Linkers and Loaders

LEON PRESSER AND JOHN R. WHITE

*University of California,**
Santa Barbara, California 93106

This is a tutorial paper on the linking and loading stages of the language transformation process First, loaders are classified and discussed Next, the linking process is treated in terms of the various times at which it may occur (i e , binding to logical space). Finally, the linking and loading functions are explained in detail through a careful examination of their implementation in the IBM System/360 Examples are presented, and a number of possible system trade-offs are pointed out.

Key words and phrases binary loaders, relocating loaders, linking loaders, linkers, compilers, assemblers, relocation, program modularity, libraries

CR categories 4 11, 4.12, 4.39

1. INTRODUCTION

A computer system includes a set of software and hardware facilities which supervises its operation, insures its coordination, and facilitates its use. Such facilities are referred to as the computer's operating system. From a functional viewpoint it is justifiable to separate from the operating system those modules which facilitate the man/computer communication process. This separation comes about since a computer understands its machine language, while it is much more natural for a user to program in a high-level language (e.g., FORTRAN, PL/I). Thus, it is necessary to transform a program written in a high-level language into a properly formatted binary string before it can be executed. In its most basic form this transformation process occurs in two stages (see Figure 1). First, a user's (source) program is translated into machine language. Then, the translated program is stored for immediate or future execution. Storing into main memory is called *loading*. In modern systems the situation is more complex. In order to obtain flexibility and better utilization of main memory, translators are required to generate *relocatable* code, that is, code that can be loaded into any section of main memory for execution Furthermore, the capability to combine subprograms into a composite program, referred to as *linking*, is of great value in modern operating systems.

This paper is intended as a tutorial on linkers and loaders. For a tutorial treatment of translators (e.g., compilers) the reader is referred to [1]. For a tutorial treatment of operating systems the reader is referred to [2, 3].

2. LOADERS

As previously stated, before a source program can be executed it must first be transformed into machine language and then loaded into main memory, if it is not already there. Since the process of loading a translated program into memory is logically distinct from the translation of that program, separate software modules, called *loaders* [4], have been developed to accom-

* Department of Electrical Engineering. This work was supported in part by the National Science Foundation, Grant GJ-31949

CONTENTS

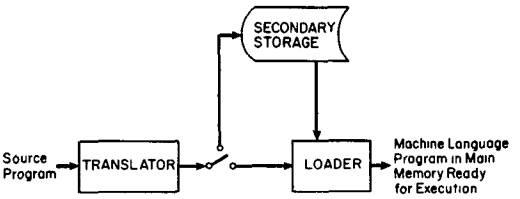

FIG 1. Basic language transformation process.

plish the loading operation. There are two types of loaders: binary loaders, and relocating loaders. Each type of loader can be distinguished by the functions it performs and by the characteristics of the inputs that it processes.

2.1 Binary Loaders

A binary (or absolute) loader, the simplest type of loader, is responsible for loading into main memory a single program in *absolute binary* form. The absolute binary form of a program is simply a binary image of the program as it will exist in memory. A program in this form is associated with specific memory locations; hence, it must always be loaded into the same memory area if it is to execute correctly.

2.2 Relocating Loaders

A program is said to be relocatable if it can be loaded into any section of main memory for execution.* The form of a relocatable program, referred to as *relocatable binary*, is similar to absolute binary except that: 1) address fields have been translated relative to zero; and 2) relocation information is associated with the program to be loaded to indicate which address fields must be relocated. There are two general approaches that have been employed to encode relocation information. In the first approach, the language translator (e.g., compiler, assembler) appends a relocation bit to each machine language instruction produced. The relocation bit is set by the translator only if the address field of the corresponding instruction must be relocated.

* It is impressive to note that John von Neumann was writing relocatable code as early as 1945 [5].

In the second approach, all relocation information is grouped into a relocation table (dictionary) by the translator. The relocation table contains a pointer to each machine language instruction that must have its address field relocated.

It should be noted that in most systems the total number of times an address field is relocated must be zero or one. The reason being that, in the case of the relocation bit approach, the translator is only introducing one additional bit (two states) to indicate "relocation" or "no relocation." The restriction that at most one relocation can be associated with an address field could, if it were meaningful, be changed to "at most n relocations" by expanding the relocation bit from one bit to $\log_2 (n + 1)$ bits. A similar reasoning applies in the case of a relocation table.

The relocating loader is responsible for loading into main memory a program in relocatable binary form and updating (relocating) all relative addresses. Note that when a relocating loader is used, the allocation of memory to a given program will remain bound (i.e., fixed) for the duration of that program's execution.

3. LINKERS

The linking† of subprograms together to form a composite program is of great value in the modular development of software. To place the linking function in perspective, let us view Figure 1 as a function of time. Source program coding must be performed first, translation second, loading third, and execution fourth. Linking, however, could be carried out at seven different times, namely: 1) at source program coding time; 2) after coding but before translation time; 3) at translation time; 4) after translation but before loading time; 5) at loading time; 6) after loading but before execution time; or 7) at execution time.

At this point it is worthwhile to introduce the concepts of physical and logical (virtual, name) address space [6, 7]. The *physical space* consists of the set of main memory locations where information may be stored. The *logical space* consists of the set of identifiers that may be used by a program to reference information. The translation or mapping of a logical into a physical address is called *address binding*.

The linking process may be viewed as binding (combining) independent logical spaces into one composite logical space. In essence, the binding of a set of subprograms in logical space is equivalent to fixing their positions relative to a common base. Let us now discuss the seven cases listed above in more detail.

To link at or before translation time implies a separate translation for each different combination of subprograms. This represents an important drawback. The IBM 1401 Autocoder is an example of a system that performed linkage at translation time. The approach of linking after translation but before loading time is employed in the IBM System/360 (Linkage Editor) and the UNIVAC 9400. In the basic case (refer to Figure 2) the input to the linker consists of one or more subprograms in *binary symbolic* form. (These subprograms can come from secondary storage and/or directly from translation.) This form is similar to relocatable binary except that an additional table (dictionary) is included with each subprogram presented to the linker to indicate the definition and use of external symbols [8]. External symbols are symbols that are declared to be "public" and, as a result, can be referenced by other programs. External symbols are the primary facility through which independently translated subprograms communicate. (External symbols are discussed in detail in section 4.1.1.) It is the responsibility of the linker to combine the input subprograms into a single (relocatable) output module in which all external references have been resolved, if the module is to be loaded for execution. Carrying out the linking process after translation but before loading time (or later) allows the composition of a set of linked subprograms to change without forcing retranslation of the entire collection;

† The linking process has also been called *binding* (Burroughs), *collecting* (UNIVAC), and *building* (IBM 1800)

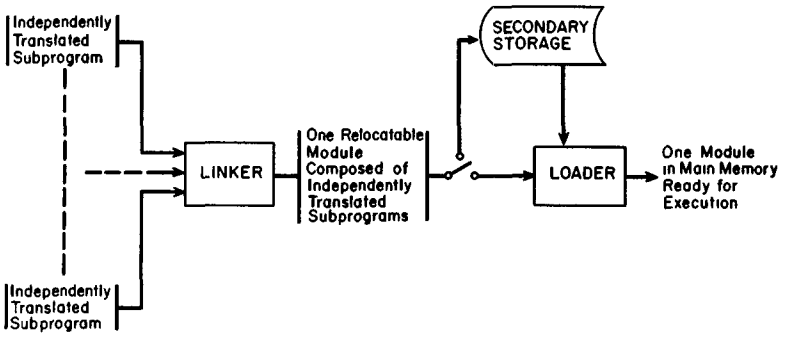

FIG. 2. Linking after translation but before loading.

only linking has to be performed anew. The output of the linker can be supplied to the loader for immediate loading or it can be stored in secondary memory for future linking and/or loading. This alternative provides for a flexible system. Furthermore, the linker represents a natural base for the incorporation of subprogram editing facilities.

