Loaders

Definition:

Loader is a system software program that performs the loading function.

Loading is the process of placing the program into memory for execution. The loader is responsible for initializing the execution of process.

Functions of loader:

1. Allocation: The space for program is allocated in the main memory, by calculating the size of the program.

2. Linking:Which combines two or more separate object programs and supplies the necessary information

3. Relocation:Adjusting all address location to object program or modifies the object program so that it can be loaded at an address different from the location originally specified.

4. Loading:Physically place the machine instruction and data into memory.

Loaders Scheme or types of Loader:

1. Compile and go loader or Assemble and go loader

2. General loader scheme

3. Absolute loader

4. Direct linking loader

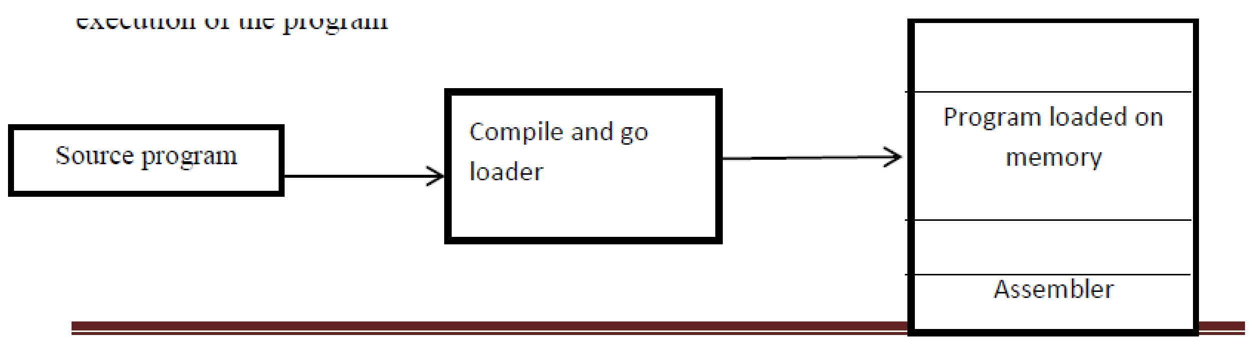
5. Relocating loader

6. Dynamic linking loader

1 Compile and go loader:

It is a link editor or program loader in which the assembler its itself places the assembled instruction directly into the designated memory location.

After completion of assembly process it assigns the starting address of the program to the location counter, and then there is no stop between the compilation, link editing, loading, and execution of the program.



Advantages:

They are simple and easier to implement

Disadvantages:

*. This loader can perform and take only one object program at a time

*. A portion of memory is wasted because of the memory occupied by the assembler for each object program due to that assembler necessary to retranslate the user program every time

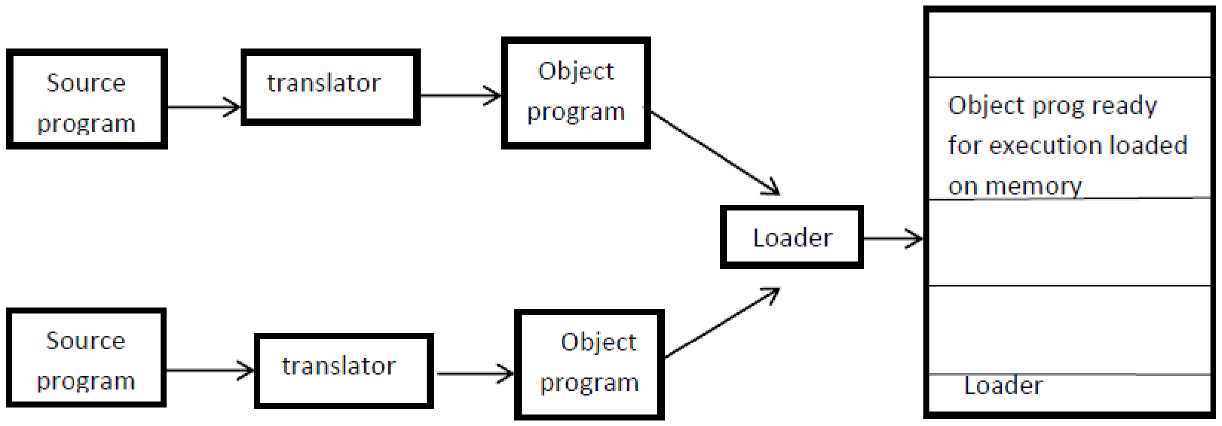
*. It is very difficult to handle multiple segments or subprograms

2 General loader schemes:

The general loader can be solve the draw backs of previous loader it is compiler and go loader.

