

# Module-4

Loaders and Linkers

# Syllabus

4	<b>Loaders and Linkers</b>	Introduction, functions of loaders, Relocation and Linking concept, Different loading schemes: Relocating loader, Direct Linking Loader, Dynamic linking and loading.
---	----------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------

# Loader

- The loader is a program, which accepts the object program decks, prepares these programs for execution by the computer and initiates the execution

# Functions of a Loader:

- Allocation
- Linking
- Relocation
- Loading

# Functions of a Loader:

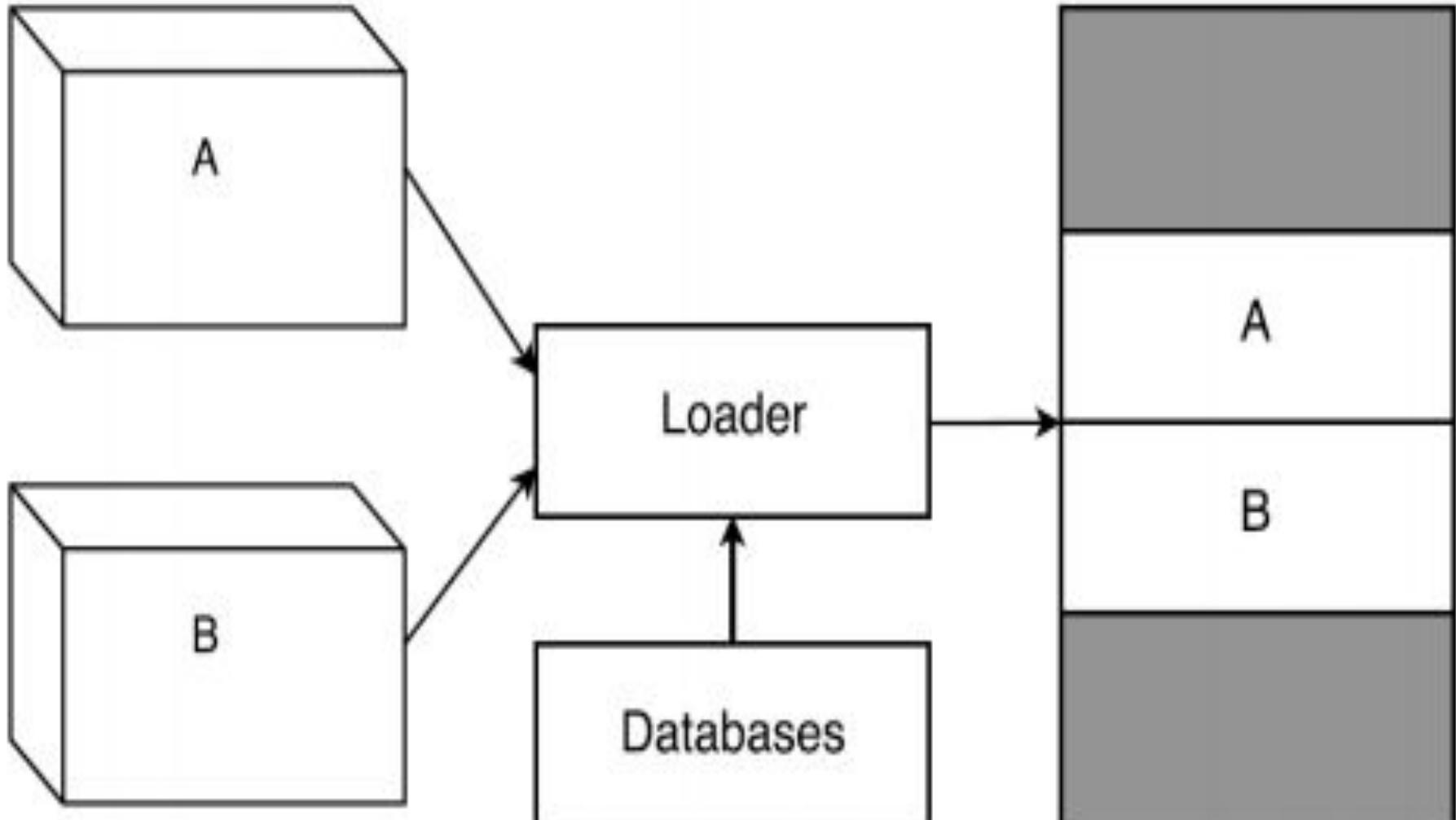
- **Allocation:** Allocate space in memory for the programs
- **Linking:** Resolve symbolic references between object decks.
- **Relocation:** Adjust all address dependent locations, such as address constants, to correspond to the allocated space.
- **Loading:** Physically place the machine instructions and data into memory.

# General Loading Scheme

- In General loader, the source program is converted to object program by some translator (assembler).
- The loader accepts these object modules and puts machine instruction and data in an executable form at their assigned memory.
- The loader occupies some portion of main memory

# General Loading Scheme

A and B are  
object  
programs



Programs loaded in  
memory for execution

# Advantages:

1. No need of retranslation each time while running program
2. No wastage of Memory
3. Possible to write source program with multiple programs and multiple languages

# Linking:

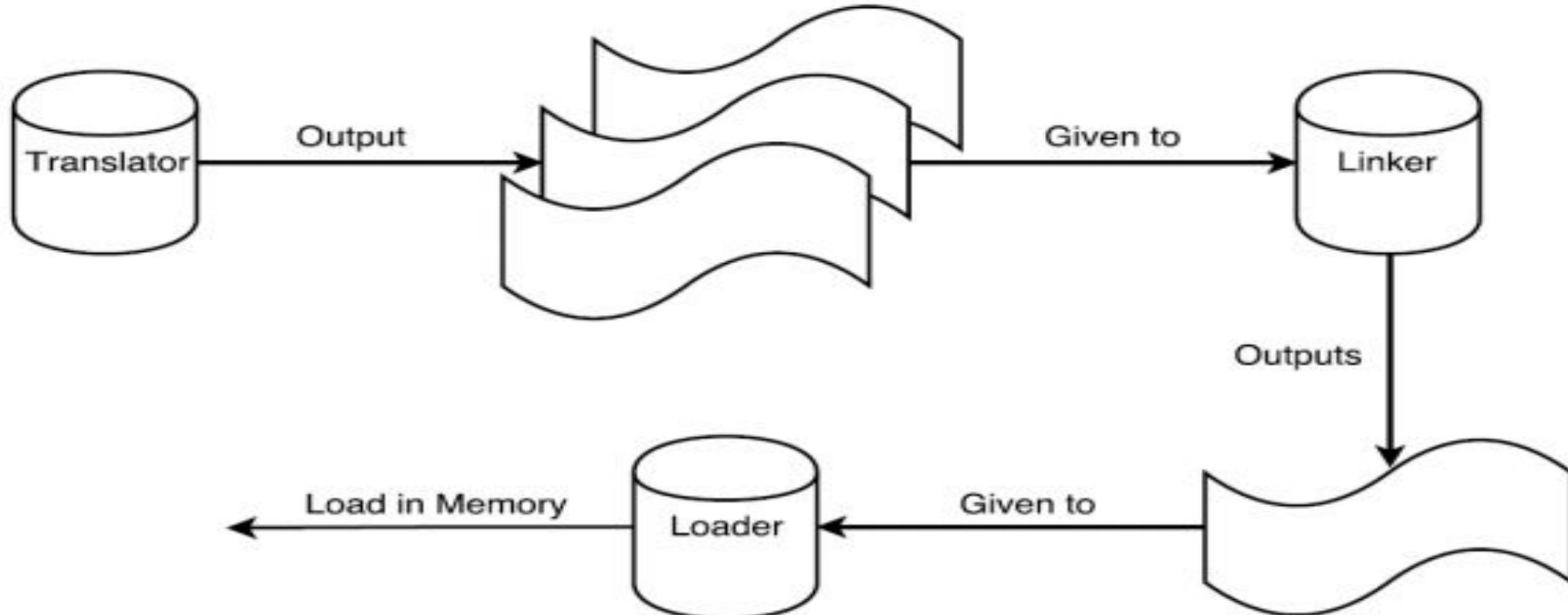
The execution of program can be done with the help of following steps:

1. **Translation:** Translation of the program (done by assembler or compiler)
2. **Linking:** Linking of the program with all other programs, which are needed for execution. This also involves preparation of a program called load module.
3. **Loading:** Loading of the load module prepared by linker to some specified memory location.

# Process of Linking:

- The output of translator is a program called object module.
- The linker processes these object modules, binds with necessary library routines and prepares a ready to execute program.
- Such a program is called binary program.
- The "binary program also contains some necessary information about allocation and relocation.
- The loader then loads this program into memory for execution purpose.

# Process of Linking:



**Process of Linking a Program**

# Tasks of a Linker

## 1. To prepare Single Load Module:

- Adjust all addresses and subroutine preferences with respect to the offset location

## 2. To prepare a load module:

- Concatenate all the object modules and adjust the
  - a) operand addresses references
  - b) External references to the offset location

## 3. To prepare ready to execute module:

- Copy the binary machine instructions and the constant data

# Segment

- A unit of information that can be treated as an entity
- It can be a program or data
- A segment corresponds to a single source deck or object deck
- It is possible to have multiple programs or data segments in a single source deck

# Loader Schemes

1. General Loader Scheme
2. Compile and Go Loaders
3. Absolute Loaders
4. Subroutine Linkages
5. Relocating Loaders
6. Direct Linking Loaders
7. Other Loader Schemes:
  1. Dynamic Loading
  2. Dynamic Linking

# General Loader Scheme

- The loader accepts the assembled machine instructions, data, and other information present in the object format, and places machine instructions and data in core in an executable computer form.
- The loader is assumed to be smaller than the assembler, so that more memory is available to the user.
- A further advantage is that reassembly is no longer necessary to run the program at a later date.

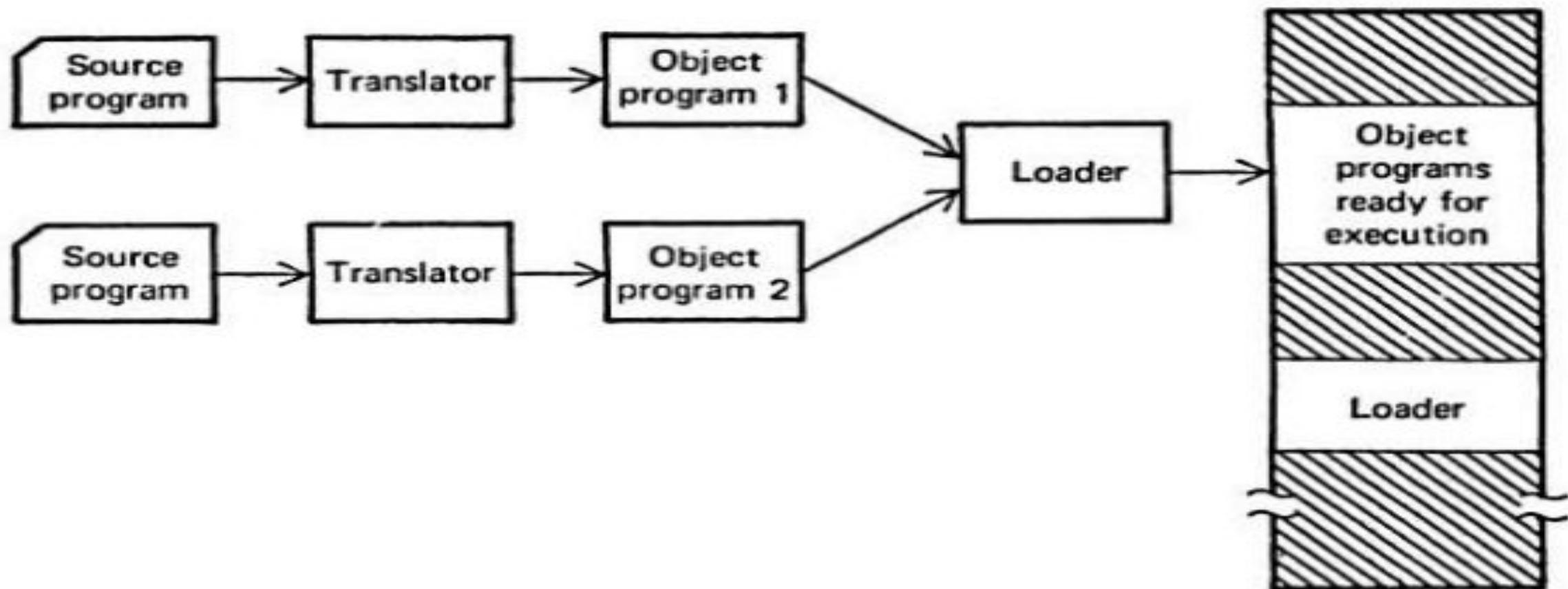
# General Loader Scheme

16

- Finally, if all the source program translators (assemblers and compilers) produce compatible object program deck formats and use compatible linkage conventions, then it is possible to write subroutines in several different languages
- This is because the object decks to be processed by the loader will all be in the same "language" (machine language).

# General Loader Scheme

17



# Relocating Loader

18

Advantage:

- To avoid possible reassembling of all subroutines when a single subroutine is changed
- To perform the tasks of allocation and linking for the programmer

# Relocating Loader

19

## Relocation

- The process of updating the addresses used in the address sensitive instructions of a program
- The object code is executed after loading at storage locations.

# Relocating Loader

20

- The addresses of such object code will be specified only after the assembly process is over.
- Therefore, after loading,  
$$\text{Address of object code} = \text{Only Address of object code} + \text{Relocation constant}$$

# Relocating Loader

21

## Example: Binary Symbolic Subroutine (BSS) Loader

- The BSS loader allows many procedure segments
- It allows only one data segment (common segment)
- The assembler assembles each procedure segment independently
- Then it passes the text and information to the loader as relocation and intersegment references.

# Relocating Loader

22

- The output of a relocating assembler using a BSS scheme is the object program and information about all other programs it references.
- In addition, there is information (relocation information) for locations in this program that need to be changed if it is to be loaded in an arbitrary place in core, i.e .. the locations which are dependent on the core allocation.
- For each source program the assembler outputs a text (machine translation of the program) prefixed by a transfer vector that consists of addresses containing names of the subroutines referenced by the source program

# Relocating Loader

23

## Process of Relocating:

- For example, if a square Root Routine (SQRT) was referenced and was the first subroutine called, the first location in the transfer vector would contain the symbolic name SQRT.
- The statement calling SQRT would be translated into a transfer instruction indicating a branch to the location of the transfer vector associated with SQRT
- The assembler would also provide the loader with additional information, such as the length of the entire program and the length of the transfer vector portion.

# Relocating Loader

24

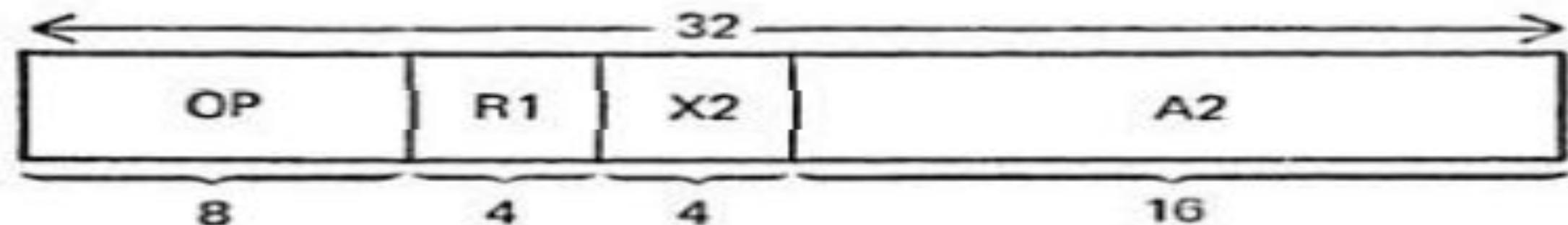
- After loading the text and the transfer vector into core, the loader would load each subroutine identified in the transfer vector.
- It would then place a transfer instruction to the corresponding subroutine in each entry in the transfer vector.
- Thus, the execution of the call SQRT statement would result in a branch to the first location in the transfer vector, which would contain a transfer instruction to the location of SQRT.

# Relocating Loader

25

## Application

- The BSS loader scheme is often used on computers with a fixed-length direct address instruction format.
- For example, if the format of the 360 RX instruction were:



- where A2 was the 16 bit absolute address of the operand, this would be a direct address instruction format.
- Such a format works if there are less than  $2^{16} = 65,536$  bytes of storage

# Relocating Loader

26

## Problem

Since it is necessary to relocate the address portion of every instruction, computers with a direct address instruction format have a much more severe relocation problem

## Solution

### Use of Relocation bits

- Relocation bits are associated with each address in the program which tells whether it needs to be relocated or not.
- 0 means NO and 1 means YES

## Solution

- The assembler associates a bit with each Instruction or address field.
- If this bit equals one, then the corresponding address field must be relocated otherwise the field is not relocated.
- These relocation indicators, are known as relocation bits and are included in the object deck.

# Relocating Loader

28

<i>Source program</i>				Program length = 48 bytes Transfer vector = 8 bytes		
				<i>Rel.</i> <i>addr.</i>	<i>Relocation</i>	<i>Object code</i>
MAIN	START			0	00	'SQRT'
	EXTRN	SQRT		4	00	'ERRb'
	EXTRN	ERR		8	01	ST 14,36
ST	14,SAVE	Save return address		12	01	L 1,40
L	1,=F'9'	Load test value		16	01	BAL 14,0
BAL	14,SQRT	Call SQRT		20	01	C 1,44
C	1,=F'3'	Compare answer		24	01	BC 7,4
BNE	ERR	Transfer to ERR		28	01	L 14,36
L	14,SAVE	Get return address		32	0	BCR 15,14
BR	14	Return to caller		34	0	(Skipped for alignment)
SAVE	DS	F	Temp. loc.	36	00	(Temp location)
END				40	00	9
				44	00	3

FIGURE 5.5 Assembly of program for “direct-address” 360



# Relocating Loader

29

1. The figure illustrates a simple assembly language program written for a hypothetical "direct-address" 360 that uses a BSS loader.
2. It supposedly calls the SQRT subroutine to get the square root of 9.
3. If the result is not 3, it transfers to a subroutine called ERR.
4. Since this is a direct-address computer, there is no base register field in the object code and no need for a USING pseudo-op in the source program.



5. The EXTRN pseudo op identifies the symbols SQRT and ERR as the names of other subroutines
6. Since the locations of these symbols are not defined in this subroutine, they are called external symbols.
7. For each external symbol the assembler generates a four byte fullword at the beginning of the program, containing the EBCDIC characters for the symbol
8. These extra words are called **transfer vectors**.



9. Every reference to an external symbol is assigned the address of the corresponding transfer vector word
10. In addition, for every halfword (two bytes) in the program, the assembler produces a separate relocation bit.

For example, the assembled instruction ST 14,36 is assigned relocation bits 01 since the first halfword contains the op-code, register field, and index field which should not be relocated but the second halfword contains the relative address 36, which must be relocated.



# Relocating Loader

32

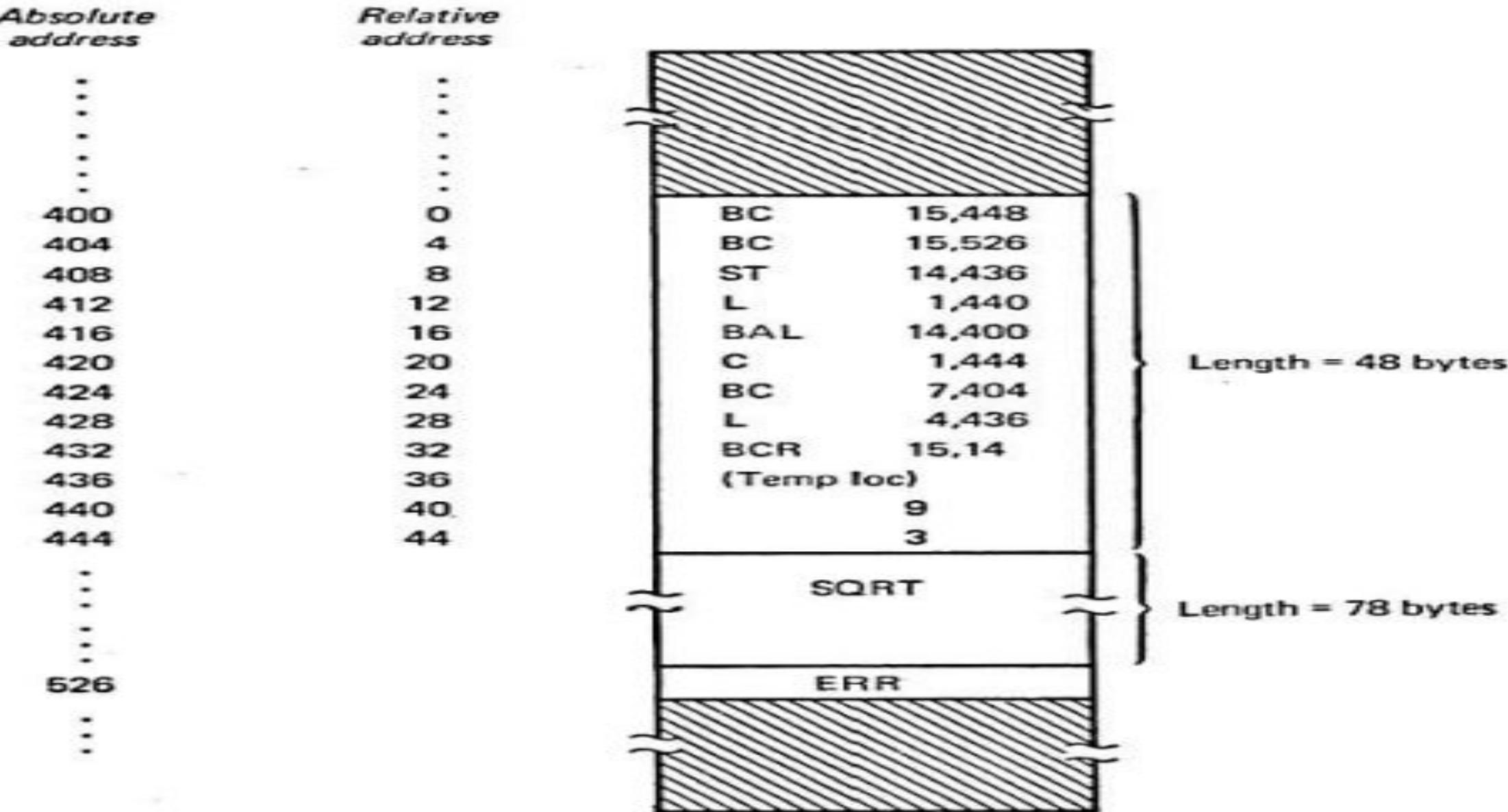


FIGURE 5.6 BSS loading of programs for "direct access" 360

# Relocating Loader

33

1. The figure illustrates the contents of memory after the programs have been loaded by the BSS loader.
2. Based upon the relocation bits, the loader has relocated the address fields to correspond to the allocated address of MAIN which is 400.
3. Using the program length information, the loader placed the sub routines SQRT and ERR at the next available locations which were 448 and 526, respectively

# Relocating Loader

34

4. Finally, the transfer vector words were changed to contain branch instructions to the corresponding subroutines

Thus, the four functions of the loader (allocation, linking, relocation, and loading) were all performed automatically by the BSS loader.

# Disadvantages

35

1. The transfer vector linkage is only useful for transfers, and is not well-suited for loading or storing external data (data located in another procedure segment).
2. The transfer vector increases the size of the object program in memory.
3. The BSS loader, processes procedure segments but does not facilitate access to data segments that can be shared.

(This last shortcoming is overcome in many BSS loaders by allowing one common data segment, often called COMMON. This facility is usually implemented by extending the relocation bits scheme to use two bits per halfword address field: if the bits are 01, the half word is relocated relative to the procedure segment, and if they are 10, it is relocated relative to the address of the single common data segment. If the bits are 00 or 11, the halfword is not relocated.)

## DIRECT LINKING LOADER

- A direct-linking loader is a general relocatable loader, and is perhaps the most popular loading scheme presently used
- The direct-linking loader has the advantage of allowing the programmer multiple procedure segments and multiple data segments and of giving him complete freedom in referencing data or instructions contained in other segments.

## DISADVANTAGE

- It is necessary to allocate, relocate, link, and load all of the subroutines each time in order to execute a program
- Since there may be tens and often hundreds of subroutines involved, this loading process can be extremely time-consuming.
- Furthermore, even though the loader program may be smaller than the assembler, it does absorb a considerable amount of space

## DYNAMIC LOADING

- If the total amount of core required by all the subroutines exceeds the amount available, then it is problem
- There are several hardware techniques, such as paging and segmentation, that attempt to solve this problem;
- For loaders: A scheme is applied which uses **Binders**

## LOADING PROCESS

Loading Process can be divided into two separate process:

- Binder: Performs the function of allocation, relocation and linking
- Module Loader: Performs the function of loading

## LOADING PROCESS

- A **binder** is a program that performs the same functions as the direct-linking loader in "binding" subroutines together, but rather than placing the relocated and linked text directly into memory, it outputs the text as a file or card deck.
- This output file is in a format ready to be loaded and is typically called a **load module**.
- The module loader has to physically load the module into core.

## DYNAMIC LOADING

- In each of the previous loader schemes we have assumed that all of the subroutines needed are loaded into core at the same time.
- If the total amount of core required by all these subroutines exceeds the amount available, then it is a problem

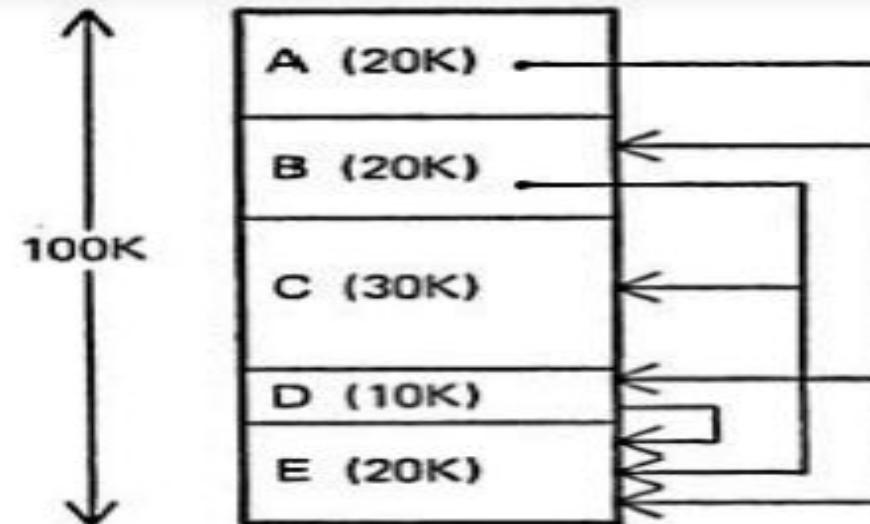
## DYNAMIC LOADING

- There are several hardware techniques, such as paging and segmentation, that attempt to solve this problem
- A dynamic loading schemes based upon the concept of a binder prior to loading can solve this problem

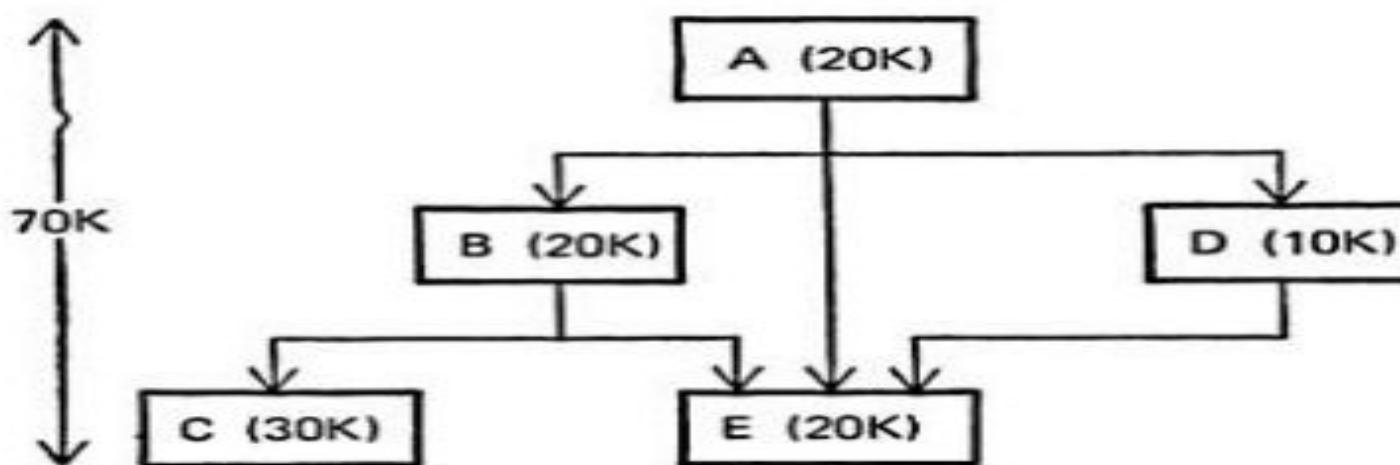
## DYNAMIC LOADING

- Usually the subroutines of a program are needed at different times
- By explicitly recognizing which subroutines call other subroutines it is possible to produce an overlay structure that identifies mutually exclusive subroutines.

