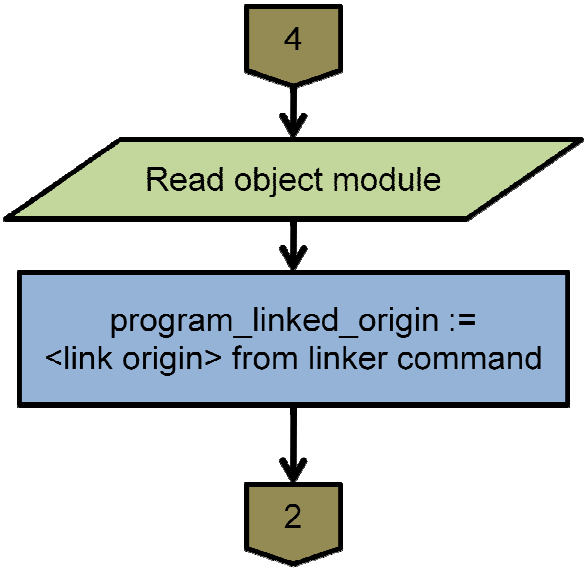
Most information in NTAB is derived from LINKTAB entries with type = PD.

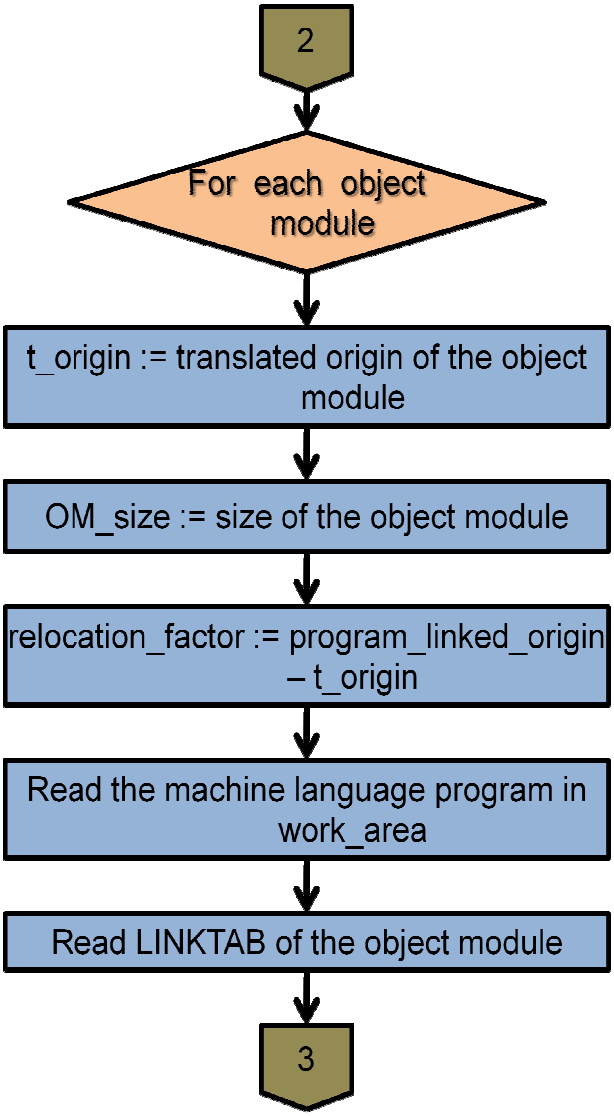
Algorithm (Program linking)