Linking could be performed at loading time (i e., *linking loader*) as in many existent systems (e.g., Loader in IBM System/360, XDS-UTS, CDC-SCOPE, and SEL 810B-BOS). The popularity of linking loaders is a result of their simplicity since the loading stage is a natural place to bind subprograms together On the other hand, since linking implies loading, there is less flexibility than when the two functions are implemented as separate modules, as in the case earlier discussed. The input to a linking loader consists of one or more subprograms in binary symbolic form. The output consists of one module in main memory ready for execution. Again, all external references must be resolved before execution. Linking after loading but before execution (case 6) would only be logical if we could keep a very large number of subprograms resident in main memory. In today's systems main memory is a scarce resource; thus, this alternative is not attractive.

Finally, it is possible to link (bind) at execution time. Such an approach is called *segmentation.* A segment is a self-contained logical entity of related information defined and named by the programmer (e.g., subprogram, data array). All intersegment references are achieved through symbolic names that are resolved at execution time. The most general implementaton of this concept is that embodied in the Honeywell 645 MULTICS system (formerly the GE-645). For an excellent discussion of segmentation as well as the MULTICS system the reader is referred to [7]. Briefly, from the linking point of view the principal advantages of segmentation include: the binding of segments to the composite logical space only when required; possible segment growth during execution since segmentation allows the management of logical space; and sharing (of segments) in its most general form since the relative position of a shared segment in one logical space is independent of its position in other logical spaces. The main problems with segmentation include: overhead costs, since execution time binding is less efficient although more flexible than earlier binding; and the importance of an integrated (hardware and software) design.

To treat linking and loading in more detail we discuss the implementation of these functions on the IBM System/360. The System/360 Operating System (OS) provides two alternatives. On one hand there exists a linking loader referred to as *Loader;* on the other hand there exists a sophisticated linker, called the *Linkage Editor,* and a simple relocating loader referred to as *Program Fetch.* The linking power and flexibility of Loader is a subset of that provided by the Linkage Editor; so that only the function and structure of the Linkage Editor and Program Fetch are examined in detail. The case in question is

that outlined in Figure 2. There is an additional facility (LINK) provided in the System/360 OS which allows two separate logical spaces to communicate at execution time through general registers. This will not be discussed here. The objective of the remaining sections of this paper is to illustrate basic implementation techniques and to point out system trade-offs. (Terms that refer to the IBM System/360 are capitalized throughout.)

4. THE LINKAGE EDITOR

With this perspective of the Linkage Editor in mind, we can examine in more detail the functions it performs. The Linkage Editor is responsible for the following functions [9]:
Primary function—(1) Linking together independently translated modules.
Secondary functions—(2) Overlay processing; (3) Program modification; (4) Library access.

The first function listed above is the main responsibility of the Linkage Editor; therefore, the bulk of this discussion centers on the manner in which independently translated modules are linked together. The other three functions represent secondary objectives and, as a result, are less thoroughly considered. Before these functions are discussed further, the inputs to the Linkage Editor should be examined.

The inputs accepted by the Linkage Editor can be divided into two groups [10]: input modules, and Linkage Editor control statements. Input modules are further classified as being either *Object Modules* or *Load Modules*. These two types of modules are similar in structure Object Modules are discussed next, while the structure of Load Modules is examined in Section 4.3. Linkage Editor control statements are covered in Section 4 5.

4.1 Object Modules

The term Object Module refers to the output produced by the (IBM System/360) language translators This output consists of the machine language code for the translated program, relocation information, and a table indicating the definition and use of external symbols. As a result, Object Modules correspond to the binary symbolic form of a program that was discussed in Section 3. Each module (see Figure 3) is divided into the following four sections [9]: External Symbol Dictionary, Text, Relocation Dictionary, and End Record.

4.1.1 External Symbol Dictionary (ESD)

The External Symbol Dictionary (ESD) is a table that contains an entry for each external symbol defined within the module [9]. As mentioned earlier, external symbols are the facility by which independently translated programs communicate An external symbol is classified as representing either an *external name* or an *external reference* [11].

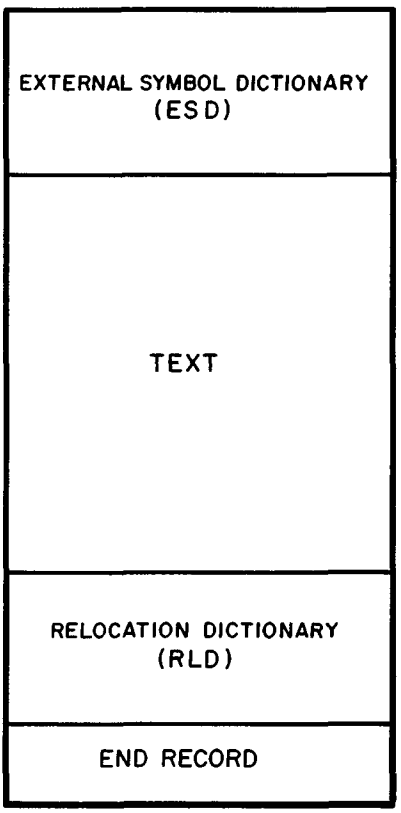

FIG 3 Object Module format

External Name

A symbol within a module is said to be an external name if that symbol can be referenced by other modules that were independently translated and are being linked together with the module containing the external name. Within the framework of the IBM System/360 there are two types of external names: Control Section names, and Entry Point names [10].

Control Section Name

A program in the System/360 is made up of one or more Control Sections. Each Control Section is a unit of coding (instructions and/or data) that is considered to be an entity. While all elements of a single Control Section are loaded and executed in a constant relationship with each other, an individual Control Section can be relocated independently of other Control Sections at load time without altering the operating logic of the program [11]. Note that sectioning allows independently coded subprograms to be translated together, thus producing a single Object Module, whereas a linker allows independently translated programs to be combined into a single Load Module.

In the System/360 Assembly Language there are three pseudo-operations (instructions to the Assembler) for identifying the beginning of a Control Section [11]:

1) CSECT—identify Control Section;
2) START—identify Control Section and specify initial location counter value; and
3) COM—define Blank Common Control Section.

A name can be associated with any of these pseudo-operations, and the corresponding Control Section is considered to be a named Control Section. The Control Section name, being external, can be referenced by other modules. An External Symbol Dictionary entry is created by the Assembler for each Control Section. Note that the beginning of unnamed Control Sections cannot be referenced by other modules since there is no external name associated with the Control Section.

