CHAPTER 7

Linkers

Execution of a program written in a language L involves the following steps:

1. *Translation* of the program
2. *Linking* of the program with other programs needed for its execution
3. *Relocation* of the program to execute from the specific memory area allocated to it
4. *Loading* of the program in the memory for the purpose of execution.

These steps are performed by different language processors. Step 1 is performed by the translator for language L. Steps 2 and 3 are performed by a *linker* while Step 4 is performed by a *loader*. The terms linking, relocation and loading are defined in a later section.

Figure 7.1 contains a schematic showing steps 1–4 in the execution of a program. The translator outputs a program form called *object module* for the program. The linker processes a set of object modules to produce a ready-to-execute program form, which we will call a *binary program*. The loader loads this program into the memory for the purpose of execution. As shown in the schematic, the object module(s) and ready-to-execute program forms can be stored in the form of files for repeated use.

Translated, linked and load time addresses

While compiling a program P, a translator is given an origin specification for P. This is called the *translated origin* of P. (In an assembly program, the programmer can specify the origin in a `START` or `ORIGIN` statement.) The translator uses the value of the translated origin to perform memory allocation for the symbols declared in P. This results in the assignment of a *translation time address* t_{symb} to each symbol *symb* in the program. The *execution start address* or simply the *start address* of a program is the address of the instruction from which its execution must begin. The start address specified by the translator is the *translated start address* of the program

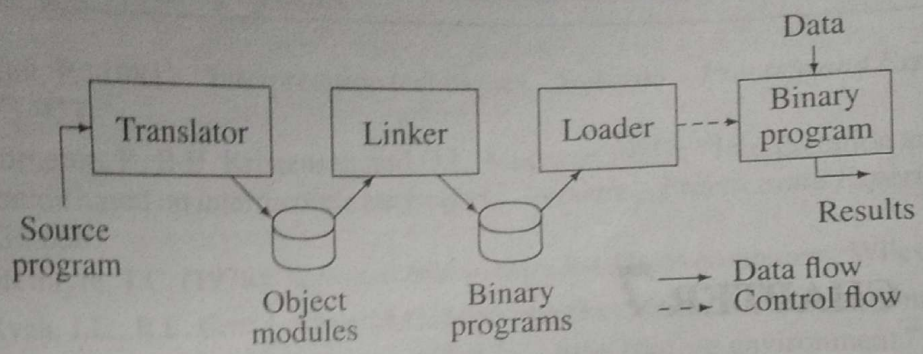

Fig. 7.1 A schematic of program execution

The origin of a program may have to be changed by the linker or loader for one of two reasons. First, the same set of translated addresses may have been used in different object modules constituting a program, e.g. object modules of library rotines often have the same translated origin. Memory allocation to such programs would conflict unless their origins are changed. Second, an operating system may require that a program should execute from a specific area of memory. This may require a change in its origin. The change of origin leads to changes in the execution start address and in the addresses assigned to symbols. The following terminology is used to refer to the address of a program entity at different times:

1. *Translation time* (or *translated*) *address:* Address assigned by the translator.
2. *Linked address:* Address assigned by the linker.
3. *Load time* (or *load*) *address:* Address assigned by the loader.

The same prefixes *translation time* (or *translated*), *linked* and *load time* (or *load*) are used with the origin and execution start address of a program. Thus,

1. *Translated origin:* Address of the origin assumed by the translator. This is the address specified by the programmer in an ORIGIN statement.
2. *Linked origin:* Address of the origin assigned by the linker while producing a binary program.
3. *Load origin:* Address of the origin assigned by the loader while loading the program for execution.

The linked and load origins may differ from the translated origin of a program due to one of the reasons mentioned earlier.

Example 7.1 Consider the assembly program and its generated code shown in Fig. 7.2. The translated origin of the program is 500. The translation time address of LOOP is therefore 501. If the program is loaded for execution in the memory area starting with the address 900. the load time origin is 900. The load time address of LOOP would be 901.

```
                Statement                    Address     Code
                START   500
                ENTRY   TOTAL
                EXTRN   MAX, ALPHA
                READ    A                    500)    + 09 0 540
LOOP                                         501)
                 ⋮
                MOVER   AREG, ALPHA          518)    + 04 1 000
                BC      ANY, MAX             519)    + 06 6 000
                 ⋮
                BC      LT, LOOP             538)    + 06 1 501
                STOP                         539)    + 00 0 000
A               DS      1                    540)
TOTAL           DS      1                    541)
                END
```

Fig. 7.2 A sample assembly program and its generated code

7.1 RELOCATION AND LINKING CONCEPTS

7.1.1 Program Relocation

Let AA be the set of absolute addresses—instruction or data addresses—used in the instructions of a program P. AA $\neq \phi$ implies that program P assumes its instructions and data to occupy memory words with specific addresses. Such a program—called an *address sensitive program*—contains one or more of the following:

1. An *address sensitive instruction*: an instruction which uses an address $a_i \in$ AA.
2. An *address constant*: a data word which contains an address $a_i \in$ AA.

In the following, we discuss relocation of programs containing address sensitive instructions. Address constants are handled analogously.

An address sensitive program P can execute correctly only if the start address of the memory area allocated to it is the same as its translated origin. To execute correctly from any other memory area, the address used in each address sensitive instruction of P must be 'corrected'.

Definition 7.1 (Program relocation) Program relocation *is the process of modifying the addresses used in the address sensitive instructions of a program such that the program can execute correctly from the designated area of memory.*

If linked origin $\neq$ translated origin, relocation must be performed by the linker. If load origin $\neq$ linked origin, relocation must be performed by the loader. In general, a linker always performs relocation, whereas some loaders do not. For simplicity, in the first part of the chapter it has been assumed that loaders do not perform

relocation—that is, load origin = linked origin. Such loaders are called *absolute loaders*. Hence the terms 'load origin' and 'linked origin' are used interchangeably. However, it would have been more precise to use the term 'linked origin'. (Loaders that perform relocation, i.e. *relocating loaders*, are discussed in Section 7.6.)

Example 7.2 The translated origin of the program in Fig. 7.2 is 500. The translation time address of symbol A is 540. The instruction corresponding to the statement READ A (existing in translated memory word 500) uses the address 540, hence it is an address sensitive instruction. If the linked origin is 900, A would have the link time address 940. Hence the address in the READ instruction should be corrected to 940. Similarly the instruction in translated memory word 538 contains 501, the address of LOOP. This should be corrected to 901. (Note that the operand addresses in the instructions with the addresses 518 and 519 also need to be corrected. This is explained in Section 7.1.2.)

