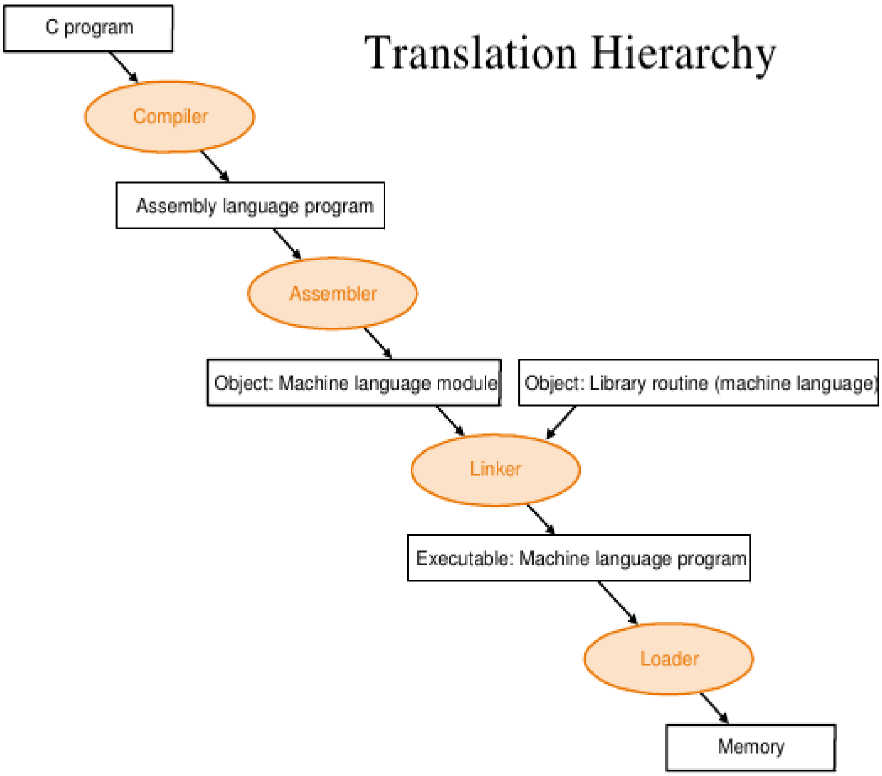
MODULE 4

LINKER AND LOADER

SYLLABUS

Basic Loader functions - Design of absolute loader, Simple bootstrap Loader, Machine dependent loader features- Relocation, Program Linking, Algorithm and data structures of two pass Linking Loader, Machine independent loader features , Loader Design Options

4.1 NEED FOR LINKING AND LOADING


Translation Hierarchy
C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

- To execute an object program, we need:
  - **Relocation -** which modifies the object program so that it can be loaded at an address different from the location originally specified
  - **Linking -** which combines two or more separate object programs and supplies the information needed to allow references between them
  - **Loading and Allocation -** which allocates memory location and brings the object program into memory for execution

- The system software which performs linking operation is called **linker.** The system software which loads the object program into memory and starts its execution is called **loader**. Linkers and loaders perform several related but conceptually separate actions.

4.2 BASIC LOADER FUNCTIONS

- Fundamental functions of a loader are **Bringing an object program into memory and starting its execution.**
- In this section, two basic loader designs are discussed
    1. **Absolute Loader**
    2. **Bootstrap Loader**

4.2.1 Design of an Absolute Loader

An absolute loader is a loader that places absolute code into main memory beginning with the initial address(absolute address) assigned by the assembler. No address manipulation is performed. That is there is no need for relocation and linking because the program will be loaded into the location specified in the program.

For a simple absolute loader, all functions are accomplished in a single pass as follows:

1) The **Header record** of object programs is checked to verify that the correct program has been presented for loading.

2) As each **Text record** is read, the object code it contains is moved to the indicated address in memory.

3) When the **End record** is encountered, the loader jumps to the specified address to begin execution of the loaded program.

Algorithm for absolute loader

```
begin
    read Header  record
    verify program name and length
    read first Text record
    while record type ≠ E
    begin
        //if object code is in character form, convert it into internal representation
        move object code to specified location in memory
        read next object program record
    end
    jump to address specified in End record
end
```

Advantages and disadvantages of absolute loader

The advantage of absolute loader is that it is simple and efficient, but the need for programmer to specify the actual address restricts the flexibility. As a result we cannot run several independent programs together, sharing memory between them. Another disadvantage is that it is difficult to use subroutine libraries while using an absolute loader.

4.2.2 A Simple Bootstrap Loader

- Given an idle computer with no program in memory, how do we get things started? Two solutions are there.
    1. On some computers, an absolute loader program is permanently resident in a read-only memory (ROM). When some hardware signal occurs, the machine begins to execute this ROM program. This is referred to as a bootstrap loader.
    2. On some computers, there's a built-in hardware which read a fixed-length record from some device into memory at a fixed location. After the read operation, control is automatically transferred to the address in memory.

When a computer is first turned on or restarted, a special type of absolute loader, called a **bootstrap loader**, is executed. This bootstrap loader loads the first program to be run by the computer – usually an operating system.

Working of a SIC Bootstrap loader

- SIC uses the above mentioned second method.
- The bootstrap begins at address 0 in the memory of the machine.
- It loads the operating system at address 80.
- Each byte of object code to be loaded is represented on device F1 as *two hexadecimal digits* just as it is in a Text record of a SIC object program.
- The object code from device F1 is always loaded into consecutive bytes of memory, starting at address 80.

- The main loop of the bootstrap keeps the address of the next memory location to be loaded in register X.
- After all of the object code from device F1 has been loaded, the bootstrap jumps to address 80, which begins the execution of the program that was loaded.

