Loader and Linker

INTRODUCTION

Earlier programmers used loaders that could take program routines stored in tapes and combine and relocate them in one program. Later, these loaders evolved into linkage editor used for linking the program routines but the program memory remained expensive and computers were slow. A little progress in linking technology helped computers become faster and disks larger, thus program linking became easier. For the easier use of memory space and efficiency in speed, you need to use linkers and loaders.

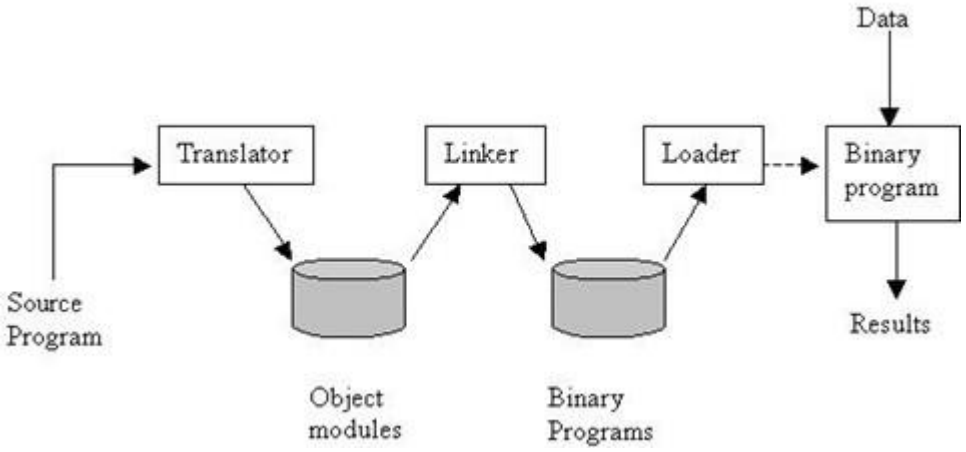
LOADERS AND LINKERS: AN OVERVIEW

An overview of loaders and linker helps you to have a schematic flow of steps that you need to follow while creating a program. Following are the steps that you need to perform when you write a program in language

1. Translation of the program, which is performed by a processor called translator.
2. Linking of the program with other programs for execution, which is performed by a separate processor known as linker.
3. Relocation of the program to execute from the memory location allocated to it, which is performed by a processor called loader.
4. Loading of the program in the memory for its execution, which is performed by a loader.

Figure 1 shows a schematic flow of program execution, in which a translator, linker and loader performs functions such as translating and linking a program for the final result of the program.

Figure 1 shows a schematic flow of program execution, in which a translator, linker and loader performs functions such as translating and linking a program for the final result of the program.

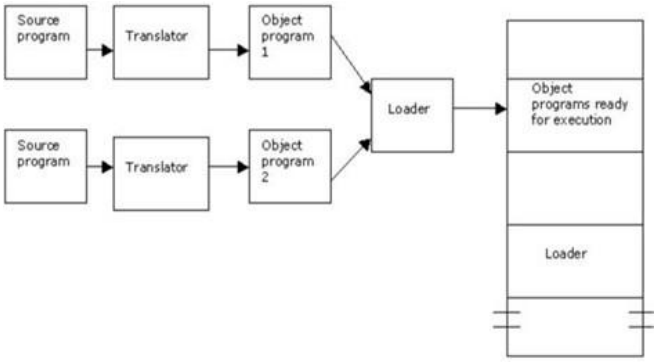


Object Modules
The object module of a program contains information, which is necessary to relocate and link the program with other programs. The object module of a program, P, consists of various components:
1. **Header** contains translated origin, size and execution start address of the program P. Translated origin is the address of the origin assumed by the translator.
2. **Program** contains the machine language program corresponding to P.
3. **Relocation table(RELOCTAB)** maintains a list of entries that contains the translated address for the corresponding entry.
4. **Linking table(LINKTAB)** contains information about the external references and public definitions of the program P.