Performing relocation

Let the translated and linked origins of program P be t_origin_P and l_origin_P, respectively. Consider a symbol *symb* in P. Let its translation time address be t_{symb} and link time address be l_{symb}. The relocation factor of P is defined as

$$relocation\_factor_P = l\_origin_P - t\_origin_P \quad (7.1)$$

Note that $relocation\_factor_P$ can be positive, negative or zero.

Consider a statement which uses *symb* as an operand. The translator puts the address t_{symb} in the instruction generated for it. Now,

$$t_{symb} = t\_origin_P + d_{symb}$$

where d_{symb} is the offset of *symb* in P. Hence

$$l_{symb} = l\_origin_P + d_{symb}$$

Using (7.1),

$$\begin{aligned} l_{symb} &= t\_origin_P + relocation\_factor_P + d_{symb} \\ &= t\_origin_P + d_{symb} + relocation\_factor_P \\ &= t_{symb} + relocation\_factor_P \end{aligned} \quad (7.2)$$

Let IRR_P designate the set of instructions requiring relocation in program P. Following (7.2), relocation of program P can be performed by computing the relocation factor for P and adding it to the translation time address(es) in every instruction $i \in IRR_P$.

Example 7.3 For the program of Fig. 7.2

$$\begin{aligned} \text{relocation factor} &= 900 - 500 \\ &= 400. \end{aligned}$$

Relocation is performed as follows: IRR_P contains the instructions with translated addresses 500 and 538. The instruction with translated address 500 contains the address 540 in the operand field. This address is changed to (540+400) = 940. Similarly, 400 is added to the operand address in the instruction with translated address 538. This achieves the relocation explained in Ex. 7.2.

7.1.2 Linking

Consider an application program AP consisting of a set of program units $SP = \{P_i\}$. A program unit P_i interacts with another program unit P_j by using addresses of P_j's instructions and data in its own instructions. To realize such interactions, P_j and P_i must contain public definitions and external references as defined in the following:

Public definition	a symbol *pub_symb* defined in a program unit which may be referenced in other program units
External reference	a reference to a symbol *ext_symb* which is not defined in the program unit containing the reference.

The handling of public definitions and external references is described in the following.

EXTRN and ENTRY statements

The ENTRY statement lists the public definitions of a program unit, i.e. it lists those symbols defined in the program unit which may be referenced in other program units. The EXTRN statement lists the symbols to which external references are made in the program unit.

Example 7.4 In the assembly program of Fig. 7.2, the ENTRY statement indicates that a public definition of TOTAL exists in the program. Note that LOOP and A are not public definitions even though they are defined in the program. The EXTRN statement indicates that the program contains external references to MAX and ALPHA. The assembler does not know the address of an external symbol. Hence it puts zeroes in the address fields of the instructions corresponding to the statements MOVER AREG, ALPHA and BC ANY, MAX. If the EXTRN statement did not exist, the assembler would have flagged references to MAX and ALPHA as errors.

Resolving external references

Before the application program AP can be executed, it is necessary that for each P_i in SP, every external reference in P_i should be bound to the correct link time address.

Definition 7.2 (Linking) Linking *is the process of binding an external reference to the correct link time address.*

An external reference is said to be *unresolved* until linking is performed for it. It is said to be *resolved* when its linking is completed.

```
                Statement          Address      Code
                START   200
                ENTRY   ALPHA
                - -
                - -
ALPHA           DS      25         231)         + 00 0 025
                END
```

Fig. 7.3 Program unit Q

Example 7.5 Let the program unit of Fig. 7.2 (referred to as program unit P) be linked with the program unit Q described in Fig. 7.3.

Program unit P contains an external reference to symbol ALPHA which is a public definition in Q with the translation time address 231. Let the link origin of P be 900 and its size be 42 words. The link origin of Q is therefore 942, and the link time address of ALPHA is 973. Linking is performed by putting the link time address of ALPHA in the instruction of P using ALPHA, i.e. by putting the address 973 in the instruction with the translation time address 518 in P.

Binary programs

Definition 7.3 (**Binary program**) *A* binary program *is a machine language program comprising a set of program units SP such that* $\forall P_i \in SP$

1. *P_i has been relocated to the memory area starting at its link origin, and*
2. *Linking has been performed for each external reference in* P_i.

To form a binary program from a set of object modules, the programmer invokes the linker using the command

```
linker   <link origin>, <object module names>
         [, <execution start address> ]
```

where *<link origin>* specifies the memory address to be given to the first word of the binary program. *<execution start address>* is usually a pair (program unit name, offset in program unit). The linker converts this into the linked start address. This is stored along with the binary program for use when the program is to be executed. If specification of *<execution start address>* is omitted the execution start address is assumed to be the same as the linked origin.

Note that a linker converts the object modules in the set of program units SP into a binary program. Since we have assumed link address = load address, the loader simply loads the binary program into the appropriate area of memory for the purpose of execution.

7.1.3 Object Module

The object module of a program contains all information necessary to relocate and link the program with other programs. The object module of a program P consists of 4 components:

1. *Header:* The header contains *translated origin*, *size* and *execution start address* of P.
2. *Program:* This component contains the machine language program corresponding to P.
3. *Relocation table:* (RELOCTAB) This table describes IRR_P. Each RELOCTAB entry contains a single field:

   Translated address : Translated address of an address sensitive instruction.

4. *Linking table* (LINKTAB): This table contains information concerning the public definitions and external references in P.

   Each LINKTAB entry contains three fields:

   Symbol : Symbolic name
   Type : PD/EXT indicating whether public definition or external reference
   Translated address : For a public definition, this is the address of the first memory word allocated to the symbol. For an external reference, it is the address of the memory word which is required to contain the address of the symbol.

Example 7.6 Consider the assembly program of Fig. 7.2. The object module of the program contains the following information:

1. *translated origin* = 500, *size* = 42, *execution start address* = 500.
2. Machine language instructions shown in Fig. 7.2.
3. Relocation table

500
538

4. Linking table

ALPHA	EXT	518
MAX	EXT	519
A	PD	540

TOTAL PD 541

Note that the symbol LOOP does not appear in the linking table. This is because it is not declared as a public definition, i.e. it does not appear in an ENTRY statement.

7.2 DESIGN OF A LINKER

7.2.1 Relocation and Linking Requirements in Segmented Addressing

The relocation requirements of a program are influenced by the addressing structure of the computer system on which it is to execute. Use of the segmented addressing structure reduces the relocation requirements of a program.