- Much of the work of the bootstrap loader is performed by the subroutine GETC.
- GETC is used to read and convert a pair of characters from device F1 representing 1 byte of object code to be loaded. For example, two bytes = C “D8” → ‘4438’H converting to one byte ‘D8’H.
- The resulting byte is stored at the address currently in register X, using STCH instruction that refers to location 0 using indexed addressing.
- The TIXR instruction is then used to add 1 to the value in X.

Bootstrap Loader for SIC/XE

This bootstrap main function reads object code from device F1 and enters it into memory starting at address 80 (hexadecimal) . After all of the code from dev F1 has been seen entered into memory, the bootstrap executes a jump to address 80 to begin execution of the program just loaded. Register X contains the next address to be loaded.

```
BOOT   START    0
       CLEAR    A          CLEAR REGISTER A TO ZERO
        LDX     #128       INITIALIZE REGISTER X TO HEX 80
LOOP   JSUB     GETC       READ HEX DIGIT FROM PROGRAM BEING LOADED
        RMO     A, S       SAVE IN REGISTER S
        SHIFTL  S , 4      MOVE TO HIGH□ORDER 4 BITS OF BYTE
        JSUB    GETC       GET NEXT HEX DIGIT
        ADDR    S ,A       COMBINE DIGITS TO FORM ONE BYTE
        STCH    0 ,X       STORE AT ADDRESS IN REGISTER X
        TIXR    X          ADD 1 TO MEMORY ADDRESS BEING LOADED
        JUMP    LOOP       LOOP UNTIL END OF INPUT IS REACHED
```

GETC subroutine read one character from input device and convert it from ASCII code to hexadecimal digit value. The converted digit value is returned in register A. When an end of file is read, control is transferred to the starting address (hex 80)

```
GETC       TD      INPUT       TEST INPUT DEVICE
           JEQ     GETC         LOOP UNTIL READY
           RD      INPUT        READ CHARACTER
           COMP  #4             IF CHARACTER IS HEX 04 (END OF FILE) ,
           JEQ     80           JUMP TO START OF PROGRAM JUST LOADED
           COMP  #48            COMPARE TO HEX 30 (CHARACTER ' 0 ' )
           JLT     GETC         SKIP CHARACTERS LESS THAN ' 0 '
           SUB     #48          SUBTRACT HEX 30 FROM ASCII CODE
           COMP  10             IF RESULT IS LESS THAN 10 , CONVERSION IS
           JLT     RETURN     COMPLETE. OTHERWISE, SUBTRACT 7 MORE
           SUB     #7         (FOR HEX DIGITS 'A' THROUGH 'F ' )
RETURN    RSUB               RETURN TO CALLER
INPUT      BYTE     X'F1 '       CODE FOR INPUT DEVICE
           END LOOP
```

4.3 MACHINE DEPENDENT LOADER FEATURES

The features of loader that depends on machine architecture are called machine dependent loader features. It includes:

1. Program Relocation
2. Program Linking

4.3.1 Program Relocation (Relocating Loader)

- The absolute loader has several disadvantages. One of the most obvious is the need for the programmer to specify the actual address at which it will be loaded into memory.
- On a simple computer with a small memory the actual address at which the program will be loaded can be specified easily.
- On a larger and more advanced machine, we often like to run several independent programs together, sharing memory between them. We do not know in advance where a program will be loaded. Hence we write relocatable programs instead of absolute ones.

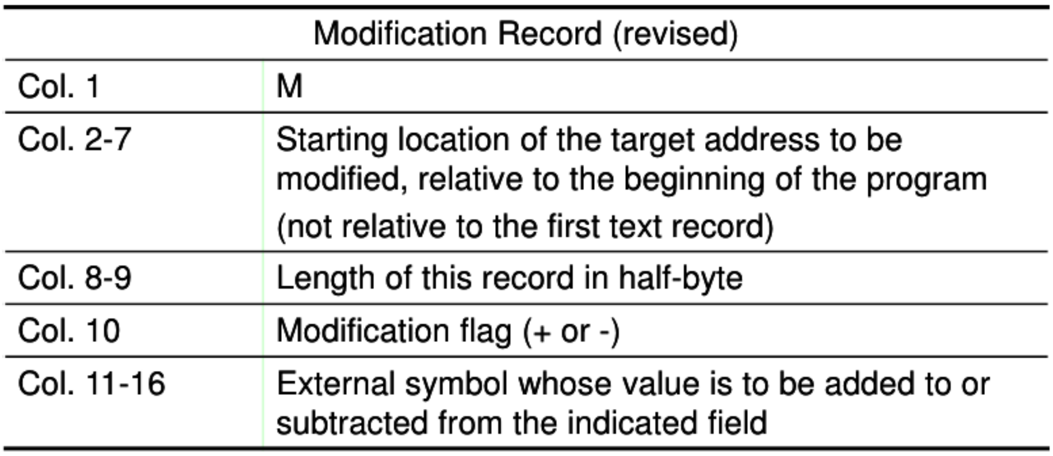
- Writing absolute programs also makes it difficult to use subroutine libraries efficiently. This could not be done effectively if all of the subroutines had preassigned absolute addresses.
- The need for *program relocation* is an indirect consequence of the change to larger and more powerful computers. The way relocation is implemented in a loader is also dependent upon machine characteristics.
- Program relocation is explained in Module 2

- **Loaders that has the capability to perform relocation are called relocating loaders or relative loaders.**

- There are two methods for specifying relocation in object program
  1. Modification Record
  2. Relocation Bit

Modification Record

- A Modification record is used to describe each part of the object code that must be changed when the program is relocated.
- The Modification has the following format:(Its explained in detail in module 2)