Entry Point Name

Since a Control Section name is an external symbol, another independently translated module can reference the beginning of any named Control Section. It is often desirable, however, to be able to reference a particular point within a Control Section. This can be accomplished by declaring the symbol to be an external name at the desired reference point. A symbol declared to be external for the above purpose is referred to as an Entry Point name In the System/360 Assembly Language the ENTRY pseudo-operation is used to identify those labels which are to be considered Entry Points [11]. The Assembler creates an ESD entry each time an ENTRY pseudo-operation is found.

External Reference

The term external reference refers to a symbol that is defined as an external name in another independently translated module but is referred to in the current module [10]. To insure correct assembly the symbol being referenced must be identified as an external symbol. In the System/360 Assembly Language this is accomplished, in general, with either the EXTRN pseudo-operation or a V-type Address Constant. (Address Constants are discussed in Section 4.1.3.) Either of these causes the Assembler to create an ESD entry for the external reference

ESD Entries

Each entry in the External Symbol Dictionary has a type assigned to it that indicates its function. There are six possible ESD types, however, for purposes of this discussion it is sufficient to limit our attention to the following five types [9]:

1) Section Definition (SD). This ESD entry represents the beginning of a named Control Section. As shown in Figure 4(a), the entry specifies the Control Section name, the fact that this entry represents a named Control Section, the assembled origin of the Control Section, and the length of the Control Section.

2) *Private Code (PC).* This ESD entry represents the beginning of an unnamed Control Section. The format of the entry, as shown in Figure 4(b), is similar to an SD type entry except that the Name field is blank Note that since the Control Section is unnamed, the beginning of the Control Section cannot be referenced by other modules.

3) *Label Definition (LD).* This ESD entry represents an Entry Point name. Figure 4(c) shows that the entry contains the name of the Entry Point, the type, the address of the Entry Point relative to the start of the input module, and a pointer (called an ESD ID) to the ESD entry for the Control Section that contains the Entry Point.

4) *Common (CM).* This ESD entry represents a Common area and specifies the name and length of the area. (See Figure 4(d).) The Assembled Origin field is undefined since space for the Common area is not created during translation. One Common area in the output module will be allocated by the Linkage Editor; thus, the value of this field is set at link time. The length of this area will equal the length of the largest Common area contained in the inputs.

5) *External Reference (ER).* This type of ESD entry represents the occurrence of an external reference. As shown in Figure 4(e), the entry need only specify the referenced symbol and the fact that the entry corresponds to an external reference

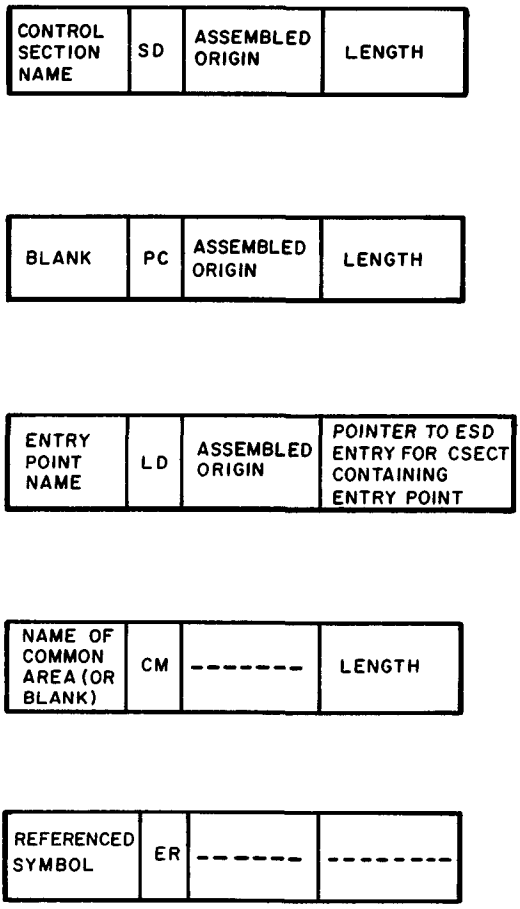

FIG 4 Format of ESD entries (a) Section definition. (b) Private code (c) Label definition (d). Common area (e) External reference.

4.1.2 Text

The Text portion of an Object Module is straightforward It contains the relocatable machine language instructions and data that were produced during translation.

4.1.3 Relocation Dictionary

The Relocation Dictionary (table) contains one entry for each address that must be relocated when the module is loaded into main memory. The number of relocatable addresses and, as a result, the amount of information that must be contained in a relocation table is a function of the machine (addressing) architecture. To illustrate this point consider for a moment a hypothetical computer with an addressing mechanism which functions such that the effective address (i.e., the actual memory location that is accessed) is taken to be the contents of the memory address field of the instruction. With such an architecture, the address field of the machine language instructions will contain the effective address of the cell to be referenced. As a result, *all* the instructions that reference memory must have their address fields modified when a program for this computer is relocated in mem-

ory. Consequently, the relocation table is of maximum length.

In the System/360, the size of the relocation table is greatly reduced because the system architecture utilizes a base register approach in calculating the effective address. The effective address is formed by *always* adding the contents of a base register to the contents of the instruction memory address field. (*When* indexing is specified, the contents of an additional index register is added in when forming the effective address.) Therefore, the address portion of machine language instructions that reference memory can be represented by the ordered pair: (Base Register, Displacement). The first element of the pair indicates which one of the 16 general-purpose registers is being used as a base register; the second element is a displacement (in bytes) from the address contained in the base register. The hardware calculates the effective memory address at execution time by adding the displacement field to the contents of the appropriate base register (and, if specified, the contents of another general-purpose register, an index, is also added). It is the responsibility of the language translator to place the proper base register and displacement in the address portion of the machine language instructions being generated. In the System/360 Assembly Language, the USING pseudo-operation exists to inform the Assembler: 1) which of the sixteen general registers is to be used as a base register, and 2) the relative address that will be in the base register at execution time [11] These two pieces of information enable the Assembler to correctly build the address portion of each memory reference instruction. When writing in 360 Assembly Language the programmer is responsible for including instructions in his program that at execution time will load the base registers with the appropriate addresses. As a result of this organization, the address fields of machine language instructions in the System/360 do not have to be modified when a module is loaded. In effect, the necessary relocation occurs at execution time when the hardware adds the displacement to the contents of the base register to obtain the effective memory address.

This discussion illustrates some important trade-offs that exist at system design time. On the one hand there is the trade-off between hardware and software as far as contributions to the relocation function are concerned (e.g., base registers). On the other hand there is the the trade-off between various software modules. For example, the loader in the System/360 has shifted some of its responsibilities to the language translators which now have to output addresses in a base plus displacement format. Moreover, note that with a base register approach the binding of a memory address is not completed until the last possible moment—execution time.

In the System/360 approach the only parts of the Text that require relocation are those entries that represent *Address Constants*. It is sufficient for us to think of an Address Constant as simply a cell that will contain an absolute memory address at execution time. Address Constants are used primarily for: 1) initializing base registers, and 2) communicating between Control Sections.

In discussing Address Constants, we must distinguish between: 1) the cell (Text entry) that contains the constant, and 2) the value of the constant (i.e., the contents of the cell).

