GC26-3813-5
File No. S370-31

Systems

OS/VS Linkage Editor and Loader

VS1 Release 6
VS2 Release 3.7

Includes Selectable Unit:
OS/VS1 Subsystem Attachment Support (5741-606)

IBM

Includes Selectable Unit:
OS/VS1 Subsystem Attachment Support (5741-606)

Sixth Edition (August 1978)

This edition replaces the previous edition (numbered GC26-3813-4) and incorporates technical newsletter GN26-0827 and System Supplement Newsletter GC26-3888 and makes them obsolete.

This edition applies to both Release 6 of OS/VS1 and to Release 3.7 of OS/VS2, and to any subsequent releases of either system unless otherwise indicated in new editions or technical newsletters.

Significant system changes are summarized under "OS/VS1 Summary of Amendments" or "OS/VS2 Summary of Amendments" following the list of figures. In addition, miscellaneous editorial and technical changes have been made throughout the publication. Each technical change is marked by a vertical line to the left of the change.

Information in this publication is subject to significant change. Any such changes will be published in new editions or technical newsletters. Before using the publication, consult the latest *IBM System/370 Bibliography,* GC20-0001, and the technical newsletters that amend the bibliography, to learn which editions and technical newsletters are applicable and current.

Requests for copies of IBM publications should be made to the IBM branch office that serves you.

Forms for readers' comments are provided at the back of the publication. If the forms have been removed, comments may be addressed to IBM Corporation, P. O. Box 50020, Programming Publishing, San Jose, California, U.S.A. 95150. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

PREFACE

This publication provides application programmers with the information necessary to use the OS/VS Linkage Editor and Loader to prepare the output of a language translator for execution. Additional information on the operation and use of the linkage editor and loader is directed to the system programmer responsible for installing and maintaining the operating system.

The "Introduction" briefly defines the functions of the linkage editor and loader and gives recommendations for the use of each. Part 1 describes the linkage editor, and should be read before Part 2, which describes the loader.

The *linkage editor* combines and edits modules to produce a single module that can be brought into storage by program fetch for execution. It operates as a processing program rather than as part of the control program. The linkage editor provides several processing facilities that are either performed automatically or invoked in response to control statements prepared by the programmer.

Part 1, which consists of six chapters and three appendixes, briefly describes the processing facilities and operation of the linkage editor. The introduction also defines linkage editor terms in reference to the source language statements that cause them to be created.

The six chapters describe the input to the linkage editor, the output from the linkage editor, module editing functions, design and specification of overlay programs, the job control language necessary to run a linkage editor job step, and the linkage editor control statements. The last two chapters are summaries of reference information to be used after the general information in the first four chapters is learned. The appendixes to Part 1 contain sample programs, a description of the linkage editor programs, and information on the invocation of the linkage editor.

The *loader* program combines the basic editing and loading functions of the linkage editor and program fetch in one job step. It is designed for high-performance loading of modules that do not require the special processing facilities of the linkage editor and fetch, such as overlay. The loader does not produce load modules for program libraries.

Part 2 of this publication describes the loader. The introduction to this part describes the functional characteristics of the loader, along with its compatibility with the linkage editor and restrictions on its use. The chapter on using the loader describes the job control language statements and invocation procedures for the loader, as well as loader input and output, and user program data. The appendixes to Part 2 contain sample input, a description of loader return codes, and storage considerations. All of these items are discussed in relation to the capabilities of the linkage editor; therefore, the reader must be familiar with Part 1 of this publication.

The diagnostic messages issued by both the linkage editor and the loader program are described in *OS/VS Message Library: Linkage Editor and Loader Messages*, GC38-1007. The description of each message includes an explanation, a system action, and a problem determination action to be taken.

Time Sharing Option (TSO)

The following publication is needed to use the linkage editor or loader under the Time Sharing Option (TSO):

OS/VS2 TSO Terminal User's Guide, GC28-0645

This manual contains procedures for invoking the linkage editor or loader from the terminal and gives a brief description of the options that can be specified under TSO.