Binary Programs
A binary program is a machine language program, which contains a set of program units P where $P_i \varepsilon P$. P_i has a memory address that is relocated at its link origin. Link origin is the address of the origin assigned by the linker while producing a binary program. You need to invoke a linker command for creating a binary program from a set of object modules. The syntax for the linker command is:
linker <link origin>, <object module names>,[<execution start address>]

LOADERS
As already discussed that assemblers and compilers are used to convert the source program developed by the user to the corresponding object program. A loader is a program that performs the functions of a linker program and then immediately schedules the resulting executable program for some kind of action. In other words, a loader accepts the object program, prepares these programs for execution by the computer and then initiates the execution. It is not necessary for the loader to save a program as an executable file. The functions performed by a loader are as follows:

1. **Memory Allocation** allocates space in memory for the program.
2. **Linking**: **Resolves** symbolic references between the different objects.
3. **Relocation** adjusts all the address dependent locations such as address constants, in order to correspond to the allocated space.
3. **Loading** places the instructions and data into memory.
The concept of loaders can be well understood if one knows the general loader scheme. It is recommended for the general loader scheme that the instructions and data should be produced in the output, as they were assembled. This strategy, if followed, prevents the problem of wasting core for an assembler. When the code is required to be executed, the output is saved and loaded in the memory. The assembled program is loaded into the same area in core that it occupied earlier. The output that contains a coded form of the instructions is called the object deck. The object deck is used as intermediate data to avoid the circumstances in which the addition of a new program to a system is required. The loader accepts the assembled machine instructions, data and other information present in the object. The loader places the machine instructions and data in core in an executable computer form.
More memory can be made available to a user, since in this scheme, the loader is assumed to be smaller than the assembler. Figure shows the general loader scheme.

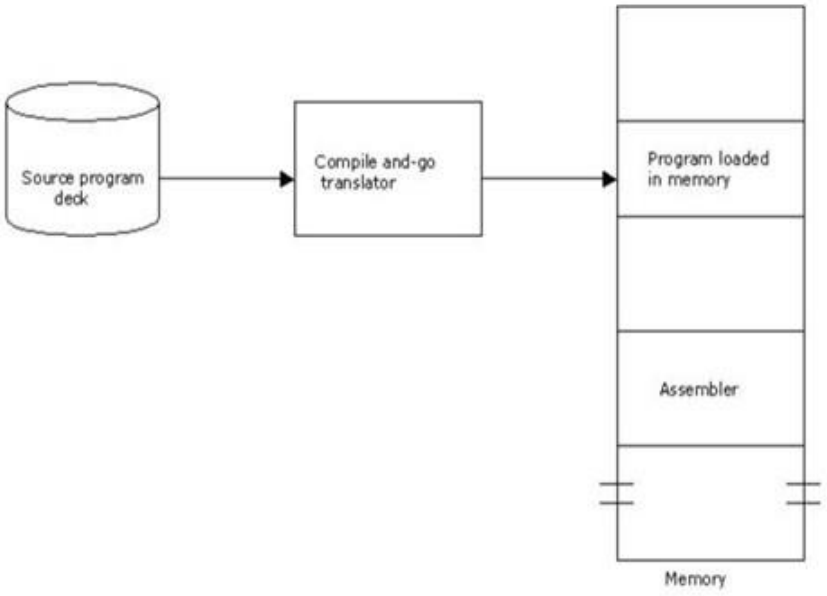


There are various other loading schemes such as compile and go loader, absolute loader, relocating loader and direct linking loader.

Compile and Go Loader

Compile and go loader is also known as “assembler -and-go”. It is required to introduce the term “segment” to understand the different loader schemes. A segment is a unit of information such as a program or data that is treated as an entity and corresponds to a single source or object deck. Figure shows the compile and go loader.

The compile and go loader executes the assembler program in one part of memory and places the assembled machine instructions and data directly into their assigned memory locations. Once the assembly is completed, the assembler transfers the control to the starting instruction of the program.