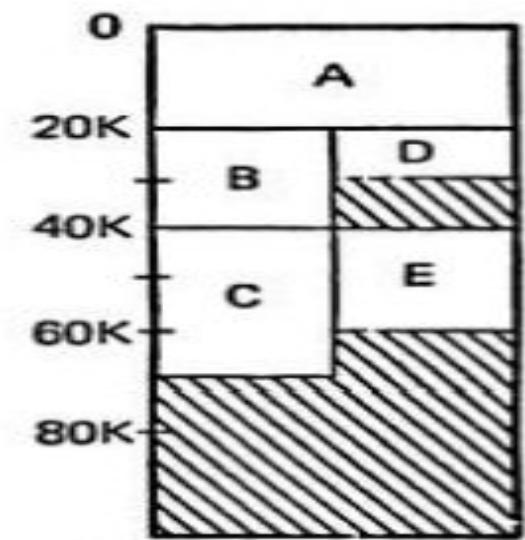
# DYNAMIC LOADING



(a) Subroutine calls between the procedures



(b) Overlay structure



(c) Possible storage assignment of each procedure

## DYNAMIC LOADING

- In the figure(a): A program consisting of five subprograms (A,B,C,D and E) that require 100K bytes of core.
- The arrows indicate that subprogram A only calls B, D and E
- Subprogram B only calls C and E; subprogram D only calls E;
- Subprograms C and E do not call any other routines.

## DYNAMIC LOADING

- In Figure 5 .9b the interdependencies between the procedures are highlighted
- Procedures B and D are never in use at the same time; neither are C and E.
- If we load only those procedures that are actually to be used at any particular time, the amount of core needed is equal to the longest path of the overlay structure.
- This happens to be 70K for the example in Figure 5.9b - procedures A, B, and C.
- Figure 5.9c illustrates a storage assignment for each procedure consistent with the overlay structure.<sup>46</sup>

## OVERLAY STRUCTURE

- In order for the overlay structure to work it is necessary for the module loader to load the various procedures as they are needed.
- There are many binders capable of processing and allocating an overlay structure.
- The portion of the loader that actually intercepts the "calls" and loads the necessary procedure is called the overlay supervisor or simply the flipper.
- This overall scheme is called dynamic loading or load-on-call (LOCAL).

## DISADVANTAGE

- If a subroutine is referenced but never executed, the loader would still incur the overhead of linking the subroutine.
- Furthermore, all of these schemes require the programmer to explicitly name all procedures that might be called.

## DYNAMIC LINKING

- This is a mechanism by which loading and linking of external references are postponed until execution time.
- The assembler produces text, binding, and relocation information from a source language deck.
- The loader loads only the main program.

## DYNAMIC LINKING

- If the main program should execute a transfer instruction to an external address, or should reference an external variable, then the loader is called.
- Only then is the segment containing the external reference is loaded.

# ADVANTAGE

- No overhead is incurred unless the procedure to be called or referenced is actually used.
- The system can be dynamically reconfigured.

# DYNAMIC LINKING VS BINDERS

	<i><b>Dynamic Linking Loader</b></i>	<i><b>Linkage Editor (Binder)</b></i>
1	<i>A Linking Loader performs linking and relocation at run time and directly loads the linked program into the memory.</i>	<i>A Linkage Editor performs linking and some relocation operations prior to load time and write the linked program to an executable for later execution.</i>
2	<i>It resolves external references every time the program is executed.</i>	<i>External references are resolved and library searching is performed only once, when the program is “link-edited”.</i>
3	<i>Executable image is not generated; the linked program is directly loaded into memory.</i>	<i>The linked programs is written into an executable image, which is later given to a relocating loader for execution.</i>
4	<i>It is more suitable for development and testing environment where the size of program is comparatively smaller.</i>	<i>It is suitable when the program has to be re-executed many times without being reassembled.</i>
5	<i>Produces output into memory.</i>	<i>Produces output on disc.</i>
6	<i>E.g. Direct-Linking Loader</i>	<i>E.g. MS-DOS Linker</i>



## MODULE-4



LOADERS AND LINKERS

# LOADER SCHEMES

1. Compile and Go  
Loaders

2. General Loader  
Scheme

3. Absolute Loaders

4. Subroutine Linkages

5. Relocating Loaders

6. Direct Linking  
Loaders

7. Other Loader  
Schemes: a) Dynamic  
Loading b) Dynamic  
Linking

## DIRECT LINKING LOADER

- A direct-linking loader is a general relocatable loader, and is perhaps the most popular loading scheme presently used
- The direct-linking loader has the advantage of allowing the programmer multiple procedure segments and multiple data segments and of giving him complete freedom in referencing data or instructions contained in other segments.

## DISADVANTAGE

- It is necessary to allocate, relocate, link, and load all of the subroutines each time in order to execute a program
- Since there may be tens and often hundreds of subroutines involved, this loading process can be extremely time-consuming.
- Furthermore, even though the loader program may be smaller than the assembler, it does absorb a considerable amount of space

## DYNAMIC LOADING

- If the total amount of core required by all the subroutines exceeds the amount available, then it is problem
- There are several hardware techniques, such as paging and segmentation, that attempt to solve this problem;
- For loaders: A scheme is applied which uses **Binders**

# LOADING PROCESS

Loading Process can be divided into two separate process:

- Binder: Performs the function of allocation, relocation and linking
- Module Loader: Performs the function of loading

## LOADING PROCESS

- A **binder** is a program that performs the same functions as the direct-linking loader in "binding" subroutines together, but rather than placing the relocated and linked text directly into memory, it outputs the text as a file or card deck.
- This output file is in a format ready to be loaded and is typically called a **load module**.
- The module loader has to physically load the module into core.

## DYNAMIC LOADING

- In each of the previous loader schemes we have assumed that all of the subroutines needed are loaded into core at the same time.
- If the total amount of core required by all these subroutines exceeds the amount available, then it is a problem

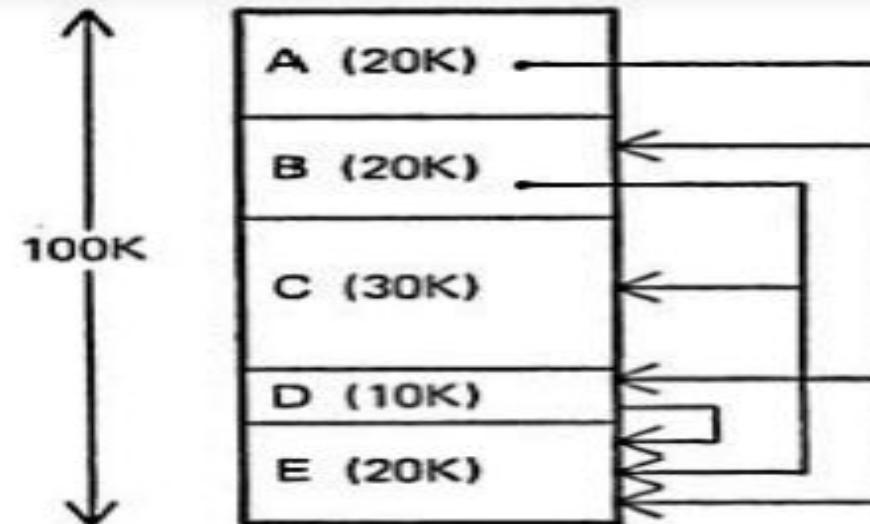
## DYNAMIC LOADING

- There are several hardware techniques, such as paging and segmentation, that attempt to solve this problem
- A dynamic loading schemes based upon the concept of a binder prior to loading can solve this problem

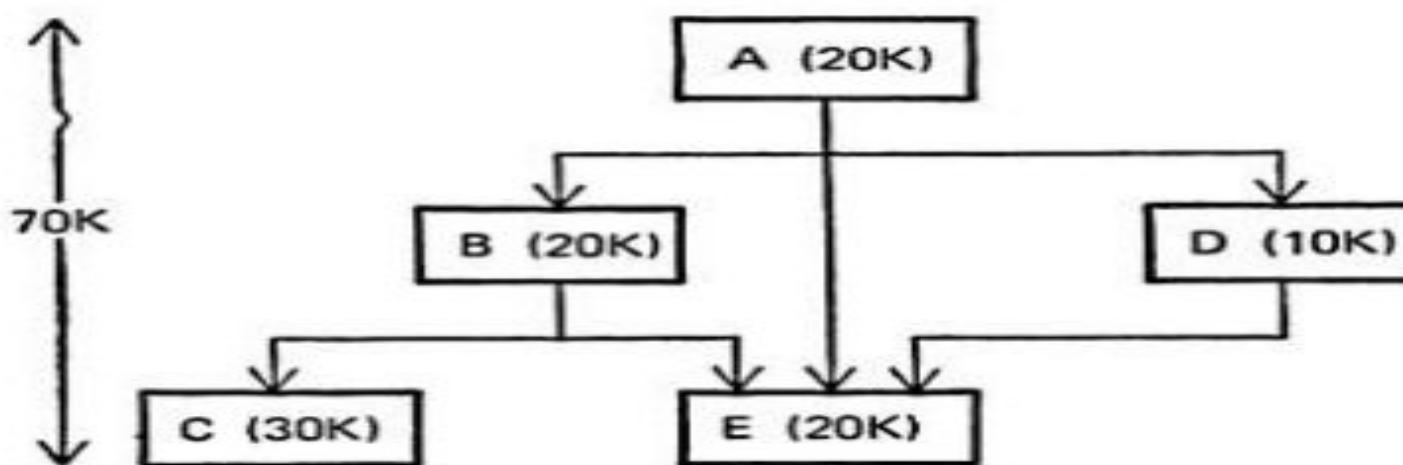
## DYNAMIC LOADING

- Usually the subroutines of a program are needed at different times
- By explicitly recognizing which subroutines call other subroutines it is possible to produce an overlay structure that identifies mutually exclusive subroutines.

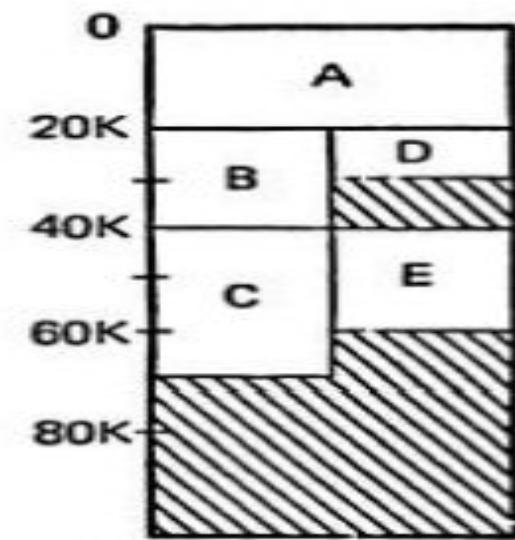
# DYNAMIC LOADING



(a) Subroutine calls between the procedures



(b) Overlay structure



(c) Possible storage assignment of each procedure

## DYNAMIC LOADING

- In the figure(a): A program consisting of five subprograms (A,B,C,D and E) that require 100K bytes of core.
- The arrows indicate that subprogram A only calls B, D and E
- Subprogram B only calls C and E; subprogram D only calls E;
- Subprograms C and E do not call any other routines.

## DYNAMIC LOADING

- In Figure 5 .9b the interdependencies between the procedures are highlighted
- Procedures B and D are never in use at the same time; neither are C and E.
- If we load only those procedures that are actually to be used at any particular time, the amount of core needed is equal to the longest path of the overlay structure.
- This happens to be 70K for the example in Figure 5.9b - procedures A, B, and C.
- Figure 5.9c illustrates a storage assignment for each procedure consistent with the overlay structure. 13

## OVERLAY STRUCTURE

- In order for the overlay structure to work it is necessary for the module loader to load the various procedures as they are needed.
- There are many binders capable of processing and allocating an overlay structure.
- The portion of the loader that actually intercepts the "calls" and loads the necessary procedure is called the overlay supervisor or simply the flipper.
- This overall scheme is called dynamic loading or load-on-call (LOCAL).

## DISADVANTAGE

- If a subroutine is referenced but never executed, the loader would still incur the overhead of linking the subroutine.
- Furthermore, all of these schemes require the programmer to explicitly name all procedures that might be called.

## DYNAMIC LINKING

- This is a mechanism by which loading and linking of external references are postponed until execution time.
- The assembler produces text, binding, and relocation information from a source language deck.
- The loader loads only the main program.

## DYNAMIC LINKING

- If the main program should execute a transfer instruction to an external address, or should reference an external variable, then the loader is called.
- Only then is the segment containing the external reference is loaded.

# ADVANTAGE

- No overhead is incurred unless the procedure to be called or referenced is actually used.
- The system can be dynamically reconfigured.

# DYNAMIC LINKING VS BINDERS

	<i><b>Dynamic Linking Loader</b></i>	<i><b>Linkage Editor (Binder)</b></i>
1	<i>A Linking Loader performs linking and relocation at run time and directly loads the linked program into the memory.</i>	<i>A Linkage Editor performs linking and some relocation operations prior to load time and write the linked program to an executable for later execution.</i>
2	<i>It resolves external references every time the program is executed.</i>	<i>External references are resolved and library searching is performed only once, when the program is “link-edited”.</i>
3	<i>Executable image is not generated; the linked program is directly loaded into memory.</i>	<i>The linked programs is written into an executable image, which is later given to a relocating loader for execution.</i>
4	<i>It is more suitable for development and testing environment where the size of program is comparatively smaller.</i>	<i>It is suitable when the program has to be re-executed many times without being reassembled.</i>
5	<i>Produces output into memory.</i>	<i>Produces output on disc.</i>
6	<i>E.g. Direct-Linking Loader</i>	<i>E.g. MS-DOS Linker</i>

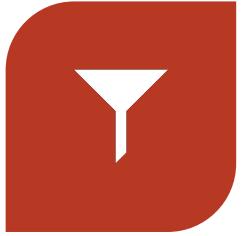


# MODULE-4

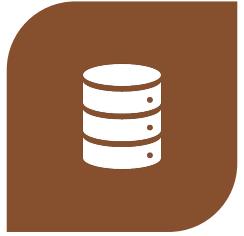
## LOADERS AND LINKERS

# LOADER SCHEMES

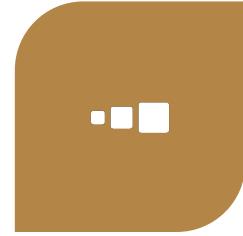
2



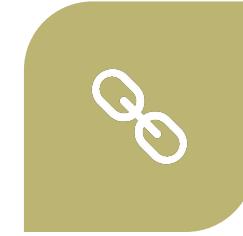
1. COMPILE AND GO  
LOADERS



2. GENERAL LOADER  
SCHEME



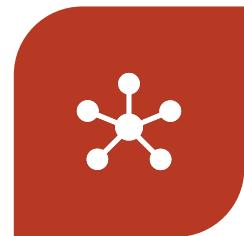
3. ABSOLUTE LOADERS



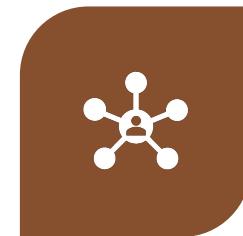
4. SUBROUTINE  
LINKAGES



5. RELOCATING  
LOADERS



6. DIRECT LINKING  
LOADERS



7. OTHER LOADER  
SCHEMES: A) DYNAMIC  
LOADING B) DYNAMIC  
LINKING

# Assemble (or Compile) and Go Loader

3

- In this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address.
- That means the assembler runs in one part of memory and the assembled machine instructions and data is directly put into their assigned memory locations.
- After completion of assembly process, assign starting address of the program to the location counter.
- This loading scheme is also called as “assemble and go”.

# Assemble (or Compile) and Go Loader

4

## Advantage:

- Easy to implement
- Assembler runs in one part of memory and assembled instructions are directly loaded into core
- No extra procedures are involved

# Assemble (or Compile) and Go Loader

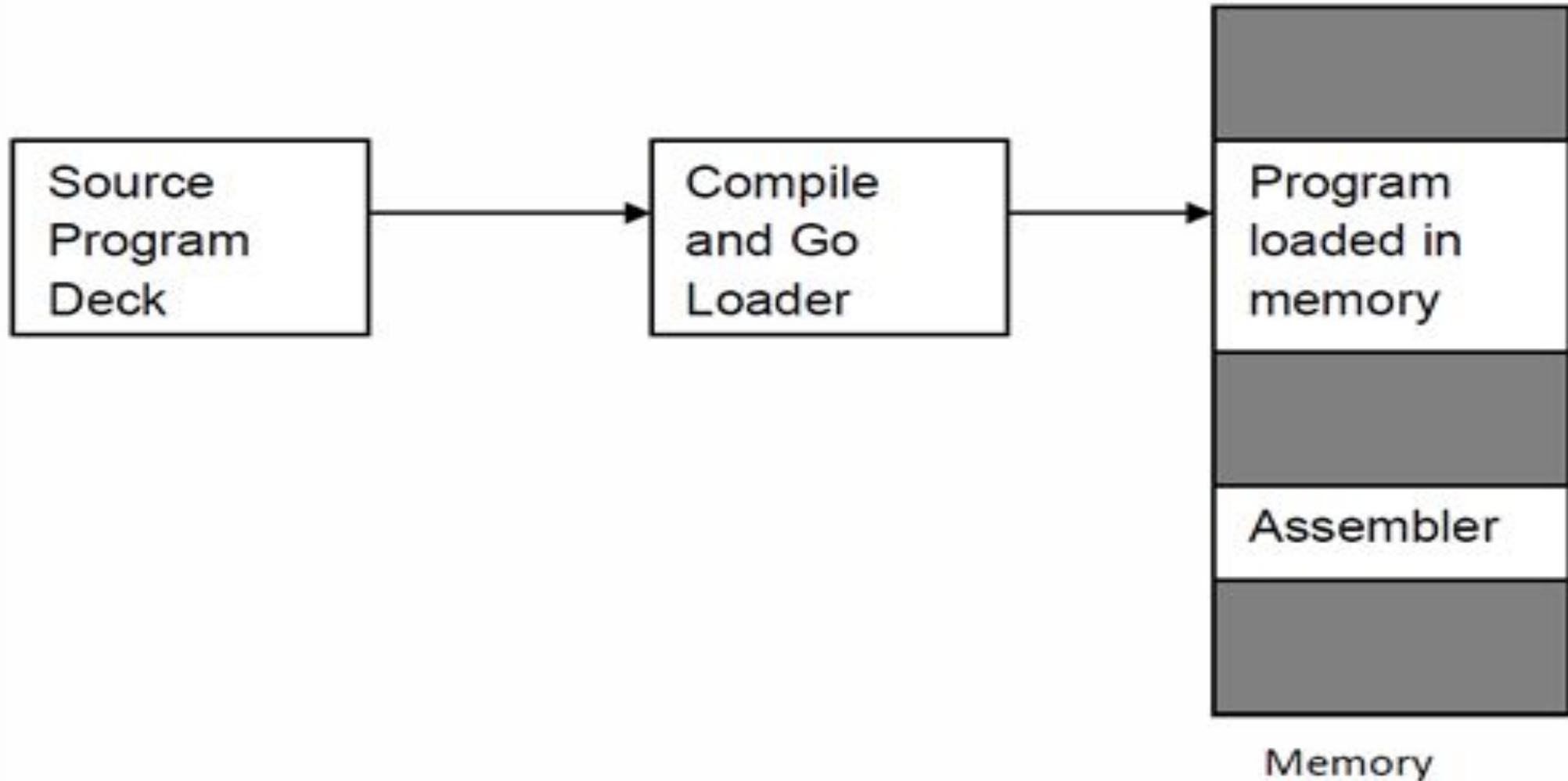
5

## Disadvantage:

- Wastage of memory as assembler and loader both occupies memory space
- Time consuming process as no production of object file
- It can't handle multiple source languages and multiple source programs
- For a programmer it is very difficult to make an orderly modulator program and also it becomes difficult to maintain such program, and the "compile and go" loader cannot handle such programs.
- Execution time is more

# Assemble (or Compile) and Go Loader

6



# Absolute Loader

7

- Absolute loader is a kind of loader in which relocated object files are created.
- Loader accepts these files and places them at specified locations in the memory.
- This type of loader is called absolute because *no relocation information is needed*
- This relocation information is obtained from the programmer or assembler.

# Absolute Loader

8

The starting address of every module is known to the programmer, and this corresponding starting address is stored in the object file

- Then task of loader becomes very simple and that is to simply place the executable form of the machine instructions at the locations mentioned in the object file.
- In this scheme, the programmer or assembler should have knowledge of memory management.
- The resolution of external references or linking of different subroutines is the issues which need to be handled by the programmer.

# Absolute Loader

## 9 Advantages:

- Simple to implement
- Process of Execution is efficient
- Multiple Programs are allowed
- Multiple source languages can be used
- Common object file will be prepared for multiple languages or multiple programs
- Task of Loader is to load the object code in main memory at given address

# Absolute Loader

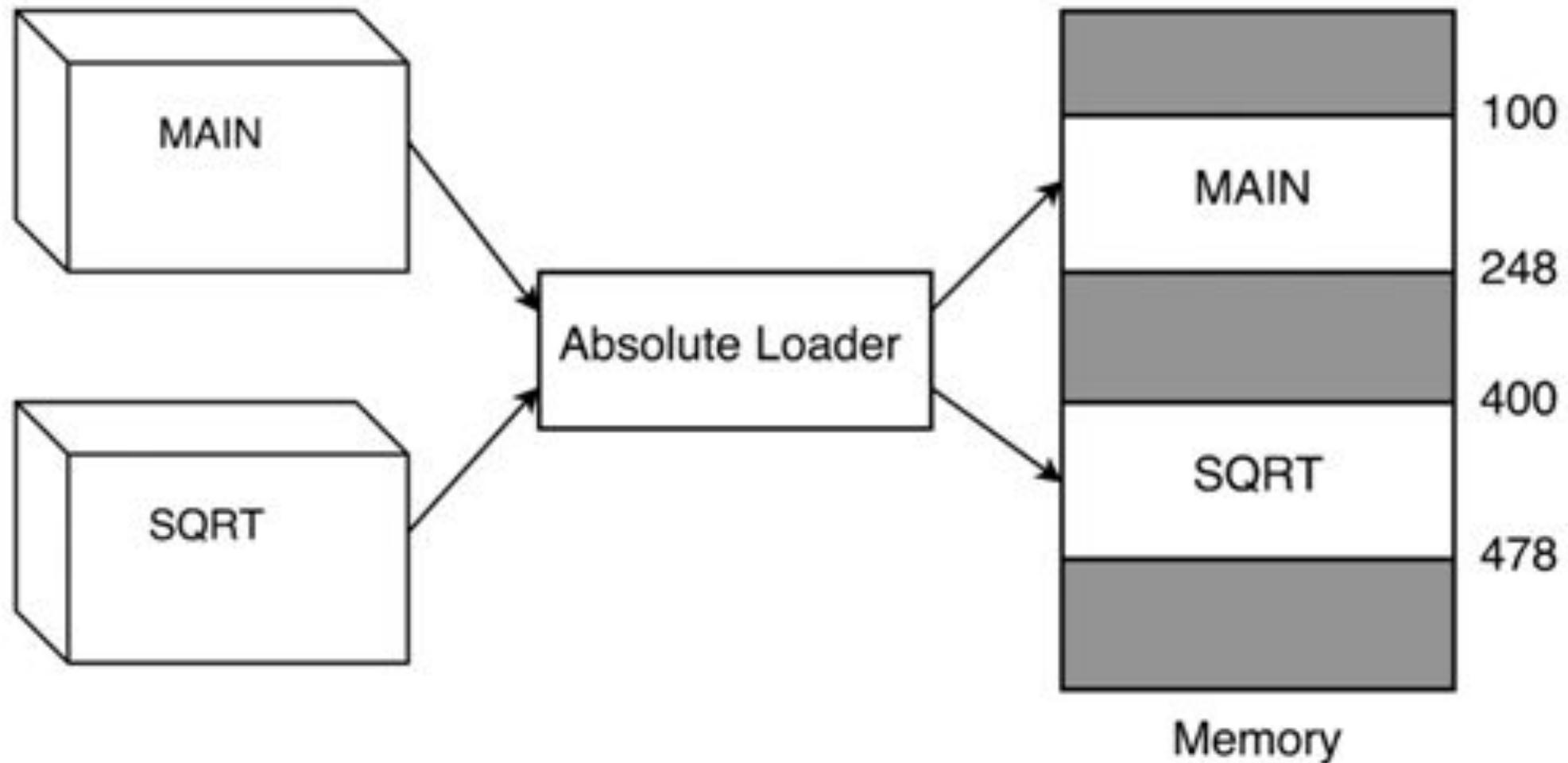
10

## Disadvantages:

- Programmer's duty is to perform all linking activity and inter-segment addresses
- Programmer must know memory management
- Programmer need to update changes in starting address of modules if any modification is in source program

# Absolute Loader

11



# Sub routine Linkages

12

- It deals with the special mechanism for calling another subroutine in an assembly language program.
- The problem of subroutine linkage is this: a main program
  - A wishes to transfer to subprogram B.
  - The programmer, in program A, could write a transfer instruction (e.g., BAL 14, B) (Branch and Link) to subprogram B.
  - However, the assembler does not know the value of this Symbol reference and will declare it as an error (undefined Symbol) unless a special mechanism has been provided.

# Sub routine Linkages

13

- There are two assembler pseudo-ops which are used for subroutine linkages :
  - **EXTRN:** It is followed by a list of Symbols which indicates that these Symbols are defined in other programs but referenced in the present program.
  - **ENTRY:** It is followed by a list of Symbols which indicates that these Symbols are defined in same programs but referenced in the other program.
- In turn, the assembler will inform the loader that these Symbols may be referenced by other programs.

# Sub routine Linkages

14

- For example, the following sequence of instructions may be a simple calling sequence to another program:

MAIN	START EXTRN .... L BALR .... END	SUBROUT .... 15, = A (SUBROUT) } CALL 14, 15 } SUBROUT
------	----------------------------------------------------	-----------------------------------------------------------------

# Sub routine Linkages

15

- The above sequence of instructions first declares SUBROUT as an external variable, that is, a variable reference but not defined in this program.
- The load instruction loads the address of that variable into register 15.
- The BALR instruction branches to the contents of register 15, which is the address of SUBROUT, and leaves the value of the next Instruction in register 14.
- In most assemblies, we may simply use a CALL SUBROUT macro, which is translated by the assembler into a calling sequence

# Design of an Absolute Loader

16

- With an absolute loading scheme, the programmer and the assembler perform the tasks of allocation, relocation, and linking.
- Therefore, it is only necessary for the loader to read cards of the object deck and move the text on the cards into the absolute locations specified by the assembler.
- There are two types of information that the object deck must communicate from the assembler to the loader.:
  - First, it must convey the *machine instructions* that the assembler has created along with the assigned core locations.
  - Second, it must convey the *entry point* of the program, which is where the loader is to transfer control when all instructions are loaded.

# Design of an Absolute Loader

17

## Card Formats for an Absolute Loader:

### □ Text Cards (for instructions and data)

Card Column	Contents
1	Card type = 0 (for text card identifier)
2	Count of number of bytes (1 byte per column) of information on card
3-5	Addresses at which data on card is to be put
6-7	Empty (could be used for validity checking)
8-72	Instructions and data to be loaded
73-80	Card sequence number

### □ Transfer Cards (to hold entry point to program)

Card Column	Contents
1	Card type = 1 (transfer card identifier)
2	Count = 0
3-5	Addresses of entry point
6-72	Empty
73-80	Card sequence number

# Design of an Absolute Loader

18

- In the card format, the instructions are stored on the card as one core byte per column.
- For each of the 256 possible contents of an eight-bit byte there is a corresponding punched card code (e.g., hexadecimal 00 is a column punched with five holes, 12-0-1-8-9, whereas a hexadecimal F1 is a column with a single punch in row 1).
- Thus, when a card is read, it is stored in core as 80 contiguous bytes.

