# MODULE- 4

**LOADERS AND LINKERS**

### Introduction

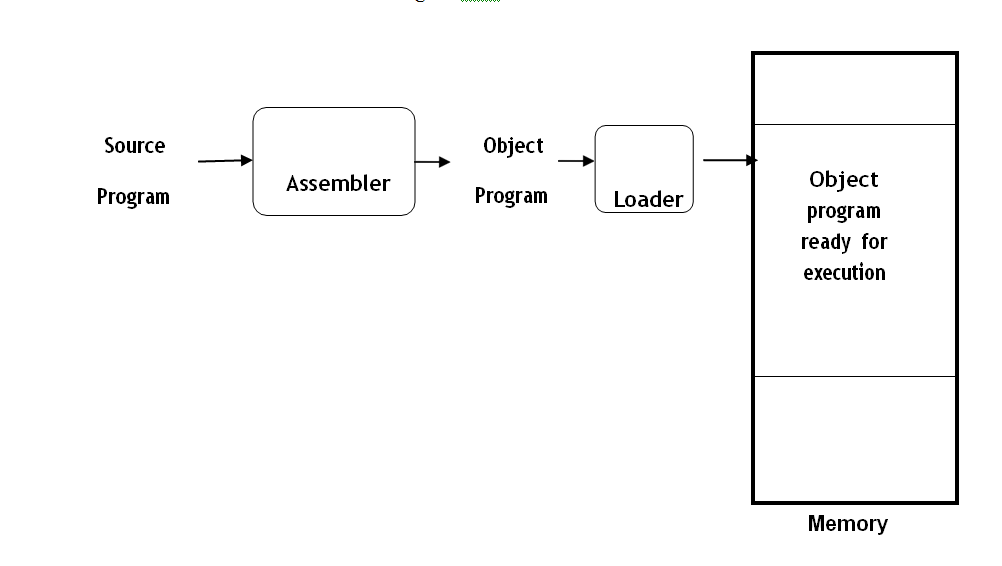
The Source Program written in assembly language or high level language will be converted to object program, which is in the machine language form for execution. This conversion either from assembler or from compiler, contains translated instructions and data values from the source program, or specifies addresses in primary memory where these items are to be loaded for execution.

This contains the following three processes, and they are,

* **Loading** - which allocates memory location and brings the object program into memory for execution - (Loader)
* **Linking**- which combines two or more separate object programs and supplies the information needed to allow references between them - (Linker)
* **Relocation** - which modifies the object program so that it can be loaded at an address different from the location originally specified - (Relocating Loader)

## 4. 1 Basic Loader Functions:

* A loader is a system software that performs the loading function. It brings object program into memory and starts its execution. The role of loader is as shown in the figure.



### Type of Loaders

The different types of loaders are, absolute loader, bootstrap loader, relocating loader (relative loader), and, linking loader. The following sections discuss the functions and design of all these types of loaders.

### **4.1.1 Design of Absolute Loader:**

* The operation of absolute loader is very simple. The object code is loaded to specified locations in the memory. At the end the loader jumps to the specified address to begin execution of the loaded program. Linking and relocation is not done.
* The algorithm for this type of loader is given here.

Begin

read Header record

verify program name and length

read first Text record

**while** record type is != ‘E’ **do begin**

{if object code is in character form, convert into internal representation} move object code to specified location in memory

