5.2 DESIGN OF AN ABSOLUTE LOADER

We introduce the general topic of loader design by presenting a design of an absolute loader.

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. Assuming that this information is transmitted on cards, a possible format is shown in Figure 5.10.

Note that in the card format shown 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.

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	Address 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	Address of entry point
6-72	Empty
73-80	Card sequence number

FIGURE 5.10 Card formats for an absolute loader

The algorithm for an absolute loader is quite simple. 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. A flowchart for this process is illustrated in Figure 5.11.

5.3 DESIGN OF A DIRECT-LINKING LOADER

In this section a design of an IBM 360-type direct-linking loader is presented. Certain obscure features (primarily related to the IBM PL/I implementation and overlay structures) have been omitted, and where alternative formats are possible, only the simplest is given.

The design steps followed will parallel those taken in the design of an assembler (Chapter 3). Note that because the direct-linking loader needs to know

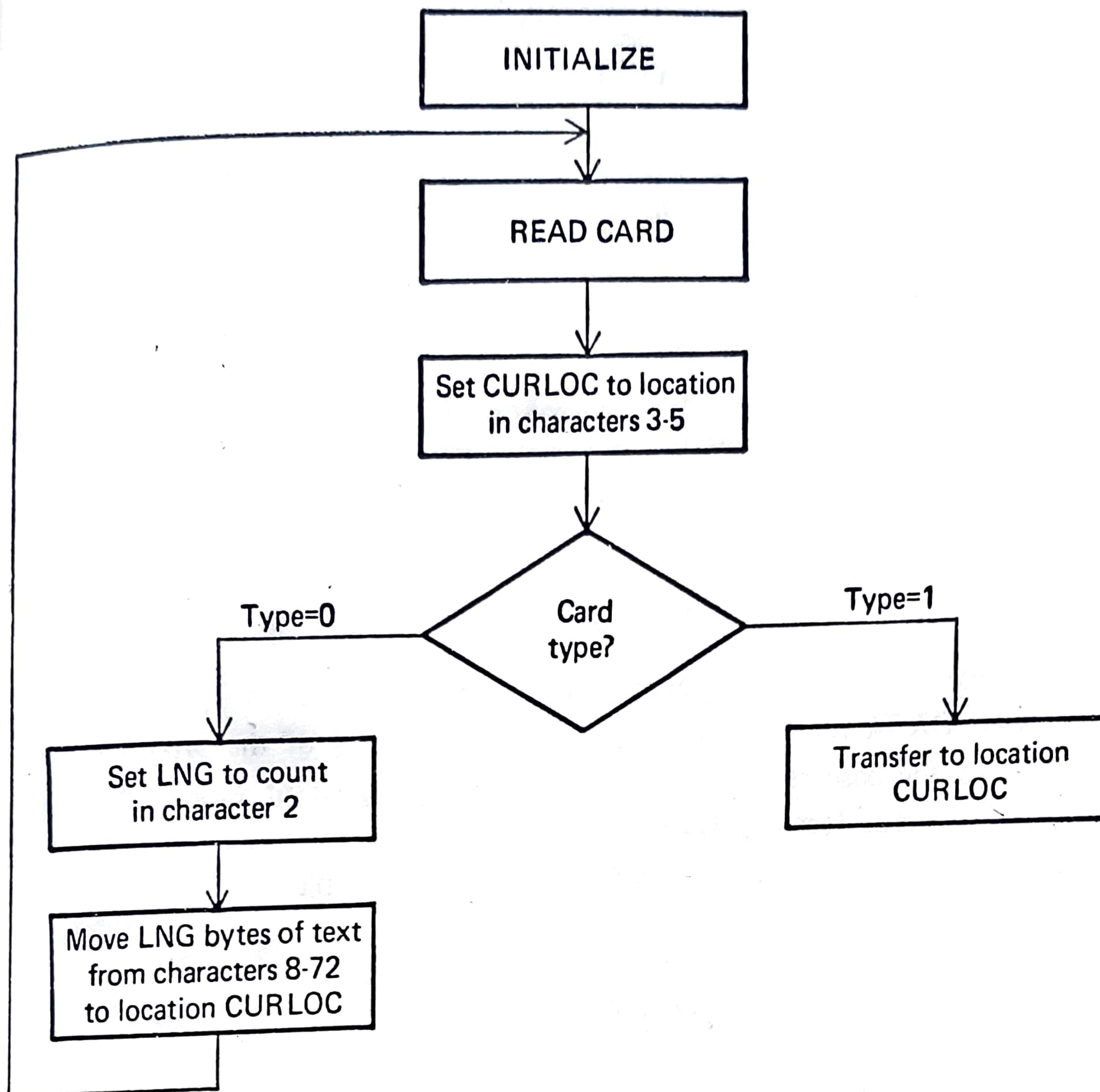

FIGURE 5.11 Absolute loader

the absolute (load time) values of some external symbols before it can perform the modifications on address constants, it requires two passes.

5.3.1 Specification of Problem

The organization of the IBM 360 facilitates the tasks to be performed by its relocating loader. On the IBM 7094, a direct access machine, it was necessary to relocate the address portion on almost all instructions. In the 360, instruction relocation is accomplished by the use of the base register, which is set by neither the assembler nor the loader. Therefore, the 360 relocating loader can treat instructions *exactly* like nonrelocatable data (full word constants, characters, etc.). However, address constants must still be relocated.

For example, the following instructions:

```
TEST      START
          USING     *,15
          L         1,DATA
            .
            .
            .
DATA      DC        F'5'
          END
```

might be assembled as:

```
Rel. loc.   Instruction/data
   0        L  1,96(0,15).
   .          .
   .          .
   .          .
  96        5
```

Regardless of where the program is loaded, the L instruction will be unchanged as long as DATA remains 96 bytes from the beginning of the program. The contents of the base register (15) obviously will be different, depending upon the program's load location.