Example 7.7 Consider the program of Fig. 7.4 written in the assembly language of Intel 8088. The ASSUME statement declares the segment registers CS and DS to be available for memory addressing. Hence all memory addressing is performed by using suitable displacements from their contents. Translation time address of A is 0196. In statement 16, a reference to A is assembled as a displacement of 196 from the contents of the CS register. This avoids the use of an absolute address, hence the instruction is not address sensitive. Now no relocation is needed if segment SAMPLE is to be loaded in the memory starting at the address 2000 because the CS register would be loaded with the address 2000 by a calling program (or by the OS). The effective operand address would be calculated as <CS> + 0196, which is the correct address 2196. A similar situation exists with the reference to B in statement 17. The reference to B is assembled as a displacement of 0002 from the contents of the DS register. Since the DS register would be loaded with the execution time address of DATA_HERE, the reference to B would be automatically relocated to the correct address.

Sr. no.	*Statement*			*Offset*
0001	DATA_HERE	SEGMENT		
0002	ABC	DW	25	0000
0003	B	DW ?		0002
⋮		⋮		
0012	SAMPLE	SEGMENT		
0013		ASSUME	CS:SAMPLE, DS:DATA_HERE	
0014		MOV	AX, DATA_HERE	0000
0015		MOV	DS, AX	0003
0016		JMP	A	0005
0017		MOV	AL,B	0008
⋮		⋮		
0027	A	MOV	AX,BX	0196
⋮		⋮		
0043	SAMPLE	ENDS		
0044		END		

Fig. 7.4 An 8088 assembly program for linking

Though use of segment registers reduces the relocation requirements, it does not completely eliminate the need for relocation. Consider statement 14 of Fig. 7.4, viz.

```
MOV          AX, DATA_HERE
```

which loads the segment base of DATA_HERE into the AX register preparatory to its transfer into the DS register. Since the assembler knows DATA_HERE to be a segment, it makes provision to load the higher order 16 bits of the address of DATA_HERE into the AX register. However, it does not know the link time address of DATA_HERE, hence it assembles the MOV instruction in the immediate operand format and puts zeroes in the operand field. It also makes an entry for this instruction in RELOCTAB so that the linker would put the appropriate address in the operand field. Inter-segment calls and jumps are handled in a similar way.

Relocation is somewhat more involved in the case of intra-segment jumps assembled in the FAR format. For example, consider the following program:

```
FAR_LAB   EQU     THIS FAR   ; FAR_LAB is a FAR label
          - - -
          JMP     FAR_LAB    ; A FAR jump
```

Here the displacement and the segment base of FAR_LAB are to be put in the JMP instruction itself. The assembler puts the displacement of FAR_LAB in the first two operand bytes of the instruction, and makes a RELOCTAB entry for the third and fourth operand bytes which are to hold the segment base address. A statement like

```
ADDR_A    DW        OFFSET A
```

(which is an 'address constant') does not need any relocation since the assembler can itself put the required offset in the bytes. In summary, the only RELOCTAB entries that must exist for a program using segmented memory addressing are for the bytes that contain a segment base address.

For linking, however, both segment base address and offset of the external symbol must be computed by the linker. Hence there is no reduction in the linking requirements.

7.2.2 Relocation Algorithm

Algorithm 7.1 (Program relocation)

1. *program_linked_origin* := <*link origin*> from `linker` command;
2. For each object module
   (a) *t_origin* := *translated origin* of the object module;
   OM_size := *size* of the object module;
   (b) *relocation_factor* := *program_linked_origin* − *t_origin*;
   (c) Read the machine language program in *work_area*.
   (d) Read RELOCTAB of the object module.
   (e) For each entry in RELOCTAB
      (i) *translated_addr* := address in the RELOCTAB entry;
      (ii) *address_in_work_area* := address of *work_area*
      + *translated_address* − *t_origin*;

(iii) Add *relocation_factor* to the operand address in the word with the address *address_in_work_area*.

(f) *program_linked_origin* := *program_linked_origin* + *OM_size*;

The computations performed in the algorithm are along the lines described in Section 7.1.1. The only new action is the computation of the work area address of the word requiring relocation (step 2(e)(ii)). Step 2(f) increments *program_linked_origin* so that the next object module would be granted the next available load address.

Example 7.8 Let the address of *work_area* be 300. While relocating the object module of Ex. 7.6, *relocation factor* = 400. For the first RELOCTAB entry, *address_in_work_area* = 300 + 500 − 500 = 300. This word contains the instruction for `READ A`. It is relocated by adding 400 to the operand address in it. For the second RELOCTAB entry, *address_in_work_area* = 300 + 538 − 500 = 338. The instruction in this word is similarly relocated by adding 400 to the operand address in it.

7.2.3 Linking Requirements

Features of a programming language influence the linking requirements of programs. In Fortran all program units are translated separately. Hence all subprogram calls and `common` variable references require linking. Pascal procedures are typically nested inside the main program. Hence procedure references do not require linking—they can be handled through relocation. References to built in functions, however, require linking. In C, program files are translated separately. Thus, only function calls that cross file boundaries and references to global data require linking.

A reference to an external symbol `alpha` can be resolved only if `alpha` is declared as a public definition in some object module. This observation forms the basis of program linking. The linker processes all object modules being linked and builds a table of all public definitions and their load time addresses. Linking for `alpha` is simply a matter of searching for `alpha` in this table and copying its linked address into the word containing the external reference.

A *name table* (NTAB) is defined for use in program linking. Each entry of the table contains the following fields:

Symbol	:	symbolic name of an external reference or an object module.
Linked_address	:	For a public definition, this field contains linked address of the symbol. For an object module, it contains the linked origin of the object module.

Most information in NTAB is derived from LINKTAB entries with *type* = PD.

Algorithm 7.2 (Program Linking)

1. *program_linked_origin* := <*link origin*> from `linker` command.
2. For each object module

(a) *t_origin* := translated origin of the object module;
OM_size := size of the object module;
(b) *relocation_factor* := *program_linked_origin* − *t_origin*;
(c) Read the machine language program in *work_area*.
(d) Read LINKTAB of the object module.
(e) For each LINKTAB entry with *type* = PD
{ *name* := *symbol*;
linked_address := *translated_address* + *relocation_factor*;
Enter (*name*, *linked_address*) in NTAB.
(f) Enter (object module name, *program_linked_origin*) in NTAB.
(g) *program_linked_origin* := *program_linked_origin* + *OM_size*;

3. For each object module
(a) *t_origin* := translated origin of the object module;
program_linked_origin := load address from NTAB;
(b) For each LINKTAB entry with *type* = EXT
(i) *address_in_work_area* := address of *work_area* +
program_linked_origin − <*link origin*>
+ *translated address* − *t_origin*;
(ii) Search *symbol* in NTAB and copy its linked address. Add the linked address to the operand address in the word with the address *address_in_work_area*.