This loader accepts multiple object programs at a time.

Generally loader is assumed smaller than the assembler so that more memory is available to the user.



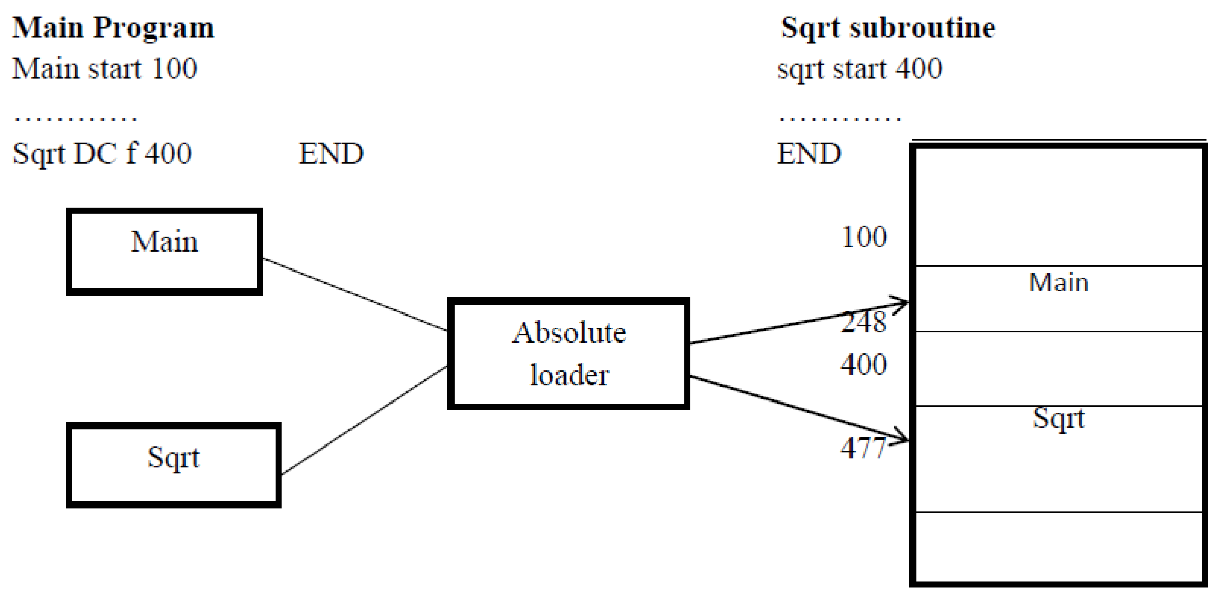
Advantages:

*. In this scheme there is no require for retranslation for each and every program because here we are storing the loader instead of assembler

3 Absolute loader:

- In this scheme the assembler outputs the machine language translation of the source program.
- The data is punched on the cards instead of being placed directly in memory
- The loaders in turns simply accept the machine language text and places into core at the location prescribed by the assembler.

The main program assigned to location 100 to 247 and the subroutine is assigned to the location 400 to 477, if the changes were made to main memory i,e., increased its length more than 300 byte at that time relocation is necessary



The loader functions are accomplished as follows in an absolute loader schemes

1. Allocation by the programmer
2. Linking is also by the programmer
3. Relocation is by assembler
4. Loading by loader

Design of an absolute loader:

We can design an absolute loader we must and should having two types cards

1. Text cards.

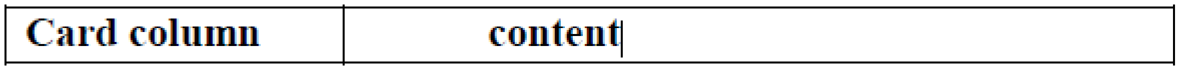
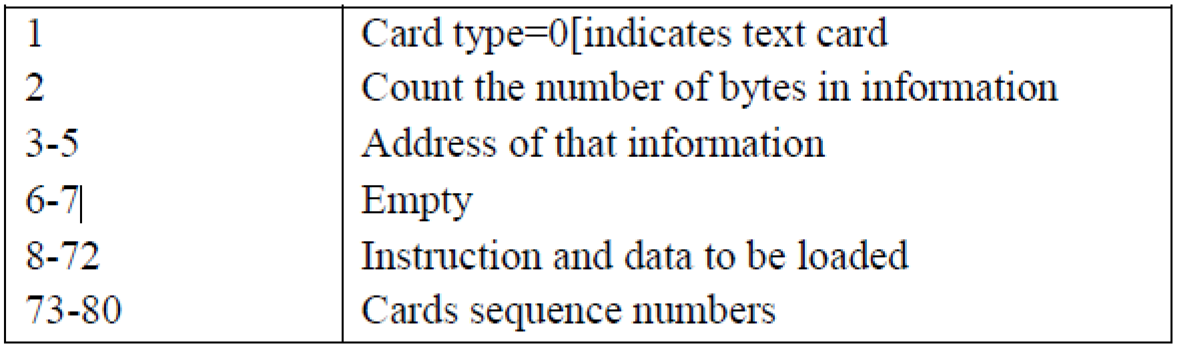
2. Transfer cards.

