

MACHINE INDEPENDENT LOADER FEATURES:

- AUTOMATIC LIBRARY SEARCH
- USE OF LOADER OPTIONS

1. Explain Automatic Library search.

- This feature allows a programmer to use standard subroutines without explicitly including them in the program to be loaded.
- The routines are automatically retrieved from a library as they are needed during linking.
- This allows programmer to use subroutines from one or more libraries. The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program and loaded.
- The loader searches the library or libraries specified for routines that contain the definitions of these symbols in the main program.
- Automatic library call -The programmer does not need to take any action beyond mentioning the subroutine names as external references
- Linking loaders that support automatic library search must keep track of external symbols that are referred to, but not defined, in the primary input to loader.
 - ✓ 1 Enter the symbols from each Refer record into ESTAB
 - ✓ 2 When the definition is encountered (Define record), the address is assigned
 - ✓ 3 At the end of Pass 1, the symbols in ESTAB that remain undefined represent unresolved external references .
 - ✓ 4 The loader searches the specified (or standard) libraries for undefined symbols or subroutines
- The library search process may be repeated
 - ✓ Since the subroutines fetched from a library may themselves contain external references
 - ✓ Programmer defined subroutines have higher priority
 - ✓ The programmer can override the standard subroutines in the library by supplying their own routines
- Library structures
 - ✓ Assembled or compiled versions of the subroutines in a library can be structured using a **directory** that gives the name of each routine and a pointer to its address within the library
- The linker searches the subroutine directory, finds the address of desired library routine. Then linker prepares a **load module** appending the user program and necessary library routines by doing the necessary relocation.

2. Explain different loader option commands.

Loader options allow the user to specify options that modify the standard processing. The options may be specified in three different ways. They are, specified using a command language, specified as a part of job control language that is processed by the operating system, and can be specified using loader control statements in the source program.

Here are the some examples of how option can be specified.

- **INCLUDE** program-name (library-name) - read the designated object program from a library
- **DELETE** csect-name – delete the named control section from the set pf programs being loaded
- **CHANGE** name1, name2 - external symbol name1 to be changed to name2 wherever it appears in the object programs
- **LIBRARY MYLIB** – search MYLIB library before standard libraries
- **NOCALL STDDEV, PLOT, CORREL** – no loading and linking of unneeded routines

Here is one more example giving, how commands can be specified as a part of object file, and the respective changes are carried out by the loader.

```
1.LIBRARY UTLIB
2.INCLUDE READ (UTLIB)
3. INCLUDE WRITE (UTLIB)
4.DELETE RDREC, WRREC
5.CHANGE RDREC, READ
6.CHANGE WRREC, WRITE
7.NOCALL SQRT, PLOT
```

The commands are, use UTLIB (say utility library),

INCLUDE READ and WRITE control sections from the library,

DELETE the control sections RDREC and WRREC from the load,

The CHANGE command causes all external references to the symbol RDREC to be changed to The symbol READ, similarly references to WRREC is changed to WRITE,

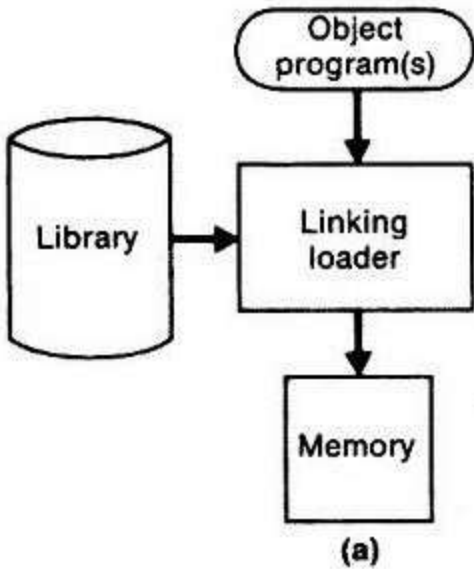
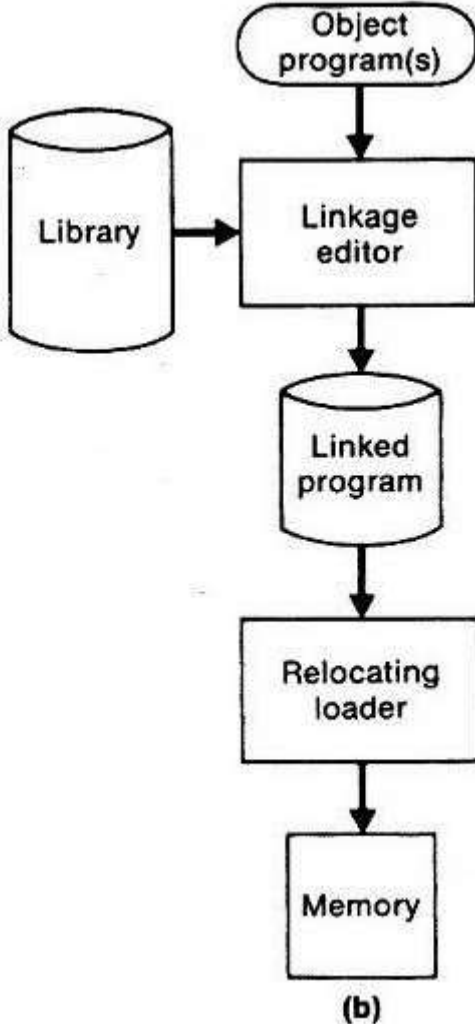
Finally, NO CALL to the functions SQRT, PLOT,(reference symbols are not resolved) if they are used in the program.