In the 360 Assembly Language, Address Constants are normally established with a DC (Define Constant) pseudo-instruction. For example,

```
JW      DC      A(LP)
```

defines a cell labeled *JW* which at execution time will contain the actual memory address of the cell labeled *LP*. The Text entry that contains the Address Constant cannot be completed at translation time since the address of the symbol specified in the reference field of the DC instruction will not be known until the corresponding module is loaded Therefore, Address Constants must be completed (relocated) by the loader.

There are two principal kinds of Address Constants that require relocation: *A-type* and *V-type* [9].

A-type Address Constants are defined with:

DC A(SYMBOL)

where SYMBOL is either 1) a name local to the module containing the DC pseudo, or 2) a name that has been declared external by the use of the EXTRN pseudo-instruction. In the first case, where the Address Constant represents a local reference, the Assembler sets the value of the constant equal to the relative address of SYMBOL. In the second case, however, the value of the constant is set to zero since the Assembler has no knowledge of the relative address of the specified symbol.

V-type Address Constants are defined with:

DC V(SYMBOL)

where SYMBOL is assumed to be an external reference. As in the second case above, the value of the constant is set to zero by the Assembler.

Since Address Constants are the only part of the Text that require relocation, the Relocation Dictionary contains an entry for each Address Constant in the program. The format of a Relocation Dictionary (RLD) entry is shown in Figure 5. Each entry contains the following four fields [9]:

1) *Relocation Pointer* (*R*). A pointer to the ESD entry for the external symbol on which the value of the Address Constant depends. If the Address Constant is an A-type that does not depend on an external symbol, then the R pointer is set to point to the ESD entry for the Control Section which contains the Address Constant.
2) *Position Pointer* (*P*). A pointer to the ESD entry for the Control Section containing the Address Constant.
3) *Flag*. A type indicator that, among other things, specifies whether the RLD entry represents an A-type or V-type Address Constant.
4) *Address*. The displacement (in bytes) from the start of the Text to the Address Constant.

The RLD entries are used by the relocating loader to relocate the corresponding Address Constants when the module is loaded. This relocation occurs prior to execution and is referred to as *static relocation* since it remains fixed for the duration of program execution. As was pointed out earlier, the address portion of memory reference instructions is bound at execution time (*dynamic relocation*) when the hardware adds the displacement field to the contents of the appropriate base register. Therefore, the running of a program in the System/360 involves both static and dynamic relocation. The software provided by IBM for the System/360 discards the Relocation Dictionary after loading the module; so it is not possible to move a program to another place in memory once it has been loaded.

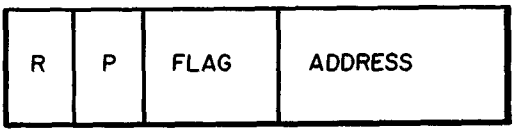

FIG 5. RLD entry format

It is interesting to note that with a somewhat different system strategy, Address Constants are not needed. For example, if a single base register is used, and if the length (in bits) of the displacement field is sufficient to allow all of main memory to be accessed (e.g., in the System/360, 24 bits instead of 12—with implicit specification of the base register), then all memory references could be done with a base plus displacement format without the need for Address Constants. For a local reference, the displacement field would be set by the translator; but the displacement field of external references would be set by the linker. This approach is essentially the one employed in the CDC 6400. There are definite system trade-offs:

1) In the single base register case, each instruction that references memory requires additional bits; however, once a program has been loaded it can be easily relocated in physical space at any time during execution since there are no Address Constants.
2) Multiple base registers under user control require that the user specify (e.g.,

USING) which base register is to be used.
3) Multiple base registers facilitate the sharing of programs and allow the splitting of programs for loading into noncontiguous areas of main memory.

In addition to the static relocation of Address Constants there are other features of the System/360 architecture that prevent the dynamic relocation of programs Among these are the fact that working registers can contain absolute addresses (e.g , return addresses from Branch-And-Link instructions), and the fact that working and base registers are drawn from the same register set.

4 1 4 End Record

The End Record indicates to the Linkage Editor that the end of the Object Module has been reached.

With the structure of Object Modules in mind (Figure 3), we can now discuss in more detail the primary Linkage Editor function of linking together one or more Object Modules.

4.2 Linking Together a Set of Modules

In linking together a set of modules, the Linkage Editor is primarily responsible for [9]: 1) assigning addresses; 2) relocating Address Constants; and 3) creating an output module (called a Load Module).

4.2.1 Assigning Addresses

Each input Object Module may consist of one or more Control Sections To produce a single loadable module, the Linkage Editor assigns consecutive relative addresses to each Control Section encountered. This is done by assigning an address of zero to the first Control Section and then assigning addresses relative to this origin to all other Control Sections. During this process the External Symbol Dictionaries of the input modules are merged together to form a Composite External Symbol Dictionary (CESD). The Assembled Origin fields of all SD, PC, and LD type entries are updated to reflect the new addresses that were assigned.

4.2.2 Relocating Address Constants

Once contiguous addresses have been assigned to the Control Sections in the input modules, all A-type and V-type Address Constants must be relocated relative to the beginning of the output module being created. This relocation is accomplished in the following manner [9]:
1) Every entry in the RLD for each individual input module is read. The R and P pointers are updated to point to the correct CESD entry. The Address field is updated by adding to it the contents of the Assembled Origin field of the CESD entry pointed to by the new P pointer. The Flag field is examined to determine the type of Address Constant represented by the RLD entry.
2) If the RLD entry represents a V-type Address Constant, then the constant corresponds to an external reference. This means that the symbol referenced is not defined in the input module containing the Address Constant, but is defined (it is hoped) in one of the other input modules being linked together In the ESD of the input module the external reference is represented by an entry with Type set to ER. When the Composite External Symbol Dictionary is formed during the first step of linking, each ER type entry should be matched by an SD, LD, or CM type entry that has the same name field. External references that are matched in this way are said to be *resolved* since the referenced symbol corresponds to either a Control Section Name (SD type entry), an Entry Point Name (LD type entry), or a Common area (CM type entry). Only one entry (the SD, LD, or CM entry) is retained in the CESD. On the other hand, if there is no matching SD, LD, or CM entry, the ER entry is placed in the CESD and the external reference is said to be *unresolved.* Relocation of a V-type Address Constant is accomplished in the following manner. The constant is accessed through the Address field of the

RLD entry. The R pointer is used to index the CESD to find the entry on which the value of the constant depends. If the Type field of the entry accessed is either SD, LD, or CM, the external reference has been resolved; and relocation is effected by setting the value of the constant equal to the contents of the Assembled Origin field of the CESD entry. If, however, the Type field of the CESD entry is ER, the external reference is unresolved and the module must be flagged as not executable

3) If the RLD entry represents an A-type Address Constant, the constant can correspond to either a local reference or an external reference. If the Address Constant corresponds to a local reference (i.e., the symbol referenced is defined in the module containing the Address Constant), relocation is accomplished in the following manner. First, the cell containing the constant is accessed through the Address field of the RLD entry. Then the value of the constant is updated (relocated) by adding to it the contents of the Assembled Origin field of the CESD entry pointed to by the R (relocation) pointer If the A-type Address Constant represents an external reference, the constant is again accessed through the Address field of the RLD entry. If the Type field of the CESD entry pointed to by the R pointer is ER, then the Address Constant corresponds to an unresolved external reference, and the module must be marked as not executable. On the other hand, if the Type field of the CESD entry is either SD, LD, or CM, the external reference has been resolved. Relocation is then effected by adding to the value of the constant the Assembled Origin field of the CESD entry.