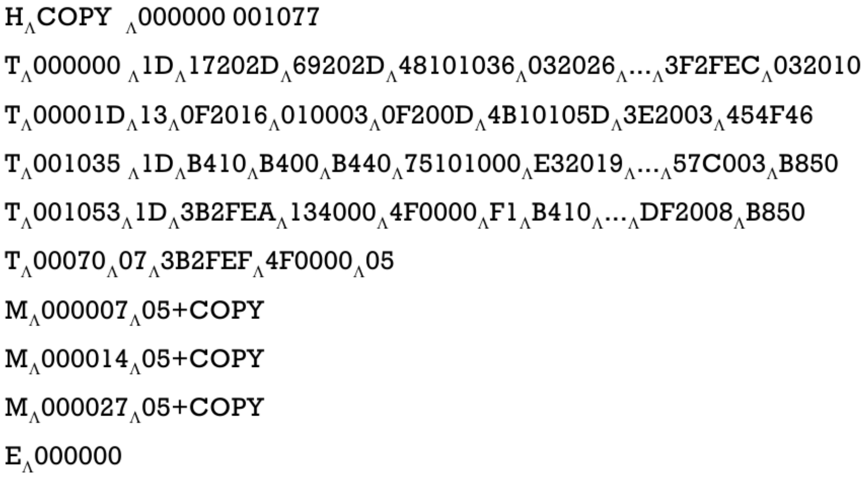


Modification Record (revised)

Col. 1	M
Col. 2-7	Starting location of the target address to be modified, relative to the beginning of the program (not relative to the first text record)
Col. 8-9	Length of this record in half-byte
Col. 10	Modification flag (+ or -)
Col. 11-16	External symbol whose value is to be added to or subtracted from the indicated field

- Each Modification record specifies the starting address and length of the field whose value is to be altered. It then describes the modification to be performed.

- Consider the following object program, here the records starting with M represents the modification record. In this example, the record M 000007 05 + COPY is the modification suggested for the statement at location 000007 and requires modification of 5-half bytes and the modification to be performed is add the value of the symbol COPY, which represents the starting address of the program.(means add the starting address of program to the statement at 000007). Similarly for other records.

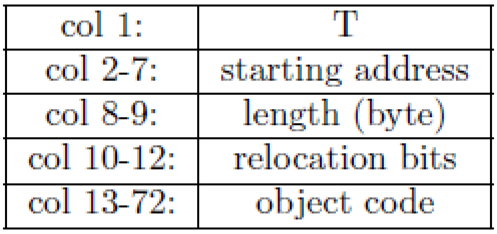


```
H∧COPY  ∧000000 001077
T∧000000 ∧1D∧17202D∧69202D∧48101036∧032026∧...∧3F2FEC∧032010
T∧00001D∧13∧0F2016∧010003∧0F200D∧4B10105D∧3E2003∧454F46
T∧001035 ∧1D∧B410∧B400∧B440∧75101000∧E32019∧...∧57C003∧B850
T∧001053∧1D∧3B2FEA∧134000∧4F0000∧F1∧B410∧...∧DF2008∧B850
T∧00070∧07∧3B2FEF∧4F0000∧05
M∧000007∧05+COPY
M∧000014∧05+COPY
M∧000027∧05+COPY
E∧000000
```

The Modification record is not well suited for certain cases. In some programs the addresses in majority of instructions need to be modified when the program is relocated. This would require large number of Modification records, which results in an object program more than twice as large as the normal. In such cases, the second method called relocation bit is used.

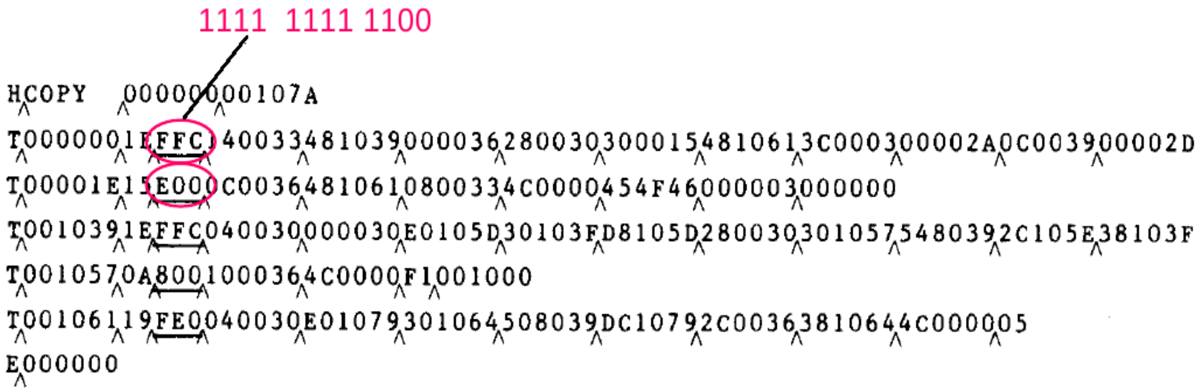
Relocation Bit

- To overcome the disadvantage of modification record, relocation bit is used.
- The Text records are the same as before except that there is a relocation bit associated with each word of object code.
- Since all SIC instructions occupy one word, this means that there is one relocation bit for each possible instruction.
- The relocation bits are gathered together into a **bit mask** following the length indicator in each Text record.
- Text record format



col 1:	T
col 2-7:	starting address
col 8-9:	length (byte)
col 10-12:	relocation bits
col 13-72:	object code

- If the relocation bit corresponding to a word of object code is set to 1, the programs starting address is to be added to this word when the program is relocated.
- A bit value of 0 indicates that no modification is necessary.
- If a Text record contains fewer than 12 words of object code, the bits corresponding to unused words are set to 0.
- In the following object code, the bit mask FFC (representing the bit string 111111111100) in the first Text record specifies that all 10 words of object code are to be modified during relocation.