Example 7.9 While linking program P of Ex. 7.2 and program Q of Ex. 7.5 with *linked_origin* = 900, NTAB contains the following information

symbol	*linked address*
P	900
Q	942
ALPHA	973

TOTAL 941

Let the address of *work_area* be 300. When the LINKTAB entry of ALPHA is processed during linking, *address_in_work_area* := 300 + 900 − 900 + 518 − 500, i.e. 318. Hence, the linked address of ALPHA, i.e. 973, is copied from the NTAB entry of ALPHA and added to the word in address 318 (step 3(b)(ii)).

EXERCISE 7.2

1. It is required to merge a set of object modules $\{om_i\}$ to construct a single object module om'. This reduces the linking and relocation time in situations where the object modules in $\{om_i\}$ are interdependent—that is, they call or use each other.
(a) What are the public definitions of om'?

(b) What are the external references in *om''*?
(c) Explain how the RELOCTAB's and LINKTAB's can be merged.

2. Comment on the feasibility of reconstructing the original object modules from *om''*. Is any additional information required to facilitate it?

7.3 SELF-RELOCATING PROGRAMS

The manner in which a program can be modified, or can modify itself, to execute from a given load origin can be used to classify programs into the following:

1. Non relocatable programs,
2. Relocatable programs,
3. Self-relocating programs.

A non relocatable program is a program which cannot be executed in any memory area other than the area starting on its translated origin. Non relocatability is the result of address sensitivity of a program and lack of information concerning the address sensitive instructions in the program. The difference between a relocatable program and a non relocatable program is the availability of information concerning the address sensitive instructions in it. A relocatable program can be processed to relocate it to a desired area of memory. Representative examples of non relocatable and relocatable programs are a hand coded machine language program and an object module, respectively.

A self-relocating program is a program which can perform the relocation of its own address sensitive instructions. It contains the following two provisions for this purpose:

1. A table of information concerning the address sensitive instructions exists as a part of the program.
2. Code to perform the relocation of address sensitive instructions also exists as a part of the program. This is called the *relocating logic*.

The start address of the relocating logic is specified as the execution start address of the program. Thus the relocating logic gains control when the program is loaded in memory for execution. It uses the load address and the information concerning address sensitive instructions to perform its own relocation. Execution control is now transferred to the relocated program.

A self-relocating program can execute in any area of the memory. This is very important in time sharing operating systems where the load address of a program is likely to be different for different executions.

EXERCISE 7.3

1. Comment on the following statements:
   (a) Self-relocating programs are less efficient than relocatable programs.

(b) There would be no need for linkers if all programs are coded as self relocating programs.

2. A self-relocating program needs to find its load address before it can execute its relocating logic. Comment on how this information can be determined by the program.

7.4 A LINKER FOR MS DOS

We discuss the design of a linker for the Intel 8088/80x86 processors which resembles LINK of MS DOS in many respects. The design uses the schematics of the previous section. It may be noted that the object modules of MS DOS differ from the Intel specifications in some respects. Further simplifications are made for the purpose of this discussion.

Object Module Format

An Intel 8088 object module is a sequence of *object records*, each object record describing specific aspects of the programs in the object module. There are 14 types of object records containing the following five basic categories of information:

1. Binary image (i.e. code generated by a translator)
2. External references
3. Public definitions
4. Debugging information (e.g. line number in source program)
5. Miscellaneous information (e.g. comments in the source program).

We only consider the object records corresponding to first three categories—a total of eight object record types.

The names and purpose of the object record types is summarized in Table 7.1. The formats of the records are shown in Fig. 7.5. The object records of an object module must appear in the specific order shown in Table 7.1. Each object record contains variable length information and may refer to the contents of previous object records. Each name in an object record is represented in the following format:

length (1 byte)	name

THEADR, LNAMES and SEGDEF records

The module name in the THEADR record is typically derived by the translator from the source file name. This name is used by the linker to report errors. An assembly programmer can specify the module name in the NAME directive. The LNAMES record lists the names for use by SEGDEF records. A SEGDEF record designates a segment name using an index into this list. The *attributes* field of a SEGDEF record indicates whether the segment is relocatable or absolute, whether (and in what manner) it can be combined with other segments, as also the alignment requirement of its base address (e.g. byte, word or paragraph, i.e. 16 byte, alignment). Stack segments

THEADR record

80H	length	T-module name	check-sum

LNAMES record

96H	length	name list	check-sum

SEGDEF record

98H	length	attributes (1-4)	segment length (2)	name index (1)	check-sum

EXTDEF record

8CH	length	external reference list	check-sum

PUBDEF record

90H	length	base (2-4)	name	offset (2)	...	check-sum

LEDATA record

A0H	length	segment index (1-2)	data offset (2)	data	check-sum

FIXUPP record

9CH	length	locat (1)	fix dat (1)	frame datum (1)	target datum (1)	target offset (2)	...	check sum

MODEND record

8AH	length	type (1)	start addr (5)	check-sum

Fig. 7.5 *Object record formats*

Table 7.1 Object Records of Intel 8088

Record type	*Id (Hex)*	*Description*
THEADR	80	Translator header record
LNAMES	96	List of names record
SEGDEF	98	Segment definition record
EXTDEF	8C	External names definition record
PUBDEF	90	Public names definition record
LEDATA	AO	Enumerated data (binary image)
FIXUPP	9C	Fixup record
MODEND	8A	Module end record

with the same name are concatenated with each other, while common segments with the same name are overlapped with one another. The *attribute* field also contains the origin specification for an absolute segment.

EXTDEF and PUBDEF records

The EXTDEF record contains a list of external references used by the programs of this module. A FIXUPP record designates an external symbol name by using an index into this list. A PUBDEF record contains a list of public names declared in a segment of the object module. The *base specification* identifies the segment. Each (*name*, *offset*) pair in the record defines one public name, specifying the name of the symbol and its offset within the segment designated by the base specification.