read next object program record

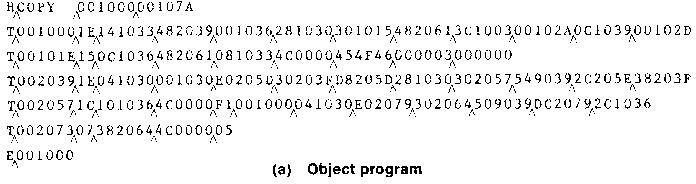
#### end

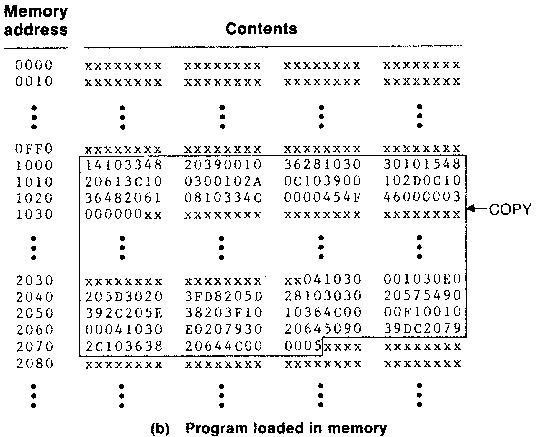
jump to address specified in End record

#### end

Algorithm for Absolute loader

* In this all functions are done in a single pass. The header is checked to veriy that the correct program has been presented for loading. As each text record is read the object code it contains is moved to the indicated address in memory. When the End record is encountered the loader jumps to the specified address to begin execution of the loaded program.





* The figure (b) shows the representation of program from figure (a) after loading.
* In the object program each byte of assembled code is given using its hexadecimal representation in character form.
* In the object program , each byte of assembled code is given using its hexadecimal representation in character form. For example, the machine opcode for an STL instruction would be represented by the pair of characters “1” and “4”. When these are read by the loader , they will occupy two bytes of memory. This opcode must be stored in a single byte with hexa decimal value 14. Thus each pair of bytes from the object program must be packed together into one byte during loading.

### **4.1.2 A simple bootstrap loader**

* When a computer is first turned on or restarted, a special type of absolute loader, called bootstrap loader is executed. This bootstrap loads the first program to be run by the computer-- usually an operating system. The bootstrap itself begins at address 0. It loads the OS starting address 80.
* Working: Consider the bootstrap loader for SIC/XE. The bootstrap loader begins at address 0 in the memory. It loads the OS starting at address 80. Each byte of object code to be loaded is represented on device F1 as two hexa decimal digits(Text record) . Object code is loaded to consecutive memory locations starting at address 80. After all the object code from device F1 has been loaded the bootstrap jumps to the address 80.
* GETC subroutine – This subroutine reads one character from device F1 and converts from ASCII to hex. This is done by subtracing 48 if the character is from 0 to 9. For characters A to F subtract 55. Subroutine jumps to address 80 when end of line is reached.
* Main loop of the bootstrap loader- This keeps the address of the next memory location to be loaded in register X. GETC is used to read and convert a pair of characters from device F1(represents one byte of object code). These two hexadecimal values are combined to a single byte by shifting the first one left by 4 bit positions and adding the second to it. The resulting byte is stored at address currently in register X

The algorithm for the bootstrap loader is as follows





## 4.2Machine-Dependent Loader Features

* Absolute loader is simple and efficient, but the scheme has potential disadvantages. One of the most disadvantage is the programmer has to specify the actual starting address, from where the program to be loaded. This does not create difficulty, if one program to run, but not for several programs. Further it is difficult to use subroutine libraries efficiently.
* This needs the design and implementation of a more complex loader. The loader must provide program relocation and linking, as well as simple loading functions. This depends on machine architecture.