This scheme is easy to implement, as the assembler simply places the code into core and the loader, which consists of one instruction, transfers control to the starting instruction of the newly assembled program. However, this scheme has several disadvantages. First, in this scheme, a portion of memory is wasted. This is mainly because the core occupied by the assembler is not available to the object program. Second, it is essential to assemble the user’s program deck every time it is executed. Finally, it is quite difficult to handle multiple segments, if the source programs are in different languages. This disadvantage makes it difficult to produce orderly

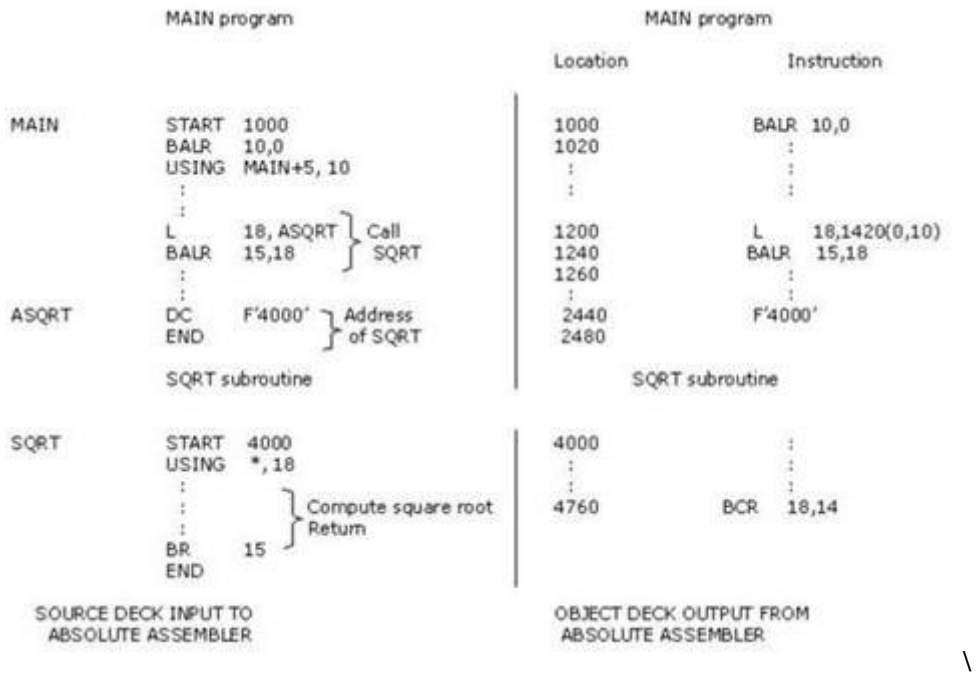
modular programs.

Absolute Loader

An absolute loader is the simplest type of loader scheme that fits the general model of loaders. The assembler produces the output in the same way as in the “compile and go loader”. The assembler outputs the machine language translation of the source program. The difference lies in the form of data, i.e., the data in the absolute loader is punched on cards or you can say that it uses object deck as an intermediate data. The loader in turn simply accepts the machine language text and places it at the location prescribed by the assembler. When the text is being placed into the core, it can be noticed that much core is still available to the user. This is because, within this scheme, the assembler is not in the memory at the load time.

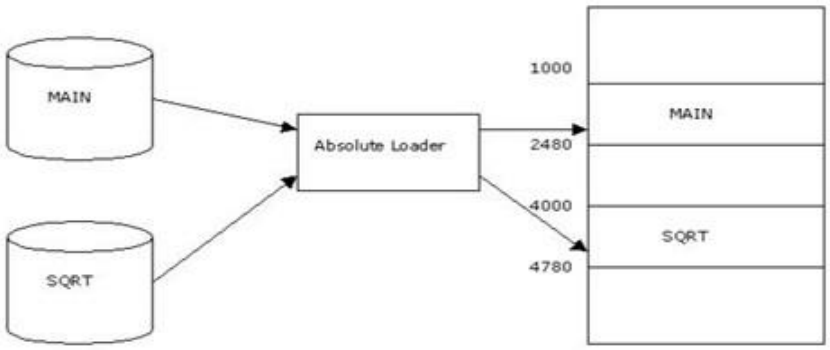
Although absolute loaders are simple to implement, they do have several disadvantages. First, it is desirable for the programmer to specify the address in core where the program is to be loaded. Furthermore, a programmer needs to remember the address of each subroutine, if there are multiple subroutines in the program. Additionally, each absolute address is to be used by the programmer explicitly in the other subroutines such that subroutine linkage can be maintained.