# Design of an Absolute Loader

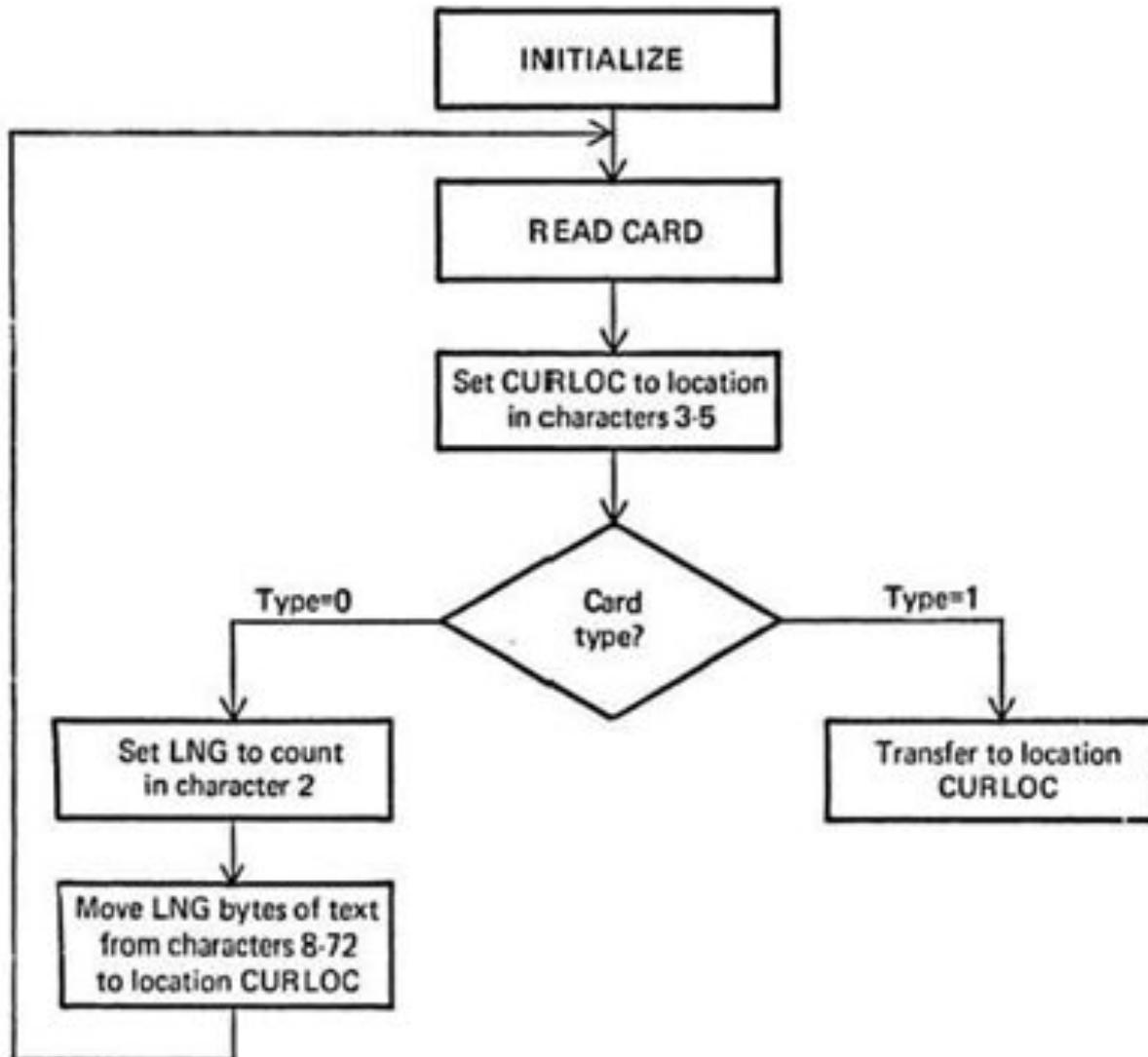
19

- Algorithm
- The object deck for this loader consists of a series of text cards terminated by a transfer card.
- Therefore, the loader should read one card at a time, moving the text to the location specified on the card, until the transfer card is reached.
- At this point, the assembled instructions are in core, and it is only necessary to transfer to the entry point specified on the transfer card.

# Design of an Absolute Loader

20

## Flowchart



# Bootstrap Loader

21

- Alternatively referred to as **bootstrapping**, **bootloader**, or **boot program**
- **Bootstrap loader** is a program that resides in the computer's EPROM, ROM, or other non-volatile memory.
- It is automatically executed by the processor when turning on the computer.
- The bootstrap loader reads the hard drives boot sector to continue the process of loading the computer's operating system.

# Bootstrap Loader

22

- The bootstrap loader is stored in the master boot record (MBR) on the computer's hard drive.
- When the computer is turned on or restarted, it first performs the power-on self-test, also known as POST.
- If the POST is successful and no issues are found, the bootstrap loader will load the operating system for the computer into memory.
- The computer will then be able to quickly access, load, and run the operating system.

# Bootstrap Loader

23

Function of Bootstrap Loader:

- Enable the user to select the OS to start
- Loading the OS file from the boot partition
- Controls the OS selection process and hardware detection prior to kernel initialization.

---

---

# **LOADERS AND LINKERS**



**MODULE-4**

## DESIGN OF A DIRECT LINKING LOADER (DLL)

- The direct linking loader is a re-locatable loader.
- The loader cannot have the direct access to the source code.
- To place the object code in the memory there are two situations: either the address of the object code could be absolute which then can be directly placed at the specified location or the address can be relative.
- If at all the address is relative, then it is the assembler who informs the loader about the relative addresses.

## SPECIFICATION OF PROBLEM

The assembler should give the following information to the loader

- The length of the object code segment
- The list of all the symbols, which are not defined in the current segment but can be used in the current segment.
- The list of all the symbols, which are defined in the current segment but can be referred by the other segments.

## SPECIFICATION OF DATA STRUCTURES

The next step in our design procedure is to identify the data bases required by each pass of the loader.

- Pass I data bases:
  - I. Input object decks.
  2. A parameter, the Initial Program Load Address (IPLA) supplied by the programmer or the operating system, that specifies the address to load the first segment.
  3. A Program Load Address (PLA) counter, used to keep track of each segment's assigned location

## SPECIFICATION OF DATA STRUCTURES

4. A table, the Global External Symbol Table (GEST), that is used to store each external symbol and its corresponding assigned core address.
5. A copy of the input to be used later by pass 2. This may be stored on an auxiliary storage device, such as magnetic tape, disk, or drum, or the original object decks may be reread by the loader a second time for pass 2.
6. A printed listing, the load map, that specifies each external symbol and its assigned value.

## SPECIFICATION OF DATA STRUCTURES

- Pass 2 data bases:
  - I. Copy of object programs inputted to pass I
  2. The Initial Program Load Address parameter (IPLA)
  3. The Program Load Address counter (PLA)
  4. The Global External Symbol Table (GEST), prepared by pass I, containing each external symbol and its corresponding absolute address value
  5. An array, 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 symbol's absolute address value

## FORMAT OF DATABASES

- Object Deck
  1. External Symbol Dictionary cards (ESD)
  2. Instructions and data cards, called "text" of program (TXT)
  3. Relocation and Linkage Directory cards (RLD)
  4. End card (END)
- End of File (EOF)/ Loader Terminate(LDT)
- Global External Symbol Table(GEST)
- Local External Symbol Array (LESA)

## OBJECT DECK

- Direct Linking Loader uses four types of records in the object decks (files) which contains all information needed for relocation and linking.
- These four sections are:
  - ESD
  - TXT
  - RLD
  - END

## EXTERNAL SYMBOL DICTIONARY CARDS (ESD)

- The ESD cards contain the information necessary to build the external symbol dictionary or symbol table.
- External symbols are symbols that can be referred beyond the subroutine level.
- The normal labels in the source program are used only by the assembler, and information about them is not included in the object deck.

## EXTERNAL SYMBOL DICTIONARY CARDS (ESD)

- Example: Assume program B has a table called NAMES; it can be accessed by program A as follows.

<b>A</b>	<b>START</b>	
	<b>EXTRN</b>	<b>NAMES</b>
	:	
	<b>L</b>	<b>1,ADDRNAME</b> get address of NAME table
	:	
<b>ADDRNAME</b>	<b>DC</b>	<b>A(NAMES)</b>
	<b>END</b>	
<hr/>		
<b>B</b>	<b>START</b>	
	<b>ENTRY</b>	<b>NAMES</b>
	:	
<b>NAMES</b>	<b>DC</b>	-----
	<b>END</b>	

# EXTERNAL SYMBOL DICTIONARY CARDS (ESD)

- There are three types of external symbols:
  - Segment Definition (SD): name on START or CSECT card.
  - Local Definition (LD): specified on ENTRY card. There must be a label in same program with same name.
  - External Reference (ER): specified on EXTRN card. There must be a corresponding ENTRY, START, or CSECT card. In another program with same name.
- Each SD and ER symbol is assigned a unique number (e.g., 1,2,3, ... ) by the assembler. This number is called the symbol's identifier, or ID, and is used in conjunction with the RLD cards.

# EXTERNAL SYMBOL DICTIONARY CARDS (ESD)

## ■ Example:

*ESD cards*

<i>Reference no.</i>	<i>Symbol</i>	<i>Type</i>	<i>Relative location</i>	<i>Length</i>
1	JOHN	SD	0	64
2	RESULT	LD	52	---
3	SUM	ER	---	---

## INSTRUCTIONS AND DATA CARDS, CALLED "TEXT" OF PROGRAM (TXT):

- The TXT cards contain blocks of data and the relative address at which the data is to be placed.
- Once the loader has decided where to load the program, it merely adds the Program Load Address (PLA) to the relative address and moves the data into the resulting location.
- The data on the TXT card may be instructions, non-relocated data, or initial values of address constants

## INSTRUCTIONS AND DATA CARDS, CALLED "TEXT" OF PROGRAM (TXT):

- The data on the TXT card may be instructions, non-relocated data, or initial values of address constants

<i>Reference no.</i>	<i>Relative location</i>	<i>Object code</i>
4	0	BALR 12.0
6	2	ST 14,54(0,12)
7	6	L 1,46(0,12)
8	10	L 15,58(0,12)
9	14	BALR 14,15
10	16	ST 1,50(0,12)
11	20	L 14,54(0,12)
12	24	BCR 15,14
13	28	1

# INSTRUCTIONS AND DATA CARDS, CALLED "TEXT" OF PROGRAM (TXT):

## ■ Example:

<i>Relative address</i>	<i>Instruction</i>
A	START EXTRN NAMES USING *,15
	:
40	L 1,ALPHA
44	BCR 15,14
46	
48 ALPHA	DC F'5'
52 ALLOC	DC A(ALPHA)
56 ADDRNAME	DC A(NAMES)
	*

(Skipped by assembler)

The TXT card produced is:

Relative address = 40

Data portion = 58 10 F0 30 07FE XX XX 00 00 00 05 00 00 00 30 00 00 00 00

Length of data portion = 20 bytes

## RELOCATION AND LINKAGE DIRECTORY CARDS (RLD):

- The RLD cards contain the following information.
  - The location and length of each address constant that needs to be changed for relocation or linking
  - The external symbol by which the address constant should be modified (added or subtracted)
  - The operation to be performed (add or subtract)

## RELOCATION AND LINKAGE DIRECTORY CARDS (RLD):

Example:

<i>ID</i>	<i>Flag</i>	<i>Length</i>	<i>Rel. Loc.</i>
01	+	4	52
02	+	4	56

- If we assume that A's assigned ID is 01 and NAMES' assigned ID is 02.
- This RLD information tells the loader to add the absolute load address of A to the contents of relative location 52 and then add the absolute load address of NAMES to the contents of relative location 56.

## **END CARD (END)**

- The END card specifies the end of the object deck. If the assembler END card has a symbol in the operand field, it specifies a start of execution point for the entire program (all subroutines). This address is recorded on the END card.

## END OF FILE (EOF) / LOADER TERMINATE(LDT)

- There is a final card required to specify the end of a collection of object decks.
- The loaders usually use either a loader terminate (LDT) or End of File (EOF) card.

## END OF FILE (EOF) / LOADER TERMINATE(LDT)

### ■ Example:

<b>Subroutine A</b>	{ ESD TXT RLD END
<b>Subroutine B</b>	{ ESD TXT RLD END
<b>Subroutine C</b>	{ ESD TXT RLD END
	EOF or LDT

## GLOBAL EXTERNAL SYMBOL TABLE (GEST)

- The Global External Symbol Table (GEST) is used to store the external symbols.
- These symbols are defined by means of a Segment Definition (SD) or Local Definition (LD) entry on an External Symbol Dictionary (ESD) card.
- When these symbols are encountered during pass I, they are assigned an absolute core address
- This address is stored, along with the symbol, in the GEST

## GLOBAL EXTERNAL SYMBOL TABLE (GEST)

■ Example:

12 bytes per entry	
External symbol (8-bytes) (characters)	Assigned core address (4-bytes) (decimal)
"PG1bbbbbb"	104
"PG1ENT1b"	124
"PG1ENT2b"	134
"PG2bbbbbb"	168
"PG2ENT1b"	184

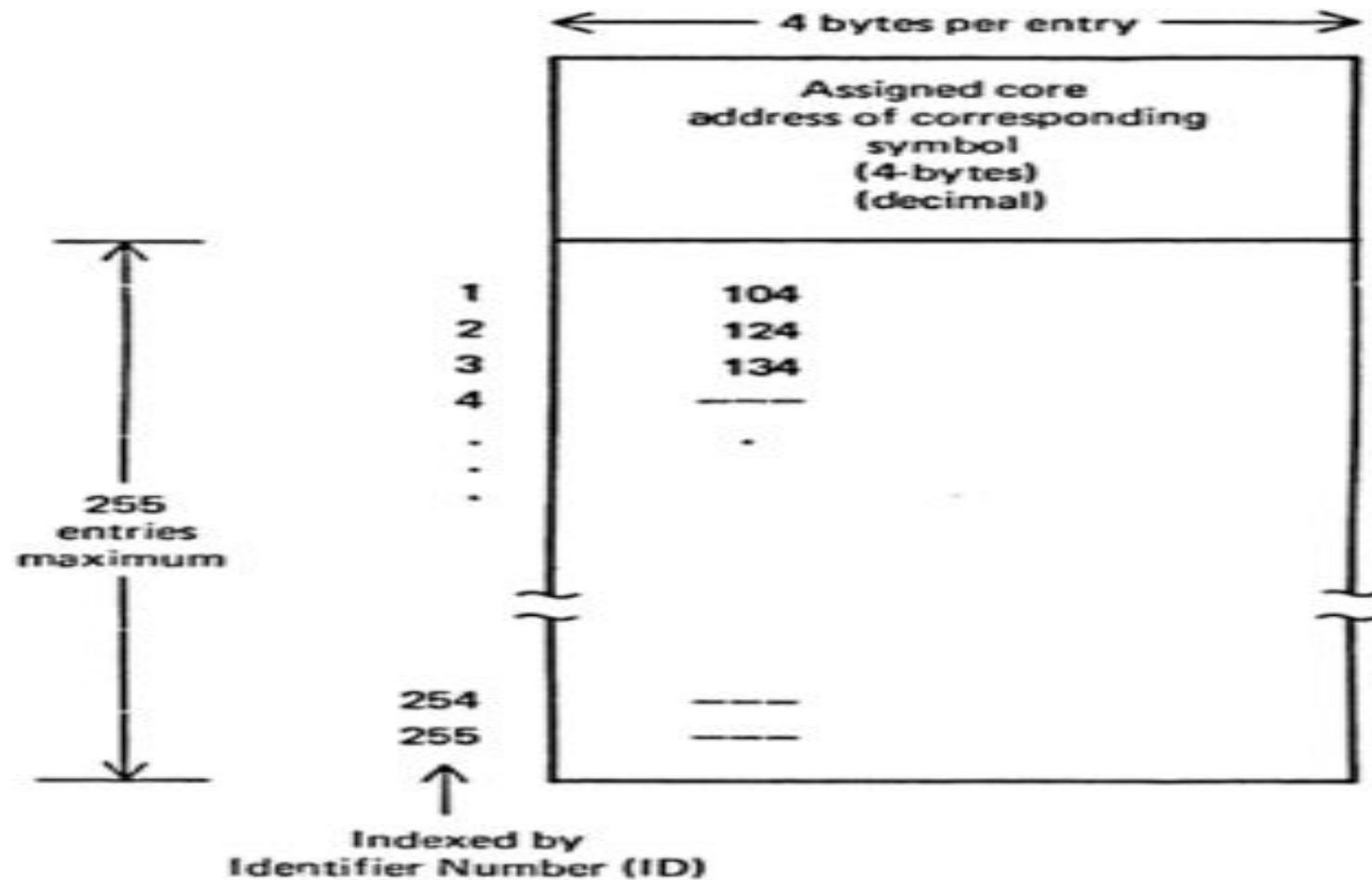
Global External Symbol Table (GEST) format

## LOCAL EXTERNAL SYMBOL ARRAY

- It is necessary to create a separate LESA for each segment, but since the LESAs are only produced one at a time, the same array can be reused for each segment.
- It is not necessary to search the LESA;
- Given an ID number, the corresponding value is written as LESA(ID) and can be immediately obtained.

# LOCAL EXTERNAL SYMBOL ARRAY

## Example:



# FLOWCHART FOR PASS-I

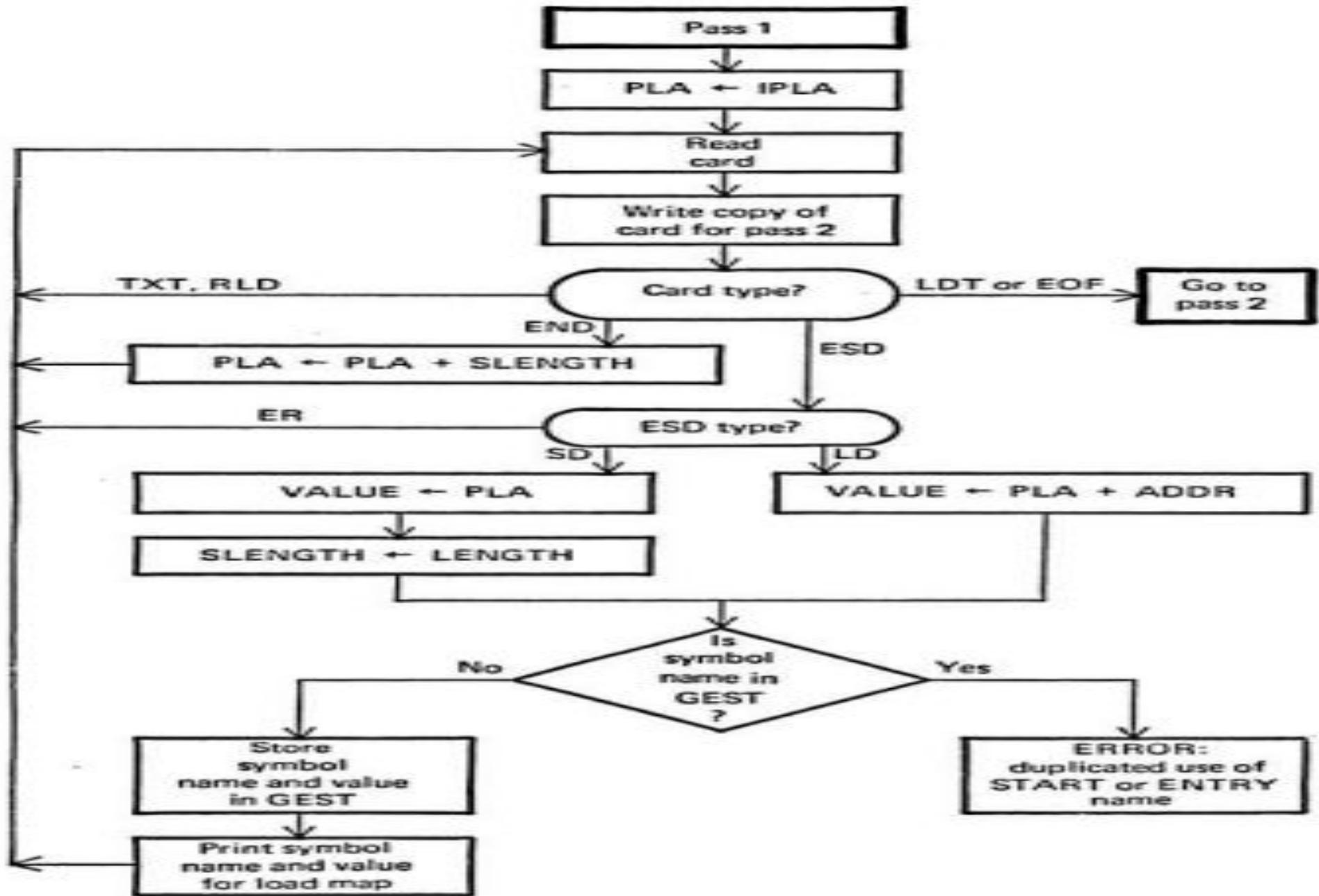
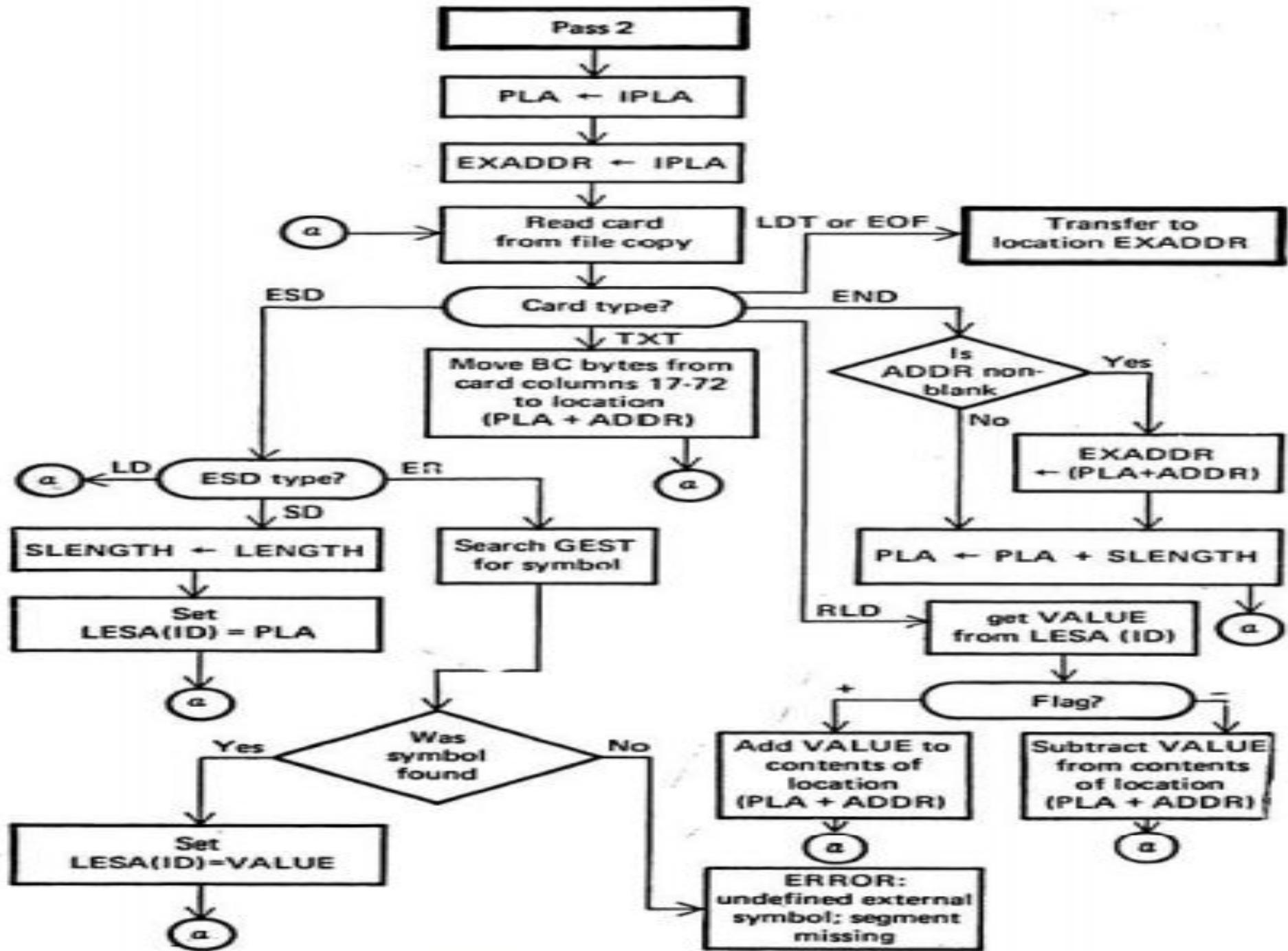


FIGURE 5.24 Detailed pass 1 flowchart

# FLOWCHAR T FOR PASS-2



# EXAMPLE

Source card reference	Relative address	Procedure	Sample program (source deck)	
1	0	PG1	START	
2			ENTRY PG1ENT1,PG1ENT2	
3			EXTRN PG2ENT1,PG2	
4	20		====	
5	30		====	
6	40		DC A(PG1ENT1)	
7	44		DC A(PG1ENT2+15)	
8	48		DC A(PG1ENT2-PG1ENT1-3)	
9	52		DC A(PG2)	
10	56		DC A(PG2ENT1+PG2-PG1ENT1+4)	
11			END	
12	0	PG2	START	
13			ENTRY PG2ENT1	
14			EXTRN PG1ENT1,PG1ENT2	
15	16		====	
16	24		DC A(PG1ENT1)	
17	28		DC A(PG1ENT2+15)	
18	32		DC A(PG1ENT2-PG1ENT1-3)	
19			END	

Sample procedures PG1 and PG2

# EXAMPLE

## ESD cards

<i>Source card reference</i>	<i>Name</i>	<i>Type</i>	<i>ID</i>	<i>Relative address</i>	<i>Length</i>
1	PG1	SD	01	0	60
2	PG1ENT1	LD	—	20	—
2	PG1ENT2	LD	—	30	—
3	PG2	ER	02	—	—
3	PG2ENT1	ER	03	—	—

## TXT cards

(only the interesting ones, i.e. those involving address constants)

<i>Source card reference</i>	<i>Relative address</i>	<i>Contents</i>	<i>Comments</i>
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	56-59	-16	= -20 + 4

## RLD cards

<i>Source card reference</i>	<i>ESD ID</i>	<i>Length (bytes)</i>	<i>Flag + or -</i>	<i>Relative address</i>
6	01	4	+	40
7	01	4	+	44
9	02	4	+	52
10	03	4	+	56
10	02	4	+	56
10	01	4	-	56

# EXAMPLE

## *ESD cards*

<i>Source card reference</i>	<i>Name</i>	<i>Type</i>	<i>ID</i>	<i>ADDR</i>	<i>Length</i>
12	PG2	SD	01	0	36
13	PG2ENT1	LD	—	16	—
14	PG1ENT1	ER	02	—	—
14	PG1ENT2	ER	03	—	—

## *TXT cards*

(only the interesting ones)

<i>Source card reference</i>	<i>Relative address</i>	<i>Contents</i>
16	24-27	0
17	28-31	15
18	32-35	-3

## *RLD cards*

<i>Source card reference</i>	<i>ESD ID</i>	<i>Length flag (bytes)</i>	<i>Flag + or -</i>	<i>Relative address</i>
16	02	4	+	24
17	03	4	+	28
18	03	4	+	32
18	02	4	-	32

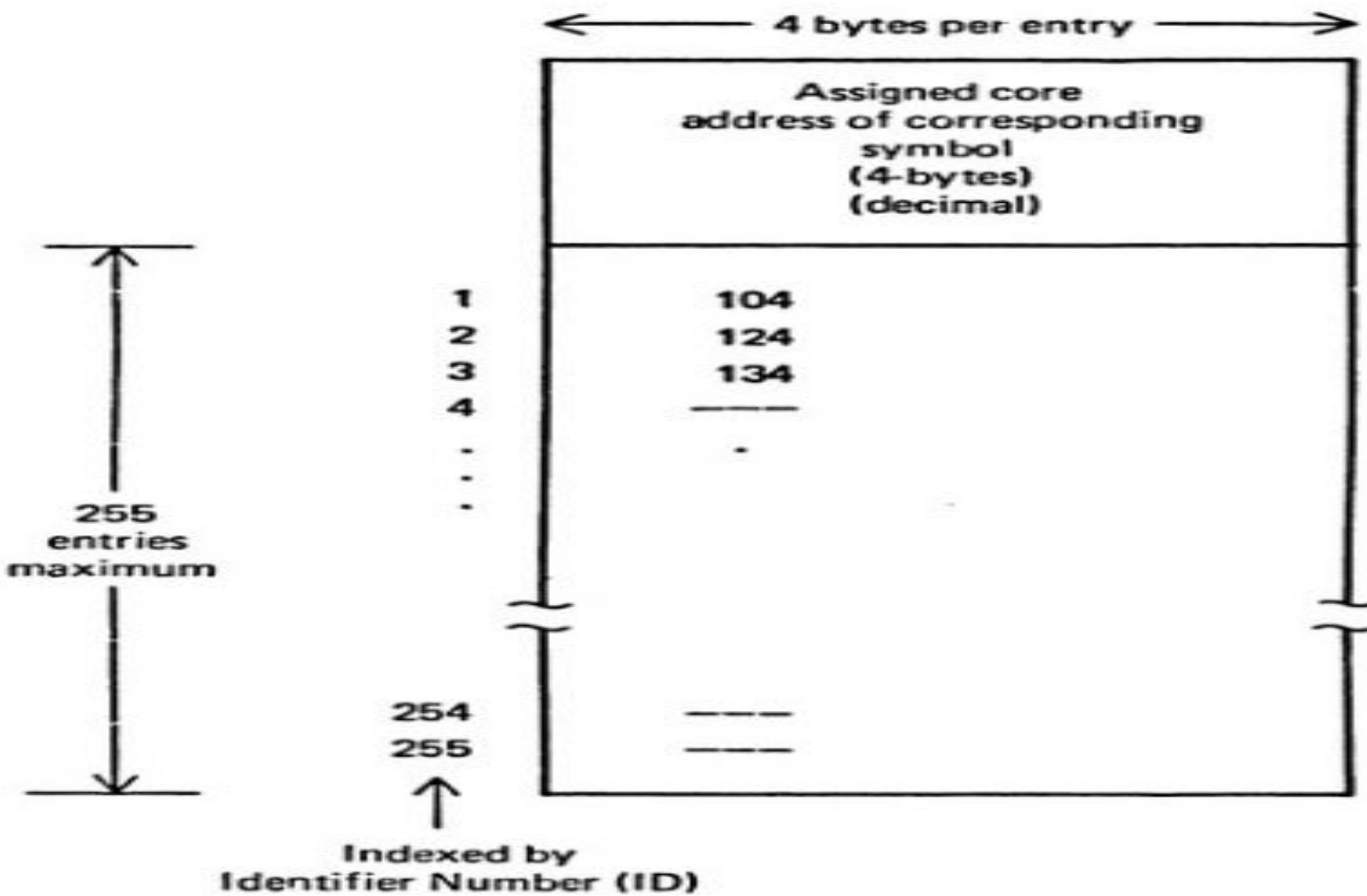
## EXAMPLE

**12 bytes per entry**

External symbol (8-bytes) (characters)	Assigned core address (4-bytes) (decimal)
"PG1bbbbbb"	104
"PG1ENT1b"	124
"PG1ENT2b"	134
"PG2bbbbbb"	168
"PG2ENT1b"	184

**Global External Symbol Table (GEST) format**

## EXAMPLE



# Module-4

Loaders and Linkers

# Syllabus

- Introduction
- Functions of loaders
- Relocation & Linking concepts
- Different Loading Schemes
  - Relocating Loader
  - Direct Linking Loader
- Dynamic Linking & Loading

# Loader

The loader is a program

- which accepts the object program decks
- prepares these programs for execution by the computer and
- initiates the execution

# Functions of a Loader

- **Allocation:** Allocate space in memory for the programs

# Functions of a Loader

- **Allocation:** Allocate space in memory for the programs
- **Linking:** Resolve symbolic references between object decks.