**4.2.1Relocation(Relocating loader)**

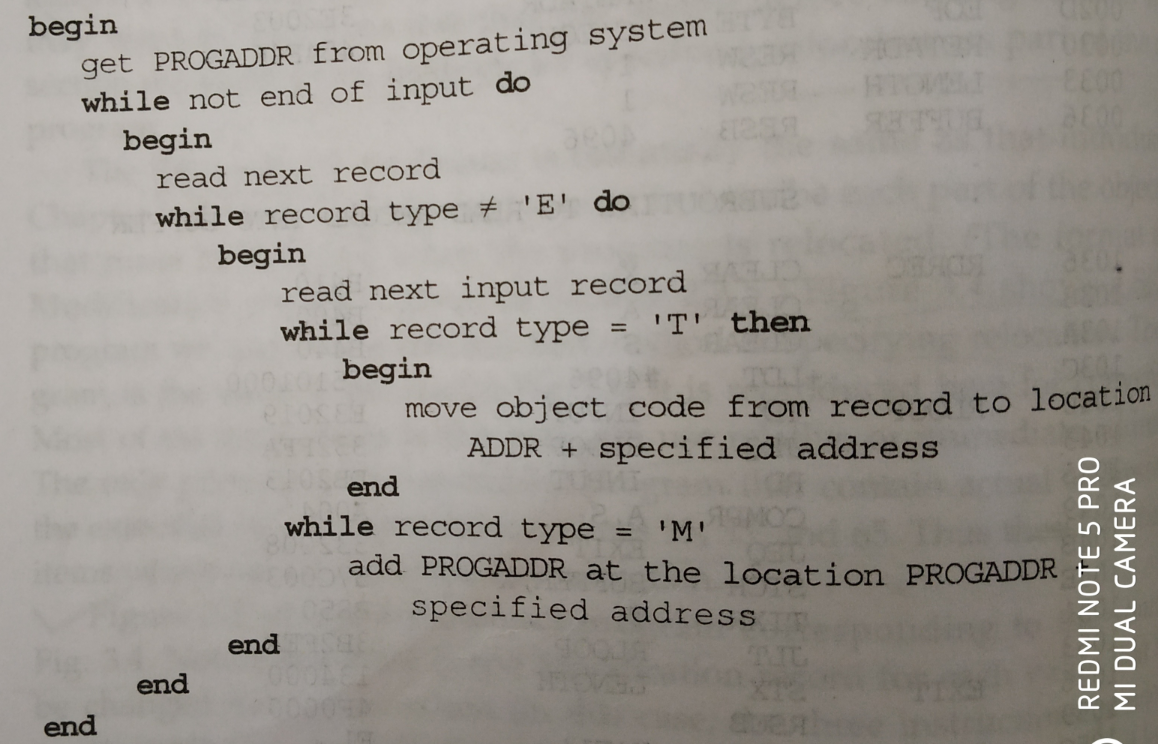
* Loaders that allow program relocation are called relocating loaders.
* There are two methods for providing relocation as part of the object program.
* Modification record
* Bit masking

**Modification Record**

* A modification record is used to describe each part of the object code that must be changed when the program is relocated.
* Consider SIC/XE programs, Most of the instructions in this program uses relative or immediate addressing. So modification not required. Only format 4 instructions require modification
* Each modification record specifies the starting address and length of the field to be modified and what modification to be performed.(adding the start address).



Algorithm for SIC/XE relocation loader



**Bitmasking**

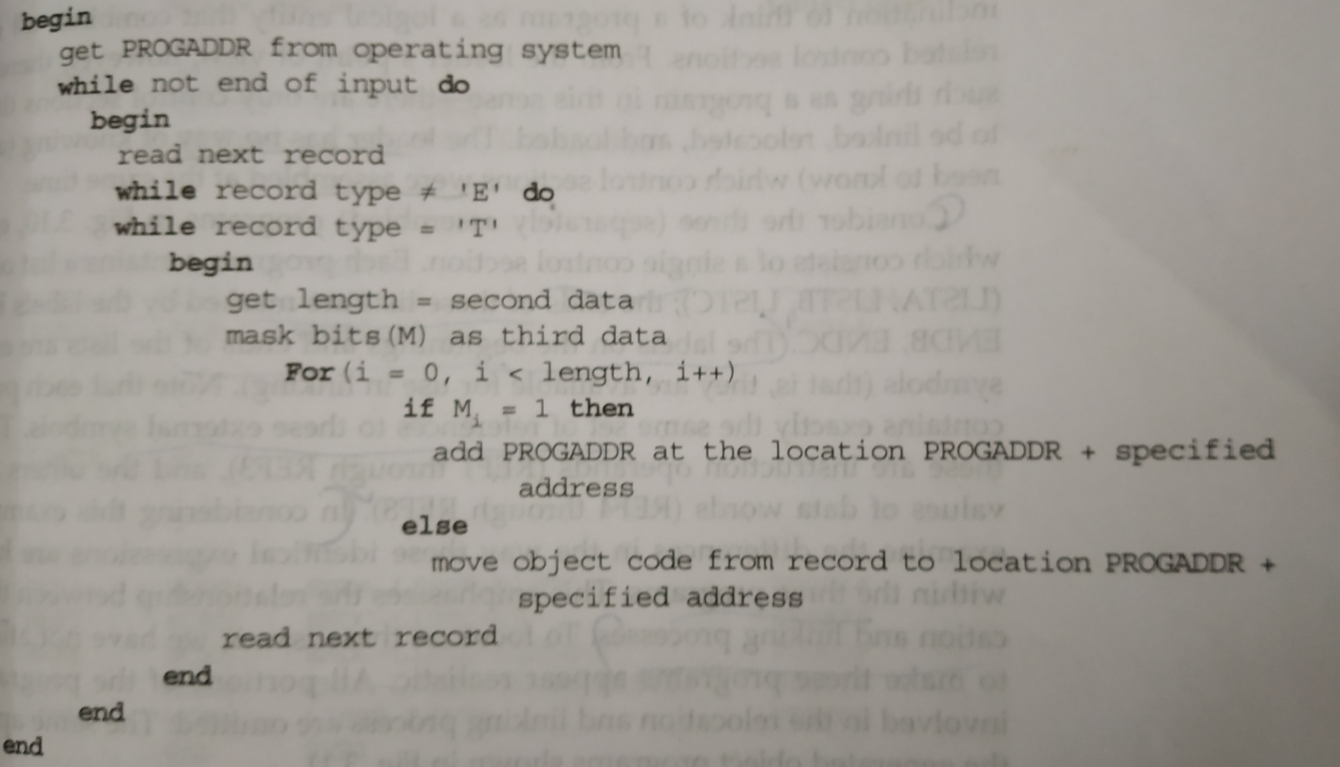
* In SIC program relative addressing is not used. So every instruction needs modification. We can not write modification records for all instructions.
* So relocation bits are used. Each instruction object code is associated with relocation bit.
* Relocation bits for each text record is written together into bitmask after the length using 3 hexadecimal digits.(12 bits)
* Example:



* If the relocation bit is 1 program starting address is to be added to this word.

FFC= 111111111100

**SIC relocation loader algorithm**



**4.2.2Program Linking**

* Consider the program of control sections. The program is made up of 3 control sections.

1. Main program
2. Read subroutine
3. Write subroutine

* These control sections could be assembled together or they could be assembled independently as separate segments of object code after assembly.
* The programmer thinks the three control sections together as a single program. But loader considers this as separate control sections which are to be linked , relocated and loaded.
* Consider the three separate programs PROGA,PROGB,PROGC. In this example, there are differences in handling the identical expressions within the 3 programs.
* Consider the references and the corresponding modification records.
* The general approach is assembler evaluate as much as of the expression it can. The remaining terms are passed on to the loader through modification records.







* Each program contains a list of items(LISTA, LISTB, LISTC). The ends of these lists are marked by ENDA, ENDB, ENDC. Each program contains the same set of references to these external symbols. Three of these are instruction operands(REF1,REF2,REF3). and the others are the values of data words.(REF4 through REF8).
* Consider first reference marked REF1.For PROGA REF1 is simply a reference to a label within the program. It is assembled in the usual way as PC relative instruction.In PROGB the same operand refers to an external symbol. The assembler uses an extended format instruction with addess field set to 00000. Object program for PROGB contains a modification record instructing the loader to add the value of the symbol LISTA to this address field when the program is linked.This reerence is handled exactly in the same way for PROGC.







* The figure below shows how the three programs are loaded into memory.





* The values of REF4 through REF8 are same in all the three programs because the same source expression appeared in each program.

**4.2.3 Algorithm and data structures for a linking loader**

* Consider the algorithm for a linking and relocating loader.
* We use modification records for both relocating and linking
* This type of loader is found on SIC/XE machines whose relative addressing makes relocation unnecessary.
* Input- consists of a set of object programs (control sections) that are to be linked together.
* Control sections or programs contain external references whose definition does not appear in the same program or control section. So linking can not be done until an address is assigned to the external symbol. So it requires two passes.
* Pass1- Assigns addresses to all external symbols.
* Pass2- performs the actual loading relocation and linking.
* The **main data structure** for the linking loader is an external symbol table **ESTAB**. It is analogous to SYMTAB. It stores the name and address of each external symbol in the control section. The table also indicates in which control section the symbol is defined.
* Two variables: PROGADDR- Program starting address in memory where the linked program should be loaded. Its value is supplied to the loader by the OS.CSADDR-contains the starting address assigned to the control section currently being scanned by the loader.
* Example: Consider the object programs of PROGA, PROGB, PROGC in fig 3.9 as input to the loader.