This is quite necessary for a programmer that no two subroutines are assigned to same or overlapping locations. Programmer performs the functions of allocation and linker, the assembler performs the relocation part and the loader in the absolute loader scheme performs loading. Consider an example to see the implementation of an absolute loader.



\

In this section, the implementation of the absolute loader scheme is discussed. Figure 4.4 shows the absolute loader scheme.



In the figure, the MAIN program is assigned to locations 1000-2470 and the SQRT subroutine is assigned locations 4000-4770. This means the length of MAIN has increased to more than 3000

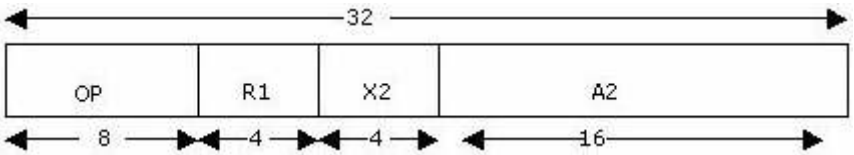
bytes, as it can be noticed from figure. If the modifications are required to be made in MAIN subroutine, then the end of MAIN subroutine, i.e., 1000+3000=4000, gets overlapped with the start of SQRT, i.e., with 4000. Therefore, it is necessary to assign a new location to SQRT. This can be made possible by changing the START pseudo-op card and reassembling it. It is then quite obvious to modify all other subroutines that refer to address of SQRT.

Relocating Loaders

Relocating loaders was introduced in order to avoid possible reassembling of all subroutines when a single subroutine is changed. It also allows you to perform the tasks of allocation and linking for the programmer. The example of relocating loaders includes the Binary Symbolic Subroutine (BSS) loader. Although the BSS loader allows only one common data segment, it allows several procedure segments. The assembler in this type of loader assembles each procedure segment independently and passes the text and information to relocation and intersegment references.

In this scheme, the assembler produces an output in the form of text for each source program. A transfer vector that contains addresses, which includes names of the subroutines referenced by the source program, prefixes the output text. The assembler would also provide the loader with additional information such as the length of the entire program and also the length of the transfer vector portion. Once this information is provided, the text and the transfer vector get loaded into the core. Followed by this, the loader would load each subroutine, which is being identified in the transfer vector. A transfer instruction would then be placed to the corresponding subroutine for each entry in the transfer vector.

The output of the relocating assembler is the object program and information about all the programs to which it references. Additionally, it also provides relocation information for the locations that need to be changed if it is to be loaded in the core. This location may be arbitrary in the core, let us say the locations, which are dependent on the core allocation. The BSS loader scheme is mostly used in computers with a fixed-length direct-address instruction format. Consider an example in which the 360 RX instruction format is as follows:



In this format, A2 is the 16-bit absolute address of the operand, which is the direct address instruction format. It is desirable to relocate the address portion of every instruction. As a result, the computers with a direct-address instruction format have much severe problems than the computes having 360-type base registers. The 360-type base registers solve the problem using relocation bits.

The relocation bits are included in the object desk and the assembler associates a bit with each instruction or address field. The corresponding address field to each instruction must be relocated if the associated bit is equal to one; otherwise this field is not relocated.

Direct-Linking Loaders

A direct-linking loader is a general relocating loader and is the most popular loading scheme presently used. This scheme has an advantage that it allows the programmer to use multiple procedure and multiple data segments. In addition, the programmer is free to reference data or instructions that are contained in other segments. The direct-linking loaders provide flexible intersegment referencing and accessing ability. An assembler provides the following information to the loader along with each procedure or data segment. This information includes:

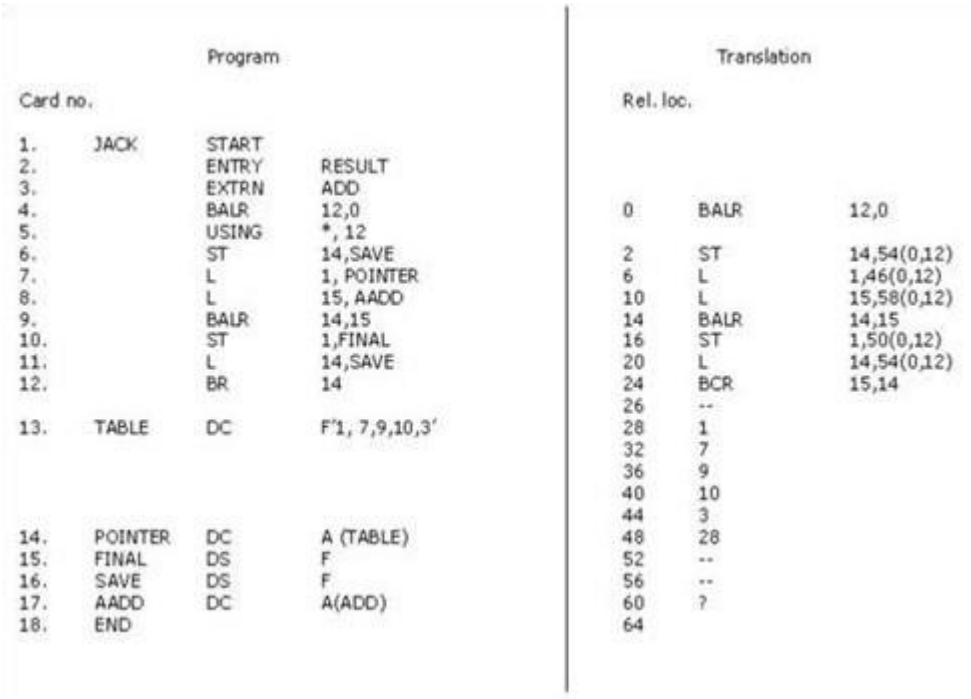
- Length of segment.
- List of all the symbols and their relative location in the segment that are referred by other segments.
- Information regarding the address constant which includes location in segment and description about the revising their values.
- Machine code translation of the source program and the relative addresses assigned.

Let us understand the implementation of the direct-linking loader with the help of an example.

In the following example, there is a source program, which is being translated by an assembler in order to produce the object code. The source program and the object code, which is being translated, both are depicted code and their var.

You must have noticed that the card number 15 in the example contains a Define Constant (DC) pseudo operation that provide instructions to the assembler. This pseudo-op helps in creating a constant with the value that contains the address of TABLE, which places this constant value at the location labelled as POINTER.

This is the point where the assembler does not know the final absolute address of TABLE. This is because the assembler does not have any idea where the program is to be loaded. However, the assembler has the knowledge that address of TABLE is 28th byte from the beginning of the program. Therefore, the assembler places 28 in POINTER. It also informs to the loader that if this program gets loaded at some other location, other than absolute location 0, then the content of location POINTER is incorrect.



Program				Translation		
Card no.				Rel. loc.		
1.	JACK	START				
2.		ENTRY	RESULT			
3.		EXTRN	ADD			
4.		BALR	12,0	0	BALR	12,0
5.		USING	*, 12			
6.		ST	14,SAVE	2	ST	14,54(0,12)
7.		L	1, POINTER	6	L	1,46(0,12)
8.		L	15, AADD	10	L	15,58(0,12)
9.		BALR	14,15	14	BALR	14,15
10.		ST	1,FINAL	16	ST	1,50(0,12)
11.		L	14,SAVE	20	L	14,54(0,12)
12.		BR	14	24	BCR	15,14
				26	--	
13.	TABLE	DC	F'1, 7,9,10,3'	28	1	
				32	7	
				36	9	
				40	10	
				44	3	
14.	POINTER	DC	A (TABLE)	48	28	
15.	FINAL	DS	F	52	--	
16.	SAVE	DS	F	56	--	
17.	AADD	DC	A(ADD)	60	?	
18.	END			64		

Another usage of DC pseudo-op in this program is on card number 17. This pseudo-op instructs the assembler to create a constant with the value of the address of the subroutine ADD. It would also cause this constant to be placed in the location, which is labelled as AADD.
The assembler cannot generate this constant, as the assembler does not have any idea of the location where the procedure ADD is loaded. It is desirable for the assembler to provide information to the loader such that the final absolute address of ADD can be loaded at the designated location, i.e., AADD, when the program gets loaded