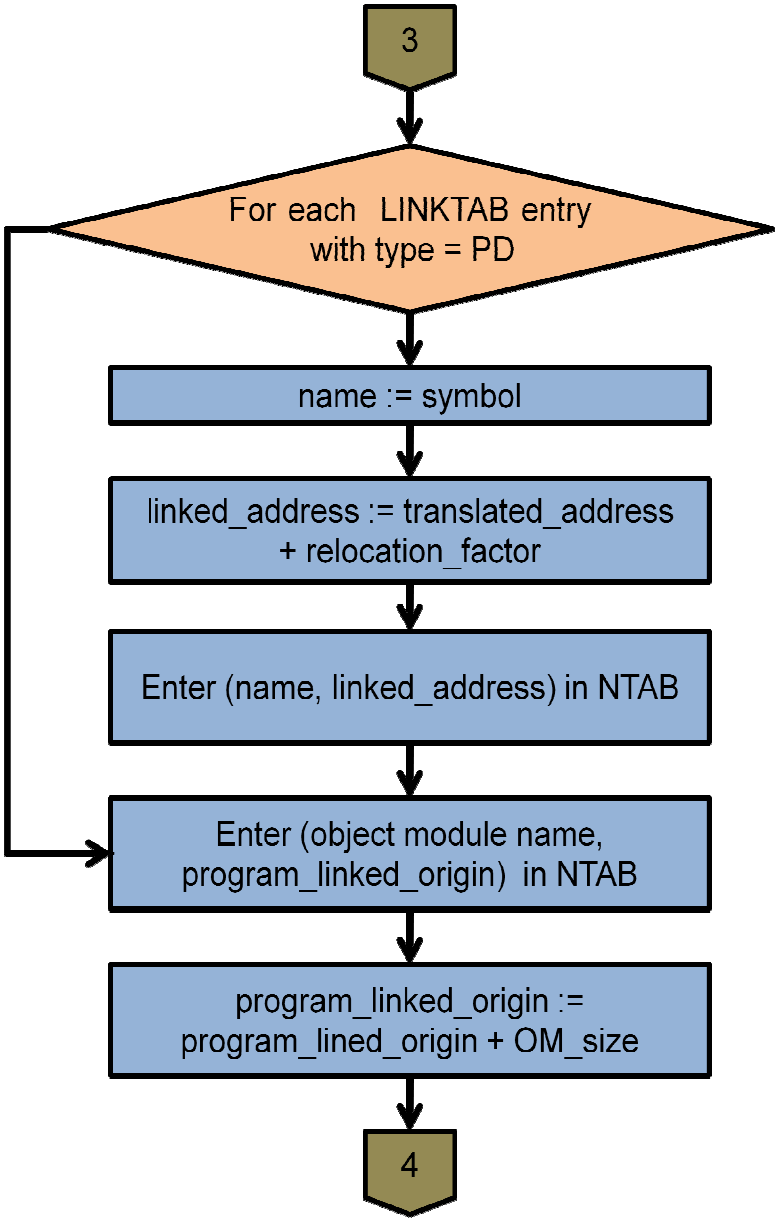
1. program_linked_origin := <link origin> from linker command.
2. For each object module
   a) t_origin := translated origin of the object module;
      OM_size := size of the object module
   b) relocation_factor := program_linked_origin – t_origin;
   c) Read the machine language program in work_area.
   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_lined_origin + OM_size;

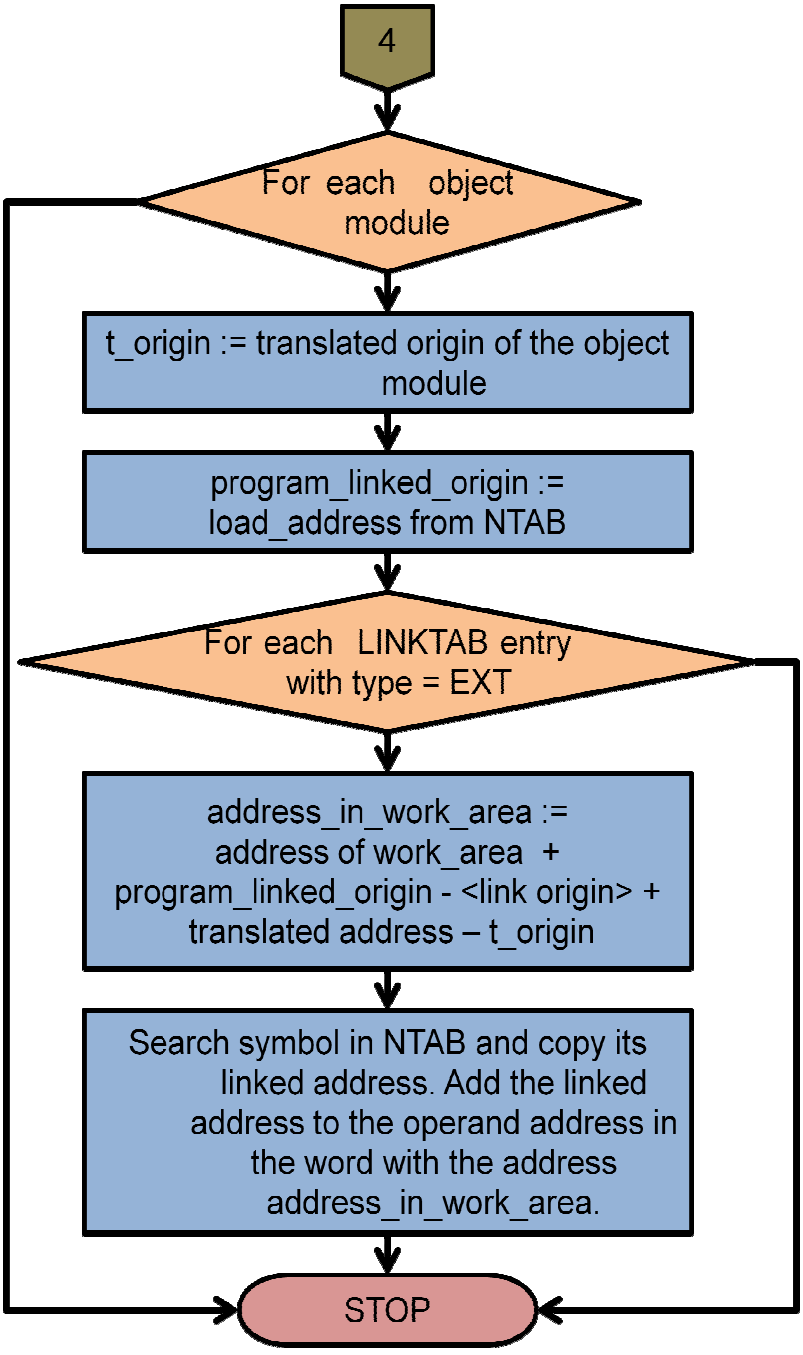
3. For each object module
    a) t_origin := translated origin of the object module;
       program_linked_origin := load_address from NTAB;
    b) For each LINKTAB entry with type = EXT