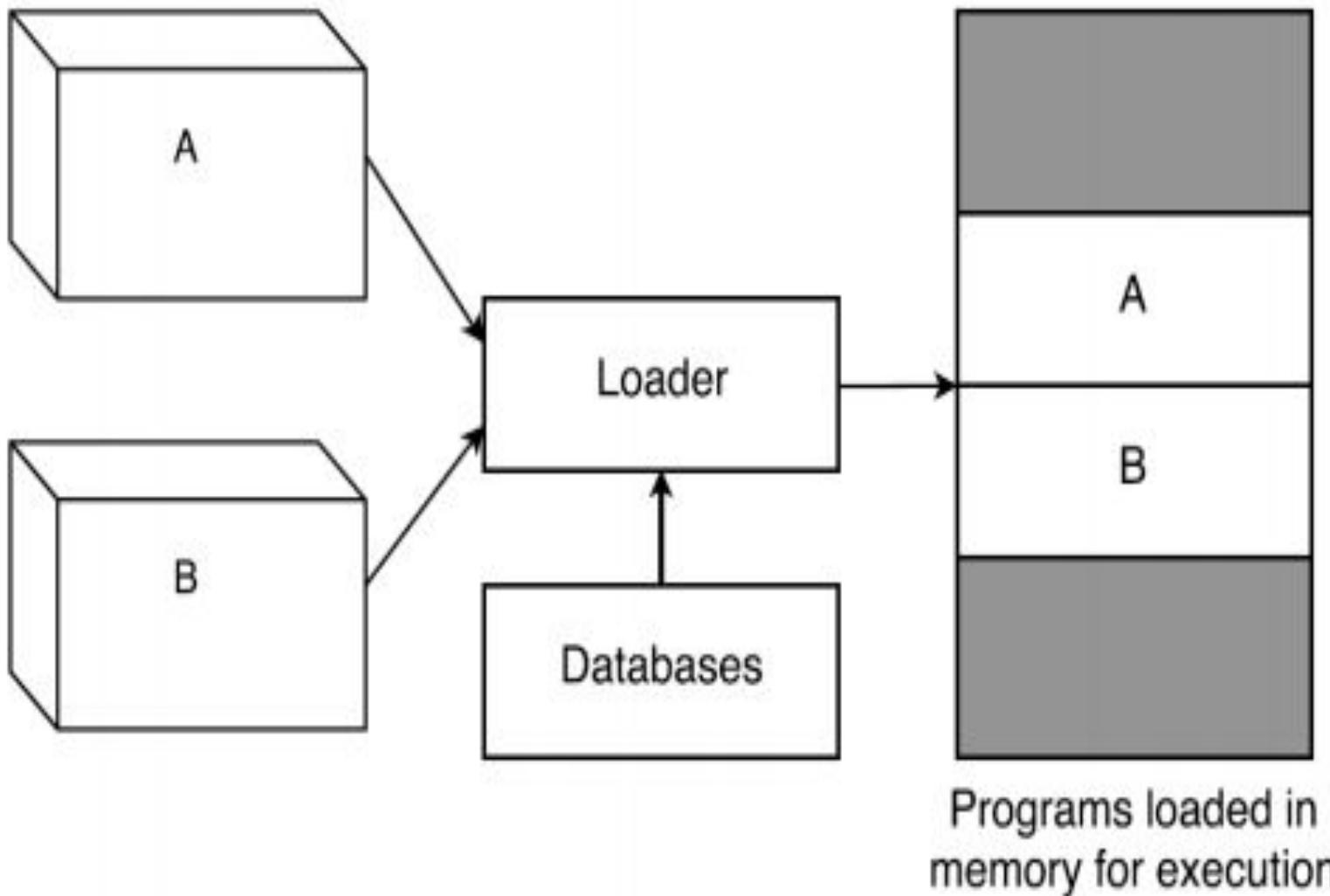
# Functions of a Loader

- **Allocation:** Allocate space in memory for the programs
- **Linking:** Resolve symbolic references between object decks.
- **Relocation:** Adjust all address dependent locations, such as address constants, to correspond to the allocated space.

# Functions of a Loader

- **Allocation:** Allocate space in memory for the programs
- **Linking:** Resolve symbolic references between object decks.
- **Relocation:** Adjust all address dependent locations, such as address constants, to correspond to the allocated space.
- **Loading:** Physically place the machine instructions and data into memory.

# General Loading Scheme



# General Loading Scheme

- In General loader, the source program is converted to object program by some translator (assembler).

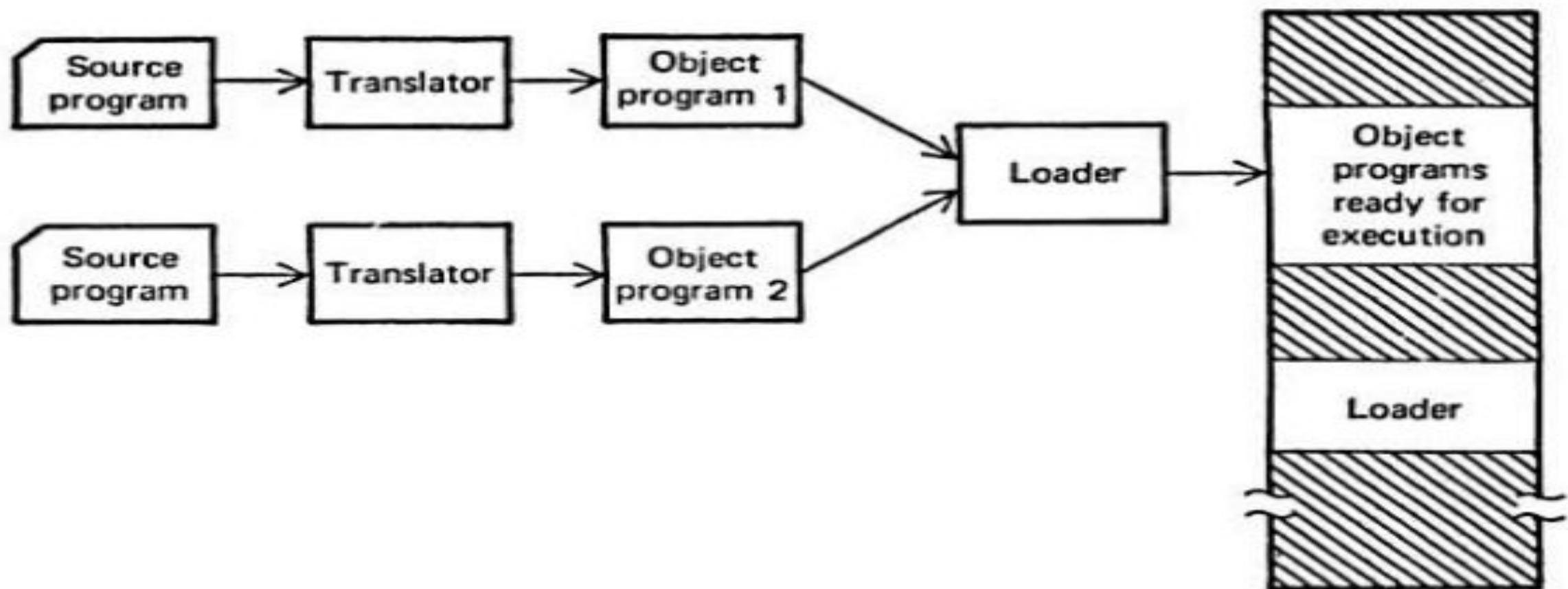
# General Loading Scheme

- In General loader, the source program is converted to object program by some translator (assembler).
- The loader accepts these object modules and puts machine instruction and data in an executable form at their assigned memory.

# General Loading Scheme

- In General loader, the source program is converted to object program by some translator (assembler).
- The loader accepts these object modules and puts machine instruction and data in an executable form at their assigned memory.
- The loader occupies some portion of main memory

# General Loading Scheme



# Advantages:

1. No need of retranslation each time while running program

# Advantages:

1. No need of retranslation each time while running program
2. No wastage of Memory
  - Loader occupies very less space in comparison with assembler

# Advantages:

1. No need of retranslation each time while running program
2. No wastage of Memory
  - Loader occupies very less space in comparison with assembler
3. Possible to write source program with multiple programs and multiple languages

# Linking

The execution of program can be done with the help of following steps:

1. **Translation:** Translation of the program (done by assembler or compiler)

# Linking

The execution of program can be done with the help of following steps:

1. **Translation:** Translation of the program (done by assembler or compiler)
2. **Linking:** Linking of the program with all other programs, which are needed for execution. This also involves preparation of a program called load module.

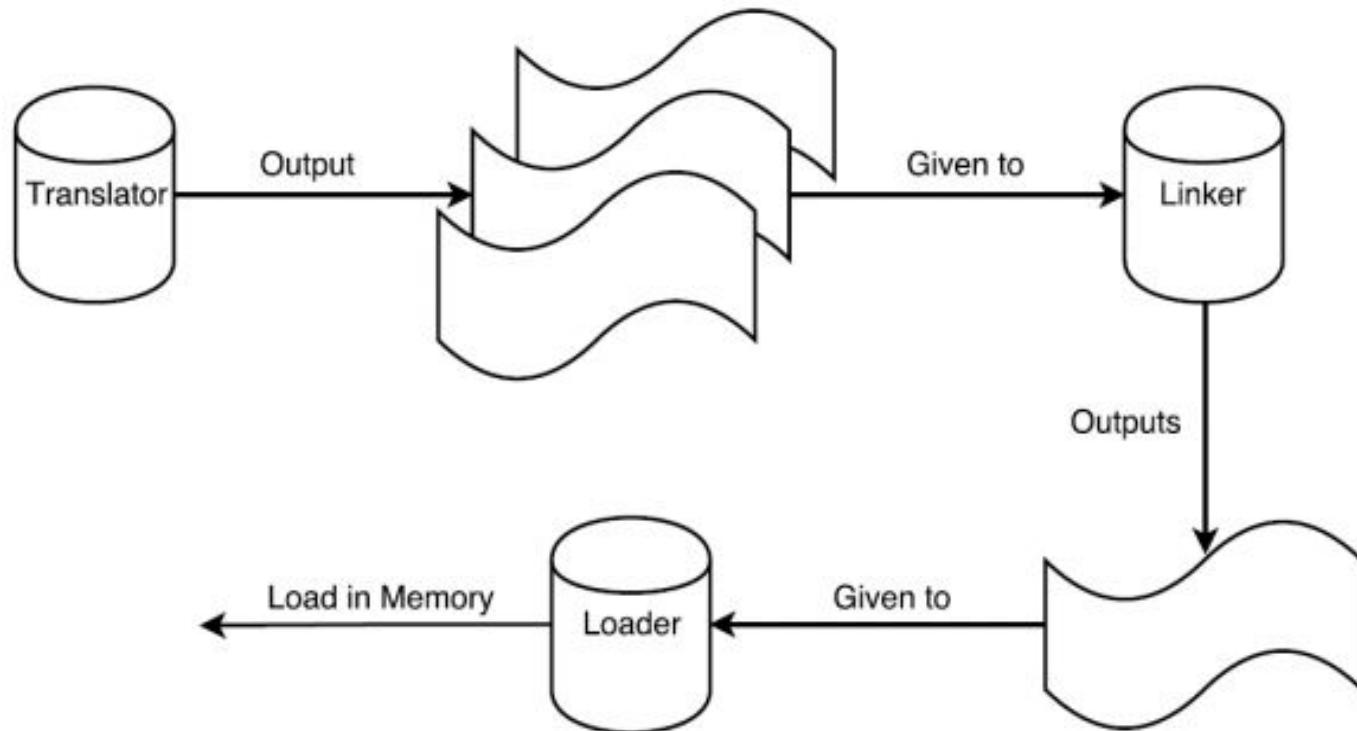
# Linking

The execution of program can be done with the help of following steps:

1. **Translation:** Translation of the program (done by assembler or compiler)
2. **Linking:** Linking of the program with all other programs, which are needed for execution. This also involves preparation of a program called load module.
3. **Loading:** Loading of the load module prepared by linker to some specified memory location.

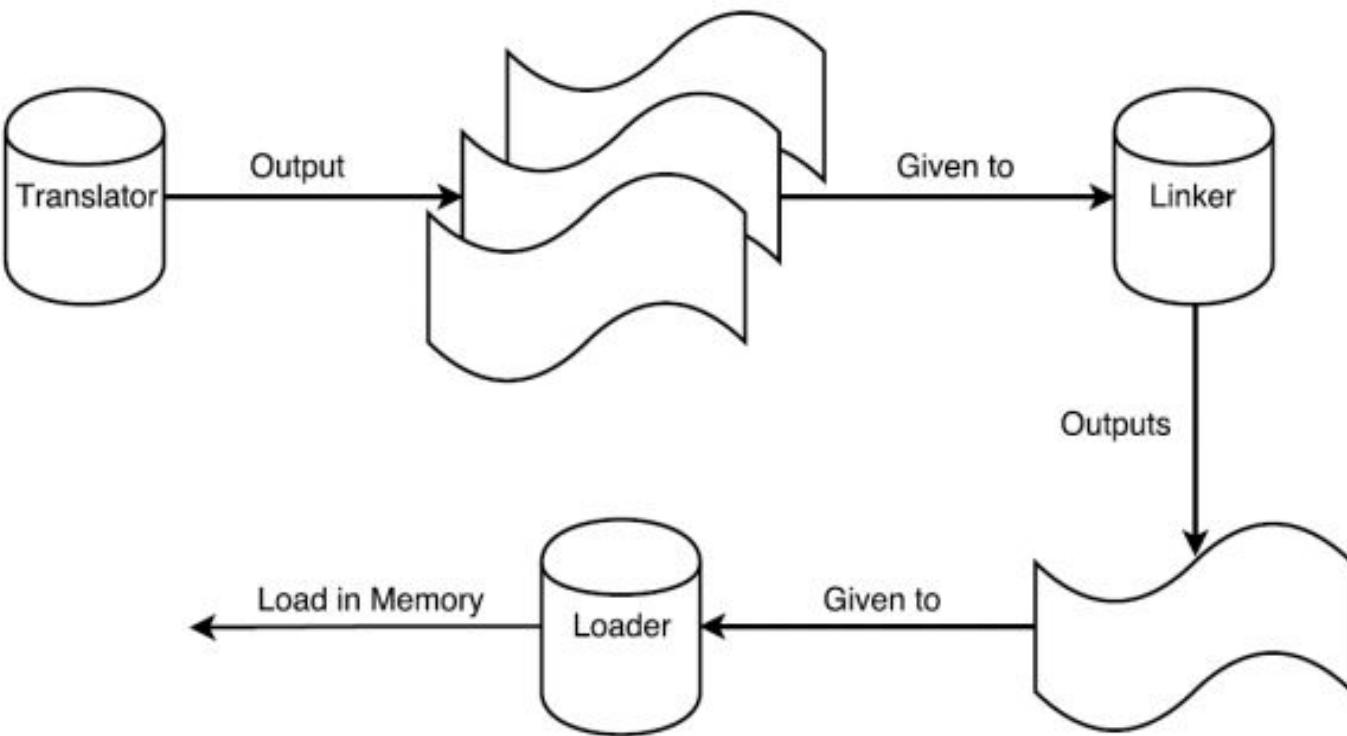
# Process of Linking

- The output of translator is a program called object module.



**Process of Linking a Program**

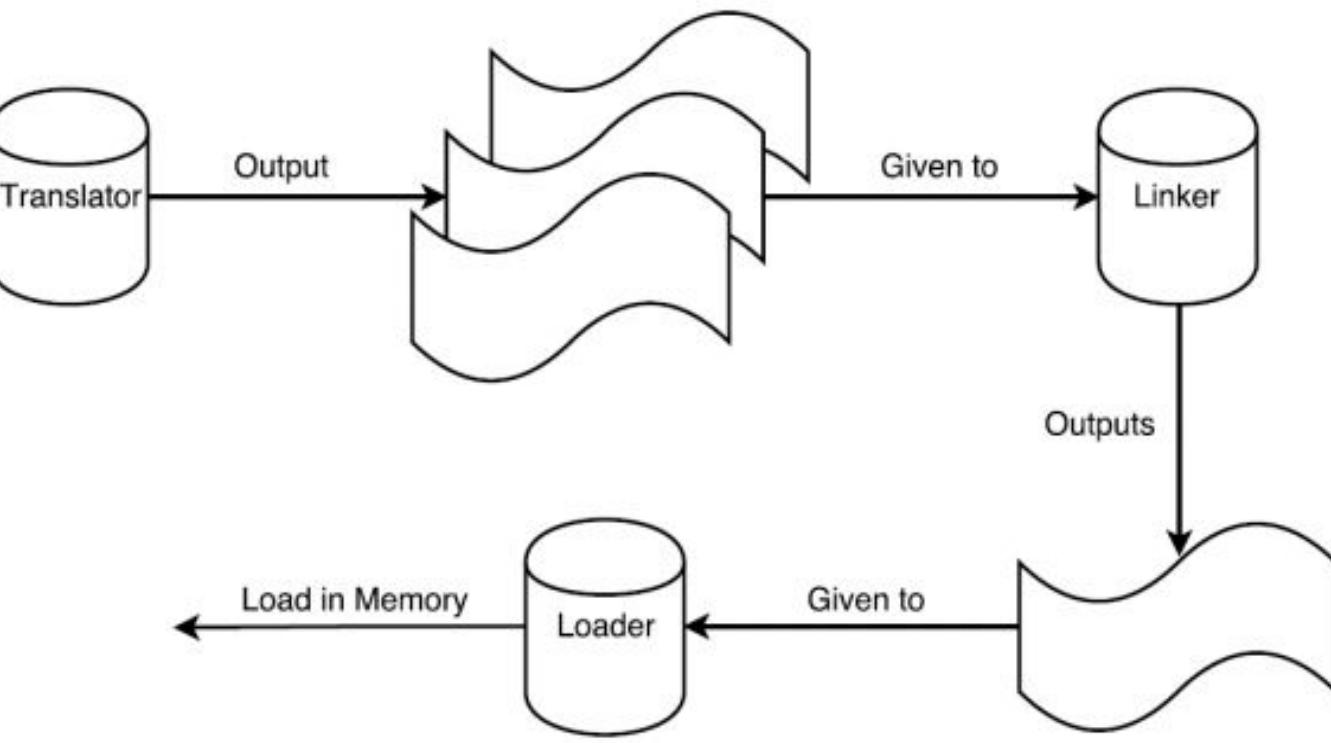
# Process of Linking



**Process of Linking a Program**

- The output of translator is a program called object module.
- The linker processes these object modules, binds with necessary library routines and prepares a ready to execute program.

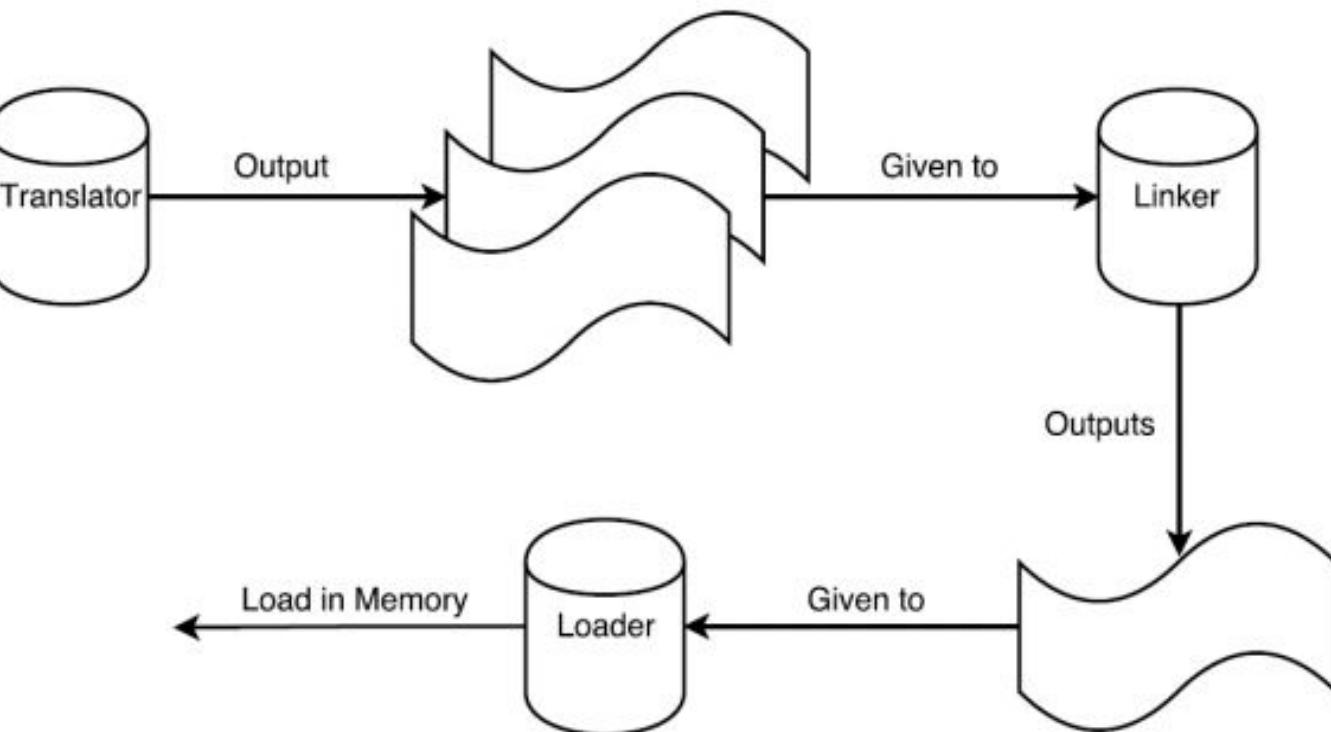
# Process of Linking



**Process of Linking a Program**

- The output of translator is a program called object module.
- The linker processes these object modules, binds with necessary library routines and prepares a ready to execute program.
- Such a program is called binary program.

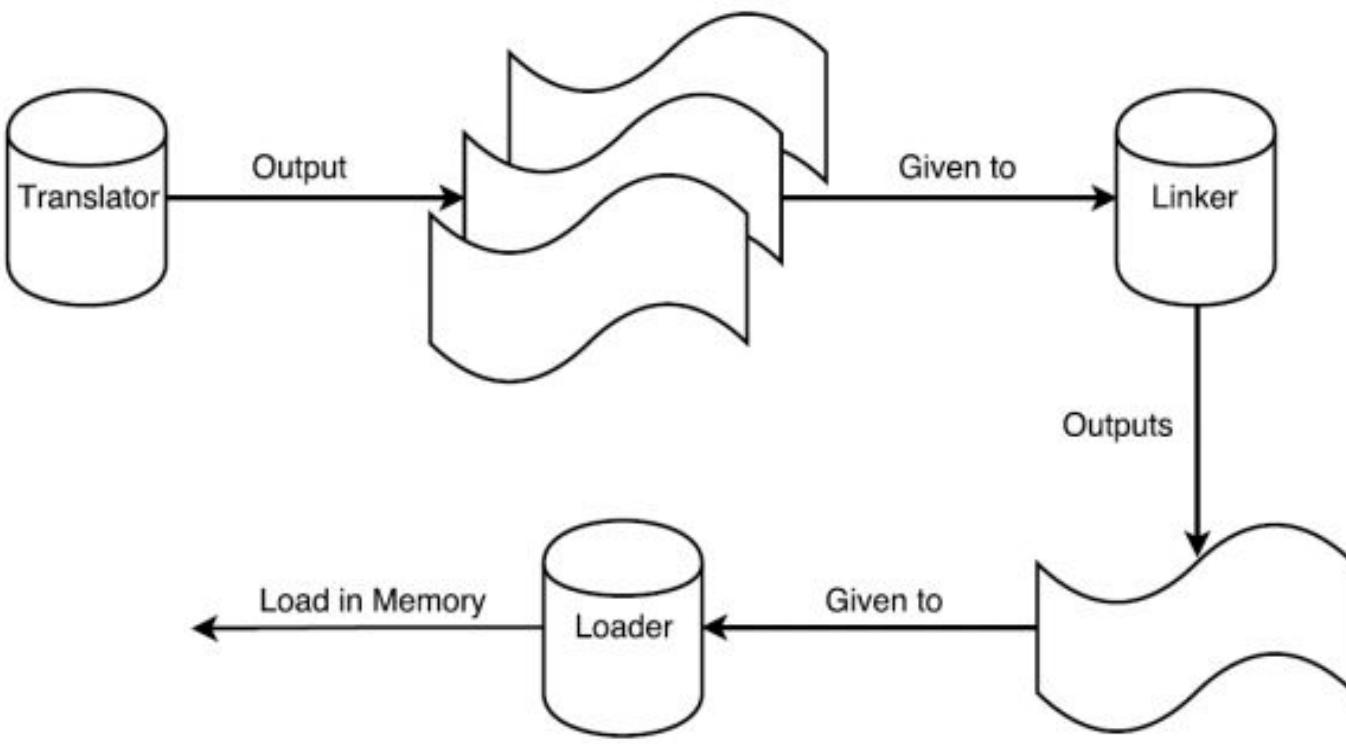
# Process of Linking



**Process of Linking a Program**

- The output of translator is a program called object module.
- The linker processes these object modules, binds with necessary library routines and prepares a ready to execute program.
- Such a program is called binary program.
- The "binary program also contains some necessary information about allocation and relocation.