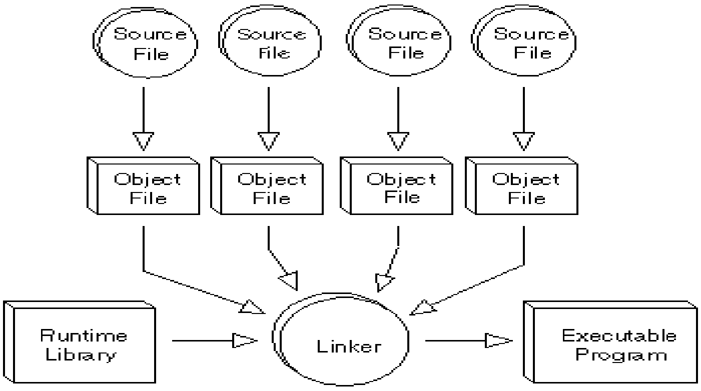
1111 1111 1100

```
HCOPY  00000000107A
T0000001EFFC1400334810390000362800303000154810613C000300002A0C003900002D
T00001E15E000C00364810610800334C0000454F460000003000000
T0010391EFFC040030000030E0105D30103FD8105D2800303010575480392C105E38103F
T0010570A8001000364C0000F1001000
T00106119FE0040030E01079301064508039DC10792C00363810644C000005
E000000
```

Object program with relocation by bit mask.

4.3.2 Program Linking (Linking Loader)

- Many programming languages allow us to write different pieces of code called *modules*, separately. This simplifies the programming task because we can break a large program into small, more manageable pieces. Eventually, though, we need to put all the modules together. Apart from this, a user code often makes references to code and data defined in some "libraries".
- Linking is the process in which references to "externally" defined symbols are processed so as to make them operational.
- A linker or link editor is a program that combines object modules to form an executable program.
- A Linking Loader is a program that has the capability to perform relocation, linking and loading. Linking and relocation is performed at load time.



Algorithm and Data Structures for a Linking Loader

- The algorithm for a *linking loader* is considerably more complicated than the *absolute loader* algorithm.
- A linking loader usually makes *two passes* over its input, just as an assembler does.
- In terms of general function, the two passes of a linking loader are quite similar to the two passes of an assembler:
    - Pass 1 assigns addresses to all external symbols.
    - Pass 2 performs the actual loading, relocation, and linking.
- The main data structure needed for our linking loader is an *external symbol table* **ESTAB.** This table, which is analogous to SYMTAB in our assembler algorithm, is used to store the *name* and *address* of each external symbol in the set of control sections being loaded.
- Two other important variables are **PROGADDR (program load address)** and **CSADDR (control section address).**

  (1) PROGADDR is *the beginning address in memory* where the linked program is to be loaded. Its value is supplied to the loader by the OS.

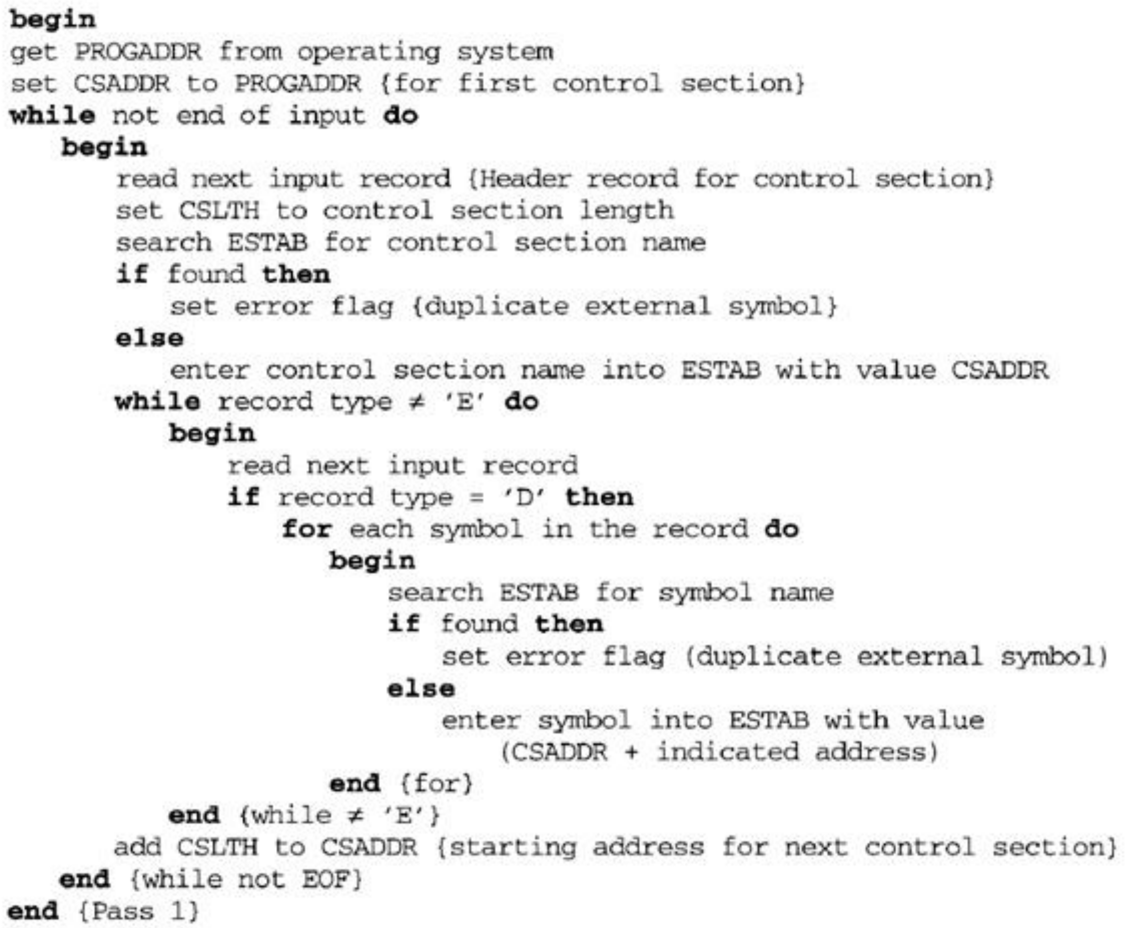
  (2) CSADDR contains *the starting address* assigned to the control section currently being scanned by the loader. This value is added to all relative addresses within the control section to convert them to actual addresses.