        i. address_in_work_area := address of work_area +
                                   program_linked_origin –
                                   <link origin> +
                                   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.


4
Read object module
program_linked_origin :=
<link origin> from linker command
2


2
For each object module
t_origin := translated origin of the object module
OM_size := size of the object module
relocation_factor := program_linked_origin – t_origin
Read the machine language program in work_area
Read LINKTAB of the object module
3


3
For each LINKTAB entry with type = PD
name := symbol
linked_address := translated_address + relocation_factor
Enter (name, linked_address) in NTAB
Enter (object module name, program_linked_origin) in NTAB
program_linked_origin := program_lined_origin + OM_size
4


4
For each object module
t_origin := translated origin of the object module
program_linked_origin := load_address from NTAB
For each LINKTAB entry with type = EXT
address_in_work_area := address of work_area + program_linked_origin - <link origin> + translated address – t_origin
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.
STOP

Example: Consider program P and Q

Program P

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		541)	
	END			

Program Q

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

The object module of the program P contains the following information:

1. The header contains the information translated origin =500, size=42, execution start address =500.

2. The machine language instruction is shown in above program.
3. The relocation table is follows:

Translated Address
500
538

4. The listing table is as follows:

Symbol	Type	Translated address
ALPHA	EXT	518
MAX	EXT	519
A	PD	540

While linking program P and Q with linked origin = 900, NTAB would contain the following information:

symbol	Linked address
P	900
A	940
Q	942
ALPHA	973

Let address of work_area be 300. When LINKTAB entry of ALPHA is processed in step 3, address_in_work_area:=300+900-900+518-500, i.e., 318. Hence the linked address of ALPHA, i.e., 973, is copied from the NTAB entry of ALPHA and added to the word in the address 318.

5.4 SELF-RELOCATING PROGRAMS:

The manner in which program can be modified or can modify itself to execute from a given load origin can be used to classify programs into the following:

- **A non-relocatable program**: The program cannot be executed in any memory area other than the area stating from its translated origin. A representative

example of a non- relocatable program is a hand coded machine language program.

- **A relocatable program**: a relocatable program can be processed top relocate it to a desired area of memory. A representative example of a non- relocatable program is an object module.
- **A self-relocating program**: A self relocation program is a program which can perform the relocation of its own address sensitive instructions. It contains the following two provisions for this purpose:
    1. A table of information concerning the address sensitive instructions exists as a part of the program.
    2. Code to perform relocation of address sensitive instruction described also exists as a part of the program. This is called the relocating logic.

The start address of relocating logic is specified as the execution start address of the program. Thus the relocating logic gains control when the program is loaded in memory for execution. It uses load address and information concerning address sensitive instruction to perform its own relocation. Execution control is now transferred to the relocated program. A self relocation program can execute in any area of the memory.

5.6 LINKING OF OVERAYS:

Overlay: An overlay is a part of a program that has the same load origin as some other part(s) of program. Overlays are used to reduce the main memory requirement of a program.

We refer to a program containing overlays as an overlay structured program. It consists of following:

- A permanently resident part, called the root
- A set of overlays that would be loaded in memory when needed.

Execution of an overlay structured program proceeds as follows: To start with, the root is loaded in memory and given control for purpose of execution. Overlays are loaded as and when needed. The loading of an overlay overwrite a previously loaded overlay with the same load origin. This reduces the memory requirement of a program.

The overlay structure of a program is designed by identifying mutually exclusive modules. Such modules do not need to reside simultaneously in memory. Hence they are located in different overlay with the same load origin.