4.2.3 Creating an Output Load Module

As a result of performing these two functions (assigning addresses and relocating Address Constants) the Linkage Editor produces a single output module that represents a concatenation of the input modules processed. This output module is called a Load Module; its format is discussed below.

4.3 Load Modules

As earlier discussed, Load Modules can also appear as inputs to the Linkage Editor. The possible reprocessing of a Load Module by the Linkage Editor requires that the structure of a Load Module be similar to that of an Object Module. The general format of a Load Module is shown in Figure 6.

The first portion of the Load Module contains the Composite External Symbol Dictionary. This table represents a combination of the ESDs of the individual input modules, as has already been discussed. The CESD is followed by a sequence of Text and RLD information; each Text/RLD pair corresponds to the Text portion and Reloca-

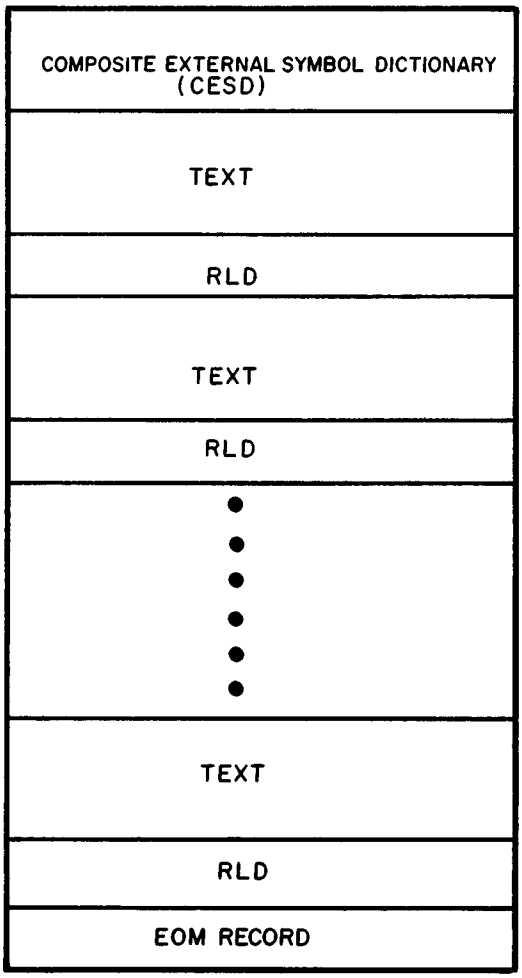

FIG 6. Load Module format

tion Dictionary of an input module. The end of the Load Module is indicated by an EOM (End-Of-Module) record.

At this point let us discuss an example in detail.

4.4 Linking Example

In this example two Object Modules, each containing one Control Section, are linked together to form one output Load Module. The format of the first module is shown in Figure 7.

Object Module One has one named Control Section (CSECT A), one Entry Point (the statement labeled BILL), and one V-type Address Constant (DC V(B)). As shown in the External Symbol Dictionary, there are three external symbols: A (a Control Section name), BILL (an Entry Point), and B (an external reference). There is one entry in the RLD for the single (relocatable) Address Constant present in Object Module One. The entry indicates that the value of the constant depends on the address of the external symbol B and that the constant is defined in Control Section A at relative byte location 300. The value of the constant has been set to zero by the Assembler.

The format of the second input module is shown in Figure 8. There are two entries in the External Symbol Dictionary, one for the Control Section name B and one for the external reference BILL. As indicated in the Relocation Dictionary, there are two Address Constants, DC A(JOE) which is a local reference, and DC V(BILL) which is an external reference. The R and P pointers in the RLD entry for the local reference, DC A(JOE), are the same since