LOADER DESIGN OPTIONS:

1. LINKAGE EDITOR
2. DYNAMIC LINKAGE.

3. Differentiate between linking loader and linkage editor:

Linking Loader	Linkage Editor
A linking loader performs all linking and relocation operations, including automatic library search if specified and loads the linked program directly into memory for execution.	The linkage editor produces a linked version of the program. Such a linked version is also called as load module. This load module is generally written in a file or library for later execution.
There is no need of relocating loader	The relocating loader loads the load module into memory
The linking loader searches the librabries and resolves the external refernces every time the the program is executed.	If the program is executed many times without being reassembled then linkage editor is the best choice.
When program is in development stage then at that time the linking loader can be used	When program development is finished or when the library is built, then linkage editor is

	the best choice.
The loading requires two passes	The loading can be done in one pass
The role of linking loader as shown below	The role of linkage editor as shown below.
 <pre> graph TD OP1([Object program(s)]) --> LL1[Linking loader] L1[(Library)] --> LL1 LL1 --> M1[Memory] </pre> <p>(a)</p>	 <pre> graph TD OP2([Object program(s)]) --> LE[Linkage editor] L2[(Library)] --> LE LE --> LP[(Linked program)] LP --> RL[Relocating loader] RL --> M2[Memory] </pre> <p>(b)</p>

Linkage Editors

The figure (b) above shows the processing of an object program using Linkage editor.

A linkage editor produces a linked version of the program – often called **a load module or an executable image** – which is written to a file or library for later execution.

The linked program produced is generally in a form that is suitable for processing by a **relocating loader**.

Some useful functions of Linkage editor are, an absolute object program can be created, if starting address is already known. New versions of the library can be included without changing the source program. Linkage editors can also be used to build packages of subroutines or other control sections that are generally used together. Linkage editors often allow the user to specify that external references are not to be resolved by automatic library search – linking will be done later by linking loader – linkage editor + linking loader – savings in space.

4.Explain Dynamic linkage with suitable diagrams.