Linking loader PASS 1

- During Pass 1, the loader is concerned only with Header and Define records.
- **Variables and Data structures used in PASS1**
    - PROGADDR (Program Load Address) from OS

- CSADDR (Control Section Address)
- CSLTH (Control Section Length)
- ESTAB (External Symbol Table)

Algorithm for Pass 1



```
begin
get PROGADDR from operating system
set CSADDR to PROGADDR {for first control section}
while not end of input do
   begin
      read next input record {Header record for control section}
      set CSLTH to control section length
      search ESTAB for control section name
      if found then
         set error flag {duplicate external symbol}
      else
         enter control section name into ESTAB with value CSADDR
      while record type ≠ 'E' do
         begin
            read next input record
            if record type = 'D' then
               for each symbol in the record do
                  begin
                     search ESTAB for symbol name
                     if found then
                        set error flag (duplicate external symbol)
                     else
                        enter symbol into ESTAB with value
                           (CSADDR + indicated address)
                  end {for}
         end {while ≠ 'E'}
      add CSLTH to CSADDR {starting address for next control section}
   end {while not EOF}
end {Pass 1}
```

Explanation of Pass 1 algorithm

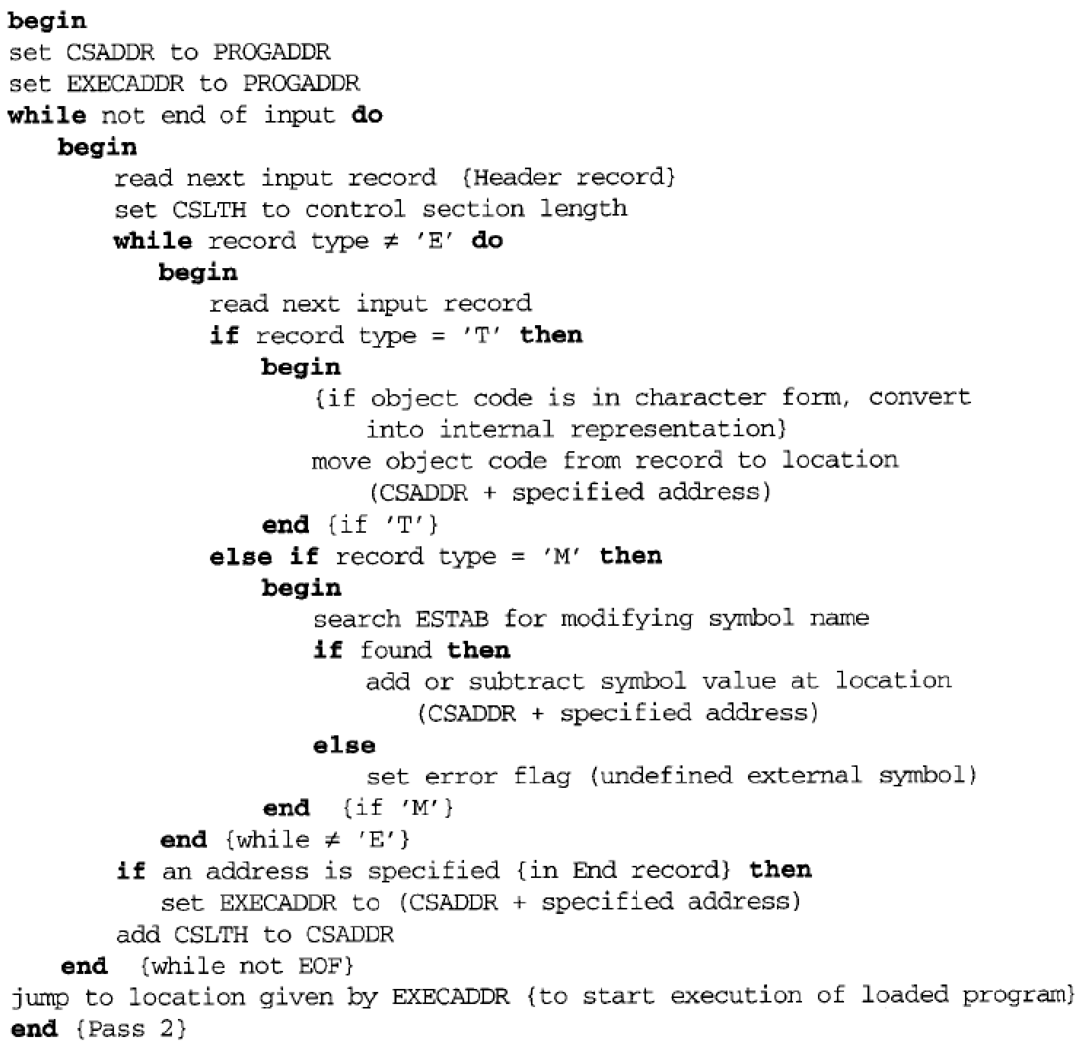
- The beginning load address for the linked program (PROGADDR) is obtained from the OS. This becomes the starting address (CSADDR) for the first control section in the input sequence.
- The control section name from Header record is entered into ESTAB, with value given by CSADDR.
- All external symbols appearing in the Define record for the control section are also entered into ESTAB. Their addresses are obtained by adding the value specified in the Define record to CSADDR.

- When the End record is read, the control section length CSLTH (which was saved from the End record) is added to CSADDR. This calculation gives the starting address for the next control section in sequence.
- At the end of Pass 1, ESTAB contains all external symbols defined in the set of control sections together with the address assigned to each.