LEDATA records

An LEDATA record contains the binary image of the code generated by the language translator. *segment index* identifies the segment to which the code belongs, and *offset* specifies the location of the code within the segment.

FIXUPP records

A FIXUPP record contains information for one or more relocation and linking fixups to be performed. The *locat* field contains a numeric code called *loc code* to indicate the type of a fixup. The meanings of these codes are given in Table 7.2.

Table 7.2 FIXUPP codes

Loc code	*Meaning*
0	low order byte is to be fixed
1	offset is to be fixed
2	segment is to be fixed
3	pointer (i.e., segment : offset) is to be fixed

locat also contains the offset of the fixup location in the previous LEDATA record. The *frame datum* field, which refers to a SEGDEF record, identifies the segment to which the fixup location belongs. The *target datum* and *target offset* fields specify the relocation or linking information. *target datum* contains a segment index or an external index, while *target offset* contains an offset from the name indicated in *target datum*. The *fix dat* field indicates the manner in which the *target datum* and *target offset* fields are to be interpreted. The numeric codes used for this purpose are given in Table 7.3.

Table 7.3 Codes in *fixdat* field

code	*contents of target datum and offset fields*
0	segment index and displacement
2	external index and target displacement
4	segment index (offset field is not used)
6	external index (offset field is not used)

Example 7.10 Consider the assembly program

```
EXTRN     ABC, XYZ
LEA       BX, ABC+25H
MOV       AX, OFFSET XYZ
- - -
```

In the LEA statement, the assembler can put zeroes in the second operand field of the instruction and use code '2' in *fix dat* to indicate the linking requirement. ABC would be identified through the external index put in *target datum*, while 25H would be put in the *target offset* field. *loc code* would be '3'. An alternative would have been to put 25H in the operand field of the instruction and simply use code '6' in *fix dat* and '2' in *loc code* to indicate the linking requirement. For the MOV statement code '6' would be used in *fix dat*. *locat* would contain '1' indicating that offset of the effective address is to be put in the designated location.

MODEND record

The MODEND record signifies the end of the module, with the *type* field indicating whether it is the main program. This record also optionally indicates the execution start address. This has two components: (a) the segment, designated as an index into the list of segment names defined in SEGDEF record(s), and (b) an offset within the segment. The exact format in which this information is represented is not important for our discussion.

Example 7.11 Figure 7.6 illustrates two assembly language programs, each consisting of a single segment, which are assembled separately. From the viewpoint of linking, only the two MOV statements of segment COMPUTE and the CALL, LEA and MOV statements of segment PART2 are of interest. Hence only the object module records concerning these statements are shown in Fig. 7.7. Note that the object modules have been given

Sr. no.		*Statement*		*Offset*
0001				
0002	COMPUTE	NAME	FIRST	
0003		SEGMENT		
0004		EXTRN	PHI:BYTE, PSI:WORD	
0007	ALPHA	PUBLIC	ALPHA, BETA	
⋮		...		
		⋮		0015
0012		MOV	AX, SEG PHI	
⋮		⋮		0028
0022		MOV	AX, OFFSET PSI	
⋮		⋮		0056
0029	BETA	...		
⋮		⋮		0084
0035		DB	25	
0036	COMPUTE	ENDS		0123
0037		END		

0001				
0002	PART2	NAME	SECOND	
0003		SEGMENT	PARA	
0004		EXTRN	ALPHA,BETA:FAR,GAMMA	
0010		PUBLIC	PHI, PSI	
⋮		JMP	BETA	0018
		⋮		
0017	PHI	...		0033
⋮		⋮		
0027	PSI	...		0059
⋮		⋮		
0051		LEA	BX, ALPHA+20H	0245
⋮		⋮		
0057		MOV	AX, SEG GAMMA	0279
⋮		⋮		
0069	PART2	ENDS		
0070		END		

Fig. 7.6 Sample MS DOS assembly language programs

Object Module FIRST

Type	*Length*	*Other fields*	*Check sum*	
80H	...	05 FIRST	...	THEADR
96H	...	07 COMPUTE	...	LNAMES
98H	...	20H 124 01	...	SEGDEF
90H	...	01 05 ALPHA 0015	...	PUBDEF
90H	...	01 04 BETA 0084	...	PUBDEF
8CH	...	03 PHI 03 PSI	...	EXTDEF
AOH	...	01 0028 A1 00 00	...	LEDATA
9CH	...	8801 06 01 01	...	FIXUPP
AOH	...	01 0056 A1 00 00	...	LEDATA
9CH	...	8401 06 01 02	...	FIXUPP
8AH	...	COH 01 00	...	MODEND

Object Module SECOND

Type	*Length*	*Other fields*	*Check sum*	
80H	...	06 SECOND	...	THEADR
96H	...	05 PART2	...	LNAMES
98H	...	60H 398 01	...	SEGDEF
90H	...	01 03 PHI 0033	...	PUBDEF
90H	...	01 03 PSI 0059	...	PUBDEF
8CH	...	05 ALPHA 04 BETA 05 GAMMA	...	EXTDEF
AOH	...	01 0018 EA 00 00 00 00	...	LEDATA
9CH	...	8C01 06 01 02	...	FIXUPP
AOH	...	01 0245 8D 1E 00 00	...	LEDATA
9CH	...	C402 02 01 01 00 20H	...	FIXUPP
AOH	...	01 0279 A1 00 00	...	LEDATA
9CH	...	8801 06 01 03	...	FIXUPP
8AH	...	80H	...	MODEND

Fig. 7.7 *MS DOS object modules*

the names FIRST and SECOND through the NAME directive. For simplicity, we show addresses in decimal.

In the object module FIRST, the SEGDEF code of 20H indicates that the segment is byte aligned and relocatable. For both MOV statements involving the use of external symbols, the assembler generates identical code with zeroes in the second operand field. In the first FIXUPP record, 88 in the first byte of *locat* implies *loc code* = '2' indicating that fixing is to be performed by putting the segment base of the external symbol in the generated code. *fix dat* contains the code '6', indicating that an external symbol is involved in the fixup. The next two fields indicate that fixing is to be performed in the first segment of the module, according to the external symbol #1 (i.e., PHI). The second FIXUPP record is similar except that 84 in *locat* implies *loc code* = '1' (offset to be put in the target location) and *target datum* = '2' implying that fixing is to be performed according to external symbol and target displacement. (Note that

code '6' could have been used as discussed in Ex. 7.10). The next two fields indicate that fixing is to be performed in the first segment of the module according to the external symbol #2 (i.e. PSI). In SECOND, the JMP statement is fixed using the code 8C in *locat*, implying *loc code* = '3' which indicates that both segment base and offset are to be put into the generated code. The statement

```
LEA                BX, ALPHA+20H
```

is assumed to have been assembled with zeroes in the operand field. It is fixed by using the code C4 in *locat* (i.e. *loc* = 2) and *code* = 2 in *fix dat*. The displacement 20H appears in *target offset* field of the FIXUPP record.