**Pass1**

* During the first pass the loader is concerned only with Header and Define record types in the control sections.
* The beginning load address for the linked program(PROGADDR) is obtained from OS. This becomes the starting address for the first control section(CSADDR).
* The control section name is entered into ESTAB with value given by CSADDR.
* All external symbols appearing in the define record for the control section are also entered into ESTAB. Their addresses are obtained by adding the value specified in the Define record to CSADDR.
* When the END record is read the control section length CSLTH which was saved from the Header record is added to CSADDR. This gives the starting address for the next control section.
* At the end of pass1 , ESTAB contains all external symbols defined in the control sections together with addresses assigned to each.
* Many loaders include the ability to print a **load map** that shows these symbols and their addresses.

Output of pass1



**Algorithm for pass1 of a linking loader**



**Pass2**

* Performs the actual loading, relocation and linking of the program.
* CSADDR holds the starting address of the control section currently being loaded.
* As each Text record is read , the object code is moved to the specified address (plus the current value of the CSADDR).
* When a modification record is encountered , the symbol whose value is to be used for modification is looked up in ESTAB. This value is then added to or subtracted from the indicated location in memory.
* The last step performed by the loader is transferring of control to the loaded program to begin execution.

**Pass2 Algorithm**



* The algorithm can be made more efficient if a slight change is made in the object program format. that is assigning a reference number to each external symbol referred to in a control section. This reference number is used in modification records.

**4.3** **Machine Independent loader features**

**4.3.1 Automatic Library search**

* This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded. The routines are automatically retrieved from library as they are needed during linking.
* Loader can automatically include routines from a library into the program being loaded.
* The programmer has to only give the subroutine name in the external reference. The routine will be automatically fetched from the library and linked with the main program.
* Working: Enter symbols from Refer record into the symbol table(ESTAB) . When the definition is encountered the address is assigned to the symbol. At the end of pass the symbols in ESTAB remain undefined represent unresolved external references . The loader searches the library for the routines and process the subroutines as if they are part of the input stream.
* The libraries to be searched by the loader contain assembled or compiled versions of the object program(sub program). A special file structure is used for libraries. This is known as directory. This contains the name of the subroutine and a pointer to its address within the file.

**4.3.2 Loader Options**

* Many loader allow the user to specify options that modify standard processing.
* Loaders have special command language that is used to specify options. Sometimes there is a separate input file to the loader that contains such control statements. The programmer can even include loader control statements in the source program.

Some of the loader options are:

1. Selection of alternative sources of input:

INCLUDE programname(libraryname)

This command direct the loader to read the designated object program from a library and treat it as if it were primary loader input.

1. Command to delete external symbols or entire control section

DELETE csectname

This instruct the loader to delete the control section from the set of programs being loaded.

1. CHANGE name1,name2

This command causes the external symbol name1 to be changed to name2 wherever it appears in the object program.

Eg: Consider the object program COPY. Here main program is COPY and the two subroutines are RDREC and WRREC. Each of these is a separate control section. Suppose that a set of utility routines are available on the computer system. Two of these READ and WRITE are are designed to perform the same functions as RDREC and WRREC. If we want to use READ and WRITE we can give the loader commands

INCLUDE READ(UTLIB)

INCLUDE WRITE(UTLIB)

DELETE RDREC, WRREC

CHANGE RDREC,READ

CHANGE WRREC,WRITE

1. Another common loader option involves the automatic inclusion of library routines to satisfy external references. Most loaders allow the user to specify alternative libraries to be searched using a statement such as LIBRARY MYLIB . Such user specified libraries are normally searched before the standard libraries. This allows the user to use special versions of the standard routines.
2. Loaders that perform automatic library search to satisfy external reference allows the user to avoid some references using the command NOCALL. Eg: NOCALL STDDEV, PLOT. This avoids the overhead of loading and linking the unwanted routines
3. Other options:

* No external reference should be resolved.
* Specify the output from the loader(load map)
* Specify the location at which the execution is to begin

**4.4 Loader Design**

* Loaders do loading , relocation and linking.
* There are 4 types
* Linkage editor- links the program stores it in a file and later loads.
* Linking loader- linking during load time
* Dynamic linking- linking during execurion time
* Bootstrap loader- loads the first program or OS.