# Process of Linking



- The output of translator is a program called object module.
- The linker processes these object modules, binds with necessary library routines and prepares a ready to execute program.
- Such a program is called binary program.
- The "binary program also contains some necessary information about allocation and relocation.
- The loader then loads this program into memory for execution purpose.

# Tasks of a Linker

## 1. To prepare Single Load Module

- Adjust all addresses & subroutine references with respect to the offset location
- Concatenate all object modules
- Adjust Operand Address References and External References with respect to offset location

# Tasks of a Linker

## 1. To prepare Single Load Module

- Adjust all addresses & subroutine references with respect to the offset location
- Concatenate all object modules
- Adjust Operand Address References and External References with respect to offset location

## 2. To prepare ready to execute module

- Copy the binary machine instruction and constant data



# MODULE-4



LOADERS AND LINKERS

# LOADER SCHEMES

**1. Compile and Go Loaders**

**2. General Loader Scheme**

**3. Absolute Loaders**

**4. Subroutine Linkages**

**5. Relocating Loaders**

**6. Direct Linking Loaders**

**7. Other Loader Schemes:**  
a) Dynamic Loading  
b) Dynamic Linking

## COMPILE AND GO LOADER

- In this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address.

## COMPILE AND GO LOADER

- In this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address.
- The assembler runs in one part of memory and the assembled machine instructions and data is directly put into their assigned memory locations.

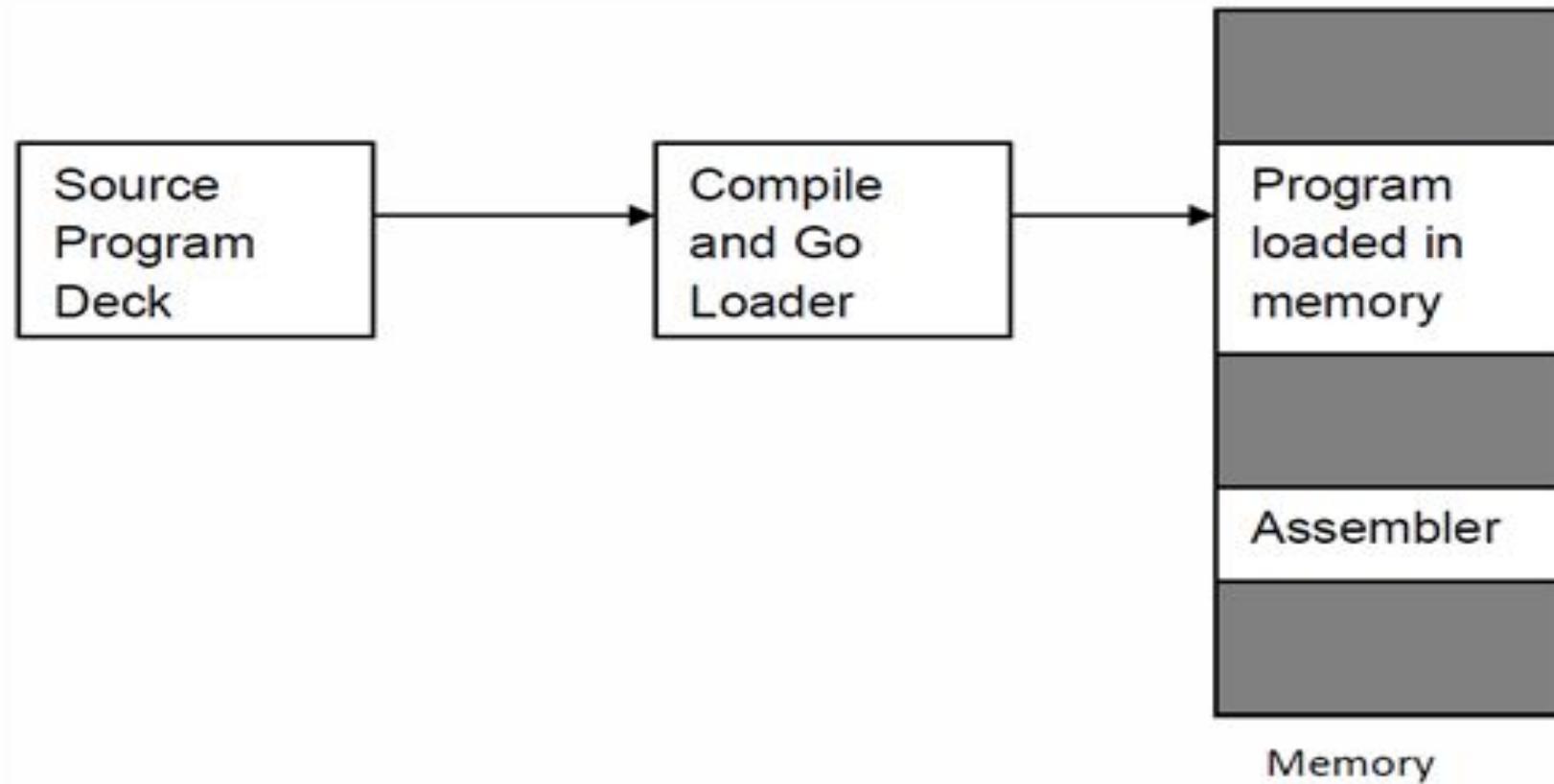
## COMPILE AND GO LOADER

- In this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address.
- The assembler runs in one part of memory and the assembled machine instructions and data is directly put into their assigned memory locations.
- After completion of assembly process, assign starting address of the program to the location counter.

## COMPILE AND GO LOADER

- In this type of loader, the instruction is read line by line, its machine code is obtained and it is directly put in the main memory at some known address.
- That means the assembler runs in one part of memory and the assembled machine instructions and data is directly put into their assigned memory locations.
- After completion of assembly process, assign starting address of the program to the location counter.
- This loading scheme is also called as “assemble and go”.

# COMPILE AND GO LOADER



# COMPILE AND GO LOADER

## **Advantage:**

- Easy to implement
- Assembler runs in one part of memory and assembled instructions are directly loaded into core
- No extra procedures are involved

# COMPILE AND GO LOADER

## **Disadvantage:**

- Wastage of memory as assembler and loader both occupies memory space

# COMPILE AND GO LOADER

## **Disadvantage:**

- Wastage of memory as assembler and loader both occupies memory space
- Time consuming process as no production of object file

# COMPILE AND GO LOADER

## **Disadvantage:**

- Wastage of memory as assembler and loader both occupies memory space
- Time consuming process as no production of object file
- It can't handle multiple source languages and multiple source programs

# COMPILE AND GO LOADER

## **Disadvantage:**

- Wastage of memory as assembler and loader both occupies memory space
- Time consuming process as no production of object file
- It can't handle multiple source languages and multiple source programs
- For a programmer it is very difficult to make an orderly modulator program and also it becomes difficult to maintain such program, and the “compile and go” loader cannot handle such programs.

# COMPILE AND GO LOADER

## **Disadvantage:**

- Wastage of memory as assembler and loader both occupies memory space
- Time consuming process as no production of object file
- It can't handle multiple source languages and multiple source programs
- For a programmer it is very difficult to make an orderly modulator program and also it becomes difficult to maintain such program, and the “compile and go” loader cannot handle such programs.
- Execution time is more

# Absolute Loader

- Absolute loader is a kind of loader in which relocated object files are created.

# Absolute Loader

- Absolute loader is a kind of loader in which relocated object files are created.
- Loader accepts these files and places them at specified locations in the memory.

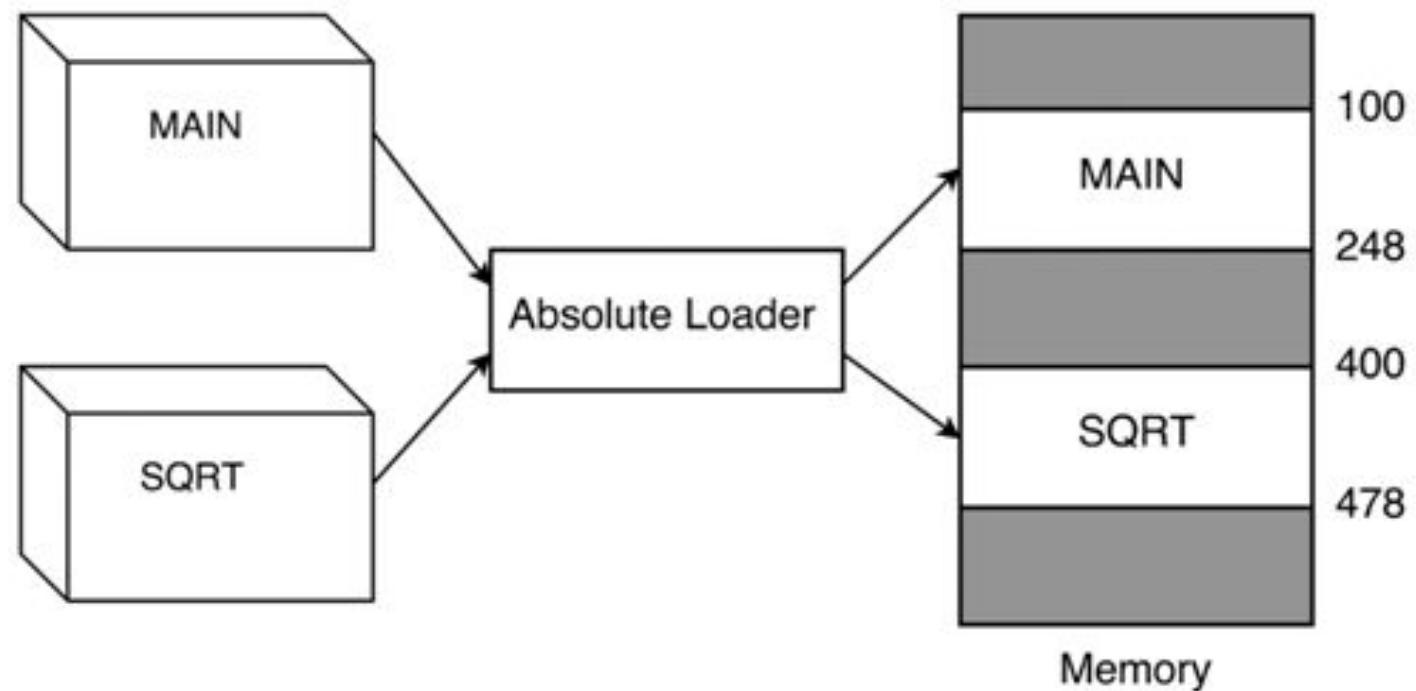
# Absolute Loader

- Absolute loader is a kind of loader in which relocated object files are created.
- Loader accepts these files and places them at specified locations in the memory.
- This type of loader is called absolute because *no relocation information is needed*

# Absolute Loader

- Absolute loader is a kind of loader in which relocated object files are created.
- Loader accepts these files and places them at specified locations in the memory.
- This type of loader is called absolute because *no relocation information is needed*
- This relocation information is obtained from the programmer or assembler.

# Absolute Loader



# Absolute Loader

- The starting address of every module is known to the programmer

# Absolute Loader

- The starting address of every module is known to the programmer
- This corresponding starting address is stored in the object file

# Absolute Loader

- The starting address of every module is known to the programmer
- This corresponding starting address is stored in the object file
- Task of loader becomes very simple: To simply place the executable form of the machine instructions at the locations mentioned in the object file.

# Absolute Loader

- The starting address of every module is known to the programmer
- This corresponding starting address is stored in the object file
- Task of loader becomes very simple: To simply place the executable form of the machine instructions at the locations mentioned in the object file.
- In this scheme, the programmer or assembler should have knowledge of memory management.

# Absolute Loader

- The starting address of every module is known to the programmer
- This corresponding starting address is stored in the object file
- Task of loader becomes very simple: To simply place the executable form of the machine instructions at the locations mentioned in the object file.
- In this scheme, the programmer or assembler should have knowledge of memory management.
- The resolution of external references or linking of different subroutines is the issues which need to be handled by the programmer. .

# Absolute Loader

## Advantages

- Simple to implement
- Process of Execution is efficient
- Multiple Programs are allowed
- Multiple source languages can be used
- Common object file will be prepared for multiple languages or multiple programs
- Task of Loader is to load the object code in main memory at given address

# Absolute Loader

## Disadvantages

- Programmer's duty is to perform all linking activity and inter-segment addresses
- Programmer must know memory management
- Programmer need to update changes in starting address of modules if any modification is in source program

# Design of an Absolute Loader

The four loader functions are accomplished as follows:

- Allocation – by Programmer
- Linking – by Programmer
- Relocation – by Assembler
- Loading – by Loader

# Design of an Absolute Loader

- It is only necessary for the loader to read cards of the object deck and move the text on the cards into the absolute locations specified by the assembler.
- There are two types of information that the object deck must communicate from the assembler to the loader:
  - First: It must convey the *machine instructions* that the assembler has created along with the assigned core locations.
  - Second: It must convey the *entry point* of the program, which is where the loader is to transfer control when all instructions are loaded.

# Design of an Absolute Loader

## Text Cards (For instruction and Data)

Card Column	Contents
1	Card type = 0 (for text card identifier)
2	Count of number of bytes (1 byte per column) of information on card
3-5	Addresses at which data on card is to be put
6-7	Empty (could be used for validity checking)
8-72	Instructions and data to be loaded
73-80	Card sequence number

# Design of an Absolute Loader

## Transfer Cards (To hold entry point to program)

Card Column	Contents
1	Card type = 1 (transfer card identifier)
2	Count = 0
3-5	Addresses of entry point
6-72	Empty
73-80	Card sequence number
1	Card type = 1 (transfer card identifier)

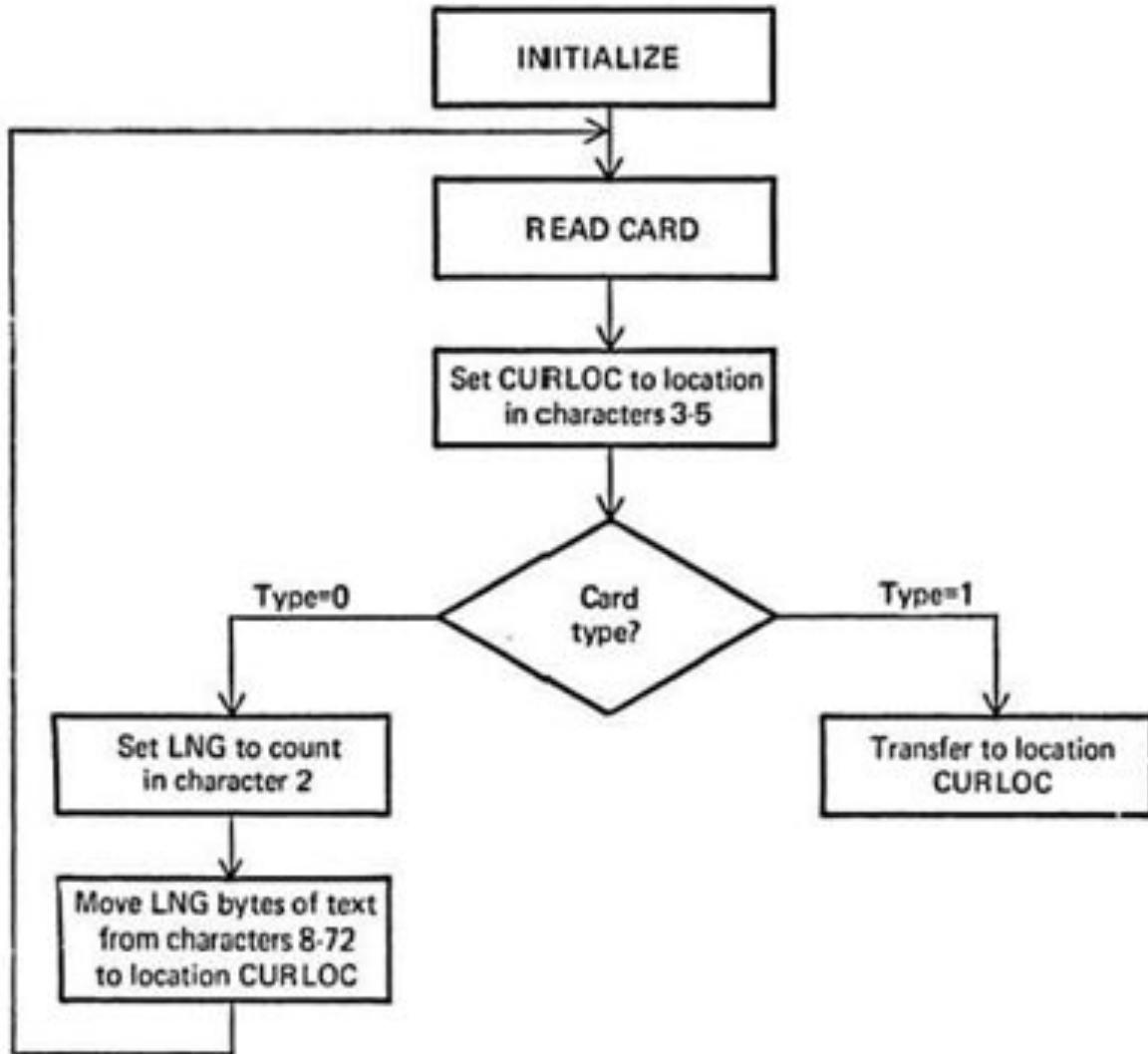
# Design of an Absolute Loader

## Algorithm

The object deck for this loader consists of a series of text cards terminated by a transfer card.

Therefore, the loader should read one card at a time, moving the text to the location specified on the card, until the transfer card is reached.

At this point, the assembled instructions are in core, and it is only necessary to transfer to the entry point specified on the transfer card..



# Bootstrap Loader

- Alternatively referred to as **bootstrapping**, **bootloader**, or **boot program**
- **Bootstrap loader** is a program that resides in the computer's EPROM, ROM, or other non-volatile memory.
- It is automatically executed by the processor when turning on the computer.
- The bootstrap loader reads the hard drives boot sector to continue the process of loading the computer's operating system

# Bootstrap Loader

- The bootstrap loader is stored in the master boot record (MBR) on the computer's hard drive.
- When the computer is turned on or restarted, it first performs the power-on self-test, also known as POST.
- If the POST is successful and no issues are found, the bootstrap loader will load the operating system for the computer into memory.
- The computer will then be able to quickly access, load, and run the operating system.

# Bootstrap Loader

Function of Bootstrap Loader:

- Enable the user to select the OS to start
- Loading the OS file from the boot partition
- Controls the OS selection process and hardware detection prior to kernel initialization.







## DIRECT LINKING LOADER

- A direct-linking loader is a general relocatable loader, and is perhaps the most popular loading scheme presently used
- The direct-linking loader has the advantage of allowing the programmer multiple procedure segments and multiple data segments and of giving him complete freedom in referencing data or instructions contained in other segments.

## DISADVANTAG

- It is necessary to allocate, relocate, link, and load all of the subroutines each time in order to execute a program
- Since there may be tens and often hundreds of subroutines involved, this loading process can be extremely time-consuming.
- Furthermore, even though the loader program may be smaller than the assembler, it does absorb a considerable amount of space

## DYNAMIC LOADING

- If the total amount of core required by all the subroutines exceeds the amount available, then it is problem
- There are several hardware techniques, such as paging and segmentation, that attempt to solve this problem;
- For loaders: A scheme is applied which uses Binders



# LOADING PROCESS

## LOADING PROCESS

- A **binder** is a program that performs the same functions as the direct-linking loader in "binding" subroutines together, but rather than placing the relocated and linked text directly into memory, it outputs the text as a file or card deck.
- This output file is in a format ready to be loaded and is typically called a **load module**.
- The module loader has to physically load the module into core.



# LOADING PROCESS

## DYNAMIC LOADING

- In each of the previous loader schemes we have assumed that all of the subroutines needed are loaded into core at the same time.
- If the total amount of core required by all these subroutines exceeds the amount available, then it is a problem

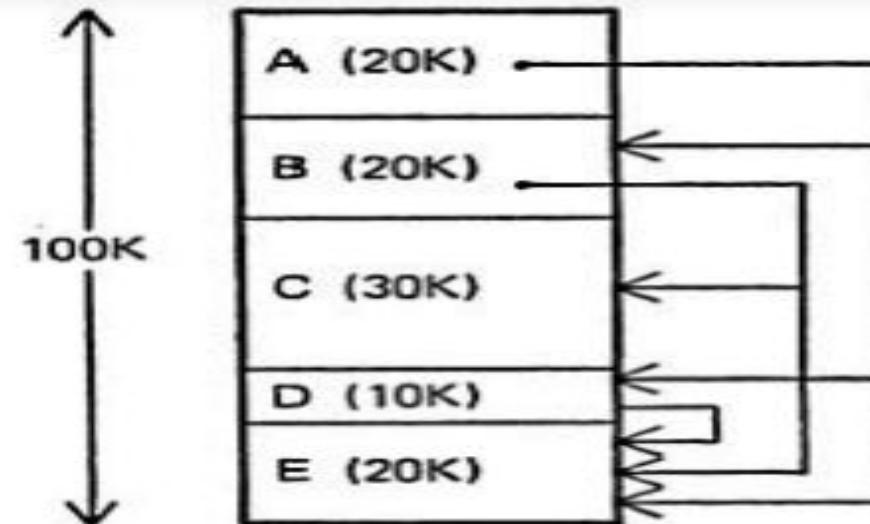
## DYNAMIC LOADING

- There are several hardware techniques, such as paging and segmentation, that attempt to solve this problem
- A dynamic loading schemes based upon the concept of a binder prior to loading can solve this problem

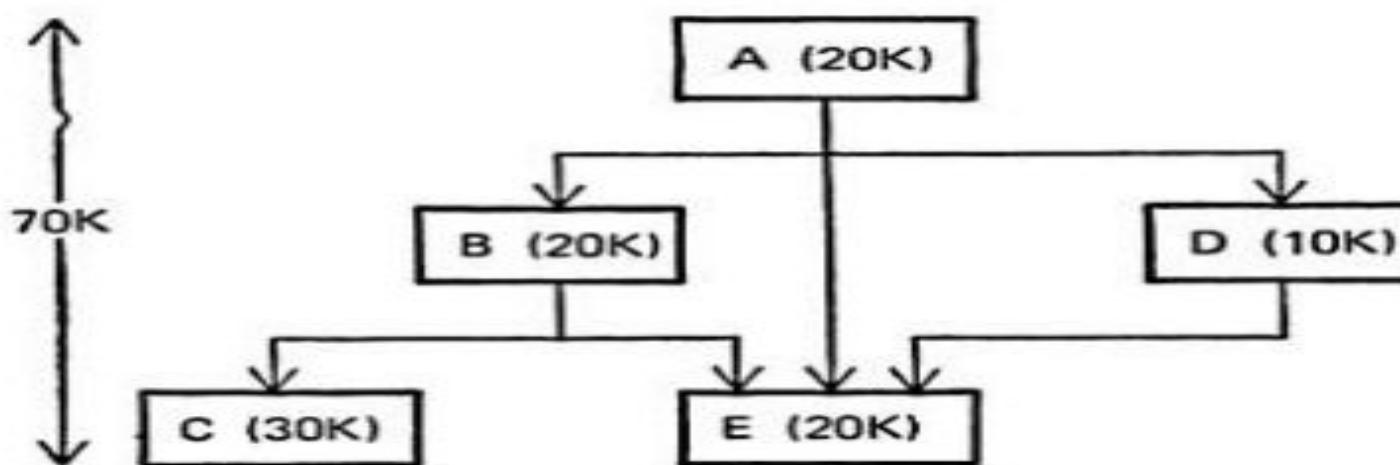
## DYNAMIC LOADING

- Usually the subroutines of a program are needed at different times
- By explicitly recognizing which subroutines call other subroutines it is possible to produce an overlay structure that identifies mutually exclusive subroutines.

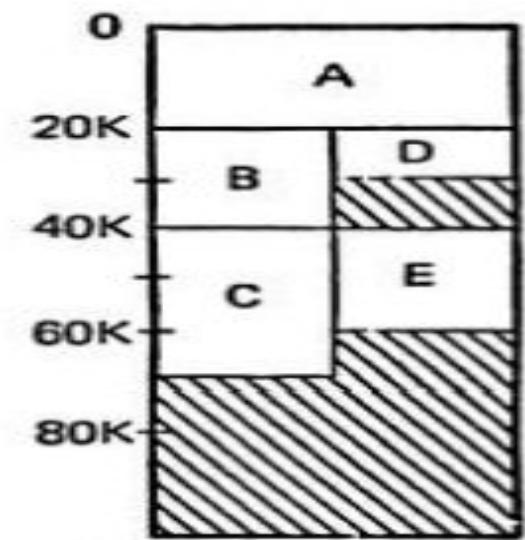
# DYNAMIC LOADING



(a) Subroutine calls between the procedures



(b) Overlay structure



(c) Possible storage assignment of each procedure

## DYNAMIC LOADING

- In the figure(a): A program consisting of five subprograms (A,B,C,D and E) that require 100K bytes of core.
- The arrows indicate that subprogram A only calls B, D and E
- Subprogram B only calls C and E; subprogram D only calls E;
- Subprograms C and E do not call any other routines.

## DYNAMIC LOADING

- In Figure 5 .9b the interdependencies between the procedures are highlighted
- Procedures B and D are never in use at the same time; neither are C and E.
- If we load only those procedures that are actually to be used at any particular time, the amount of core needed is equal to the longest path of the overlay structure.
- This happens to be 70K for the example in Figure 5.9b - procedures A, B, and C.
- Figure 5.9c illustrates a storage assignment for each procedure consistent with the overlay structure.<sup>48</sup>

## OVERLAY STRUCTURE

- In order for the overlay structure to work it is necessary for the module loader to load the various procedures as they are needed.
- There are many binders capable of processing and allocating an overlay structure.
- The portion of the loader that actually intercepts the "calls" and loads the necessary procedure is called the overlay supervisor or simply the flipper.
- This overall scheme is called dynamic loading or load-on-call (LOCAL).

## DISADVANTAGE

- If a subroutine is referenced but never executed, the loader would still incur the overhead of linking the subroutine.
- Furthermore, all of these schemes require the programmer to explicitly name all procedures that might be called.

## DYNAMIC LINKING

- This is a mechanism by which loading and linking of external references are postponed until execution time.
- The assembler produces text, binding, and relocation information from a source language deck.
- The loader loads only the main program.

## DYNAMIC LINKING

- If the main program should execute a transfer instruction to an external address, or should reference an external variable, then the loader is called.
- Only then is the segment containing the external reference is loaded.

# ADVANTAGE

- No overhead is incurred unless the procedure to be called or referenced is actually used.
- The system can be dynamically reconfigured.

# DYNAMIC LINKING VS BINDERS

	<i><b>Dynamic Linking Loader</b></i>	<i><b>Linkage Editor (Binder)</b></i>
1	<i>A Linking Loader performs linking and relocation at run time and directly loads the linked program into the memory.</i>	<i>A Linkage Editor performs linking and some relocation operations prior to load time and write the linked program to an executable for later execution.</i>
2	<i>It resolves external references every time the program is executed.</i>	<i>External references are resolved and library searching is performed only once, when the program is “link-edited”.</i>
3	<i>Executable image is not generated; the linked program is directly loaded into memory.</i>	<i>The linked programs is written into an executable image, which is later given to a relocating loader for execution.</i>
4	<i>It is more suitable for development and testing environment where the size of program is comparatively smaller.</i>	<i>It is suitable when the program has to be re-executed many times without being reassembled.</i>
5	<i>Produces output into memory.</i>	<i>Produces output on disc.</i>
6	<i>E.g. Direct-Linking Loader</i>	<i>E.g. MS-DOS Linker</i>



# MODULE-4



LOADERS AND LINKERS

# LOADER SCHEMES

**1. Compile and Go Loaders**

**2. General Loader Scheme**

**3. Absolute Loaders**

**4. Subroutine Linkages**

**5. Relocating Loaders**

**6. Direct Linking Loaders**

**7. Other Loader Schemes:**  
a) Dynamic Loading  
b) Dynamic Linking

# Content

- Subroutine Linkage
- Relocating Loader

# Subroutine Linkage

- It deals with the special mechanism for calling another subroutine in an assembly language program.

# Subroutine Linkage

- It deals with the special mechanism for calling another subroutine in an assembly language program.
- The problem of subroutine linkage is this: a main program
  - A wishes to transfer to subprogram B.