7.4.1 Design of the Linker

We shall present the design of a program named LINKER which performs both linking and relocation. The output of LINKER is a binary program which resembles a program with .COM extension in MS DOS. It is assumed that the binary program cannot be relocated by the loader. (Note that LINK program of MS DOS produces a program with .EXE extension, which is relocated by the loader prior to execution. The reader would do well to note this difference.)

Specification

The LINKER invocation command has the following format:

```
LINKER   <object module names>, <executable file>,
         <load origin>, <list of library files>
```

LINKER performs relocation and linking of all named object modules to produce a binary program with the specified load origin. The program is stored in *<executable file>*. When LINKER comes across an external name ext_symb_i not defined in any of the object modules named in the LINKER command, it locates an object module om_i in *<list of library files>* which contains ext_symb_i as a public definition. om_i is now included in the set of object modules to be linked and relocated. This process of resolving an external reference by including the object module containing its definition from a library file is called *autolinking*. LINKER execution terminates when all external references have been resolved, or when the unresolved external references cannot be found in any of the library files.

Example 7.12 In the LINKER command

```
LINKER alpha+beta+min, calculate, 10000, pas.lib
```

`alpha`, `beta` and `min` are names of the object modules to be linked and `calculate` is the name to be given to the executable file generated by LINKER. The load origin of the executable program is 10,000. Any external names not defined in object modules `alpha`, `beta` and `min` are to be searched in the library `pas.lib`.

Data structures and algorithm

LINKER uses a two pass strategy. In the first pass, the object modules are processed to collect information concerning segments and public definitions. The second pass performs relocation and linking.

First pass

In the first pass, LINKER only processes the object records relevant for building NTAB.

Algorithm 7.3 (First pass of LINKER)

1. *program_linked_origin* := <*load origin*>; (Use a default value if <*load origin*> is not specified.)
2. Repeat step 3 for each object module to be linked.
3. Select an object module and process its object records.
   (a) If an LNAMES record, enter the names in NAMELIST.
   (b) If a SEGDEF record
      (i) *i* := *name index*; *segment_name* := NAMELIST [i];
      segment_addr := start address in *attributes* (if any);
      (ii) If an absolute segment, enter (*segment_name*, *segment_addr*) in NTAB.
      (iii) If the segment is relocatable and cannot be combined with other segments
         – Align the address contained in *program_linked_origin* on the next word or paragraph as indicated in the *attributes* field.
         – Enter (*segment_name*, *program_linked_origin*) in NTAB.
         – *program_linked_origin* := *program load origin* + *segment length*;
   (c) For each PUBDEF record
      (i) *i* := *base*; *segment_name* := NAMELIST [i];
      symbol := *name*;
      (ii) *segment_addr* := load address of *segment_name* in NTAB;
      (iii) *sym_addr* := *segment_addr* + *offset*;
      (iv) Enter (*symbol*, *sym_addr*) in NTAB.

Example 7.13 For the LINKER command

```
LINKER FIRST+SECOND, demo, 10000, math.lib
```

where object modules `FIRST` and `SECOND` are as shown in Ex. 7.11, NTAB at the end of first pass is as follows:

Symbol	*Load address*
COMPUTE	10000
ALPHA	10015
BETA	10084
PART2	10128
PHI	10161
PSI	10187

Note that PART2 has the load address of 10128 even though the first byte following the segment COMPUTE has the address 10124. This is because PART2 has a paragraph alignment.

Second pass

Second pass of LINKER constructs the executable program in *work_area*. Data from the LEDATA records is moved to appropriate parts of *work_area*. FIXUPP records are then processed to effect relocation and linking. At the end of the pass, the executable program is written into the current directory under the name <*executable file*> specified in the LINKER command.

It is simple to move the data from an LEDATA record into the appropriate part of *work_area* using *segment index* and *data offset*. The segment index is used to obtain the load origin of the segment from NTAB. Adding the data offset to it gives load address of the first byte of the data in the LEDATA record. Address of this byte in *work_area* is now obtained by a computation analogous to step 2(e)(ii) of Algorithm 7.1.

Some complications arise from the fact that *segment index* contains a numeric value rather than the name of a segment. The segment name can be obtained by using this value as an index into LNAMES record(s). This name can be searched in NTAB to obtain its load origin. Similarly, while processing a FIXUPP record, the target datum can be used as an index into EXTDEF or SEGDEF records to obtain the target name. This name can be searched in NTAB to obtain its load origin.

The second pass constructs a number of tables to eliminate indirect references to NTAB. To eliminate search in the LNAMES records, the NAMELIST table can be constructed once again in the second pass, and *segment index* can be used to index NAMELIST. However, the search in NTAB is still needed. To overcome this problem, LINKER builds another table called the *segment table* (SEGTAB) to contain all segment names defined in the object module. A SEGTAB entry has the following format:

segment name	*load address*

where the load address information is copied from NTAB. Now, while processing an LEDATA record, the segment index can be used to index SEGTAB to obtain

the segment's load address. A similar use can be made while processing FIXUPP records.

Since a linking specification in a FIXUPP record may contain a reference to an external symbol, LINKER builds an *external symbols table* (EXTTAB) with a similar entry-format, viz.

external symbol	*load address*

EXTTAB is built by processing the EXTDEF records and copying the load addresses from NTAB. (Actually the *name* fields of the SEGTAB and EXTTAB entries are redundant since access to an entry in these tables is through an index rather than through a name.)

When some external name `alpha` is not found in the set of object modules named in the LINKER command, LINKER performs autolinking. The object module containing `alpha` is included in the set of object modules being linked. To implement this, the first pass of LINKER is performed on the new object module. Processing of the FIXUPP record which triggered off autolinking for `alpha` is then resumed.