1 Text cards:

*. This type of card is used to store instructions and data.

*. The capacity of this card is 80bytes.

*. It must convey the machine instructions that assembler has created along with assigned core location.

Card column	content
1	Card type=0[indicates text card
2	Count the number of bytes in information
3-5	Address of that information
6-7	Empty
8-72	Instruction and data to be loaded
73-80	Cards sequence numbers

2 Transfer cards:

These cards must convey the entry point of the program, which is where the

Loader is to transfer the control when all instructions are loaded.

Card column	content
1	Card type=1[indicates transfer card
2	Count type=0
3-5	Address of entry points
6-72	Empty
73-80	Cards sequence numbers

Algorithm for an absolute loader:

Statement 1: start

Statement 2: read header record[first record or first line]

Statement 3: program length

Statement 4: if[it is text card or transfer card]

If it is text card,then store the data and instruction

Else

Transfer instructions

Statement 5: code is in character for then it will convert in to internal representation

Statement 6: read next object program

Statement 7: end

Advantages:

1. It is very simple and easy to implement.

2. More memory available to the user.

3. Multiple segments can be allowed at a time

Disadvantages:

1. In this loader program adjust all internal segment addresses. So that programmers must and should know the memory management and address of the programs.

2. If any modification is done in one segment then starting address is also changed.

3. If there are multiple segments the programmer must and should remember the addressesof all sub-routine.

Sub-routine linkages:

If one main program is transfer to sub program and that sub program also transfer to another program.

The assembler does not know this mechanism [symbolic reference] hence it will declare the error message.

That situation assembler provides two pseudo-op codes. They are

1 EXTRN

2 ENTRY

The assembler will inform the loader that these symbols may be referred by other programs

1 EXTRN:

The EXTRN pseudo op code is used to maintain the reference between 2 or more subroutines.

OR

The assembler pseudo-op code EXTRN followed by a list of symbols indicates that these symbols are defined in other programs but referenced in the present program.

2 ENTRY: The assembler pseudo-op code ENTRY followed by a list of symbols indicates that these symbols are defined in present program and referenced in other program.

ENTRY pseudo-op code is optional which is used to defining entry locations of sub-routines.

Ex:

```
A START                   B START
EXTRN B                   USING *15
…………                      …………
L 15, =A (B)               …………
BALR 14, 15                   BR 14
…………                      ……
END                       END
```

Relocating loaders

- In order to avoid the disadvantages of reassembling in absolute loader another type of loader called relocating loader is introduced.
- BSS [Binary Symbolic Subroutine] is one of the examples of relocating loader.
- The BSS loader allows many procedure segments but only one data segment,
- The assembler assembles each procedure segments independently and then passes to loader the text and information as to relocation

and inter segment reference

- The output of a relocating assembler using a BSS scheme is the object program and information about all other programs it reference
- For each source program the assembler output a text prefixed by transfer vector that consist of address containing names of the subroutines referenced by the source program.
- The assembler would also provide to loader with additional information the length of the entire program and length of the transfers" vector.

This BSS scheme uses RX type instruction format.

OP	R1	X	D

- It is necessary to relocate the address portion of every instruction, the assembler associate a bit with each instruction or address field called relocation bits.
- If relocation bit ==1 the corresponding address field must be relocated. If (rb==0) the field is not relocated.
- The relocation bits are used to solve the problem of relocation, the transfer vector is used to solve the problem of linking and the program length information is used to solve the problem of allocation.

Advantage of relocating loader:

1. Reassembling is not necessary

2. All the function of the loader are implemented only by the BSS loader

Disadvantage of relocating loader:

1. The transfer vector increases the size of the object program in memory.

2. There is no facility for accessing common data segment.

Direct Linking Loader

It is general relocating loader and it is most popular loading scheme presently used.

The main difference between direct linking loader and relocating loader is “relocating loaders one data segment and support multiple procedure segment but only In direct linking loader support multiple procedure segment and also multiple data segments”

Linker: it is system software which is used to link the object programs and that output will be sent to the loader.