# Subroutine Linkage

- It deals with the special mechanism for calling another subroutine in an assembly language program.
- The problem of subroutine linkage is this: a main program
  - A wishes to transfer to subprogram B.
  - The programmer, in program A, could write a transfer instruction (e.g., BAL 14, B) (Branch and Link) to subprogram B.

# Subroutine Linkage

- It deals with the special mechanism for calling another subroutine in an assembly language program.
- The problem of subroutine linkage is this: a main program
  - A wishes to transfer to subprogram B.
  - The programmer, in program A, could write a transfer instruction (e.g., BAL 14, B) (Branch and Link) to subprogram B.
  - However, the assembler does not know the value of this Symbol reference and will declare it as an error (undefined Symbol) unless a special mechanism has been provided.

# Subroutine Linkage

- There are two assembler pseudo-ops which are used for subroutine linkages :
  - **EXTRN:** It is followed by a list of Symbols which indicates that these Symbols are defined in other programs but referenced in the present program.
  -

# Subroutine Linkage

- There are two assembler pseudo-ops which are used for subroutine linkages :
  - **EXTRN:** It is followed by a list of Symbols which indicates that these Symbols are defined in other programs but referenced in the present program.
  - **ENTRY:** It is followed by a list of Symbols which indicates that these Symbols are defined in same programs but referenced in the other program.

# Subroutine Linkage

- There are two assembler pseudo-ops which are used for subroutine linkages :
  - **EXTRN:** It is followed by a list of Symbols which indicates that these Symbols are defined in other programs but referenced in the present program.
  - **ENTRY:** It is followed by a list of Symbols which indicates that these Symbols are defined in same programs but referenced in the other program.
- In turn, the assembler will inform the loader that these Symbols may be referenced by other programs.

# Subroutine Linkage

- For example, the given sequence of instructions may be a simple calling sequence to another program

```
MAIN      START
          EXTRN    SUBROUT
          ...
          ...
          L        15,=A(SUBROUT) } CALL
          BALR    14,15           } SUBROUT
          ...
          ...
          END
```

# Subroutine Linkage

```
MAIN      START
          EXTRN    SUBROUT
          ...
          ...
          L        15,=A(SUBROUT) } CALL
          BALR    14,15       } SUBROUT
          ...
          ...
          END
```

- For example, the given sequence of instructions may be a simple calling sequence to another program
- The above sequence of instructions first declares SUBROUT as an external variable, that is, a variable reference but not defined in this program.

# Subroutine Linkage

```
MAIN      START
          EXTRN      SUBROUT
          ...
          ...
          L          15, =A (SUBROUT) } CALL
          BALR      14, 15        } SUBROUT
          ...
          ...
          END
```

- For example, the given sequence of instructions may be a simple calling sequence to another program
- The above sequence of instructions first declares SUBROUT as an external variable, that is, a variable reference but not defined in this program.
- The load instruction loads the address of that variable into register 15.

# Subroutine Linkage

```
MAIN      START
          EXTRN      SUBROUT
          ...
          ...
          L          15,=A(SUBROUT) } CALL
          BALR      14,15           } SUBROUT
          ...
          ...
          END
```

- For example, the given sequence of instructions may be a simple calling sequence to another program
- The above sequence of instructions first declares SUBROUT as an external variable, that is, a variable reference but not defined in this program.
- The load instruction loads the address of that variable into register 15.
- The BALR instruction branches to the contents of register 15, which is the address of SUBROUT, and leaves the value of the next Instruction in register 14.

# Subroutine Linkage

```
MAIN      START
          EXTRN      SUBROUT
          ...
          ...
          L          15, = A (SUBROUT) } CALL
          BALR      14, 15           } SUBROUT
          ...
          ...
          END
```

- For example, the given sequence of instructions may be a simple calling sequence to another program
  - The above sequence of instructions first declares SUBROUT as an external variable, that is, a variable reference but not defined in this program.
  - The load instruction loads the address of that variable into register 15.
  - The BALR instruction branches to the contents of register 15, which is the address of SUBROUT, and leaves the value of the next Instruction in register 14.
  - In most assemblies, we may simply use a CALL SUBROUT macro, which is translated by the assembler into a calling sequence



# RELOCATING LOADER

# Relocating Loader

Relocation:

- The process of updating the addresses used in the address sensitive instruction of program

# Relocating Loader

Relocation:

- The process of updating the addresses used in the address sensitive instruction of program
- The object code is executed after loading at storage locations

# Relocating Loader

Relocation:

- The process of updating the addresses used in the address sensitive instruction of program
- The object code is executed after loading at storage locations
- The addresses of such object code will be specified only after the assembly process is over.

# Relocating Loader

Relocation:

- The process of updating the addresses used in the address sensitive instruction of program
- The object code is executed after loading at storage locations
- The addresses of such object code will be specified only after the assembly process is over.
- Therefore, after loading,

**Address of object code = Only Address of object code +  
Relocation constant**

# Relocating Loader

## Advantages:

- To avoid possible reassembling of all subroutines when a single subroutine is changed
- To perform the tasks of allocation and linking for the programmer

# Relocating Loader

## Advantages:

- To avoid possible reassembling of all subroutines when a single subroutine is changed
- To perform the tasks of allocation and linking for the programmer

## Tasks Performed:

- Allocation
- Relocation
- Linking
- Loading

# Relocating Loader

## Binary Symbolic Subroutine Loader (BSS)

- The BSS loader allows many procedure segments

# Relocating Loader

## Binary Symbolic Subroutine Loader (BSS)

- The BSS loader allows many procedure segments
- It allows only one data segment (common segment)

# Relocating Loader

Binary Symbolic  
Subroutine Loader  
(BSS)

- The BSS loader allows many procedure segments
- It allows only one data segment (common segment)
- The assembler assembles each procedure segment independently

# Relocating Loader

Binary Symbolic  
Subroutine Loader  
(BSS)

- The BSS loader allows many procedure segments
- It allows only one data segment (common segment)
- The assembler assembles each procedure segment independently
- Then it passes the text and information to the loader as relocation and intersegment references.

# Binary Symbolic Subroutine Loader (BSS)

- The output of a relocating assembler using a BSS scheme is the object program and information about all other programs it references.

# Binary Symbolic Subroutine Loader (BSS)

- The output of a relocating assembler using a BSS scheme is the object program and information about all other programs it references.
- In addition, there is information (relocation information) for **locations in this program that need to be changed** if it is to be loaded in an arbitrary place in core, i.e . the locations which are dependent on the core allocation.

# Binary Symbolic Subroutine Loader (BSS)

- The output of a relocating assembler using a BSS scheme is the object program and information about all other programs it references.
- In addition, there is information (relocation information) for **locations in this program that need to be changed** if it is to be loaded in an arbitrary place in core, i.e . the locations which are dependent on the core allocation.
- For each source program the assembler outputs a text (machine translation of the program) prefixed by a transfer vector that consists of addresses containing **names of the subroutines** referenced by the source program

# Binary Symbolic Subroutine Loader (BSS)

Process of Relocation:

For example, if a square Root Routine (SQRT) was referenced and was the first subroutine called,

# Binary Symbolic Subroutine Loader (BSS)

Process of Relocation:

For example, if a square Root Routine (SQRT) was referenced and was the first subroutine called,

- I. The first location in the transfer vector would contain the symbolic name SQRT.

# Binary Symbolic Subroutine Loader (BSS)

Process of Relocation:

For example, if a square Root Routine (SQRT) was referenced and was the first subroutine called,

1. The first location in the transfer vector would contain the symbolic name SQRT.
2. The statement calling SQRT would be translated into a transfer instruction
  - It indicates a branch to the location of the transfer vector associated with SQRT

# Binary Symbolic Subroutine Loader (BSS)

Process of Relocation:

For example, if a square Root Routine (SQRT) was referenced and was the first subroutine called,

1. The first location in the transfer vector would contain the symbolic name SQRT.
2. The statement calling SQRT would be translated into a transfer instruction
  - It indicates a branch to the location of the transfer vector associated with SQRT
3. The assembler would also provide the loader with additional information
  - The length of the entire program
  - The length of the transfer vector portion.

# Binary Symbolic Subroutine Loader (BSS)

Process of Relocation:

For example, if a square Root Routine (SQRT) was referenced and was the first subroutine called,

1. The first location in the transfer vector would contain the symbolic name SQRT.
2. The statement calling SQRT would be translated into a transfer instruction
  - It indicates a branch to the location of the transfer vector associated with SQRT
3. The assembler would also provide the loader with additional information
  - The length of the entire program
  - The length of the transfer vector portion.
4. After loading the text and the transfer vector into core, the loader would load each subroutine identified in the transfer vector.

# Binary Symbolic Subroutine Loader (BSS)

Process of Relocation:

For example, if a square Root Routine (SQRT) was referenced and was the first subroutine called,

1. The first location in the transfer vector would contain the symbolic name SQRT.
2. The statement calling SQRT would be translated into a transfer instruction
  - It indicates a branch to the location of the transfer vector associated with SQRT
3. The assembler would also provide the loader with additional information
  - The length of the entire program
  - The length of the transfer vector portion.
4. After loading the text and the transfer vector into core, the loader would load each subroutine identified in the transfer vector.
5. It would then place a transfer instruction to the corresponding subroutine in each entry in the transfer vector.

# Binary Symbolic Subroutine Loader (BSS)

- The execution of the call SQRT statement would result in
  - a branch to the first location in the transfer vector,
  - It would contain a transfer instruction to the location of SQRT.

# Binary Symbolic Subroutine Loader (BSS)

- The execution of the call SQRT statement would result in
  - a branch to the first location in the transfer vector,
  - It would contain a transfer instruction to the location of SQRT.
- Problem:
  - it is necessary to relocate the address portion of every instruction
  - computers with a direct address instruction format have a much more severe relocation problem

# Binary Symbolic Subroutine Loader (BSS)

- The execution of the call SQRT statement would result in
  - a branch to the first location in the transfer vector,
  - It would contain a transfer instruction to the location of SQRT.
- Problem:
  - it is necessary to relocate the address portion of every instruction
  - computers with a direct address instruction format have a much more severe relocation problem
- Solution:
  - The assembler associates a bit with each Instruction or address field.
  - If this bit equals one, then the corresponding address field must be relocated otherwise the field is not relocated.
  - These relocation indicators, are known as relocation bits and are included in the object deck.

# Binary Symbolic Subroutine Loader (BSS)

Source program						
MAIN	START					
	EXTRN	SQRT				
	EXTRN	ERR				
	ST	14,SAVE	Save return address			
	L	1,=F'9'	Load test value	0	00	'SQRT'
	BAL	14,SQRT	Call SQRT	4	00	'ERRb'
	C	1,=F'3'	Compare answer	8	01	ST 14,36
	BNE	ERR	Transfer to ERR	12	01	L 1,40
	L	14,SAVE	Get return address	16	01	BAL 14,0
	BR	14	Return to caller	20	01	C 1,44
	SAVE	DS F	Temp. loc.	24	01	BC 7,4
	END			28	01	L 14,36
				32	0	BCR 15,14
				34	0	(Skipped for alignment)
				36	00	(Temp location)
				40	00	9
				44	00	3

- The figure illustrates a simple assembly language program written for a hypothetical "direct-address" 360 that uses a BSS loader.

FIGURE 5.5 Assembly of program for "direct-address" 360

# Binary Symbolic Subroutine Loader (BSS)

<i>Source program</i>				Program length = 48 bytes Transfer vector = 8 bytes		
				Rel. addr.	Relocation	Object code
MAIN	START			0	00	'SQRT'
	EXTRN	SQRT		4	00	'ERRb'
	EXTRN	ERR		8	01	ST 14,36
	ST	14,SAVE	Save return address	12	01	L 1,40
	L	1,=F'9'	Load test value	16	01	BAL 14,0
	BAL	14,SQRT	Call SQRT	20	01	C 1,44
	C	1,=F'3'	Compare answer	24	01	BC 7,4
	BNE	ERR	Transfer to ERR	28	01	L 14,36
	L	14,SAVE	Get return address	32	0	BCR 15,14
	BR	14	Return to caller	34	0	(Skipped for alignment)
	SAVE	DS F	Temp. loc.	36	00	(Temp location)
		END		40	00	9
				44	00	3

- The figure illustrates a simple assembly language program written for a hypothetical "direct-address" 360 that uses a BSS loader.
- It supposedly calls the SQRT subroutine to get the square root of 9.

FIGURE 5.5 Assembly of program for "direct-address" 360

# Binary Symbolic Subroutine Loader (BSS)

Source program				Object code		
				Rel. addr.	Relocation	Object code
MAIN	START			0	00	'SQRT'
	EXTRN	SQRT		4	00	'ERRb'
	EXTRN	ERR		8	01	ST 14,36
	ST	14,SAVE	Save return address	12	01	L 1,40
	L	1,=F'9'	Load test value	16	01	BAL 14,0
	BAL	14,SQRT	Call SQRT	20	01	C 1,44
	C	1,=F'3'	Compare answer	24	01	BC 7,4
	BNE	ERR	Transfer to ERR	28	01	L 14,36
	L	14,SAVE	Get return address	32	0	BCR 15,14
	BR	14	Return to caller	34	0	(Skipped for alignment)
	SAVE	DS F	Temp. loc.	36	00	(Temp location)
		END		40	00	9
				44	00	3

FIGURE 5.5 Assembly of program for "direct-address" 360

- The figure illustrates a simple assembly language program written for a hypothetical "direct-address" 360 that uses a BSS loader.
- It supposedly calls the SQRT subroutine to get the square root of 9.
- If the result is not 3, it transfers to a subroutine called ERR.

# Binary Symbolic Subroutine Loader (BSS)

Source program				Object code		
				Rel. addr.	Relocation	Object code
MAIN	START	SQRT		0	00	'SQRT'
	EXTRN	ERR		4	00	'ERRb'
	ST	14,SAVE	Save return address	8	01	ST 14,36
	L	1,=F'9'	Load test value	12	01	L 1,40
	BAL	14,SQRT	Call SQRT	16	01	BAL 14,0
	C	1,=F'3'	Compare answer	20	01	C 1,44
	BNE	ERR	Transfer to ERR	24	01	BC 7,4
	L	14,SAVE	Get return address	28	01	L 14,36
	BR	14	Return to caller	32	0	BCR 15,14
SAVE	DS	F	Temp. loc.	34	0	(Skipped for alignment)
	END			36	00	(Temp location)
				40	00	9
				44	00	3

FIGURE 5.5 Assembly of program for "direct-address" 360

- The figure illustrates a simple assembly language program written for a hypothetical "direct-address" 360 that uses a BSS loader.
- It supposedly calls the SQRT subroutine to get the square root of 9.
- If the result is not 3, it transfers to a subroutine called ERR.
- Since this is a direct-address computer, there is no base register field in the object code and no need for a USING pseudo-op in the source program.

# Binary Symbolic Subroutine Loader (BSS)

<i>Source program</i>				Program length = 48 bytes Transfer vector = 8 bytes		
				Rel. addr.	Relocation	Object code
MAIN	START			0	00	'SQRT'
	EXTRN	SQRT		4	00	'ERRb'
	EXTRN	ERR		8	01	ST 14,36
	ST	14,SAVE	Save return address	12	01	L 1,40
	L	1,=F'9'	Load test value	16	01	BAL 14,0
	C	1,=F'3'	Compare answer	20	01	C 1,44
	BNE	ERR	Transfer to ERR	24	01	BC 7,4
	L	14,SAVE	Get return address	28	01	L 14,36
	BR	14	Return to caller	32	0	BCR 15,14
	SAVE	DS F	Temp. loc.	34	0	(Skipped for alignment)
	END			36	00	(Temp location)
				40	00	9
				44	00	3

- The EXTRN pseudo op identifies the symbols SQRT and ERR as the names of other subroutines

FIGURE 5.5 Assembly of program for "direct-address" 360

# Binary Symbolic Subroutine Loader (BSS)

<i>Source program</i>				Program length = 48 bytes Transfer vector = 8 bytes		
				Rel. addr.	Relocation	Object code
MAIN	START			0	00	'SQRT'
	EXTRN	SQRT		4	00	'ERRb'
	EXTRN	ERR		8	01	ST 14,36
	ST	14,SAVE	Save return address	12	01	L 1,40
	L	1,=F'9'	Load test value	16	01	BAL 14,0
	C	14,SQRT	Call SQRT	20	01	C 1,44
	BAL	1,=F'3'	Compare answer	24	01	BC 7,4
	C	1,=F'3'		28	01	L 14,36
	BNE	ERR	Transfer to ERR	32	0	BCR 15,14
	L	14,SAVE	Get return address	34	0	(Skipped for alignment)
	BR	14	Return to caller	36	00	(Temp location)
	SAVE	DS	F Temp. loc.	40	00	9
		END		44	00	3

- The EXTRN pseudo op identifies the symbols SQRT and ERR as the names of other subroutines
- Since the locations of these symbols are not defined in this subroutine, they are called external symbols.

FIGURE 5.5 Assembly of program for "direct-address" 360

# Binary Symbolic Subroutine Loader (BSS)

<i>Source program</i>				Program length = 48 bytes Transfer vector = 8 bytes		
				Rel. addr.	Relocation	Object code
MAIN	START			0	00	'SQRT'
	EXTRN	SQRT		4	00	'ERRb'
	EXTRN	ERR		8	01	ST      14,36
	ST	14,SAVE	Save return address	12	01	L      1,40
	L	1,=F'9'	Load test value	16	01	BAL    14,0
	BAL	14,SQRT	Call SQRT	20	01	C      1,44
	C	1,=F'3'	Compare answer	24	01	BC     7,4
	BNE	ERR	Transfer to ERR	28	01	L      14,36
	L	14,SAVE	Get return address	32	0	BCR    15,14
	BR	14	Return to caller	34	0	(Skipped for alignment)
	SAVE	DS	F Temp. loc.	36	00	(Temp location)
		END		40	00	9
				44	00	3

- The EXTRN pseudo op identifies the symbols SQRT and ERR as the names of other subroutines
- Since the locations of these symbols are not defined in this subroutine, they are called external symbols.
- For each external symbol the assembler generates a four byte fullword at the beginning of the program, containing the EBCDIC characters for the symbol

FIGURE 5.5 Assembly of program for "direct-address" 360

# Binary Symbolic Subroutine Loader (BSS)

<i>Source program</i>				Program length = 48 bytes Transfer vector = 8 bytes		
				Rel. addr.	Relocation	Object code
MAIN	START			0	00	'SQRT'
	EXTRN	SQRT		4	00	'ERRb'
	EXTRN	ERR		8	01	ST      14,36
	ST	14,SAVE	Save return address	12	01	L      1,40
	L	1,=F'9'	Load test value	16	01	BAL    14,0
	BAL	14,SQRT	Call SQRT	20	01	C      1,44
	C	1,=F'3'	Compare answer	24	01	BC     7,4
	BNE	ERR	Transfer to ERR	28	01	L      14,36
	L	14,SAVE	Get return address	32	0	BCR    15,14
	BR	14	Return to caller	34	0	(Skipped for alignment)
	SAVE	DS	F Temp. loc.	36	00	(Temp location)
		END		40	00	9
				44	00	3

FIGURE 5.5 Assembly of program for "direct-address" 360

- The EXTRN pseudo op identifies the symbols SQRT and ERR as the names of other subroutines
- Since the locations of these symbols are not defined in this subroutine, they are called external symbols.
- For each external symbol the assembler generates a four byte fullword at the beginning of the program, containing the EBCDIC characters for the symbol
- These extra words are called transfer vectors.

# Binary Symbolic Subroutine Loader (BSS)

- Every reference to an external symbol is assigned the address of the corresponding transfer vector word

Source program				Program length = 48 bytes Transfer vector = 8 bytes		
				Rel. addr.	Relocation	Object code
MAIN	START			0	00	'SQRT'
	EXTRN	SQRT		4	00	'ERRb'
	EXTRN	ERR		8	01	ST 14,36
	ST	14,SAVE	Save return address	12	01	L 1,40
	L	1,=F'9'	Load test value	16	01	BAL 14,0
	C	1,=F'3'	Compare answer	20	01	C 1,44
	BNE	ERR	Transfer to ERR	24	01	BC 7,4
	L	14,SAVE	Get return address	28	01	L 14,36
	BR	14	Return to caller	32	0	BCR 15,14
	SAVE	DS F	Temp. loc.	34	0	(Skipped for alignment)
	END			36	00	(Temp location)
				40	00	9
				44	00	3

FIGURE 5.5 Assembly of program for "direct-address" 360

# Binary Symbolic Subroutine Loader (BSS)

<i>Source program</i>				Program length = 48 bytes Transfer vector = 8 bytes		
				<i>Rel.</i> <i>addr.</i>	<i>Relocation</i>	<i>Object code</i>
MAIN	START			0	00	'SQRT'
	EXTRN	SQRT		4	00	'ERRb'
	EXTRN	ERR		8	01	ST 14,36
	ST	14,SAVE	Save return address	12	01	L 1,40
	L	1,=F'9'	Load test value	16	01	BAL 14,0
	C	1,=F'3'	Compare answer	20	01	C 1,44
	BNE	ERR	Transfer to ERR	24	01	BC 7,4
	L	14,SAVE	Get return address	28	01	L 14,36
	BR	14	Return to caller	32	0	BCR 15,14
SAVE	DS	F	Temp. loc.	34	0	(Skipped for alignment)
	END			36	00	(Temp location)
				40	00	9
				44	00	3

- Every reference to an external symbol is assigned the address of the corresponding transfer vector word
- In addition, for every halfword (two bytes) in the program, the assembler produces a separate relocation bit.

FIGURE 5.5 Assembly of program for "direct-address" 360

# Binary Symbolic Subroutine Loader (BSS)

<i>Source program</i>				<i>Object code</i>		
				<i>Rel.</i> <i>addr.</i>	<i>Relocation</i>	
MAIN	START			0	00	'SQRT'
	EXTRN	SQRT		4	00	'ERRb'
	EXTRN	ERR		8	01	ST      14,36
	ST	14,SAVE	Save return address	12	01	L      1,40
	L	1,=F'9'	Load test value	16	01	BAL    14,0
	C	14,SQRT	Call SQRT	20	01	C      1,44
	BAL	1,=F'3'	Compare answer	24	01	BC     7,4
	C	1,=F'3'		28	01	L      14,36
	BNE	ERR	Transfer to ERR	32	0	BCR   15,14
	L	14,SAVE	Get return address	34	0	(Skipped for alignment)
	BR	14	Return to caller	36	00	(Temp location)
	SAVE	DS	F Temp. loc.	40	00	9
		END		44	00	3

FIGURE 5.5 Assembly of program for "direct-address" 360

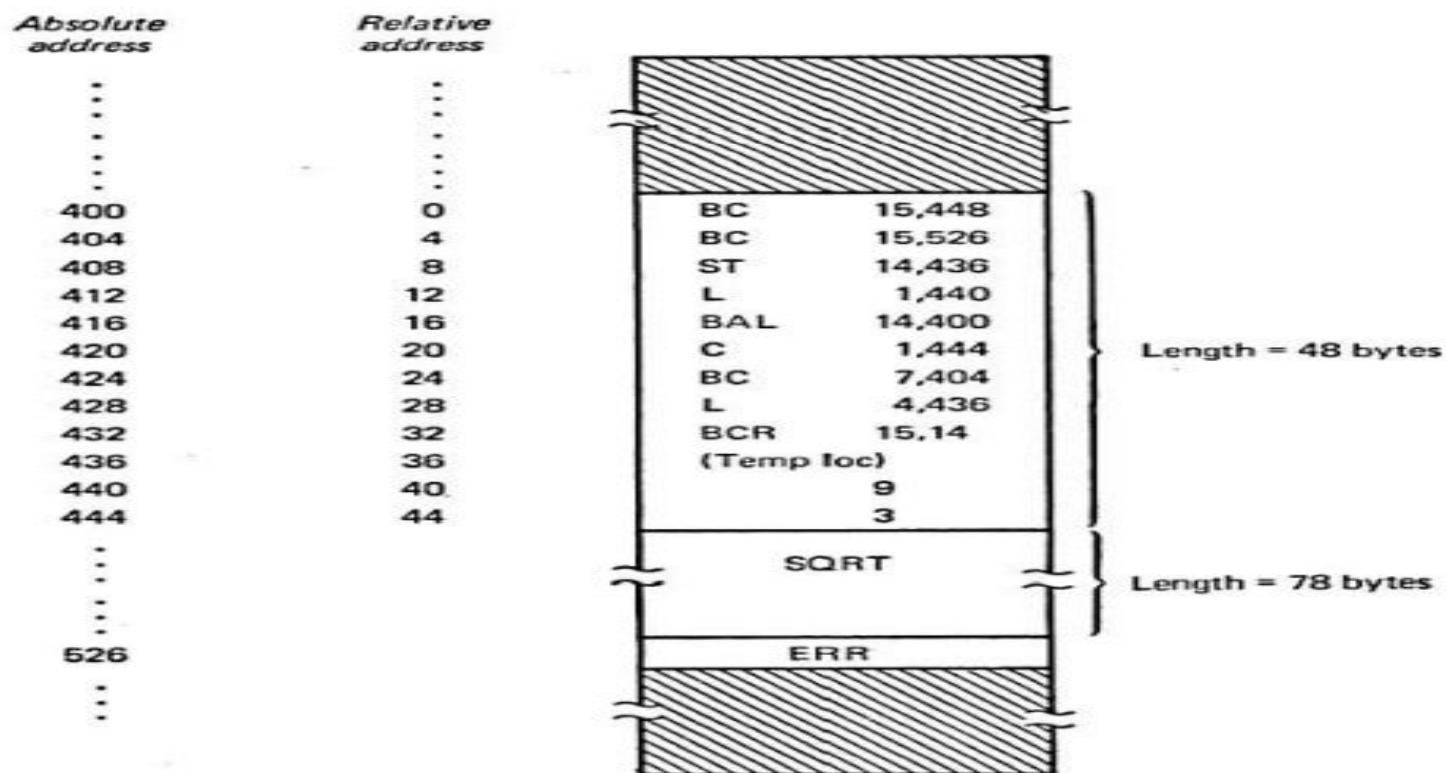
- Every reference to an external symbol is assigned the address of the corresponding transfer vector word
- In addition, for every halfword (two bytes) in the program, the assembler produces a separate relocation bit.

For example,

The assembled instruction ST 14,36 is assigned relocation bits 01 since the first halfword contains the op-code, register field, and index field which should not be relocated

but the second halfword contains the relative address 36, which must be relocated.

# Binary Symbolic Subroutine Loader (BSS)



- The figure illustrates the contents of memory after the programs have been loaded by the BSS loader.