**4.4.1Differences between Linkage editor and linking loader**

**Linking loader linkage editor**

|  |  |
| --- | --- |
| 1. Performs all linking and relocation operations and loads the linked program directly into memory for execution 2. A linking loader searches the library and resolves external references every time the program is executed. 3. More than one pass required. | 1. Produces a linked version of the program called load module which is written to a file for later execution 2. Resolution of external references and library searching are only performed once. 3. The loading can be accomplished in one pass and no external symbol table required, much less overhead than a linking loader. |



**Advantages of Linkage editors**

* Linkage editors can perform many useful functions besides simply preparing an object program for execution. Consider the example, a program PLANNER that uses a large number of subroutines. Suppose that one subroutine called PROJECT is changed. After new version of PROJECT is assembled the linkage editor can be used to replace this subroutine in the linked version of PLANNER.

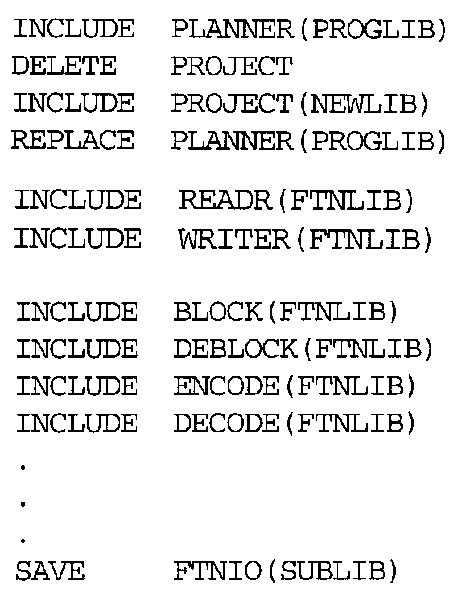
INCLUDE PLANNER(PROGLIB)

DELETE PROJECT (delete from existing planner)

INCLUDE PROJECT(NEWLIB) (include new version)

REPLACE PLANNER(PROGLIB)

* Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. Eg: For FORTRAN programs there are a number of subroutines that are used for input and output. They are read and write datablocks, encode and decode data items etc. Linkage editor can be used to combine these subroutines into a package with the following commands.



* Linkage editors can also allow the user to specify that external references are not to be resolved by automatic library search.

**4.4.2 Dynamic Linking**

* In dynamic linking the linking function is done at execution time. That is a subroutine is loaded and linked to the rest of the program when it is first called.
* Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library. For eg: in C such fuctions are stored in dynamic linking library.. A single copy of the routines in this library could be loaded into memory and all programs share this.
* In object oriented program dynamic linking is often used for references to software objects.
* Advantage:- Dynamic linking provide the ability to load the routines only when they are required. For eg: consider the subroutine which diagnose the error in input data during execution. If such errors are rare these subroutines need not be used.
* Consider the following example of dynamic linking. Here the routines that are to be dynamically loaded must be called via an OS service request.

**Loading and calling a subroutine via dynamic linking**





* When the dynamic linking is used the association of an actual address with the symbolic name of the called routine is done at execution time.. This is known as **dynamic binding**.

**4.3.3 Bootstrap loaders**

* Consider how the loader itself is loaded into memory. OS loads the loader. How the OS gets loaded.
* In an idle system if we specify the absolute address the program can be loaded at that location. that is a mechanism of absolute loader is required.
* One solution to this is to have a built in hardware function that reads a fixed length record from some device into memory at some fixed location. This device can be selected via console switches. After the read operation is complete the control is automatically transferred to the address in memory where the record was stored. This record contains machine instructions that load the absolute program that follows.
* If the loading process requires more instructions than can be read in a single record this first record causes the reading of others and in turn other records . Hence the name **Bootstrap.**