There are 4 types of cards available in the direct linking loader. They are

1. ESD-External symbol dictionary

2. TXT-card

3. RLD-Relocation and linking dictionary

4. END-card

1 ESD card:

It contains information about all symbols that are defined in the program but that may be referenced elsewhere.

It contains:

- Reference number
- Symbol name
- Type Id
- Relative location
- Length

There are again ESD cards classified into 3 types of mnemonics. They are:

1. **SD : [Segment Definition]:** It refers to the segment definition [01]

2. **LD : It refers to the local definition** [ENTRY] [02]

3. **ER: [External reference]** it refers to the external reference they are used in the [EXTRN] pseudo op code [03]

2 TXT Card: It contains the actual information are text which are already translated.

3 RLD Card: This card contains information about location in the program whose contexts depends on the address at which the program is placed.

In this we are used “+” and “–“sign, when we are using the “+“ sign then no need of relocation, when we are using “-“sign relocation is necessary.

The format of RLD contains:

1. Reference number

2. Symbol

3. Flag

4. Length

5. Relative location

4 END Card: It indicates end of the object program.

Note: The size of the above 4 cards is 80 bytes

Design of direct linking loader:

Here we are taking PG1 and PG2 are two programs

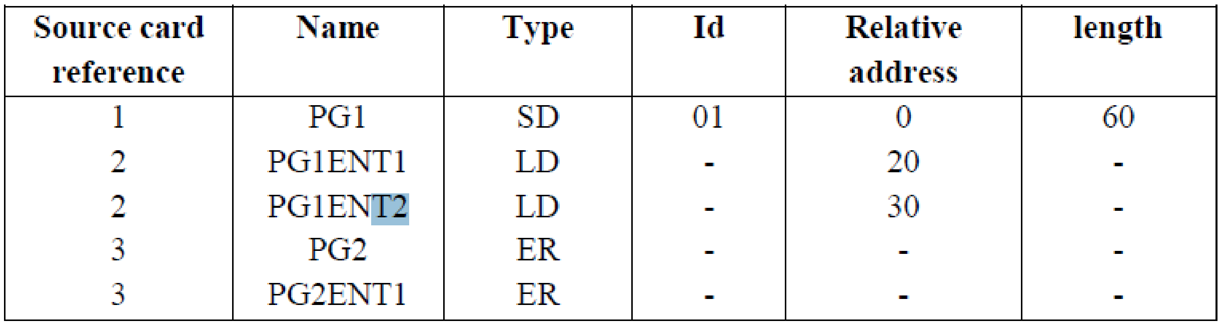
The relative address and secure code of above two programs is written in the below.

ESD Cards:

In a ESD card table contains information necessary to build the external symbol dictionary or symbols table

In the above source code the symbols are PG1, PG1ENT2, PG2, and PG2ENT1

Format of ESD Card for PG1:



Source card reference	Name	Type	Id	Relative address	length
1	PG1	SD	01	0	60
2	PG1ENT1	LD	-	20	-
2	PG1ENT2	LD	-	30	-
3	PG2	ER	-	-	-
3	PG2ENT1	ER	-	-	-

- Here, the PG1 is the segment definition it means, the header of program1
- PG1ENT1 and PG1ENT2 those are the local definition of program1, so that the we are using the type LD.
- PG2 and PG2ENT1 those are using the EXTRN pseudo op code, so that we are using the type ER

Text card for PG1:

The format of card will be

Source card reference	Relative address	Content	Comments
6	40-43	20	
7	44-47	45	=30+15
8	48-51	7	=30-20-3
9	52-55	0	Unknown to PG1
10	56-60	-16	-20+4

6= A(PG1ENT1)=20

7=A (PG1ENT2+15)=30+15=45

8=A (PG1ENT2-PG1ENT1-3)=30-20-3=7

9=A (PG2)=0

10=A (PG2ENT1+PG2-PG1ENT1+4)=0+0-20+4= -16

RLD Card Format for PG1:

Source card reference address	ESD ID	Length [bytes]	Flag + or -	relative
6	02	4	+	40
7	02	4	+	44
9	03	4	+	52
10	02	4	+	56
10	03	4	+	56
10	02	4	-	56

ESD Card for program2 [PG2]:

Source Card reference	Name	Type	Id	ADDR	Length
12	PG2	SD	01	0	36
13	PG2ENT1	LD	-	16	-
14	PG1ENT1	ER	03	-	-
14	PG1ENT2	ER	03	-	-

Text Card for PG2:

Source card reference	Relative address	content
16	24-27	0
17	28-31	15
18	32-35	-3

16=A (PG1ENT1) =0
17=A (PG1ENT2+15) =0+15=15
18=A (PG1ENT2-PG1ENT1-3) =0-0-3=-3

RLD Card for PG2:

Source card reference	ESD ID	Length[flag] [bytes]	Flag + or -	Relative address
16	01	4	+	24
17	-	4	+	28
18	03	4	+	32
18	03	4	-	32

Specification of data structure:

1 Pass1 database:

1. Input object decks
2. The initial program load addresses [IPLA]: The IPLA supplied by the programmer or operating system that specifies the address to load the first segment.
3. Program load address counter [PLA]: It is used to keep track of

each segments assigned location
4. **Global external symbol table [GEST]**: It is used to store each external symbol and its corresponding assigned core address
5. A copy of the input to be used later by pass2
6. A printed listing that specifies each external symbol and its assigned value

2 Pass2 database:

1. A copy of object program is input to pass2
2. The initial program load address [IPLA]
3. The program load address counter [PLA]
4. A table the global external symbol table [GEST]
5. The local external symbol array [LESA]: which is used to establish a correspondence between the ESD ID numbers used on ESD and RLD cards and the corresponding External symbols , Absolute address value

Format of data bases:

Object deck:
The object deck contains 4 types of cards

1 ESD Card format:

Source card reference	Name	Type	ID	Relative address	length

Type	**Hexa-decimal**
SD	01
LD	02
ER	03

2 TEXT Card:

Source card reference address	Relative address	content

3 RLD Card:

Source card references	ESD ID	Length	Flag + or -	Relative address

Note: The length of each card is 80-bytes

4 Global External Symbol Table [GEST]:

It is used to store each external symbol and its corresponding core address.

External symbol [8 bytes] character	Assigned core [4 bytes] address decimal
"PG1bbbbb" "PG1ENT1b"	104 124

5 Local external symbol array[LESA]:

The external symbol is used for relocation and linking purpose. This is used to identify the RLD card by means of an ID number rather than the symbols name. The ID number must match an SD or ER entry on the ESD card

Assigned core address of corresponding symbol [4 bytes]
104
124
134
….
….

This technique saves space and also increases the processing speed.

Pass1 Algorithm and Flowchart:

The purpose of pass1 is to assign location to each segment and also finding the values of all symbols

1. Initial program load address [PLA] it's set to the initial program load address [IPLA]
2. Read the object card
3. Write copy of card for pass2
4. The card can be any one of the following type
    - i) Text card or RLD card- there is no processing required during pass1 and then read the next card
    - ii) ESD card is processed based on the type of the external symbols
        - a) SD is read the length field LENGTH from the card is temporarily saved in the variable SLENGTH. The value, VALUE to assign to this symbol is set to the current value of the PLA. The symbol and its assigned value are then stored in the GEST. If the symbol already existed in the GEST then this is an error.

        - b) The symbol and its value are printed as part of the load map. LD is read the value to be assigned is set to the current PLA+ the relative address [ADDR]. The ADDR indicates on the ESD card.

c) ER symbols do not require any processing duration pass1.

iii) When an END card is encountered the program load address is incremented by length of the segment and saved on SLENGTH.

iv) EOF card is read pass1 is completed and control transfer to pass2

Pass2 Algorithm and Flowchart:

1. If an address is specified in the END card then that address is used as the executed start address otherwise, the execution will begin from the first segment

2. In pass2 the five cards are read one by one described as follows At the beginning of pass2 the program load address is initialized as in pass1 and the execution start address [EXADDR] is set to IPLA.

i. ESD card

a) SD type=>the length of the segment is temporarily saved in the variable

SLENGTH. The LESA [ID] is set to the current value of the PLA.

b) LD type=> Does not requires any processing during pass2

c) ER type=>the guest is searched for a match with the ER symbols. If it is not found then there is an error. If found in the GEST its value is extracted and the corresponding LESA entry is set.

ii. Text card: The text card is copied from the card to the appropriate relocated memory location [PLA+ADDR]

iii. RLD card: The value to be used for relocation and linking is executed from the

LESA [ID] as specified by the ID field.

Depending upon the flag values is either added to or subroutine from the address

constants

iv. END card: If an execution start address is specified on the END card it is saved

in a variable EXADDR. The PLA is incremented by length of the segment and

saved in SLENGTH, becoming the PLA for the next segment[PLA=PLA+SLENGTH]

v. EOF card: The loader transfers control to the loaded program at the address
specified by the current contents of the execution address variable [EXADDR].