FIGURE 5.6 BSS loading of programs for "direct access" 360

# Binary Symbolic Subroutine Loader (BSS)

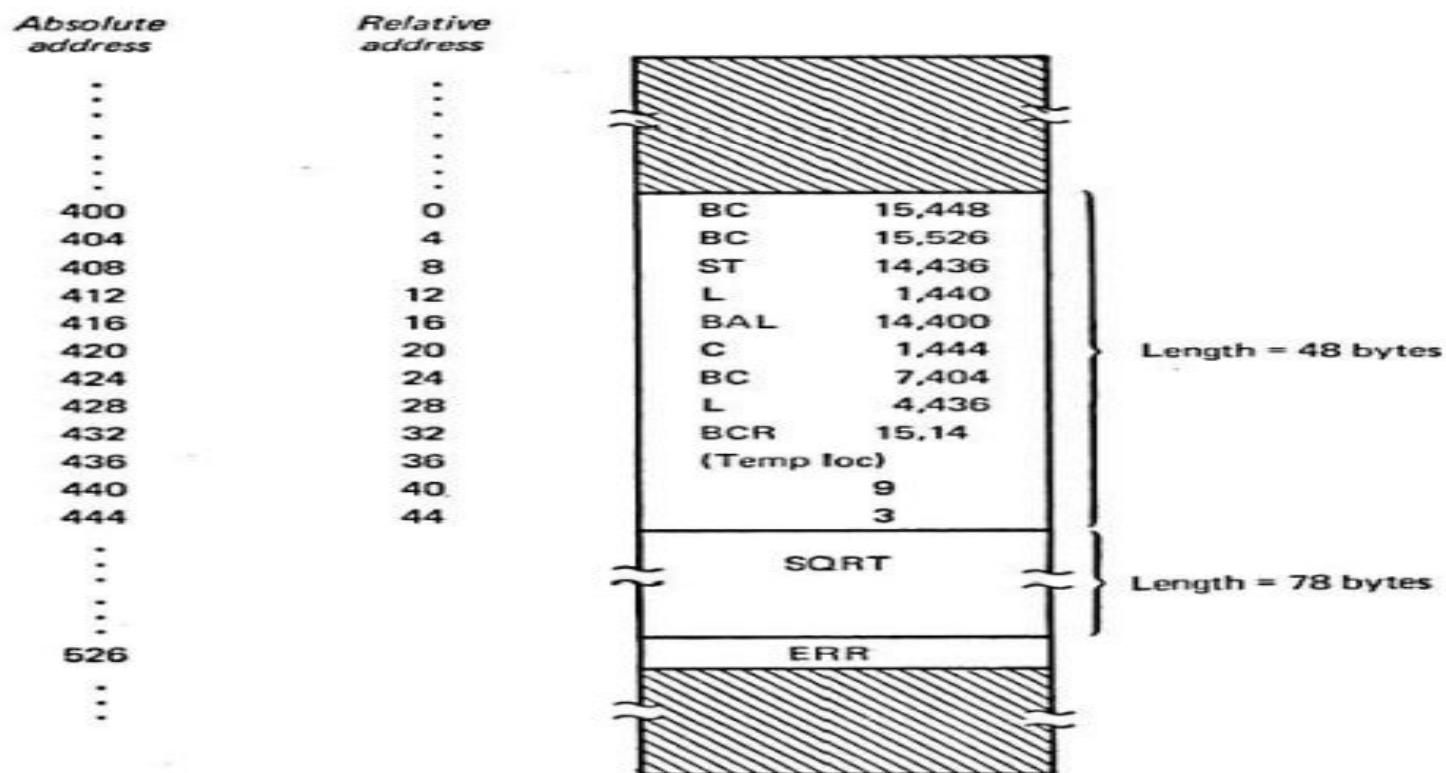
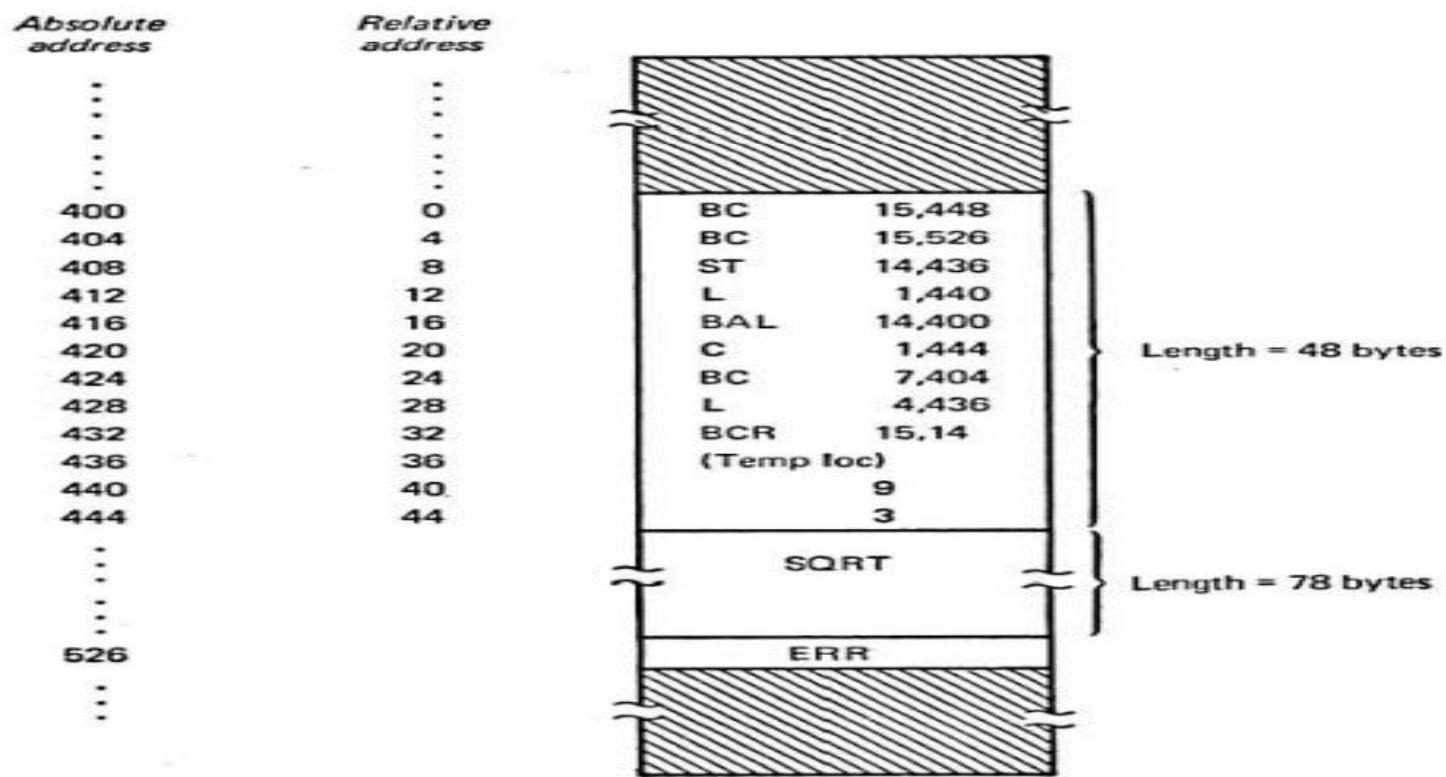


FIGURE 5.6 BSS loading of programs for "direct access" 360

- The figure illustrates the contents of memory after the programs have been loaded by the BSS loader.
- Based upon the relocation bits, the loader has relocated the address fields to correspond to the allocated address of MAIN which is 400.

# Binary Symbolic Subroutine Loader (BSS)



- The figure illustrates the contents of memory after the programs have been loaded by the BSS loader.
- Based upon the relocation bits, the loader has relocated the address fields to correspond to the allocated address of MAIN which is 400.
- Using the program length information, the loader placed the sub routines SQRT and ERR at the next available locations which were 448 and 526, respectively

FIGURE 5.6 BSS loading of programs for "direct access" 360

# Binary Symbolic Subroutine Loader (BSS)

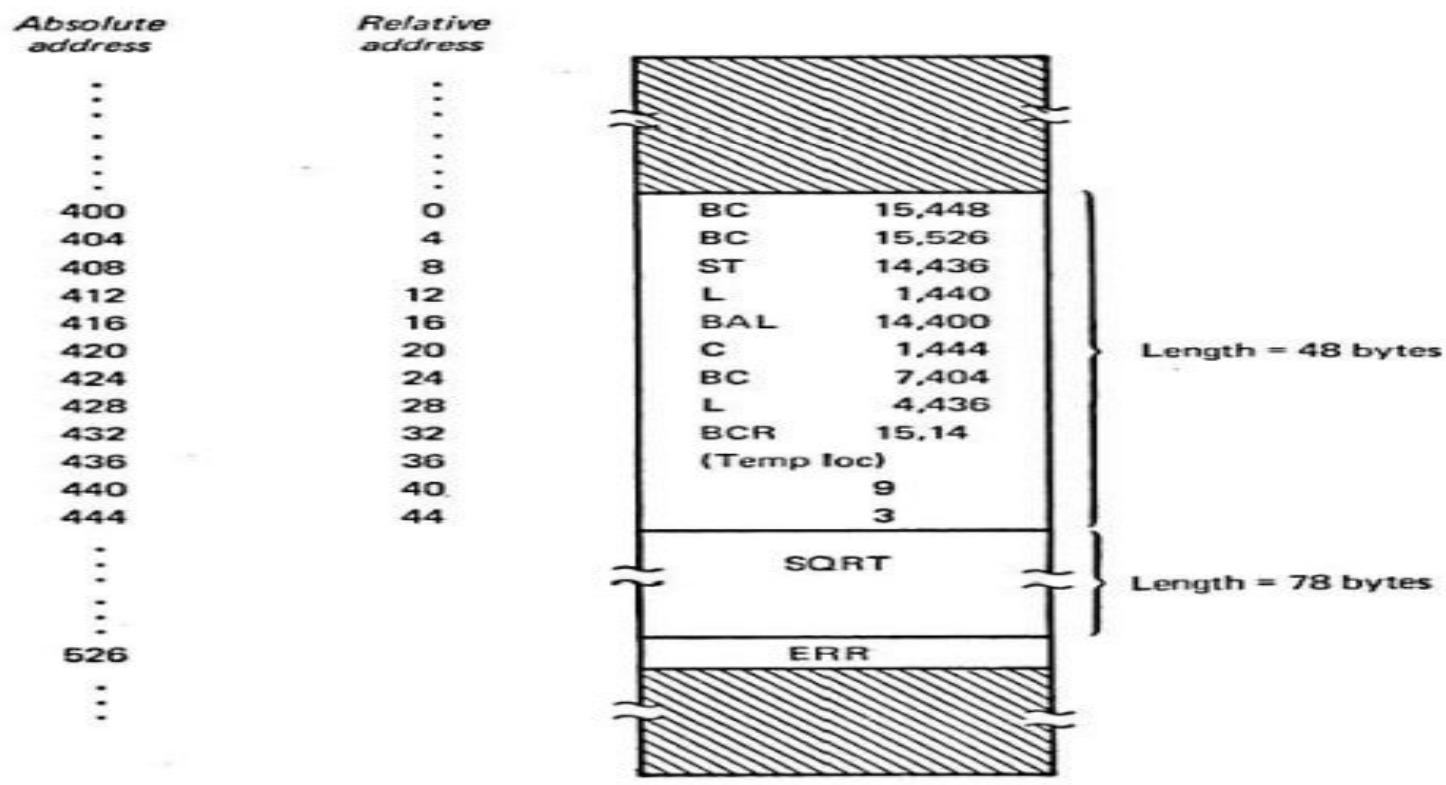


FIGURE 5.6 BSS loading of programs for "direct access" 360

- The figure illustrates the contents of memory after the programs have been loaded by the BSS loader.
- Based upon the relocation bits, the loader has relocated the address fields to correspond to the allocated address of MAIN which is 400.
- Using the program length information, the loader placed the sub routines SQRT and ERR at the next available locations which were 448 and 526, respectively
- Finally, the transfer vector words were changed to contain branch instructions to the corresponding subroutines

# Binary Symbolic Subroutine Loader (BSS)

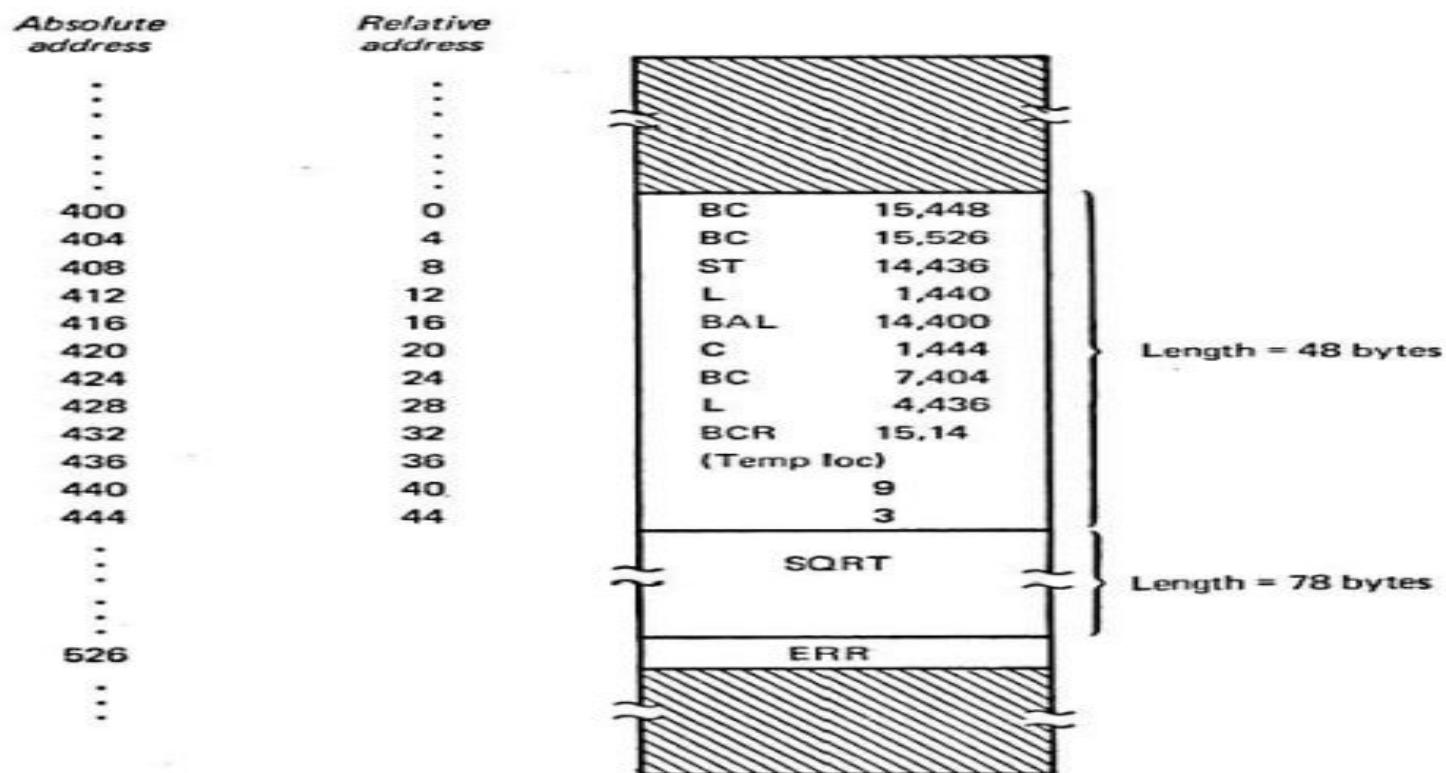


FIGURE 5.6 BSS loading of programs for "direct access" 360

Thus, the four functions of the loader (allocation, linking, relocation, and loading) were all performed automatically by the BSS loader.

# Binary Symbolic Subroutine Loader (BSS)

Disadvantages:

1. The transfer vector linkage is only useful for transfers, and is not well-suited for loading or storing external data (data located in another procedure segment).
2. The transfer vector increases the size of the object program in memory.
3. The BSS loader, processes procedure segments but does not facilitate access to data segments that can be shared.

# Dynamic Loading

- If the total amount of core required by all the subroutines exceeds the amount available, then it is problem
- There are several hardware techniques, such as paging and segmentation, that attempt to solve this problem;
- For loaders: A scheme is applied which uses Binders

# Loading Process

- A **binder** is a program that performs the same functions as the direct-linking loader in "binding" subroutines together, but rather than placing the relocated and linked text directly into memory, it outputs the text as a file or card deck.
- This output file is in a format ready to be loaded and is typically called a load module.
- Binder: Allocation, Relocation, Linking
- **The module loader** has to physically load the module into core.
- Module Loader: Task of Loading

# Dynamic Loading

- In each of the previous loader schemes we have assumed that all of the subroutines needed are loaded into core at the same time.

# Dynamic Loading

- In each of the previous loader schemes we have assumed that all of the subroutines needed are loaded into core at the same time.
- If the total amount of core required by all these subroutines exceeds the amount available, then it is a problem

# Dynamic Loading

- In each of the previous loader schemes we have assumed that all of the subroutines needed are loaded into core at the same time.
- If the total amount of core required by all these subroutines exceeds the amount available, then it is a problem
- There are several hardware techniques, such as paging and segmentation, that attempt to solve this problem
-

# Dynamic Loading

- In each of the previous loader schemes we have assumed that all of the subroutines needed are loaded into core at the same time.
- If the total amount of core required by all these subroutines exceeds the amount available, then it is a problem
- There are several hardware techniques, such as paging and segmentation, that attempt to solve this problem
- A dynamic loading schemes based upon the concept of a binder prior to loading can solve this problem

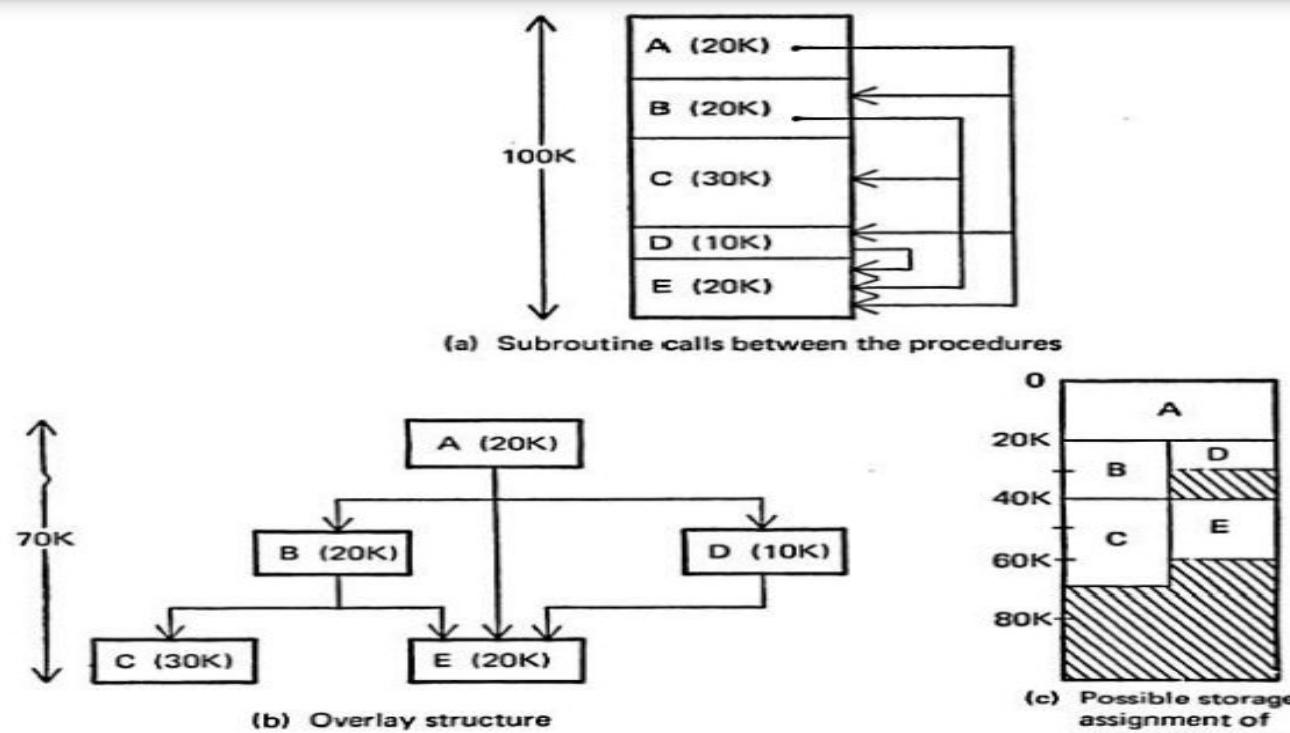
# Dynamic Loading

- In each of the previous loader schemes we have assumed that all of the subroutines needed are loaded into core at the same time.
- If the total amount of core required by all these subroutines exceeds the amount available, then it is a problem
- There are several hardware techniques, such as paging and segmentation, that attempt to solve this problem
- A dynamic loading schemes based upon the concept of a binder prior to loading can solve this problem
- Usually the subroutines of a program are needed at different times

# Dynamic Loading

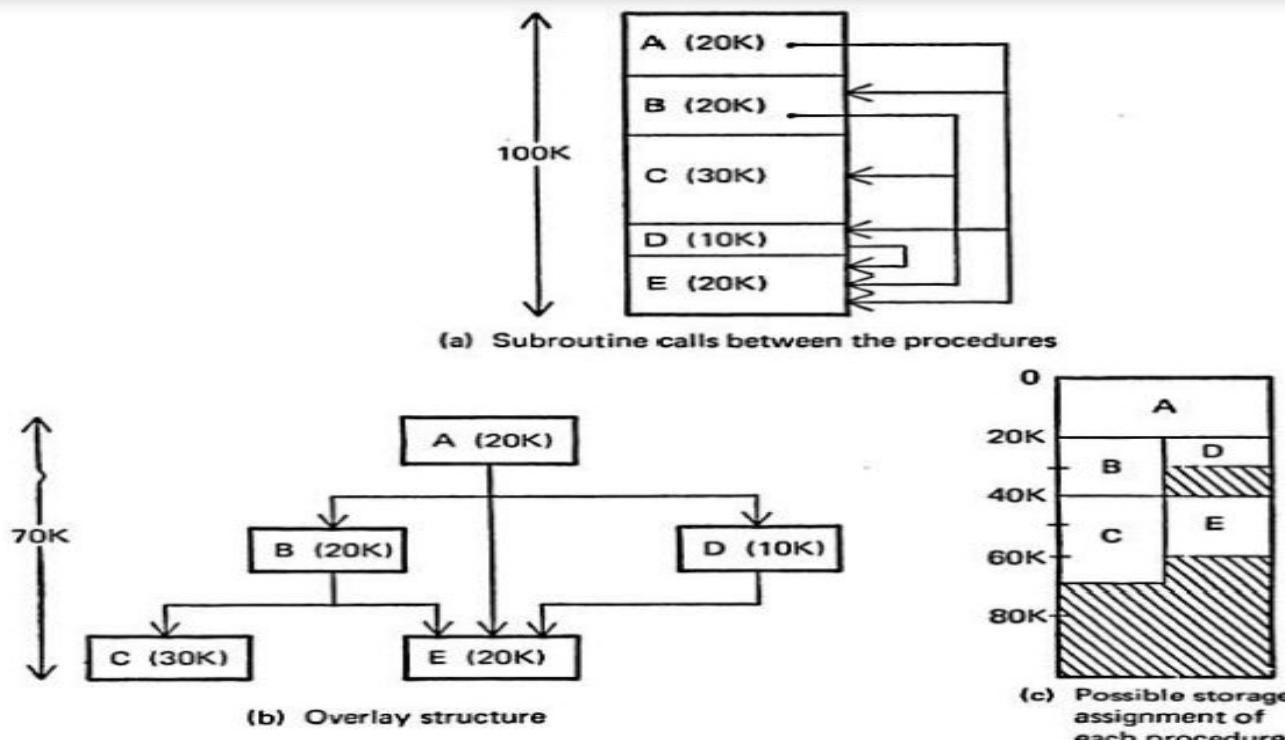
- By explicitly recognizing which subroutines call other subroutines it is possible to produce an overlay structure that identifies mutually exclusive subroutines.

# Dynamic Loading



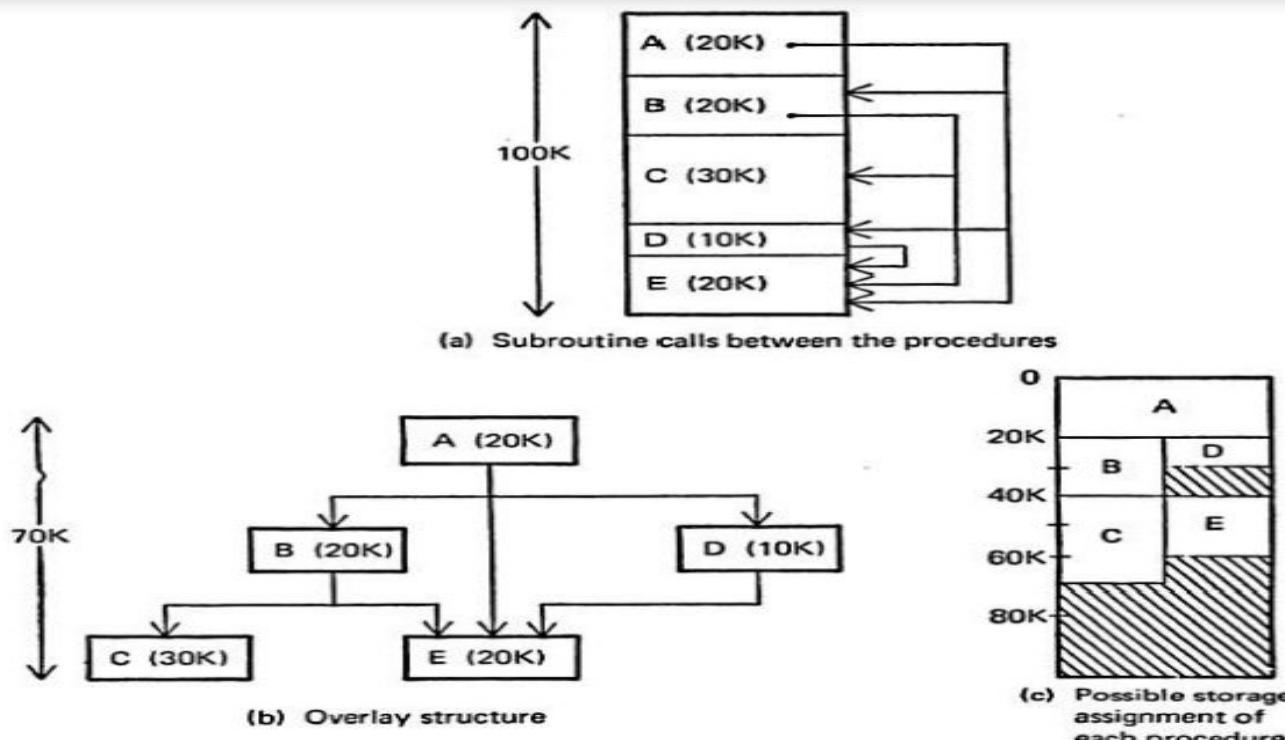
- In the figure(a): A program consisting of five subprograms (A,B,C,D and E) that require 100 K bytes of core.
- The arrows indicate that subprogram A only calls B, D and E
- Subprogram B only calls C and E; subprogram D only calls E;
- Subprograms C and E do not call any other routines.

# Dynamic Loading



- In Figure 5.9b the inter-dependencies between the procedures are highlighted
- Procedures B and D are never in use at the same time; neither are C and E.
- If we load only those procedures that are actually to be used at any particular time, the amount of core needed is equal to the longest path of the overlay structure.
- This happens to be 70K for the example in Figure 5.9b - procedures A, B, and C.
- Figure 5.9c illustrates a storage assignment for each procedure consistent with the overlay structure.

# Dynamic Loading



- In Figure 5.9b the inter-dependencies between the procedures are highlighted
- Procedures B and D are never in use at the same time; neither are C and E.
- If we load only those procedures that are actually to be used at any particular time, the amount of core needed is equal to the longest path of the overlay structure.
- This happens to be 70K for the example in Figure 5.9b - procedures A, B, and C.
- Figure 5.9c illustrates a storage assignment for each procedure consistent with the overlay structure.

# Overlay Structure

- In order for the overlay structure to work it is necessary for the module loader to load the various procedures as they are needed.

# Overlay Structure

- In order for the overlay structure to work it is necessary for the module loader to load the various procedures as they are needed.
- There are many binders capable of processing and allocating an overlay structure.

# Overlay Structure

- In order for the overlay structure to work it is necessary for the module loader to load the various procedures as they are needed.
- There are many binders capable of processing and allocating an overlay structure.
- The portion of the loader that actually intercepts the "calls" and loads the necessary procedure is called the overlay supervisor or simply the flipper.

# Overlay Structure

- In order for the overlay structure to work it is necessary for the module loader to load the various procedures as they are needed.
- There are many binders capable of processing and allocating an overlay structure.
- The portion of the loader that actually intercepts the "calls" and loads the necessary procedure is called the overlay supervisor or simply the flipper.
- This overall scheme is called dynamic loading or load-on-call (LOCAL).

# Overlay Structure

## Disadvantage

- If a subroutine is referenced but never executed, the loader would still incur the overhead of linking the subroutine.
- Furthermore, all of these schemes require the programmer to explicitly name all procedures that might be called.

# Dynamic Linking

- This is a mechanism by which loading and linking of external references are postponed until execution time.

# Dynamic Linking

- This is a mechanism by which loading and linking of external references are postponed until execution time.
- The assembler produces text, binding, and relocation information from a source language deck.

# Dynamic Linking

- This is a mechanism by which loading and linking of external references are postponed until execution time.
- The assembler produces text, binding, and relocation information from a source language deck.
- The loader loads only the main program.

# Dynamic Linking

- This is a mechanism by which loading and linking of external references are postponed until execution time.
- The assembler produces text, binding, and relocation information from a source language deck.
- The loader loads only the main program.
- If the main program should execute a transfer instruction to an external address, or should reference an external variable, then the loader is called.

# Dynamic Linking

- This is a mechanism by which loading and linking of external references are postponed until execution time.
- The assembler produces text, binding, and relocation information from a source language deck.
- The loader loads only the main program.
- If the main program should execute a transfer instruction to an external address, or should reference an external variable, then the loader is called.
- Only then is the segment containing the external reference is loaded.

# Dynamic Linking

## Advantage:

- No overhead is incurred unless the procedure to be called or referenced is actually used.
- The system can be dynamically reconfigured.