On the other hand, consider modifying the above example:

```
                .
                .
                .
DATA        DC      F'5'
DATALOC     DC      A(DATA)
            END
```

DATALOC must contain the absolute address of DATA. The assembler knows only that DATA is 96 bytes from the beginning of the program, so the loader must add to this the load address of the program in finding the actual absolute address to be contained in DATALOC.

Let us clarify the scope of the address constant problem. An address constant may be (1) absolute; (2) simple relocatable; or (3) complex relocatable.

For example, the address constant A(LOC1-LOC2) will be:

1. *Absolute* if LOC1 and LOC2 are two relocatable symbols defined internal to program – the assembler can calculate the actual value, their difference.
2. *Simple relocatable* if LOC1 is a relocatable symbol within this procedure and LOC2 is an absolute number (e.g., LOC2 EQU 5). The assembler can calculate the difference between relative location of LOC1 and value of LOC2, but the loader must perform relocation by adding the program load address.
3. *Complex relocatable* if LOC1 and LOC2 are entries to some other program. The assembler can do nothing, and the value must be calculated by the loader.

The 360 direct-linking loader processes programs generated by the assembler, FORTRAN compiler, or PL/I compiler. Recall that neither the original source program nor the assembler symbol table is available to the loader. Therefore, the object deck must contain all information needed for relocation and linking.

There are four sections to the object deck (and four corresponding card formats):

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)

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.

EXAMPLE

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

```
A           START
            EXTRN     NAMES
              .
            L         1,ADDRNAME get address of NAME table
              .
              .
ADDRNAME    DC        A(NAMES)
            END
-------------------------------
B           START
            ENTRY     NAMES
              .
              .
NAMES       DC        ----
            END
```

There are three types of external symbols, as illustrated in the above:

1. *Segment Definition* (SD) – name on START or CSECT card.
2. *Local Definition* (LD) – specified on ENTRY card. There must be a label in same program with same name.
3. *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.

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, nonrelocated data, or initial values of address constants.