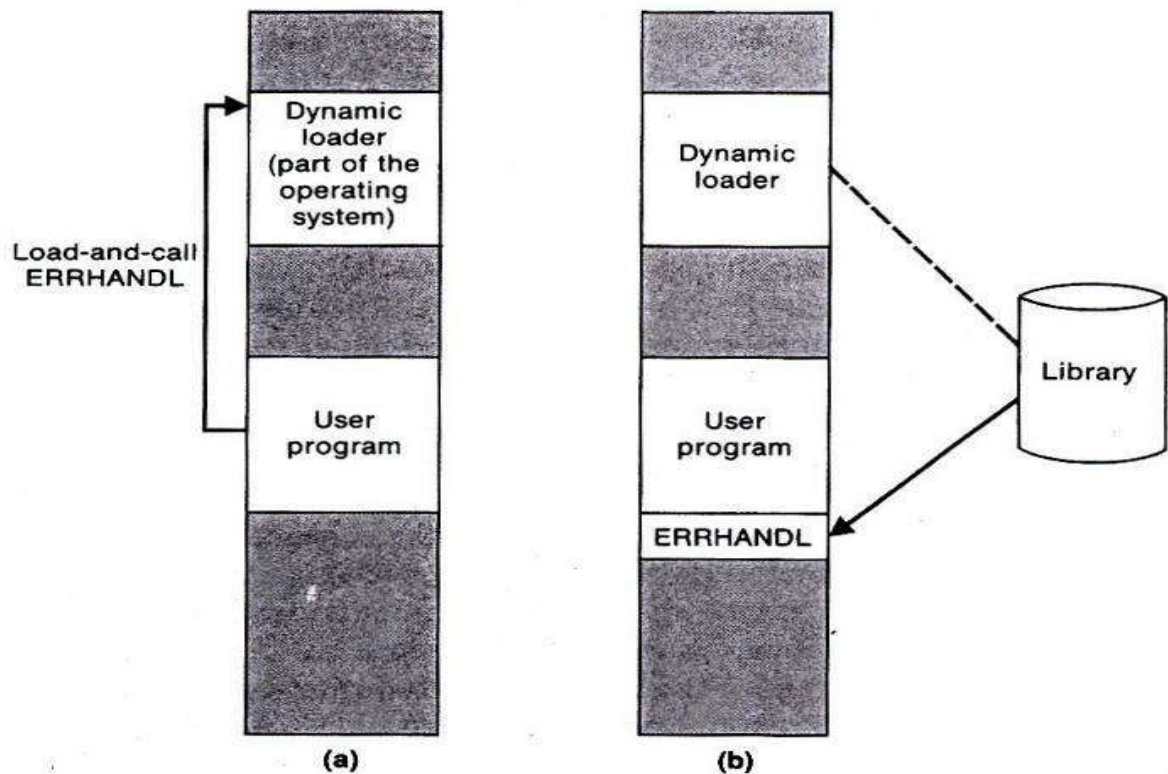
Or

Explain the process of loading and calling of subroutine using dynamic binding.

In overlay structure certain selective subroutines can be resident in the memory. That means it is not necessary to resident all the subroutines in the memory for all time. only necessary routines can be present in the main memory and during execution the required subroutines can be loaded in the memory.

The process of postponing linking and loading of external references until execution is called dynamic linking dynamic loading or load on call.

- Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library. for ex run time support routines for a high level language like C could be stored in a dynamic link library. A single copy of the routines in this library could be loaded into the memory of the computer. All c programs currently in execution could be linked to this one copy , instead of linking a separate copy into each object program.
- Suppose that in any one execution a program uses only a few out of large number of possible routines, but the exact routine needed cannot be predicted until program receives its input. dynamic linking avoids the necessity of loading the entire library for each execution.