# DYNAMIC LINKING VS BINDERS

	<i><b>Dynamic Linking Loader</b></i>	<i><b>Linkage Editor (Binder)</b></i>
1	<i>A Linking Loader performs linking and relocation at run time and directly loads the linked program into the memory.</i>	<i>A Linkage Editor performs linking and some relocation operations prior to load time and write the linked program to an executable for later execution.</i>
2	<i>It resolves external references every time the program is executed.</i>	<i>External references are resolved and library searching is performed only once, when the program is “link-edited”.</i>
3	<i>Executable image is not generated; the linked program is directly loaded into memory.</i>	<i>The linked programs is written into an executable image, which is later given to a relocating loader for execution.</i>
4	<i>It is more suitable for development and testing environment where the size of program is comparatively smaller.</i>	<i>It is suitable when the program has to be re-executed many times without being reassembled.</i>
5	<i>Produces output into memory.</i>	<i>Produces output on disc.</i>
6	<i>E.g. Direct-Linking Loader</i>	<i>E.g. MS-DOS Linker</i>



## **LOADERS AND LINKERS**



**MODULE-4**

# **Design of a Direct Linking Loader (DLL)**

---

- The direct linking loader is a re-locatable loader.

# **Design of a Direct Linking Loader (DLL)**

---

- The direct linking loader is a re-locatable loader.
- The loader cannot have the direct access to the source code.

# **Design of a Direct Linking Loader (DLL)**

---

- The direct linking loader is a re-locatable loader.
- The loader cannot have the direct access to the source code.
- To place the object code in the memory there are two situations:
  - The address of the object code could be absolute: Directly place code at specified location
  - The address can be relative.
- If at all the address is relative, then it is the assembler who informs the loader about the relative addresses.

# **Design of a Direct Linking Loader (DLL)**

---

- The direct linking loader is a re-locatable loader.
- The loader cannot have the direct access to the source code.
- To place the object code in the memory there are two situations:
  - The address of the object code could be absolute: Directly place code at specified location
  - The address can be relative.
- If at all the address is relative, then it is the assembler who informs the loader about the relative addresses.

# Specification of problem

---

The assembler should give the following information to the loader

- The length of the object code segment

# Specification of problem

---

The assembler should give the following information to the loader

- The length of the object code segment
- The list of all the symbols, which are not defined in the current segment but can be used in the current segment.

# Specification of problem

---

The assembler should give the following information to the loader

- The length of the object code segment
- The list of all the symbols, which are not defined in the current segment but can be used in the current segment.
- The list of all the symbols, which are defined in the current segment but can be referred by the other segments.

# **Specification of Data Structures**

---

Identify the data bases required by each pass of the loader:

## **Pass 1 data bases:**

1. Input object decks.

# Specification of Data Structures

---

Identify the data bases required by each pass of the loader:

## Pass 1 data bases:

1. Input object decks.
2. A parameter, the **Initial Program Load Address (IPLA)** supplied by the programmer or the operating system, that specifies the address to load the first segment.

# Specification of Data Structures

---

Identify the data bases required by each pass of the loader:

## Pass 1 data bases:

1. Input object decks.
2. A parameter, the **Initial Program Load Address (IPLA)** supplied by the programmer or the operating system, that specifies the address to load the first segment.
3. A **Program Load Address (PLA)** counter, used to keep track of each segment's assigned location

# Specification of Data Structures

---

## Pass 1 data bases:

4. A table, the **Global External Symbol Table (GEST)**, that is used to store each external symbol and its corresponding assigned core address.

# Specification of Data Structures

---

## Pass 1 data bases:

4. A table, the **Global External Symbol Table (GEST)**, that is used to store each external symbol and its corresponding assigned core address.
5. A copy of the input to be used later by pass 2.

This may be stored on an auxiliary storage device, such as magnetic tape, disk, or drum, or the original object decks may be reread by the loader a second time for pass 2.

# Specification of Data Structures

---

## Pass 1 data bases:

4. A table, the **Global External Symbol Table (GEST)**, that is used to store each external symbol and its corresponding assigned core address.
5. A copy of the input to be used later by pass 2.

This may be stored on an auxiliary storage device, such as magnetic tape, disk, or drum, or the original object decks may be reread by the loader a second time for pass 2.

6. A printed listing, **the load map**, that specifies each external symbol and its assigned value.

# Specification of Data Structures

---

## Pass 2 data bases:

1. Copy of object programs inputted to pass I

# Specification of Data Structures

---

## Pass 2 data bases:

1. Copy of object programs inputted to pass I
2. The Initial Program Load Address parameter (IPLA)

# Specification of Data Structures

---

## Pass 2 data bases:

1. Copy of object programs inputted to pass I
2. The Initial Program Load Address parameter (IPLA)
3. The Program Load Address counter (PLA)

# Specification of Data Structures

---

## Pass 2 data bases:

1. Copy of object programs inputted to pass I
2. The Initial Program Load Address parameter (IPLA)
3. The Program Load Address counter (PLA)
4. The Global External Symbol Table (GEST), prepared by pass I, containing each external symbol and its corresponding absolute address value

# Specification of Data Structures

---

## Pass 2 data bases:

1. Copy of object programs inputted to pass I
2. The Initial Program Load Address parameter (IPLA)
3. The Program Load Address counter (PLA)
4. The Global External Symbol Table (GEST), prepared by pass I, containing each external symbol and its corresponding absolute address value
5. An array, 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 symbol's absolute

# **Format of databases**

---

- Object Deck
  1. External Symbol Dictionary cards (ESD)
  2. Instructions and data cards, called "text" of program (TXT)
  3. Relocation and Linkage Directory cards (RLD)
  4. End card (END)
- End of File (EOF)/ Loader Terminate(LDT)
- Global External Symbol Table(GEST)
- Local External Symbol Array (LESA)

# Object deck

---

- Direct Linking Loader uses four types of records in the object decks (files) which contains all information needed for relocation and linking.
- These four sections are:
  1. External Symbol Dictionary cards (ESD)
  2. Instructions and data cards, called "text" of program (TXT)
  3. Relocation and Linkage Directory cards (RLD)
  4. End card (END)

# **External Symbol Dictionary cards (ESD)**

---

- The ESD cards contain the information necessary to build the external symbol dictionary or symbol table.

# **External Symbol Dictionary cards (ESD)**

---

- The ESD cards contain the information necessary to build the external symbol dictionary or symbol table.
- External symbols are symbols that can be referred beyond the subroutine level.

# External Symbol Dictionary cards (ESD)

---

- The ESD cards contain the information necessary to build the external symbol dictionary or symbol table.
- External symbols are symbols that can be referred beyond the subroutine level.
- The normal labels in the source program are used only by the assembler, and information about them is not included in the object deck.

# External Symbol Dictionary cards (ESD)

- Example: Assume program B has a table called NAMES; it can be accessed by program A as follows.

<b>A</b>	START	
	EXTRN	NAMES
	:	
	L	1,ADDRNAME get address of NAME table
	:	
	ADDRNAME	DC
		A(NAMES)
		END
<hr/>		
<b>B</b>	START	
	ENTRY	NAMES
	:	
	NAMES	DC
		----
		END

# External Symbol Dictionary cards (ESD)

---

- There are three types of external symbols:
  - **Segment Definition (SD):** name on START or CSECT card.
  - **Local Definition (LD):** specified on ENTRY card. There must be a label in same program with same name. ·
  - **External Reference (ER):** specified on EXTRN card. There must be a corresponding ENTRY, START, or CSECT card in another program with same name.



# External Symbol Dictionary cards (ESD)

---

- There are three types of external symbols:
  - **Segment Definition (SD):** name on START or CSECT card.
  - **Local Definition (LD):** specified on ENTRY card. There must be a label in same program with same name. ·
  - **External Reference (ER):** specified on EXTRN card. There must be a corresponding ENTRY, START, or CSECT card in another program with same name.
- Each SD and ER symbol is assigned a unique number (e.g., 1,2,3, ... ) by the assembler.

# External Symbol Dictionary cards (ESD)

---

- There are three types of external symbols:
  - **Segment Definition (SD):** name on START or CSECT card.
  - **Local Definition (LD):** specified on ENTRY card. There must be a label in same program with same name. ·
  - **External Reference (ER):** specified on EXTRN card. There must be a corresponding ENTRY, START, or CSECT card in another program with same name.
- Each SD and ER symbol is assigned a unique number (e.g., 1,2,3, ... ) by the assembler.
- This number is called the symbol's identifier, or ID, and is used in conjunction

# External Symbol Dictionary cards (ESD)

## Example:

*ESD cards*

<i>Reference no.</i>	<i>Symbol</i>	<i>Type</i>	<i>Relative location</i>	<i>Length</i>
1	JOHN	SD	0	64
2	RESULT	LD	52	---
3	SUM	ER	---	---

# **TXT Cards: "text" of program**

---

- Instructions and data cards, called "text" of program (TXT)

## **TXT Cards: "text" of program**

---

- Instructions and data cards, called "text" of program (TXT)
- The TXT cards contain blocks of data and the relative address at which the data is to be placed.

## **TXT Cards: "text" of program**

---

- Instructions and data cards, called "text" of program (TXT)
- The TXT cards contain blocks of data and the relative address at which the data is to be placed.
- Once the loader has decided where to load the program, it merely adds the Program Load Address (PLA) to the relative address and moves the data into the resulting location.

## **TXT Cards: "text" of program**

---

- Instructions and data cards, called "text" of program (TXT)
- The TXT cards contain blocks of data and the relative address at which the data is to be placed.
- Once the loader has decided where to load the program, it merely adds the Program Load Address (PLA) to the relative address and moves the data into the resulting location.
- The data on the TXT card may be instructions, non-relocated data, or initial values of address constants

# Instructions and data cards, called "text" of program (TXT):

- The data on the TXT card may be instructions, non-relocated data, or initial values of address constants

<i>Reference no.</i>	<i>Relative location</i>	<i>Object code</i>	
4	0	BALR	12,0
6	2	ST	14,54(0,12)
7	6	L	1,46(0,12)
8	10	L	15,58(0,12)
9	14	BALR	14,15
10	16	ST	1,50(0,12)
11	20	L	14,54(0,12)
12	24	BCR	15,14
13	28	1	

## **Relocation and Linkage Directory cards (RLD):**

---

- The RLD cards contain the following information.
  - The location and length of each address constant that needs to be changed for relocation or linking



## **Relocation and Linkage Directory cards (RLD):**

---

- The RLD cards contain the following information.
  - The location and length of each address constant that needs to be changed for relocation or linking
  - The external symbol by which the address constant should be modified (added or subtracted)

## **Relocation and Linkage Directory cards (RLD):**

---

- The RLD cards contain the following information.
  - The location and length of each address constant that needs to be changed for relocation or linking
  - The external symbol by which the address constant should be modified (added or subtracted)
  - The operation to be performed (add or subtract)

# Relocation and Linkage Directory cards (RLD):

Example:

<i>ID</i>	<i>Flag</i>	<i>Length</i>	<i>Rel. loc.</i>
01	+	4	52
02	+	4	56

- If we assume that A's assigned ID is 01 and NAMES' assigned ID is 02.
- This RLD information tells the loader to add the absolute load address of A to the contents of relative location 52 and then add the absolute load address of NAMES to the contents of relative location 56.

## **End card (END)**

---

- The END card specifies the end of the object deck.

## **End card (END)**

---

- The END card specifies the end of the object deck.
- If the assembler END card has a symbol in the operand field, it specifies a start of execution point for the entire program (all subroutines).

## End card (END)

---

- The END card specifies the end of the object deck.
- If the assembler END card has a symbol in the operand field, it specifies a start of execution point for the entire program (all subroutines).
- This address is recorded on the END card.

# **End of File (EOF) / Loader Terminate (LDT)**

---

- There is a final card required to specify the end of a collection of object decks.
- The loaders usually use either a loader terminate (LDT) or End of File (EOF) card.

# End of File (EOF) / Loader Terminate(LDT)

---

- Example:

<b>Subroutine A</b>	{	ESD TXT RLD END
<b>Subroutine B</b>	{	ESD TXT RLD END
<b>Subroutine C</b>	{	ESD TXT RLD END
		<b>EOF or LDT</b>

# **Global External Symbol Table (GEST)**

---

- The Global External Symbol Table (GEST) is used to store the external symbols.

## **Global External Symbol Table (GEST)**

---

- The Global External Symbol Table (GEST) is used to store the external symbols.
- These symbols are defined by means of a Segment Definition (SD) or Local Definition (LD) entry on an External Symbol Dictionary (ESD) card.

# Global External Symbol Table (GEST)

---

- The Global External Symbol Table (GEST) is used to store the external symbols.
- These symbols are defined by means of a Segment Definition (SD) or Local Definition (LD) entry on an External Symbol Dictionary (ESD) card.
- When these symbols are encountered during pass 1, they are assigned an absolute core address

# Global External Symbol Table (GEST)

---

- The Global External Symbol Table (GEST) is used to store the external symbols.
- These symbols are defined by means of a Segment Definition (SD) or Local Definition (LD) entry on an External Symbol Dictionary (ESD) card.
- When these symbols are encountered during pass 1, they are assigned an absolute core address
- This address is stored, along with the symbol, in the GEST

# Global External Symbol Table (GEST)

- Example:

12 bytes per entry	
External symbol (8-bytes) (characters)	Assigned core address (4-bytes) (decimal)
"PG1bbbbbb"	104
"PG1ENT1b"	124
"PG1ENT2b"	134
"PG2bbbbbb"	168
"PG2ENT1b"	184

Global External Symbol Table (GEST) format

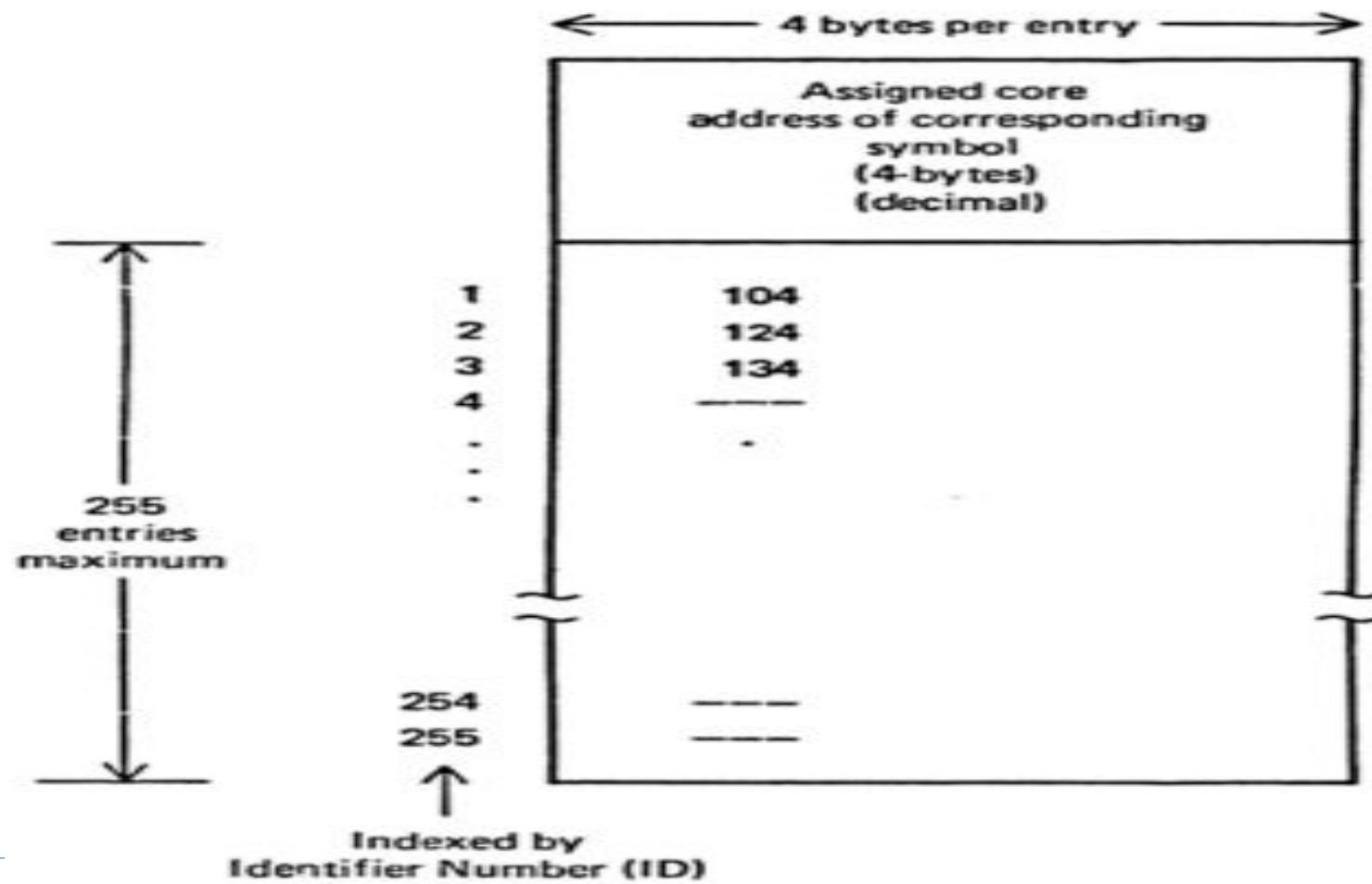
## LOCAL EXTERNAL SYMBOL ARRAY

---

- It is necessary to create a separate LESA for each segment, but since the LESAs are only produced one at a time, the same array can be reused for each segment.
- It is not necessary to search the LESA;
- Given an ID number, the corresponding value is written as LESA(ID) and can be immediately obtained.

# LOCAL EXTERNAL SYMBOL ARRAY

- Example:



# Algorithm

---

Pass 1:

Allocate segments and define symbols

Function:

1. Assign a location to each segment
2. Define the values of all external symbols

# Flowchart for Pass-1

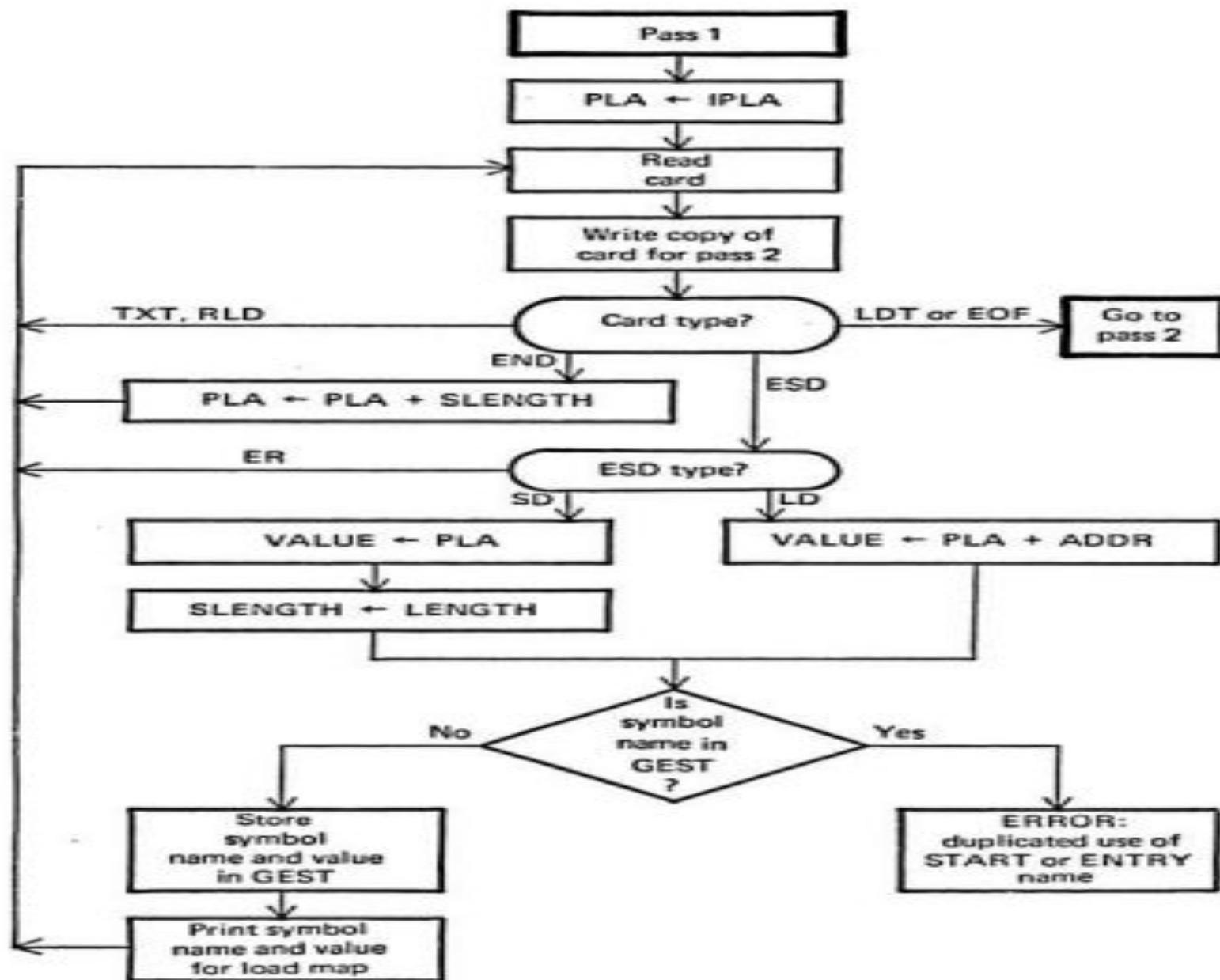


FIGURE 5.24 Detailed pass 1 flowchart

# Algorithm

---

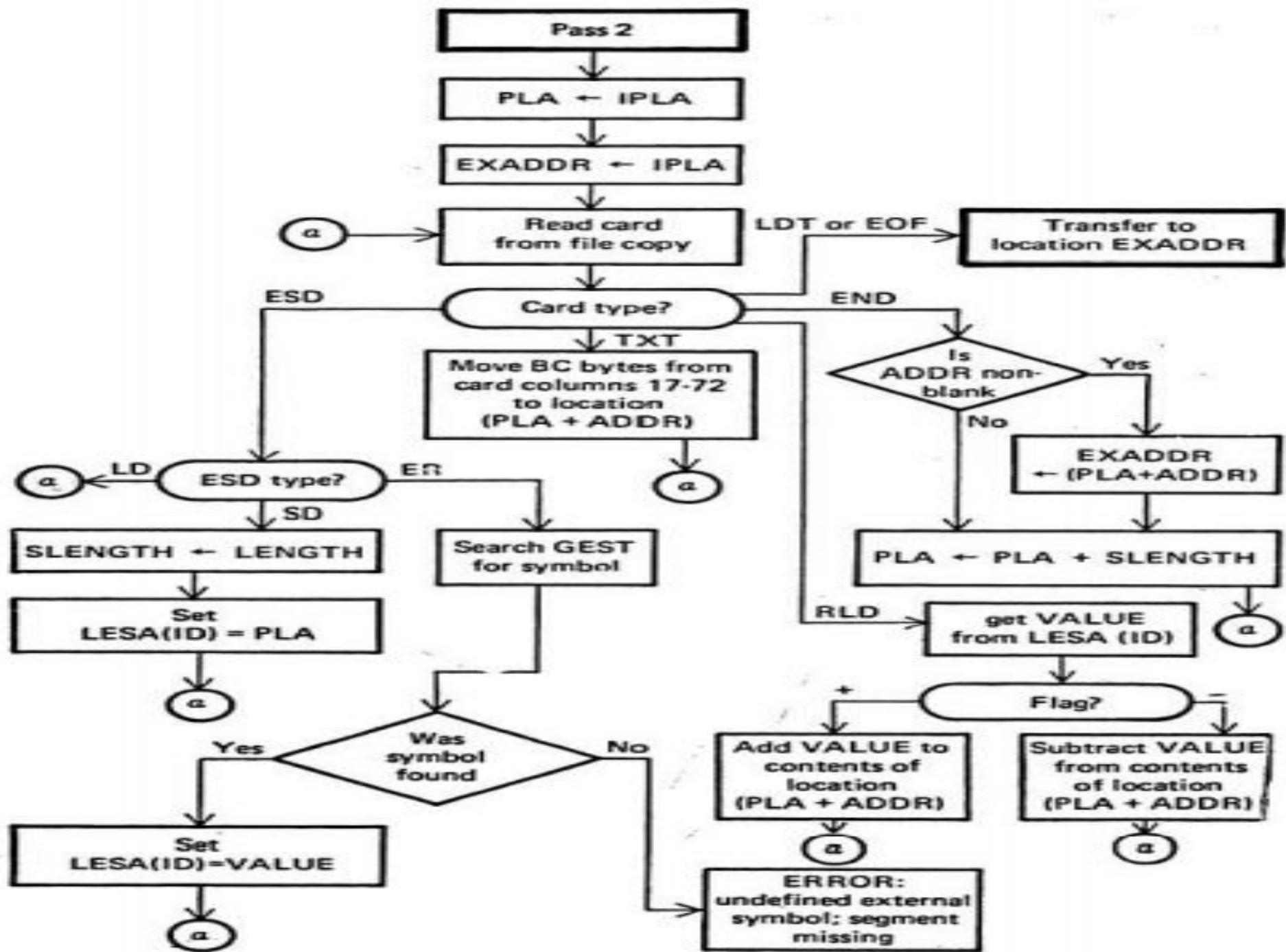
Pass 2:

Load Text and Relocate/Link Address content

Functions:

Complete the loading process by loading the text and adjusting  
(Relocation or Linking) address constant

# FLOWCHART FOR PASS-2



# Example

Source card reference	Relative address	Procedure name	Sample program (source deck)	
1	0	PG1	START	(a) Procedure PG1
2			ENTRY PG1ENT1,PG1ENT2	
3			EXTRN PG2ENT1,PG2	
4	20	PG1ENT1	---	
5	30	PG1ENT2	---	
6	40		DC A(PG1ENT1)	
7	44		DC A(PG1ENT2+15)	
8	48		DC A(PG1ENT2-PG1ENT1-3)	
9	52		DC A(PG2)	
10	56		DC A(PG2ENT1+PG2-PG1ENT1+4)	
11			END	
12	0	PG2	START	(b) Procedure PG2
13			ENTRY PG2ENT1	
14			EXTRN PG1ENT1,PG1ENT2	
15	16	PG2ENT1	---	
16	24		DC A(PG1ENT1)	
17	28		DC A(PG1ENT2+15)	
18	32		DC A(PG1ENT2-PG1ENT1-3)	
19			END	

Sample procedures PG1 and PG2

# Example

## *ESD cards*

<i>Source card reference</i>	<i>Name</i>	<i>Type</i>	<i>ID</i>	<i>Relative address</i>	<i>Length</i>
1	PG1	SD	01	0	60
2	PG1ENT1	LD	—	20	—
2	PG1ENT2	LD	—	30	—
3	PG2	ER	02	—	—
3	PG2ENT1	ER	03	—	—

## *TXT cards*

(only the interesting ones, i.e. those involving address constants)

<i>Source card reference</i>	<i>Relative address</i>	<i>Contents</i>	<i>Comments</i>
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	56-59	-16	= -20 + 4

## *RLD cards*

<i>Source card reference</i>	<i>ESD ID</i>	<i>Length (bytes)</i>	<i>Flag + or -</i>	<i>Relative address</i>
6	01	4	+	40
7	01	4	+	44
9	02	4	+	52
10	03	4	+	56
10	02	4	+	56
10	01	4	-	56

# Example

## *ESD cards*

<i>Source card reference</i>	<i>Name</i>	<i>Type</i>	<i>ID</i>	<i>ADDR</i>	<i>Length</i>
12	PG2	SD	01	0	36
13	PG2ENT1	LD	—	16	—
14	PG1ENT1	ER	02	—	—
14	PG1ENT2	ER	03	—	—

## *TXT cards*

(only the interesting ones)

<i>Source card reference</i>	<i>Relative address</i>	<i>Contents</i>
16	24-27	0
17	28-31	15
18	32-35	-3

## *RLD cards*

<i>Source card reference</i>	<i>ESD ID</i>	<i>Length flag (bytes)</i>	<i>Flag + or -</i>	<i>Relative address</i>
16	02	4	+	24
17	03	4	+	28
18	03	4	+	32
18	02	4	-	32

## Example

← 12 bytes per entry →

External symbol (8-bytes) (characters)	Assigned core address (4-bytes) (decimal)
"PG1bbbbbb"	104
"PG1ENT1b"	124
"PG1ENT2b"	134
"PG2bbbbbb"	168
"PG2ENT1b"	184

Global External Symbol Table (GEST) format

# Example