LINKAGE EDITOR

Linkage editor allows you to combine two or more objects defined in a program and supply information needed to allow references between them. A linkage editor is also known as linker.
To allow linking in a program, you need to perform:
• Program relocation
• Program linking

Program relocation

Program relocation is the process of modifying the addresses containing instructions of a program. You need to use program relocation to allocate a new memory address to the instruction. Instructions are fetched from the memory address and are followed sequentially to execute a program. The relocation of a program is performed by a linker and for performing relocation you need to calculate the relocation_factor that helps specify the translation time address in every instruction. Let the translated and linked origins of a program P be t_ origin and l_origin, respectively. The relocation factor of a program P can be defined as:
relocation_factor = l_origin – t_origin
The algorithm that you use for program relocation is:
1. program_linked_origin:=<link origin> from linker command;
2. For each object module
3. t_origin :=translated origin of the object module; OM_size

:=size of the object module;
4. relocation_factor :=program_linked_origin-t_origin;
5. Read the machine language program in work_area;
6. Read RELOCTAB of the object module
7. For each entry in RELOCTAB
A. **translated_addr**: = address in the RELOCTAB entry;
B. **address_in_work**: =address of work_area + translated_address – t_origin;
C. add relocation_factor to the operand in the wrd with the address address_in_work_area.

C. **program_linked_origin**: = program_linked_origin + OM_size;

Program Linking

Linking in a program is a process of binding an external reference to a correct link address. You need to perform linking to resolve external reference that helps in the execution of a program. All the external references in a program are maintained in a table called name table (NTAB), which contains the symbolic name for external references or an object module. The information specified in NTAB is derived from LINKTAB entries having type=PD. The algorithm that you use for program linking is:

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_linked_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
address_in_work_area: =address of work_area + program_linked_origin - <link origin> + translated address – t_origin
f 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.

DESIGN OF A LINKER

You need to perform linking and relocation to explain the design of a linker. For example, the design of a program LINKER helps explain linking and relocation. The output of a LINKER

program is in binary form with a .COM extension. You need to use the following syntax to create a LINKER program:

LINKER <object module name>,<executable file>,<load origin>,<list of binary files>
The LINKER program allows you to perform linking and relocation of all the named object modules, which are contained in <object module name>. The program is *Self-Instructional Material*

stored in <executable file>, the <list of binary files> contains a list of external object module name that are not specified in <object module name>. You need to resolve all the external object module names for terminating the execution of LINKER program. The LINKER program is explained using an example:

LINKER alpha+beta+min, calculate, 1000, pas.lib

Alpha, beta and min are the names of the object modules that you need to link, and calculate is the name given to the executable file, which is generated by LINKER. The load origin of the program that you want to execute is 1000. The external names, which are not defined in object modules alpha, beta and min needs to be searched in pas.lib library

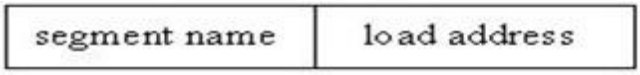
For proper execution of the LINKER program you need to use a two-pass strategy:
• First pass
• Second pass

First Pass
In the first pass, the object module collects information about the segments and definitions that are defined in the program. The algorithm that you use for the first pass is:
Algorithm (first pass of LINKER)

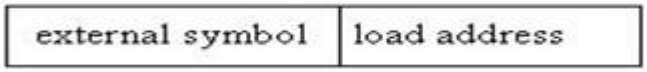
1. **Program_linked_origin**: =<load origin>; // a default value is used if <load origin is not specified>.
2. Repeat step 3 for each object module to be linked.
3. Select an object module and process its object records.
A. If an LANAME record, enter the names in NAMELIST.
B. If a SEGDEF record
ƒ **i**: = name index; segment_name := NAMELIST[i];
ƒ **segment_addr**: =start address in attributes(if present);
ƒ If an absolute segment, enter (segment_name, segment_addr) in NTAB.
ƒ If segment is relocatable and cannot be combined with other segments:
-align the address contained in program_linked_origin on the next word or paragraph as indicated in the attributes field. -enter (segment_name, program_linked_origin) in NTAB.
-program_linked_origin :=program load origin+segment length;
C. For each PUBDEF record
ƒ **i**:=base; segment_name := NAMELIST[i];
ƒ **symbol**:=name;
ƒ **segment_addr**:=load address of segment_name in NTAB;
ƒ **sym_addr**:=segment_addr+offset;
ƒ Enter (symbol, sym_addr) in NTAB.