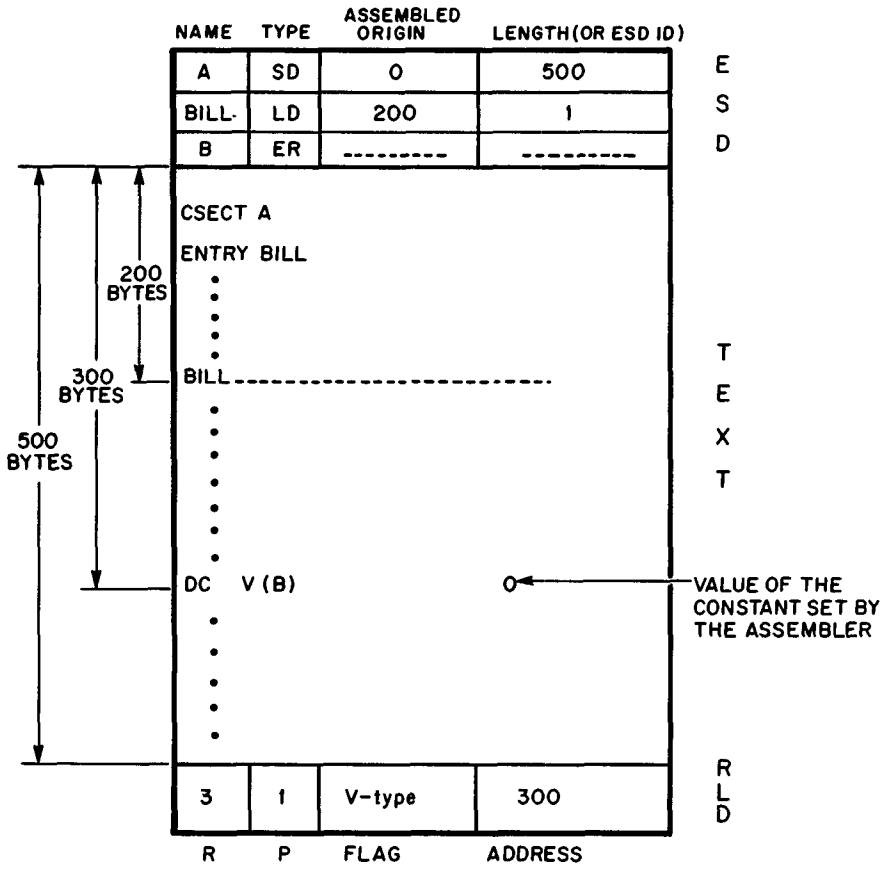

FIG. 7. Object Module One

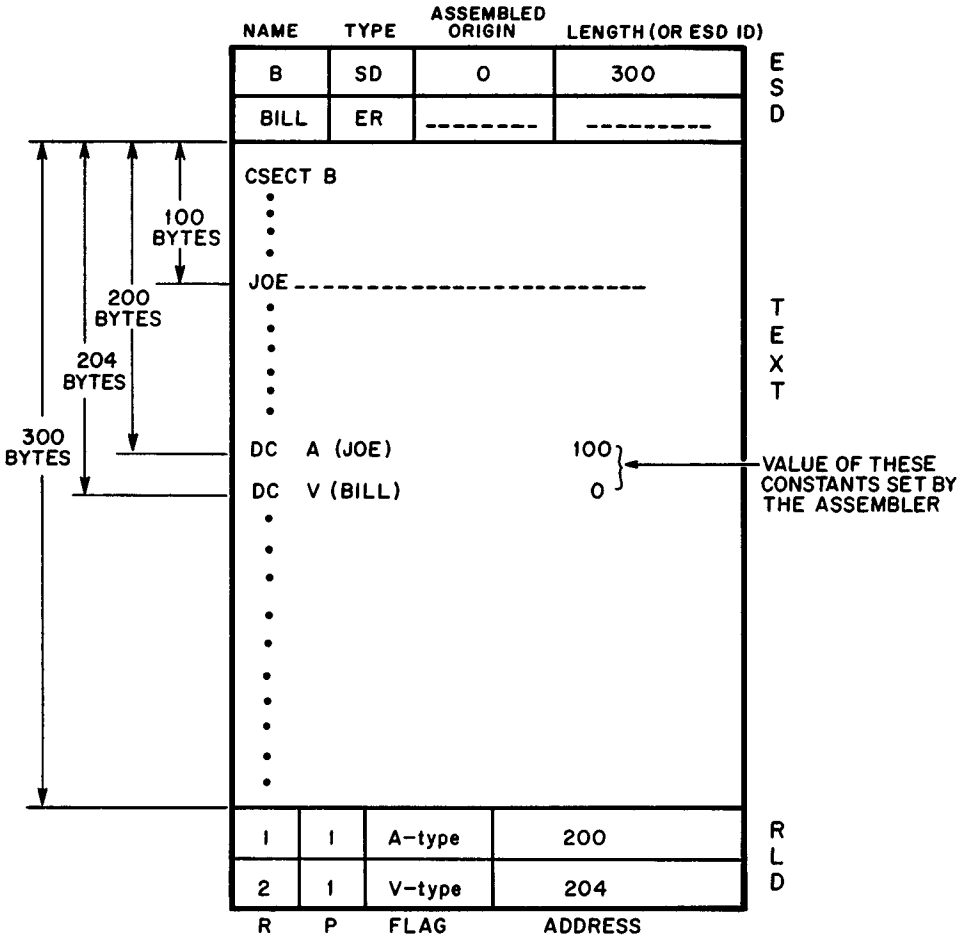

FIG. 8. Object Module Two

the Address Constant is contained in Control Section B, and the value of the constant depends on the address assigned to Control Section B.

As said, it is the responsibility of the language translator (e.g., compiler, assembler) to create the Object Module in the format defined. Thus, Object Module One and Object Module Two would have been produced by two previous and independent translation processes.

Processing by the Linkage Editor yields one output Load Module, the format of which is shown in Figure 9. As indicated in the Assembled Origin field of the Composite External Symbol Dictionary, Control Section B had been assigned an address relative to Control Section A. The R and P fields in the Relocation Dictionaries have been updated to point to the correct CESD entries, and the Address fields have been changed to reflect the new addresses assigned. The three Address Constants have been relocated relative to the start of the module, and the two constants that represented external references have been resolved.

Having discussed the primary function of the Linkage Editor (linking together independently translated modules), and having also described one class of Linkage Editor inputs (input modules), let us now briefly discuss the secondary functions of the Linkage Editor and the other class of Linkage Editor inputs (control statements).

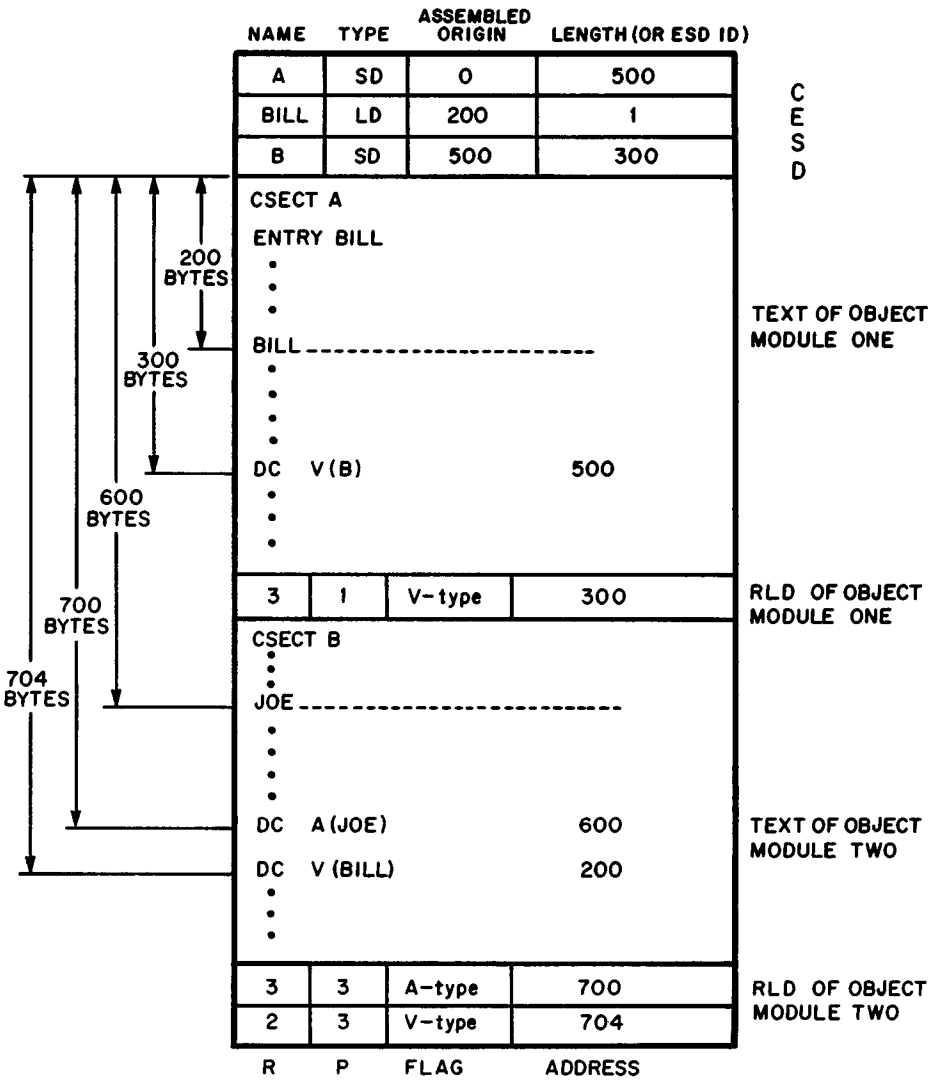

FIG 9. Format of output Load Module

4.5 Linkage Editor Control Statements

In general, Linkage Editor control statements modify or augment the processing performed by the Linkage Editor in its primary function of linking modules. Control statements specify to the Linkage Editor which of the secondary functions (Overlay Processing, Program Modification, or Library Access) are to be performed. To convey the flavor of what takes place, some key Linkage Editor control statements will be discussed as they relate to the appropriate secondary function. Since we are more interested in concepts than in details we will not concern ourselves with the syntax of control statements

4.5.1 Overlay Processing

The tendency in modern computer architecture is toward systems with hardware-aided (main) memory management (e.g., paging) [6]. As a result of these hardware facilities the software effort necessary to implement a dynamic (i.e., execution time)

memory management scheme is sizeably reduced. However, even though such memory management can proceed completely transparent to the programmer, it may be inefficient without his cooperation. Other than base registers, the System/360 (except for models 67, 85, and 195) does not provide hardware features to perform dynamic memory management. Rather, the software incorporated into the operating system to manage memory is extensive and complex. The Linkage Editor through a facility referred to as *Overlay* allows a programmer the option of specifying certain dynamic memory management. In essence, the programmer points out to the Linkage Editor, via control statements, those Control Sections in his program that need not reside in main memory at the same time. Based on this information the Linkage Editor structures the module it outputs so that at execution time an Overlay Supervisor (part of the operating system) will be able to overlay Control Sections.