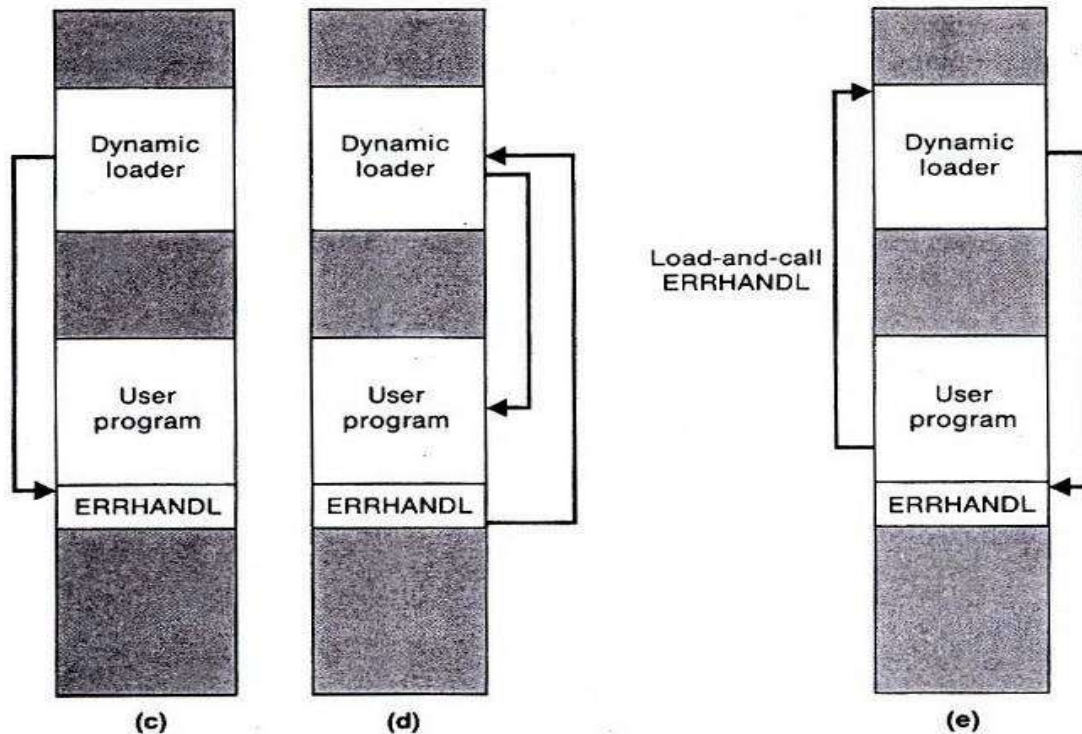


Figure 3.14 Loading and calling of a subroutine using dynamic linking.

Pass of control

1. **User program → OS**
2. **OS → load the subroutine**
3. **OS → Subroutine**
4. **Subroutine → OS**
5. **OS → User program**

1. The program makes a load and call service request to the operating system. The parameter of this request is the symbolic name of the routine to be called. Ex here is ERRHANDL(symbolic name of routine).
 2. The OS examines its internal tables to determine whether or not the routine is already loaded. If necessary the routine is loaded from the specified user or system libraries.
 3. Control is then passed from the OS to the routine being called.[ERRHANDL]
 4. When the called subroutine completes its processing , it returns to its caller(OS).
 5. The OS then returns control to the program that issued the request(User Program).
- After the subroutine is completed the memory that was allocated to load it, may be released and used for other purposes..

If a subroutine is still in memory a second call to it may not require another load operation. Control may simple be passed from the dynamic loader to the called routine.

Advantages:

1. The overhead on the loader is reduced. The required subroutine will be loaded in the main memory only at the time of execution.
2. The system can be dynamically configured.

Disadvantages

The linking and loading has to be suspended until execution. During the execution if at all any subroutine is needed then the process of execution needs to be suspended until the require subroutine gets loaded in the main memory.

5. Explain MS DOS Linker.

Or

Explain the object module of MS-DOS Linker.

Microsoft MS-DOS Linker.

Most MS-DOS compilers and assemblers (MASM) produce object modules, and they are stored in .OBJ files. MS-DOS LINK is a linkage editor that combines one or more object modules to produce a complete executable program - .EXE file; this file is later executed for results.

The following table illustrates the typical MS-DOS object module

Record Types	Description
THEADR	Translator Header
TYPDEF,PUBDEF, EXTDEF	External symbols and references
LNAMES, SEGDEF, GRPDEF	Segment definition and grouping
LEDATA, LIDATA	Translated instructions and data
FIXUPP	Relocation and linking information
MODEND	End of object module

THEADR (Translator Header) specifies the name of the object module.

Generally correspond HEADER record used in SIC/XE.

MODEND(Module End) specifies the end of the module.

Generally correspond END record used in SIC/XE

PUBDEF (Public Name Definition record) contains list of the external symbols that are defined in this object module.(called public names).

Correspond to Define record used in SIC/XE.

EXTDEF(External Name Definition record) contains list of external symbols referred in this module, but defined elsewhere.

Correspond to Refer record used in SIC/XE.

TYPDEF(Type Definition Record) the data types are defined here.

SEGDEF(Segment Definition Record) describes segments in the object module (includes name, length, and alignment).

GRPDEF(Group definition Record) includes how segments are combined into groups.

LNAMES(list of names record) contains all segment and class names.

LEDATA (logical enumerated data)contains translated instructions and data.

LIDATA(logical iterated data) has above in repeating pattern.

FIXUPP is used to resolve external references.

LINK performs its processing in two forms.

PASS 1 computes a starting address for each segment in the program.

It constructs a symbol table that associates an address with each segment (using LNames,SEGDEF,GRPDEF records) and each external symbol (using the EXTDEF and PUBDEF records).

If unresolved external symbols remain after all the object modules have been processed, LINK searches the specified libraries.

PASS2:

LINK extracts the translated data and instructions from the object modules and builds an image of executable program in memory.

It processes each LEDATA and LIDATA record along with the FIXUPP record.

It places the binary data from LEDATA and LIDATA records into memory image at locations that reflect the segment addresses computed during Pass 1.

Relocation within a segment is performed and external symbol references are resolved.

After the memory image is complete LINK writes it to the executable (.EXE) file.

This file includes a header that contains the table of segment fixups, information about memory requirements and entry points and initial contents of register CS and SP.