Second Pass

The second pass performs relocation and linking of the program. Second pass helps execute program in work_area. You need to fetch the data from LEDATA records using segment index and data offset and send them to the correct parts of work_area. The segment index helps obtain the load origin of the segment from NTAB while data offset gives the load address of the first byte of the data in the LEDATA record. A segment index can also contain a numeric value rather than the name of the segment, LNAMES records help obtain the segment name. The obtained segment name, the segment name is searched in NTAB for its load origin. For the relocation and linking you need to FIXUPP records, while processing a FIXUPP the target data can be searched in EXTDEF or SEGDEF to obtain the target name. The obtained target name can be searched in NTAB for its load origin. LINKER constructs a table called the segment table (SEGTAB), which contains the name of all the segments defined in the object module. A SEGTAB entry has the following syntax:



The information about the load address is copied from NTAB.
For the relocation and linking, you need to FIXUPP records, while processing a FIXUPP the target data can be searched in EXTDEF or SEGDEF to obtain the target name. The obtained target name can be searched in NTAB for its load origin. A FIXUPP record may also contain a reference to an external symbol, LINKER constructs an external symbols table (EXTTAB), which has the following syntax:



The information about the load address is copied from NTAB.
At the end of the second pass, the executable program is stored in <executable file>, which is specified in the LINKER command.

The algorithm that you use for the second pass is:
1. **list_of_object_modules**: = 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] ;
Enter (segment_name, load address from NTAB) in SEGTAB.
C. If an EXTTAB record
ƒ **external_name**: =name from EXTTAB record;
ƒ If external_name is not found in NTAB, then
-Locate an object module in the library which contains external_name as a segment or public definition.
-Add name of object module to list_of_object_modules.
-Perform first pass of LINKER, for the new object module.
ƒ Enter (external_name, load address from NTAB) in EXTTAB
D. If an LEDATA record

ƒ **i** := segment index; d:=data offset;
ƒ **program_load_origin**:=SEGTAB[i].load address;
ƒ **address_in_work_area**:= address of work_area + program_load_origin-

<load origin> + d;
ƒ Move the data from LEDATA into the memory area starting at the address address_in_work_area.
E. If a FIXUPP record, for each FIXUPP specification
ƒ **f**:=offset from locat field;
ƒ **fix_up_address**:= address in work_area + f;
ƒ perform required fix up using a load address from SEGTAB or EXTTAB and the value of the code in locat and fix dat.
F. If a MODEND record
If start address is specified, compute the corresponding load address (analogous to the computation while processing an LEDATA record) and record it in the executable file being generated.

DYNAMIC LINKING

Sophisticated operating systems, such as Windows allow you to link executable object modules to be linked to a program while a program is running. This is known as dynamic linking. The operating system contains a linker that determines functions, which are not specified in a progam. A linker searches through the specified libraries for the missing function and helps extract the object modules containing the missing functions from the libraries. The libraries are constructed in a way to be able to work with dynamic linkers. Such libraries are known as dynamic link libraries (DLLs). Technically, dynamic linking is not like static linking, which is done at build time. DLLs contain functions or routines, which are loaded and executed when needed by a program. The advantages of DLLs are:

• **Code sharing**: Programs in dynamic linking can share an identical code instead of creating an individual copy of a same library. Sharing allows executable functions and routines to be shared by many application programs. For example, the object linking and embedding (OLE) functions of OLE2.DLL can be invoked to allow the execution of functions or routines in any program.
• **Automatic updating**: Whenever you install a new version of dynamic link library, the older version is automatically overridden. When you run a program the updated version of the dynamic link library is automatically picked.
• **Securing**: Splitting the program you create into several linkage units makes it harder for crackers to read an executable file.