Overlay Structure

A program in overlay form consists of a set of Segments, each of which is composed of one or more Control Sections [10]. The overlay structure of a program can be represented by a tree, as the example in Figure 10 indicates. The Root Segment (Segment 1) contains all Control Sections that must remain in main memory throughout execution of the program. Segments that lie in a path are logically related; when control is passed to a Segment, all Segments in the path between the Root and the Segment in question are loaded into main memory if not already there For example, when control is passed to Segment 4, the Overlay Supervisor must insure that both Segment 4 and Segment 2 are in main memory. Segments that lie on the same level are not logically related and, thus, can overlay each other (e.g., Segments 2 and 3 in Figure 10).

Once the programmer has designed the overlay structure of his program, he must indicate that structure to the Linkage Editor, this is accomplished with the Overlay Linkage Editor control statement. Each

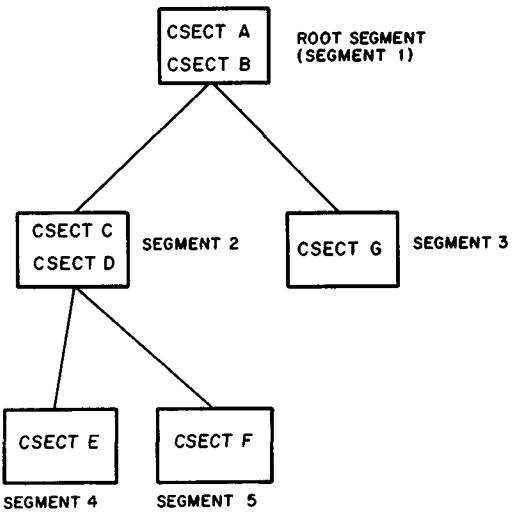

FIG 10 Example of an Overlay Structure in tree form

Overlay statement specifies 1) a set of Control Sections that are to be grouped into a Segment, and 2) the relationship of that segment to other Segments. The Linkage Editor structures this information for the Overlay Supervisor. In fact, this information becomes part of the Linkage Editor output module; the module is termed an Overlay Load Module.

4.5.2 Program Modification

During Linkage Editor processing the user can edit (thus the name Linkage *Editor*) his input modules on a Control Section basis. This makes it possible to modify a Control Section in an Object or Load Module without retranslating the entire source program [10] Two Linkage Editor control statements that facilitate program modification are *Replace* and *Change*.

The Replace control statement is employed to specify one of the following:

1) the replacement of one Control Section with another;
2) the deletion of a Control Section; or
3) the deletion of an Entry Point name.

The Change control statement allows the programmer to change an external symbol. The symbol to be changed can be a Control

Section name, an Entry Point name, or an external reference.

4.5.3 Library Access

It is possible for the Linkage Editor to obtain input modules from sources other than its primary input. The Linkage Editor incorporates such modules either automatically or upon request [10]

Automatic Library Call

If, after linking together a set of modules, the Linkage Editor detects any unresolved external references, it automatically searches a specified library—the Call Library—in an attempt to resolve these external references All such references must be resolved before a Load Module can be executed. The Call Library (e g., FORTRAN library) is specified through a job control language statement. With the *Library* Linkage Editor control statement it is possible to:

1) instruct the Linkage Editor to search a library other than the Call Library for the resolution of specific unresolved external references. The control statement indicates both the library and the specific external references that are to be resolved by a search of that library.
2) indicate those unresolved external references for which no search of the Call Library is to be performed during this run of the Linkage Editor.

These facilities allow a programmer to translate, link, and check out his code before it is complete The incomplete module may contain references to modules that will be incorporated at a later time

Requested Library Call

The Linkage Editor control statement *Include* allows a user to request that a specific module (from some specified file) be included in the Load Module being produced.

4.6 Diagnostics

As previously discussed, the principal output of the Linkage Editor is a Load Module which is placed in a file specified through the job control language. In addition, the Linkage Editor outputs diagnostic information which is also placed in a file specified through the job control language. The diagnostic information consists of three parts. The first part, which is always output, indicates options (e.g., overlay) and attributes (e.g., re-entrant) valid for the Load Module, as well as messages describing the handling of the Load Module (e.g., Module Has Become Not Executable). The second part of the diagnostic information, which may or may not exist, consists of error/warning messages. The third part contains additional diagnostic information requested at the user's option. This optional output includes a listing of all Linkage Editor control statements, a module map of the Load Module (indicating such facts as the origin and length of each Control Section in the Load Module, the point of definition of each Entry Point name in each Control Section, those Control Sections obtained from Automatic Library Call, etc.), and a cross-reference table (listing the cross-references between the Control Sections in the Load Module).

5. THE RELOCATING LOADER

The relocating loader, a portion of the Control Program that is always resident in main memory, is functionally much simpler than the Linkage Editor. Basically, with a single Load Module as input, the functions of the relocating loader are to acquire sufficient space in main memory for the Load Module, to load the module into main memory, and to update (relocate) all Address Constants in the module.

5.1 Requesting Main Memory

The relocating loader requests from the Control Program the main memory necessary to load the module [12]. If the Control Program cannot satisfy the storage request, either 1) the program that called the loader is terminated, or 2) an operation (called *Rollout*) is initiated in which the Control

Program must find another job in the system (i.e., a program in memory other than the one that has just requested main storage) and write all the memory allocated to that job (its *Region*) onto secondary storage. The space occupied by the rolled out job is then made available to the requesting program

Once the request for memory is satisfied, the appropriate amount of main storage is allocated. This storage will be used to hold the Text of the module being loaded

5.2 Loading and Relocating the Text

For each Text/RLD pair in the Load Module (see Figure 6) the following actions are performed [12].

1) The Text is read into the next available section of the memory allocated.
2) The RLD (Relocation Dictionary) is read into a buffer in the relocating loader's work area.
3) Each Address Constant in the Text just loaded is updated in the following manner. The cell that represents the corresponding Address Constant is accessed from the Address field (a pointer) in the RLD entry; the starting Text address is then added to the contents of the Address Constant.

These steps are repeated until an End-Of-Module (EOM) indication is found. At that point, the Load Module is in main memory ready for execution.