Linking loader PASS 2

- Pass 2 of linking loader performs the actual loading, relocation, and linking of the program.

Algorithm for Pass 2



```
begin
set CSADDR to PROGADDR
set EXECADDR to PROGADDR
while not end of input do
   begin
      read next input record  {Header record}
      set CSLTH to control section length
      while record type ≠ 'E' do
         begin
            read next input record
            if record type = 'T' then
               begin
                  {if object code is in character form, convert
                     into internal representation}
                  move object code from record to location
                     (CSADDR + specified address)
               end {if 'T'}
            else if record type = 'M' then
               begin
                  search ESTAB for modifying symbol name
                  if found then
                     add or subtract symbol value at location
                        (CSADDR + specified address)
                  else
                     set error flag (undefined external symbol)
               end  {if 'M'}
         end {while ≠ 'E'}
      if an address is specified {in End record} then
         set EXECADDR to (CSADDR + specified address)
      add CSLTH to CSADDR
   end  {while not EOF}
jump to location given by EXECADDR {to start execution of loaded program}
end {Pass 2}
```

Explanation of Pass 2 Algorithm

- As each Text record is read, the object code is moved to the specified address (plus the current value of CSADDR).

- When a Modification record is encountered, the symbol whose value is to be used for modification is looked up in ESTAB.
- This value is then added to or subtracted from the indicated location in memory.
- The last step performed by the loader is usually the transferring of control to the loaded program to begin execution. The End record for each control section may contain the address of the first instruction in that control section to be executed. Loader takes this as the transfer point to begin execution.

4.3 MACHINE INDEPENDENT LOADER FEATURES

The features of loader that doesn't depends the architecture of machine are called machine independent loader features. It includes:

- Automatic Library search
- Loader Options that can be selected at the time of loading and linking

4.3.1 Automatic Library Search

One of the important machine independent feature of loader is **to use an automatic library search process for handling external reference.**Many linking loaders can automatically incorporate routines from a subprogram library into the program being loaded. The subroutines called by the program being loaded are automatically fetched from the library, linked with the main program, and loaded. Linking loaders that supp ort automatic library search must keep track of external symbols that are referred to, but not defined, in the primary input to the loader.

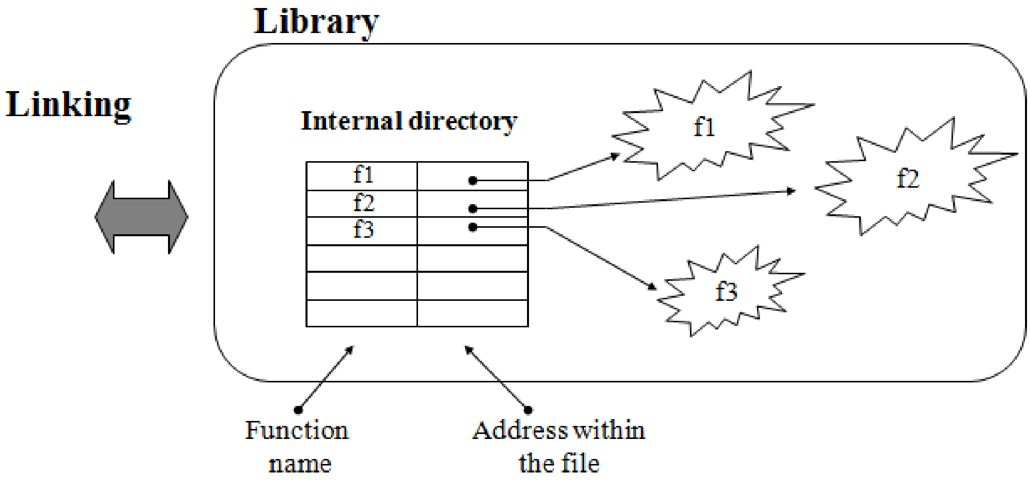
At the end of Pass 1, the symbols in ESTAB that remain undefined represent unresolved external references. The loader searches the library or libraries specified for routines that contain the definitions of these symbols, and processes the subroutines found by this search exactly as if they had been part of the primary input stream. Note that the subroutines fetched from a library in this way may themselves contain external references. It is therefore necessary to repeat the library search process until all references are resolved. If unresolved external references remain after the library search is completed, these must be treated as errors.

Automatic Library search process is described below:

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 libraries specified (or standard) 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. So the programmer can override the standard subroutines in the library by supplying their own routines. Searching on the libraries is done by scanning through the define records of all the object programs in the library. This method is quiet inefficient. So we go for a directory structure. Assembled or compiled versions of the subroutines in a library is structured using a directory that gives the name of each routine and a pointer to its address within the library. Thus the library search involves only a search on the directory, followed by reading the object programs indicated by this search.



The library contains an internal directory where each files along with their address are stored. This facilitates the linking of library functions more easy, because whenever a library function is needed its address can be directly obtained from internal directory.

4.3.2 Loader Options

Many loaders allow the user to specify options that modify the standard processing.

Option 1:

- allows the selection of alternative sources of input.
- Ex. INCLUDE program-name (library-name)

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

Option 2:

- allows the user to delete external symbols or entire control sections.
- Ex. DE LETE csect-name

  This instruct the loader to delete the named control section(s) from the set of programs being loaded.

Option 3:

- allows the user to change the name of external symbol
- Ex: CHANGE name1, name2

  this cause the external symbol name1 to be changed to name2 wherever it appears in the object programs.

Option 4:

- This involves the automatic inclusion of library routines to satisfy external references.
- Ex: LIBRARY MYLIB

  Such user-specified libraries are normally searched before the standard system libraries. This allows the user to use special versions of the standard routines.