Algorithm 7.4 (Second pass of LINKER)

1. *list_of_object_modules* := Object modules named in LINKER command;
2. Repeat step 3 until *list_of_object_modules* is empty.
3. Select an object module and process its object records.
   (a) If an LNAMES record
   Enter the names in NAMELIST.
   (b) If a SEGDEF record
   i := *name index*; *segment_name* := NAMELIST [i];
   Enter (*segment_name*, *load address* from NTAB) in SEGTAB.
   (c) If an EXTDEF record
   (i) *external_name* := name from EXTDEF record;
   (ii) If *external_name* is not found in NTAB, then
   – Locate an object module in the library which contains *external_name* as a segment or public definition.
   – Add name of object module to *list_of_object_modules*.
   – Perform first pass of LINKER, i.e. Algorithm 7.3, for the new object module.
   (iii) Enter (*external_name*, *load address* from NTAB) in EXTTAB.
   (d) If an LEDATA record
   (i) *i* := *segment index*; *d* := *data offset*;
   (ii) *program_load_origin* := SEGTAB [i].*load address*;
   (iii) *address_in_work_area* := address of *work_area* +
   program_load_origin − <*load origin*> + *d*;

(iv) Move data from LEDATA into the memory area starting at the address *address_in_work_area*.

(e) If a FIXUPP record, for each FIXUPP specification

(i) *f* := *offset* from *locat* field;

(ii) *fix_up_address* := *address_in_work_area* + *f*;

(iii) Perform required fix up using a load address from SEGTAB or EXTTAB and the value of code in *locat* and *fix dat*.

(f) If a MODEND record

If start address is specified, compute the corresponding load address (analogous to the computation while processing an LEDATA record) and record it in the executable file being generated.

Example 7.14 Figure 7.8(a) illustrates the contents of the various data structures before object module SECOND is processed by the second pass of LINKER. While linking the object modules of Ex. 7.11 autolinking would be performed for the external name GAMMA of object module SECOND. Let GAMMA be a public definition in object module THIRD. First pass of LINKER is performed for THIRD. Thus, the segments and public symbols defined in THIRD are added to NTAB. This includes PRECISE, a segment name and EMP_DATA, another public definition (see Fig. 7.8(b)). The name THIRD is also added to the list of object modules. This ensures that the second pass would be performed for THIRD.

EXERCISE 7.4

1. Explain the purpose of the *segment index* field in an LEDATA record.
2. Answer problems 1 and 2 of Exercise 7.2 with respect to the Intel 8086 object module.
3. Modify Algorithm 7.4 to incorporate the processing of common and stack segments as follows:
   (a) All common segments with the same name overlap one another.
   (b) All stack segments with the same name are concatenated.
4. After a program has been translated to obtain the object module, it is found that certain *n* instructions from the translated address *aaaa* are redundant. These instructions can be replaced by *no-op* instructions, by introducing an LEDATA record with *aaaa* in its *data offset* field. However, this would leave the program length unchanged and also waste execution time due to the *no-op* instructions. It is therefore required to physically delete these *n* instructions from the object module.
   Indicate what modifications need to be made to implement physical deletion.
5. Suggest an entry format for the symbol table of an assembler which incorporates all information required to generate the SEGDEF, PUBDEF, EXTDEF and FIXUPP records of the object module.
6. Autolinking is a slow and laborious process because it involves library searches for every unresolved address constant. Comment on the effectiveness of the following schemes aimed at speeding up autolinking:

a) After processing FIRST

NTAB

Symbol	*Load address*
COMPUTE	10000
ALPHA	10015
BETA	10084
PART2	10128
PHI	10161
PSI	10187

NAMELIST

Symbol
COMPUTE

SEGTAB

Segment name	*Load address*
COMPUTE	10000

EXTTAB

External symbol	*Load address*
PHI	10161
PSI	10187

b) After processing SECOND

NTAB

Symbol	*Load address*
COMPUTE	10000
ALPHA	10015
BETA	10084
PART2	10128
PHI	10161
PSI	10187
PRECISE	10528
GAMMA	10586
EMP_DATA	10619

NAMELIST

Symbol
PART2

SEGTAB

Segment name	*Load address*
PART2	10128

EXTTAB

External symbol	*Load address*
ALPHA	10015
BETA	10084
GAMMA	10586

Fig. 7.8 LINKER data structures for autolinking

(a) A few unresolved address constants are grouped together for library searching.
(b) The directory entries for the object modules library are extended to include information about all public definitions in an object module.

7. In Section 7.4.1 we have assumed that *work_area* is large enough to accommodate the entire binary program. Comment on the changes required if *work_area* is smaller than the binary program.

7.5 LINKING FOR OVERLAYS

Definition 7.4 (Overlay) *An overlay is a part of a program (or software package) which has the same load origin as some other part(s) of the program.*

Overlays are used to reduce the main memory requirement of a program.

Overlay structured programs

We refer to a program containing overlays as an *overlay structured program*. Such a program consists of

1. A permanently resident portion, called the *root*
2. A set of overlays.

Execution of an overlay structured program proceeds as follows: To start with, the *root* is loaded in memory and given control for the purpose of execution. Other overlays are loaded as and when needed. Note that the loading of an overlay overwrites a previously loaded overlay with the same load origin. This reduces the memory requirement of a program. It also makes it possible to execute programs whose size exceeds the amount of memory which can be allocated to them.

The overlay structure of a program is designed by identifying mutually exclusive modules—that is, modules which do not call each other. Such modules do not need to reside simultaneously in memory. Hence they are located in different overlays with the same load origin.

Example 7.15 Consider a program with 6 sections named `init`, `read`, `trans_a`, `trans_b`, `trans_c`, and `print`. `init` performs some initializations and passes control to `read`. `read` reads one set of data and invokes one of `trans_a`, `trans_b` or `trans_c` depending on the values of the data. `print` is called to print the results.

`trans_a`, `trans_b` and `trans_c` are mutually exclusive. Hence they can be made into separate overlays. `read` and `print` are put in the root of the program since they are needed for each set of data. For simplicity, we put `init` also in the root, though it could be made into an overlay by itself. Figure 7.9 shows the proposed structure of the program. The overlay structured program can execute in 40 K bytes though it has a total size of 65 K bytes. It is possible to overlay parts of `trans_a` against each other by analyzing its logic. This will further reduce the memory requirements of the program.

The overlay structure of an object program is specified in the linker command.