5.3 Loading Example

At this point, the program associated with the Load Module in the previous example (Figure 9) is loaded into main memory The relocating loader starts by requesting 800 bytes of main memory from the Control Program (the amount of storage required by the Load Module) Assume that the request is satisfied and that the storage allocated starts at absolute address 2000 (Figure 11(a)) The relocating loader then reads the first section of Text (CSECT A) into memory, starting at location 2000 The RLD for this Text is read into the loader's work area, and the Address Constant in CSECT A is updated (relocated) by adding the starting Text address (2000) to the value of the constant (Figure 11(b)). The Text for CSECT B is then read into the next available section of memory (location 2500), and the two Address Constants are relocated (Figure 11(c)).

It should be noted that the relocating loader does not reference the Composite External Symbol Dictionary of the module being loaded As mentioned earlier, the CESDs are retained to allow the reprocessing of Load Modules by the Linkage Editor. Actually, for purposes of relocation it is only the address field of the RLD entries that is of interest.

Thus, in essence, the relocatable binary form of a program in the IBM System/360 consists of the Text and RLD portions of the Load Module. As previously described, when a program is loaded into memory, the relocation information (RLD) employed to load (relocate) it is discarded Therefore, a program that has been rolled out to secondary storage cannot be brought back into main memory (*Rollin* operation) until the space that it previously occupied is made available; this represents a serious disadvantage. (Other reasons that require a rolled out program to be returned—rolled in —to the exact space from which it was removed are mentioned in Section 4 1.3)

6. SUMMARY

In this paper we have discussed the linking and loading functions, and the implementation of linkers and relocating loaders. In so doing, we have placed in perspective the fact that the language processing responsibility of an operating system extends beyond translation (e g , compilation), and that the translators are strongly influenced by the environment in which they function. A number of possible system trade-offs have been pointed out. For example, in the System/360 (software and hardware) architecture the work of the relocating loader is rather simple since: machine addressing follows the base plus displacement form;

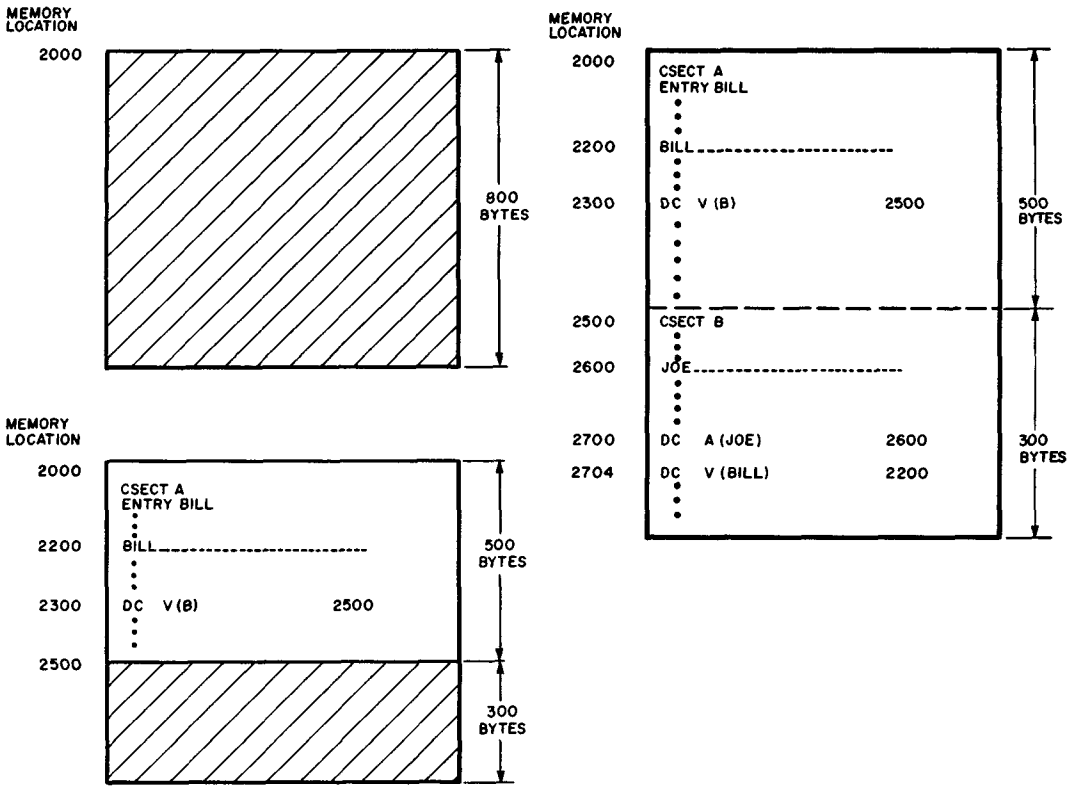

FIG 11 Loading a module (a). Storage allocated by Control Program (b). First section of Text loaded and relocated (c). Module in memory ready for execution.

the linking together of independently translated programs is the responsibility of the Linkage Editor; and a major part of the language processing burden is on the language translators whose responsibility is not only to translate source programs into a form which is very close to machine language, but also to format addresses in a base plus displacement form and to create the Object Module.

Another important trade-off involves binding time [2]. If the various stages of the language transformation process are viewed as a function of time, it is generally true that early binding allows more efficient implementations, while late binding facilitates program debugging and modification.

The high cost of such features as elaborate editing capabilities and overlay processing, which produce a powerful and sophisticated linker like IBM's Linkage Editor, presents a question of practicality in a great many computer center environments. This point is substantiated by the existence of IBM's simple loader which supposedly reduces editing and loading time by about one half [10].

In conclusion, it is our opinion that the flexibility provided by simple linkers and relocating loaders has a definite place in modern operating systems.

7. ACKNOWLEDGMENTS

We wish to make it clear that our description of the IBM System/360 modules is based on the manuals listed in the references, as well as on our experience with the system. The information presented here is correct to the best of our knowledge.

We are grateful to the reviewers and to

Ed Balkovich, Willy Chiu, Don Dumont, Rex Kerley, Dick Mandell, and Ed Prichard for many helpful comments.

REFERENCES

1 PRESSER, L. "The translation of programming languages" In *Computer science,* CARDENAS, PRESSER, AND MARIN (Eds.), John Wiley & Sons, New York, 1972

2 BRADEN, R. "Operating systems" In *Computer science,* CARDENAS, PRESSER, AND MARIN (Eds.), John Wiley & Sons, New York, 1972.

3 BARRON, D. W. *Computer operating systems* Chapman and Hall, London, 1971

4 BARRON, D. W. *Assemblers and loaders* American Elsevier, New York, 1969

5 KNUTH, D. E. "Von Neumann's first computer program" *Computing Surveys* **2,** 4 (Dec. 1970), 247–260

6 DENNING, P. J. "Virtual memory" *Computing Surveys* **2,** 3 (Sept. 1970), 153–189

7 WATSON, R. W. *Time-sharing system design concepts.* McGraw-Hill, New York, 1970

8 MCCARTHY, J., CORBATO, F. J.; AND DAGGETT, M. M. "The linking segment subprogram language and linking loader" *Comm. ACM* **6,** 7 (July 1963), 391–395

9 IBM System/360 operating sysem linkage editor program logic manual IBM Form No. Y28-6667-0

10 IBM System/360 operating system linkage editor and loader IBM Form No. C28-6538-8

11 IBM System/360 operating system assembler language IBM Form No. C28-6514-5.

12 IBM System/360 operating system MVT supervisor program logic manual IBM Form No. GY28-6659-4