EXAMPLE

```
Relative address                  Instruction

        A                         START
                                  EXTRN      NAMES
                                  USING      *,15
                                  .
                                  .
                                  .
   40                             L          1,ALPHA
   44                             BCR        15,14
   46                             ------------------- (Skipped by assembler)
   48     ALPHA                   DC         F'5'
   52     ALLOC                   DC         A(ALPHA)
   56     ADDRNAME                DC         A(NAMES)
                                  .
                                  .
                                  .
```

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
```

The RLD cards contain the following information.

1. The location and length of each address constant that needs to be changed for relocation or linking
2. The external symbol by which the address constant should be modified (added or subtracted)
3. The operation to be performed (add or subtract)

Rather than using the actual external symbol's name on the RLD card, as implied in section 5.1.6 and Figure 5.8, the external symbol's identifier, or ID, is used. There are various reasons for this, the major one probably being that the ID is only a single byte long, compared to the eight bytes occupied by the

symbol name, so that a considerable amount of space is saved on the RLD cards. Unfortunately, this space-saving technique causes increased loader complexity as will be shown later.

The preceding program segment used as an example for TXT cards would result in the following RLD cards

ID	*Flag*	*Length*	*Rel. loc.*
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.

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.

There is a final card required to specify the end of a collection of object decks. The 360 loaders usually use either a loader terminate (LDT) or End of File (EOF) card.

Subroutine A	ESD TXT RLD END
Subroutine B	ESD TXT RLD END
Subroutine C	ESD TXT RLD END
	EOF or LDT

The simple programs PG1 and PG2 in Figure 5.12 illustrate a wide range of relocation and linking situations. Figures 5.13 and 5.14 display the ESD, TXT, and RLD cards produced by the assembler for PG1 and PG2, respectively. Finally, Figure 5.15 depicts the contents of main storage after the programs have been allocated space, relocated, linked, and loaded. The reader should examine these figures carefully and validate the correctness and reasons for each value.

A few specific points in these examples should be noted. Both PG1 and PG2 contain an address constant of the form A(PG1ENT2-PG1ENT1-3). It should be

(a) Procedure PG1

Source card reference	Relative address		Sample program (source deck)	
1	0	PG1	START	
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	

(b) Procedure PG2

Source card reference	Relative address		Sample program (source deck)	
12	0	PG2	START	
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	

FIGURE 5.12 Sample procedures PG1 and PG2

reassuring to note in Figure 5.15 that both instances of this address constant (location 152, 200) have the same value — 7. Since both PG1ENT2 and PG1ENT1 are symbols internal to PG1, the assembler, processing PG1, can compute the entire expression and determine the value of 7. We see in Figure 5.13 that the TXT card for location 48-51 contains the 7 and there are no associated RLD cards for this address constant. On the other hand, these symbols are external to PG2; thus, the assembler, while processing PG2, has no means of evaluating the address constant. This is illustrated in Figure 5.14. The TXT card for relative locations 32-35 contains a -3, the only part of the address constant that can be calculated by the assembler. The last two RLD cards tell the loader to add the value of ID 03, which is PG1ENT2, to locations 32-35 and then subtract the value of ID 02, which is PG1ENT1. When processed by the loader, this address constant in PG2 will indeed have the same value as the one in PG1.

Since the direct-linking loader may encounter external references in an object deck which cannot be evaluated until a later object deck is processed, this type of loader requires two passes. Their functions are very similar to those of the two passes of an assembler. The major function of pass 1 of a direct-linking loader is to allocate and assign each program a location in core and create a

ESD cards

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

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

Source card reference	Relative address	Contents	Comments
6	40-43	20	
7	44-47	45	= 30 + 15
8	48-51	7	= 30-20-3
9	52-55	0	unknown to PG1
10	56-59	-16	= -20 + 4

RLD cards

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

FIGURE 5.13 Object deck program PG1

symbol table filling in the values of the external symbols. The major function of pass 2 is to load the actual program text and perform the relocation modification of any address constants needing to be altered.

The first pass allocates and assigns storage locations to all segments and stores the values of all external symbols in a symbol table. These external symbols appear as local definitions on the ESD cards of another assembled program. For every external reference symbol there must be a corresponding internal symbol in some other program. The loader inserts the absolute address of all of these external symbols in the symbol table. In the second pass, the loader places the text into the assigned locations and performs the relocation task, modifying relocatable constants. Figure 5.16 depicts the interplay between the passes of a loader in a direct-linking loading scheme.

ESD cards

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

TXT cards
(only the interesting ones)

Source card reference	Relative address	Contents
16	24-27	0
17	28-31	15
18	32-25	-3

RLD cards

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

FIGURE 5.14 Object deck program PG2

5.3.2 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 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.
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.