Example: Consider program with 6 section init, read, trans_a, trans_b, trans_c and print. init perform some initializations and passes control to read. read reads one set of data and invokes one of trans_a, trans_b or trans_c depending on the values of the data. print is called to print the results.

trans_a, trans_b, trans_c are mutually exclusive. Hence they can be made into separate overlays. read and print are put in the root of the program since they are needed for each set of data. For simplicity, we put init also in root, though it could be made into an overlay by itself. Figure 2 shows the proposed structure of the program. The overlay structured program can execute in 40K byte though it has a total size of 60 K bytes. It may be possible to overlay parts of trans_a against each other by analyzing its logic. This would further reduce the memory requirements of the program.

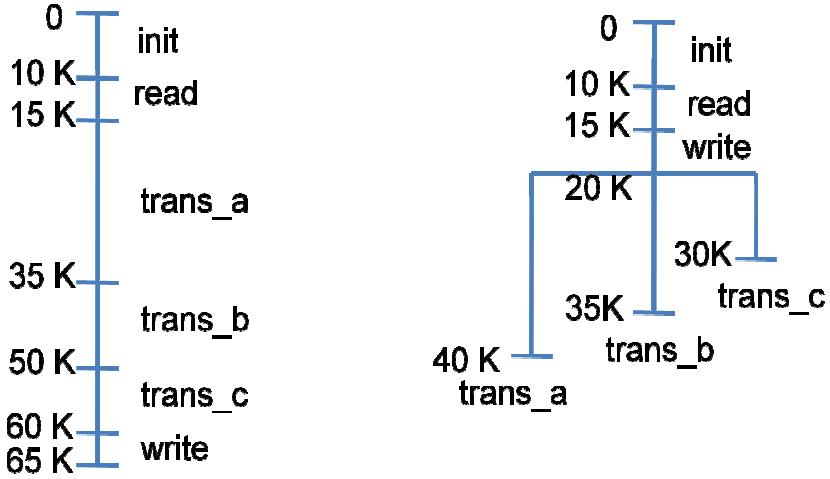


Figure 2: An overlay tree

The overlay structure of an object program is specified in the linker command.

Example: The MS-DOS LINK command to implement the overlay structure of above program is

```
LINK    init + read + write + (trans_a) + (trans_b) + (trans_c),
        <executable file>,<library files>
```

The object module(s) within pair of parentheses becomes one overlay of the program. The object modules not included in any overlay become part of the root. LINK produce a single binary program containing all overlays and stores it in <executable file>

Example: The IBM mainframe linker command for the overlay structure of figure 2 is as follows:

```
Phase main:      PHASE     MAIN, +10000
                 INCLUDE   INIT
                 INCLUDE   READ
                 INCLUDE   WRITE
Phase a_trans:   PHASE     A_TRANS, *
                 INCLUDE   TRANS_A
Phase b_trans:   PHASE     B_TRANS, A_TRANS
                 INCLUDE   TRANS_B
Phase c_trans:   PHASE     C_TRANS, A_TRANS
                 INCLUDE   TRANS_C
```

Each overlay forms a binary program by itself. A PHASE statement specifies the object program name and linked origin for one overlay. The linked origin can be specified in many different ways:

- It can be an absolute address specification as in the first PHASE statement.
- In second PHASE statement, the '*' indicates content of the counter program_linked_origin. Thus the linked origin is the next available memory bytes.

Execution of an overlay structured program

For linking and execution of an overlay structured program in MSDOS, the linker produces a single executable file at the output which contains two provisions

- An overlay manager module for loading the overlays when needed

- All calls that cross overlay boundaries are replaced by an interrupt producing instruction

To start with, the overlay manager receives control and loads the root. A procedure call which crosses overlay boundaries leads to an interrupt. This interrupt is processed by the overlay manager and the appropriate overlay is loaded into memory.

When each overlay is structured into a separate binary program, as in IBM mainframe system, a call which crosses overlay boundaries leads to an interrupt which is attended by the OS kernel. Control is now transferred to the OS loader to load the appropriate binary program.

Changes in the LINKER algorithm

The basic change required in the LINKER algorithm is in assignment of linked addresses to segments. The variable program_load_origin can be used before while processing the root portion of a program. The size of root would decide the load address of the overlays. program_load_origin would be initialized to this value while processing every overlay.

Another change in LINKER algorithm would be in handling of procedural calls that cross overlays boundaries. The LINKER has to identify an inter overlay call and determine the destination overlay. This information should be made available to the overlay manager and kernel of OS which is activated though the interrupt instruction.

An open issue in the linking of overlay structured program is the handling object modules that would be added through autolinking. Should these objects modules be added to current overlay or to the root of program? The former has the advantage of cleaner semantics however it may increase the memory requirement of the program.