Option 5:

- NOCALL STDDEV, PLOT, CORREL
- To instruct the loader that these external references are to remain unresolved. This avoids the overhead of loading and linking the unneeded routines, and saves the memory space that would otherwise be required.

Example:

If we would like to use the utility routines READ and WRITE instead of RDREC and WRREC in our programs, for a temporary measure, we use the following loader commands

```
INCLUDE READ(UTLIB)
INCLUDE WRITE(UTILB)
DELETE RDREC, WRREC
CHANGE RDREC, READ
CHANGE WRREC, WRITE
```

These commands would ask the loader to include control sections READ and WRITE from the library UTLIB and to delete the control sections WRREC and RDREC. The first CHANGE command would change all the external references to the symbol RDREC to be changed to refer to READ and second CHANGE will cause references to WRREC to be changed to WRITE.

4.4 LOADER DESIGN OPTIONS (or Variants of basic loader model)

In this section some alternatives for basic loader models are discussed.

1. **Linkage Editors** – which perform linking prior to load time
2. **Dynamic Linking** – which perform the linking function at execution time.
3. **Bootstrap Loaders** – used to load operating system or the loader into the memory.

4.4.1 Linkage Editors

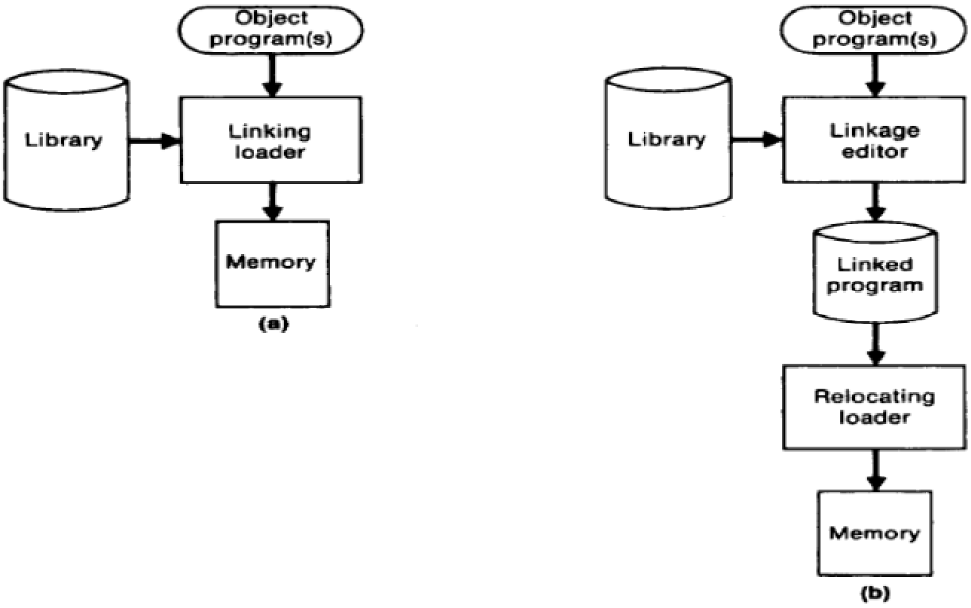
Linking loaders perform all linking and relocation at load time. There are two alternatives: Linkage editors, which perform linking prior to load time, and dynamic linking, in which the linking function is performed at execution time. Difference between linkage editor and linking loader is explained below:

• 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.

• A **linkage editor** produces a linked version of the program (load module or executable image), which is written to a file or library for later execution.

- A linkage editor produces a linked version of the program (load module or executable image), which is written to a file or library for later execution. When the user is ready to run the linked program, a simple relocating loader can be used to load the program into memory. The only object code modification necessary is the addition of an actual load address to relative values within the program.

Figure: Processing of an object program using a) linking loader and b)linkage editor



- The Linkage Editor(LE) performs relocation of all control sections relative to the start of the linked program. Thus, all items that need to be modified at load time have values that are relative to the start of the linked program. This means that the loading can be accomplished in one pass with no external symbol table required.
- If a program is to be executed many times without being reassembled, the use of a linkage editor substantially reduces the overhead required. Linkage editors can perform many useful functions besides simply preparing an object program for execution. Resolution of external reference and library searching are only performed once for linkage editor.
- If a program is under development or is used infrequently, the use of a linking loader outperforms a linkage editor.
- Consider a program PLANNER with a number of subroutines. You want to improve a subroutine (PROJECT) of the program (PLANNER) without going back to the original versions of all of the other subroutines. For that you can use linkage editor commands as follows:

```
INCLUDE PLANNER (PROGLIB)
DELETE PROJECT          // delete from existing PLANNER
INCLUDE PROJECT (NEWLIB)             // include new version
REPLACE PLANNER (PROGLIB)
```

4.4.2 Dynamic Linking/Dynamic Loading/Load-on-call

- **Linkage editors** perform linking operations before the program is loaded for execution.
- **Linking loaders** perform these same operations at load time.
- **Dynamic linking, dynamic loading, or load on call** postpones the linking function until execution time. That is a subroutine is loaded and linked to the rest of the program when it is first called.

- **Dynamic linking, dynamic loading, or load on call** postpones the linking function until execution time. That is a subroutine is loaded and linked to the rest of the program when it is first called.
- Dynamic linking is often used to allow several executing programs to share one copy of a subroutine or library (eg. run-time support routines for a high-level language like C.)
- With a program that allows its user to interactively call any of the subroutines of a large mathematical and statistical library, all of the library subroutines could potentially be needed, but only a few will actually be used in any one execution. Dynamic linking can avoid the necessity of loading the entire library for each execution except those necessary subroutines.
- For example, that a program contains subroutines that correct or clearly diagnose error in the input data during execution. If such error are rare, the correction and diagnostic routines may not be used at all during most execution of the program. However, if the program were completely linked before execution, these subroutines need to be loaded and linked every time.
- Fig 3.14 illustrates a method in which routines that are to be dynamically loaded must be called via an OS service request.

