Chap 3 Loaders and Linkers
A loader is a system program that performs the loading function. Some loaders also support relocation and linking.
3.1 Basic loader function

(1) Fig 3.2 shows algorithm for an absolute loader.

begin
  read Header record
  verify program name and length
  read first Text record
  while record type != 'E' do
    begin
      {if object code is in character form, convert into internal representation}
      move object code to specified location in memory
      read next object program record
    end
  end
  jump to address specified in End record
end

(Figure 3.2 Algorithm for an absolute loader)
(2) When a computer is first turned on started, a special type of absolute loader, called a bootstrap loader, is executed. It loads the first program to be the first program to be run by the computer--usually an operating system.

(3) Fig 3.3 shows bootstrap loader for SIC/XE.
3.2 Machine-dependent loader feature

(1) Since
• efficient sharing of the machine (i.e. relocation),
• loading the demanded routines. loaders that allow for program relocation are built.
(i.e. relative loaders)
3.2 Machine-dependent loader feature

(2) How relocation? A relocation bit associated with each word of object code is needed to modify the address is to be added to the word which relocation bit is set to 1 when the program is relocated.

Ex.

```
FFC 1111 1111 1100 (前10個都須做relocation)
140033 481039  M___ + copy    M___ + copy
```
3.2 Machine-dependent loader feature

(3) Some computers provide a hardware solution to perform program relocation.

Ex. X86 電腦 … CS : IP …
3.2 Machine-dependent loader feature

(4) Program linking
* REF1: PROGA ➔ pc relative ➔ no relocation or linking.
  PROGB (and PROGC) ➔ modification record
* REF2: PROGA (and PROGC) ➔ modification record
  PROGB ➔ pc relative no relocation or linking
* REF3: PROGA ➔ immediate operation
  PROGB (and PROGC) ➔ two external modification
* REF4: PROGA ➔ simple external reference
  PROGB ➔ complication of relocation and external reference

Figure 3.10(b) shows the relocation and linking operations performed on REF4 from PROGA.
Relocation and linking operations performed on REF4 from PROGA

Memory contents

0000

(REF4)

004126

4050

(Actual address of LI STC)

LOAD addresses

PROGA 004000
PROGB 004063
PROGC 0040E2

Fig3.10(b)
3.2.3 Tables and logic for a linking loader

(1) The linking loader usually makes two passes over its input:

Pass 1 assigns addresses to all external symbols.
Pass 2 performs the actual loading, relocation, and linking.
3.2.3 Tables and logic for a linking loader

(2) Algorithm for Pass 1 of a linking loader:

Pass 1:
begin
get PROOADDR from operating system
set CSADDR to PROOADDR {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)
d {for}
d {while ~ 'E'}
add CSLTH to CSADDR {starting address for next control section}
d {while not EOF}
d {Pass 1}
end {Pass 1}
Figure 3.11 (a) Algorithm for Pass 1 of a linking loader.
(3) Algorithm for Pass 2 of a linking loader:

**Pass 2:**

```
begin
  set CSADDR to PROOADDR
  set EXECADDR to PROOADDR
  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 {if 'M'}
        end {while != 'E'}
      if an address is specified {in End record} then
        begin
          set EXECADDR to (CSADDR + specified address)
          add CSLTH to CSADDR
        end {while not EOF}
    end {while not EOF}
jump to location given by EXECADDR {to start execution of loaded program}
end {Pass 2}
```

Figure 3.11(b) Algorithm for Pass 2 of a linking loader.
(1) Automatic library search allows a programmer to use standard subroutines without explicitly including them in the program to be loaded.
(2) Linking loader with automatic library search -> keep track of the referred external symbols but not defined -> At the end of Pass 1, there remains the unresolved external reference -> Repeat the library search process until all reference are resolved -> If there exists unresolved external references, these must be treated as errors.
3.3 MACHINE-INDEPENDENT LOADER FEATURES

(3) Many loaders have a special command language that is used to specify options:

- Selection of alternative sources of input:
  INCLUDE program-name (library-name)
- Deletion of the named control section(s):
  DELETE csect-name
- Change external symbol name:
  CHANGE name1,name2
- Automatic inclusion of library routines:
  LIBRARY MYLIB
- Unresolved external references:
  NOCALL STDDEV,PL T,CORREL
- No external references are resolved by library search.
- Output from the loader
- Executed starting address
- Errors handling
(4) The actual loading of segment during program execution is handled by an overlay manager, which is automatically included in the root segment of the overlay program by the loader.
(5) Overlay rule: If a segment S is present in memory, all of the other segments that lie on the path from S to the root must also be present.
3.3 MACHINE-INDEPENDENT LOADER FEATURES

(6) Example of overlay management (Fig. 3.16):
Segments 1, 2 and 4 are loaded -> +JSUB B
-> SEGTAB -> segment 2 -> return to segment 1
-> + JSUB D -> SEGTAB -> OVLMGR
Segment 6 is loaded from SEGFILE into memory and SEGTAB is updated and segments 4 and 2 are removed
-> segment 6 -> return to segment 1.
(1) Linking loaders:
it performs all linking and relocation at load time.

Linkage editors:
it performs linking prior to load time.

Linkage linking:
it performs linking at execution time.
Processing of an object program using (a) linking loader and (b) linkage editor
(2)
If a program is to be executed many times without being reassembled, the use of a linkage editor substantially reduces the overhead required.
(3)
Linkage editors often allow the user to specify that external references are not be resolved by automatic library search. Ex. consider 100 FORTRAN programs using the I/O routines stored on a library.
Dynamic linking avoids the necessity of loading the entire library for each execution. Instead of executing a JSUB instruction that refers to an external symbol, the program makes a load-and-call service request to the operating system (delayed binding).
Bootstrap loader working !!

1. Power on
2. Load Bootstrap loader from ROM to Memory
3. Bootstrap loader loads Loader from Disk
4. Loader loads OS from Disk
5. Transfer control form loader to OS
Dynamic Linking working

- Dynamic loader
- User program
- Library
- Load-and-call
- ERRHANDL
Dynamic Linking working

Dynamic loader

User program

ERRHANDL

Control transfer

Library
Dynamic Linking working

Dynamic loader

Library

User program

Load-and-call

ERRHANDL

ERRHANDL