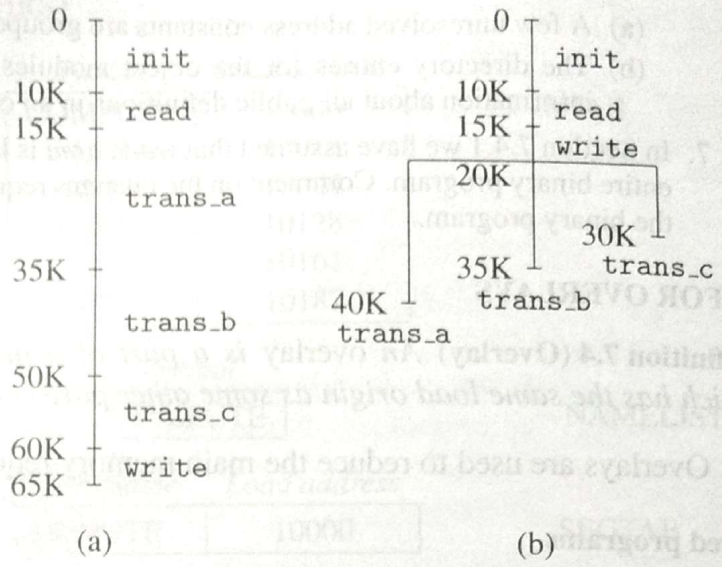

Fig. 7.9 An overlay tree

Example 7.16 The MS-DOS LINK command to implement the overlay structure of Fig. 7.9 is

```
LINK  init + read + write + (trans_a) + (trans_b)
      + (trans_c), <executable file>, <library files>
```

The object module(s) within parenthesis become one overlay of the program. The object modules not included in any overlay become part of the root. LINK produces a single binary program containing all overlays and stores it in *<executable file>*.

Example 7.17 The IBM mainframe linker commands for the overlay structure of Fig. 7.9 are as follows:

```
Phase main:      PHASE    MAIN,+10000
                 INCLUDE  INIT
                 INCLUDE  READ
                 INCLUDE  WRITE
Phase a_trans:   PHASE    A_TRANS,*
                 INCLUDE  TRANS_A
Phase b_trans:   PHASE    B_TRANS,A_TRANS
                 INCLUDE  TRANS_B
Phase c_trans:   PHASE    C_TRANS,A_TRANS
                 INCLUDE  TRANS_C
```

Each overlay forms a binary program (i.e. a *phase*) by itself. A PHASE statement specifies the object program name and linked origin for one overlay. The linked origin can be specified in many different ways. It can be an absolute address specification as in the first PHASE statement. In the second PHASE statement, the '*' indicates contents of the counter *program_linked_origin*. Thus, the linked origin is the next available memory byte. The specifications in the last two PHASE statements indicate the same linked origin as that of A_TRANS.

Execution of an overlay structured program

For linking and execution of an overlay structured program in MS DOS the linker produces a single executable file at the output, which contains two provisions to support overlays. First, an overlay manager module is included in the executable file. This module is responsible for loading the overlays when needed. Second, all calls that cross overlay boundaries are replaced by an interrupt producing instruction. To start with, the overlay manager receives control and loads the root. A procedure call which crosses overlay boundaries leads to an interrupt. This interrupt is processed by the overlay manager and the appropriate overlay is loaded into memory.

When each overlay is structured into a separate binary program, as in IBM mainframe systems, a call which crosses overlay boundaries leads to an interrupt which is attended by the OS kernel. Control is now transferred to the OS loader to load the appropriate binary program.

Changes in LINKER algorithms

The basic change required in LINKER algorithms of Section 7.4.1 is in the assignment of load addresses to segments. *program_load_origin* can be used as before while processing the root portion of a program. The size of the root would decide the load address of the overlays. *program_load_origin* would be initialized to this value while processing every overlay. Another change in the LINKER algorithm would be in the handling of procedure calls that cross overlay boundaries. LINKER has to identify an inter-overlay call and determine the destination overlay. This information must be encoded in the software interrupt instruction.

An open issue in the linking of overlay structured programs is the handling of object modules added during autolinking. Should these object modules be added to the current overlay or to the root of the program? The former has the advantage of cleaner semantics (think of procedures with *static* or *own* data), however it may increase the memory requirement of the program.

EXERCISE 7.5

1. A call matrix (CM) is a square matrix with boolean values in which CM [*i*, *j*] = *true* only if procedure *i* calls procedure *j* (directly or indirectly). Describe how a programmer can use this information to design the overlay structure of a program.
2. Modify the passes of LINKER to perform the linking of an overlay structured program. (*Hint:* Should each overlay have a separate NTAB?)
3. This chapter has discussed *static relocation*. *Dynamic relocation* implies relocation during the execution of a program, e.g. a program executing in one area of memory may be suspended, relocated to execute in another area of memory and resumed there. Discuss how dynamic relocation can be performed.
4. An inter-overlay call in an overlay structured program is expensive from the execution viewpoint. It is therefore proposed that if sufficient memory is available during program execution, all overlays can be loaded into memory so that each inter-overlay call can simply be executed as an inter-segment call. Comment on the feasibility of

this proposal and develop a design for it.

7.6 LOADERS

As described in Section 7.1.1, an absolute loader can only load programs with load origin = linked origin. This can be inconvenient if the load address of a program is likely to be different for different executions of a program. A *relocating loader* performs relocation while loading a program for execution. This permits a program to be executed in different parts of the memory.

Linking and loading in MS DOS

MS DOS operating system supports two object program forms. A file with a .COM extension contains a non relocatable object program whereas a file with a .EXE extension contains a relocatable program. MS DOS contains a program EXE2BIN which converts a .EXE program into a .COM program. The system contains two loaders—an absolute loader and a relocating loader. When the user types a filename with the .COM extension, the absolute loader is invoked to simply load the object program into the memory. When a filename with the .EXE extension is given, the relocating loader relocates the program to the designated load area before passing control to it for execution.

EXERCISE 7.6

1. Compare the .COM files in MS DOS with an object module. (MS DOS object modules are contained in files with .OBJ extension.)
2. It is proposed to perform dynamic linking and loading of the program units constituting an application. Comment on the nature of the information which would have to be maintained during the execution of a program.

BIBLIOGRAPHY

Most literature on loaders and linkage editors is proprietary. Some public domain literature is listed below.

1. Barron, D. W. (1969): *Assemblers and Loaders,* Macdonald Elsevier, London.
2. Presser, L. and J. R. White (1972): "Linkers and Loaders," *Computing Surveys,* **4** (3).
3. Wilder, W. L. (1980): "Comparing load and go and link/load compiler organizations," *Proc. AFIPS NCC,* **49**, 823-825.
4. Schwartz, R. L. (1978): "Parallel compilation: A design and its application to SIMULA-67," *Computer Languages*, **3** (2), 75-94.