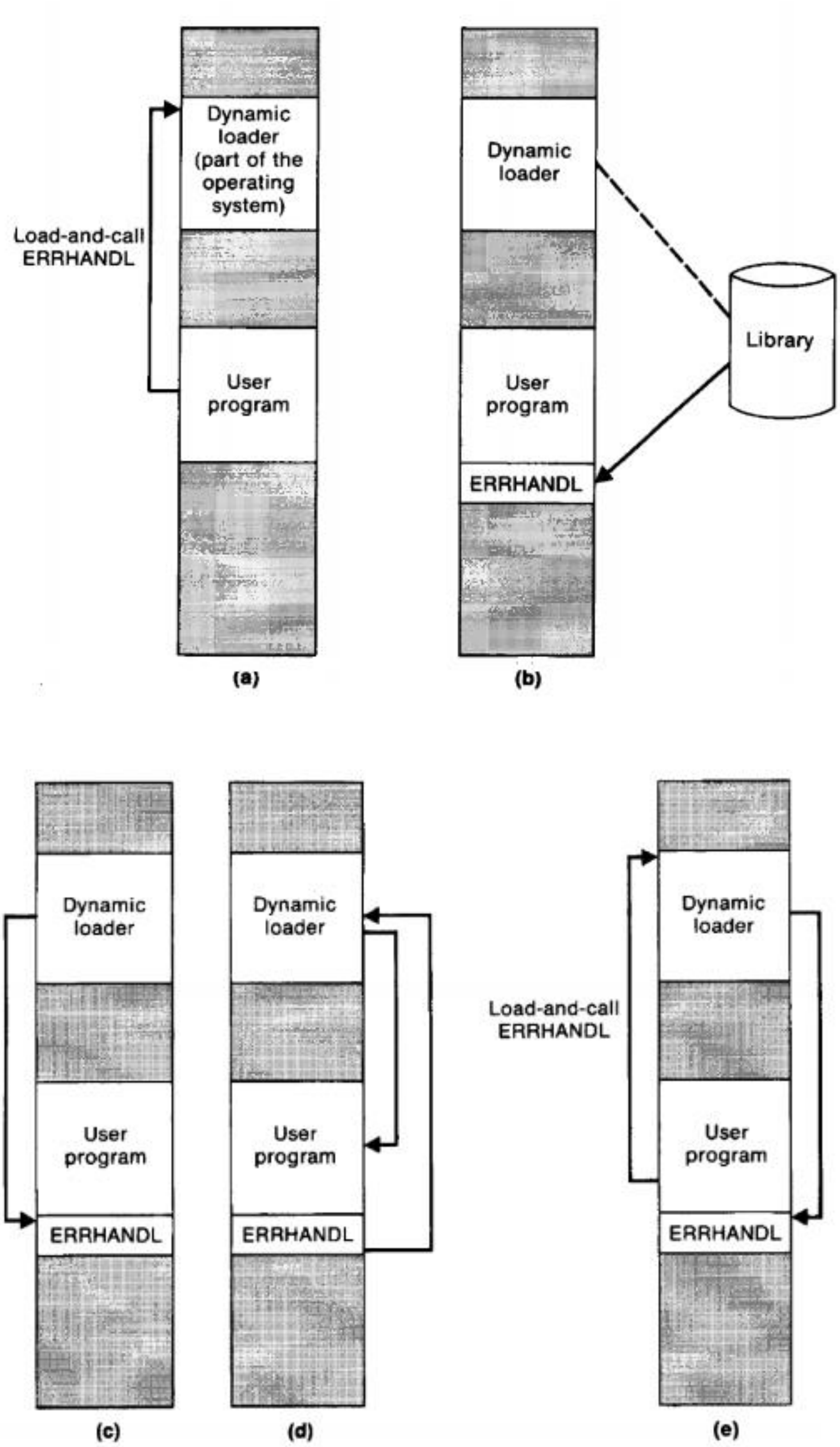


Figure: Loading and calling of a subroutine using dynamic linking

- Fig (a): Whenever the user program needs a subroutine for its execution, the program makes a load-and-call service request to OS(instead of executing a JSUB instruction referreing to an external symbol) . The parameter of this request is the symbolic name(ERRHANDL) of the routine to be called.
- Fig (b): 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.
- Fig (c): Control is then passed from OS to the routine being called.
- Fig (d): When the called subroutine completes it processing, it returns to its caller (i.e., OS). OS then returns control to the program that issued the request.
- Fig (e): If a subroutine is still in memory, a second call to it may not require another load operation. Control may simply be passed from the dynamic loader to the called routine.

4.4.3 Bootstrap Loaders

- Given an idle computer with no program in memory, how do we get things started?Two solutions are there.

1. On some computers, an absolute loader program is permanently resident in a read-only memory (ROM). When some hardware signal occurs, the machine begins to execute this ROM program. This is referred to as a bootstrap loader.

2. On some computers, there's a built-in hardware which read a fixed-length record from some device into memory at a fixed location. After the read operation, control is automatically transferred to the address in memory. If the loading process requires more instructions than can be read in a single record, this first record causes the reading of others, and these in turn can cause the reading of more records

When a computer is first turned on or restarted, a special type of absolute loader, called a **bootstrap loader**, is executed. This bootstrap loader loads the first program to be run by the computer – usually an operating system.

Module 4 Summary:

- **Basic Loader functions** - Design of absolute loader, Simple bootstrap Loader
- **Machine dependent loader features-** Relocation, Program Linking, Algorithm and data structures of two pass Linking Loader
- **Machine independent loader features** – Automatic Library Search, Loader options
- **Loader Design Options** – Linkage Editors, Dynamic Linking, Bootstrap Loaders