Further information on TSO can be found in the following two manuals:

- *OS/VS2 System Programming Library: TSO,* GC28-0629
- *OS/VS2 TSO Command Language Reference*, GC28-0646

Additional Publications

Within the text, references are made to the following publications:

- *OS/VS1 Data Management Services Guide*, GC26-3874
- *OS/VS2 MVS Data Management Services Guide*, GC26-3875
- *OS/VS1 Planning and Use Guide*, GC24-5090
- *OS/VS2 System Programming Library: Initialization and Tuning Guide*, GC28-0681
- *OS/VS2 Planning Guide for Release 2*, GC28-0667
- *OS/VS1 Service Aids*, GC28-0665
- *OS/VS2 System Programming Library: Service Aids*, GC28-0674
- *OS/VS1 Storage Estimates*, GC24-5094
- *OS/VS2 System Programming Library: Storage Estimates*, GC28-0604
- *OS/VS1 Supervisor Services and Macro Instructions*, GC24-5103
- *OS/VS2 Supervisor Services and Macro Instructions*, GC28-0683
- *OS/VS1 System Data Areas*, SY28-0605
- *OS/VS2 Data Areas*, SYB8-0606
- *OS/VS1 System Generation Reference*, GC26-3791
- *OS/VS2 System Programming Library: System Generation Reference*, GC26-3792
- *OS/VS1 Utilities*, GC26-3901
- *OS/VS2 MVS Utilities*, GC26-3902
- *OS/VS Message Library: VS1 System Codes*, GC38-1003
- *OS/VS Message Library: VS2 System Codes*, GC38-1008
- *OS/VS Message Library: Routing and Descriptor Codes*, GC38-1004
- *OS/VS Message Library: Linkage Editor and Loader Messages*, GC38-1007
- *OS/VS1 JCL Reference,* GC24-5099
- *OS/VS2 JCL,* GC28-0692

CONTENTS

FIGURES

OS/VS1 SUMMARY OF AMENDMENTS

Release 6

- This revision incorporates information formerly contained in technical newsletter GN26-0827 and System Supplement Newsletter GC26-3888 (SU 5741-606). It also includes miscellaneous technical and editorial changes.

Release 5

- The appropriate figures have been updated to include specifications for the IBM 3350 Direct Access Storage and the IBM 3344 Direct Access Storage Device.

OS/VS2 SUMMARY OF AMENDMENTS

Release 3.7

- OS/VS2 Release 3.7 supports the IBM 3350 and 3344 Direct Access Storage Devices. Miscellaneous technical and editorial changes have been made to this publication.

Release 2

- The appropriate figures have been updated to include specifications for the 3330-1 and 3340 disk storage devices.
- The format for the load modules produced by the linkage editor has been included in this edition. See Appendix G.
- The "SIZE option" has been rewritten to make it easier for the user to determine the correct values for the option. Appendix H is a summary of this section.

Release 1

- A more efficient EXEC statement has been added for use in the LKEDG procedure when the programmer wishes to specify the LET parameter in the LKED step. This change applies to both OS/VS1 and OS/VS2.

INTRODUCTION

The linkage editor and the loader processing programs prepare the output of language translators for execution. The linkage editor prepares a load module that is to be brought into storage for execution by program fetch. The loader prepares the executable program in storage and passes control to it directly.

The linkage editor provides several processing facilities such as creating overlay programs, and aiding program modification. (The linkage editor is also used to build and edit system libraries.) The loader provides high performance loading of programs that do not require the special processing facilities of the linkage editor.

Use of the linkage editor is recommended in the following cases:

- If the program requires linkage editor services in addition to the MAP, LET, NCAL, and SIZE options
- If the program uses linkage editor control statements such as INCLUDE, NAME, OVERLAY
- If a load module is to be produced for a program library

Use of the loader is recommended if the program only requires the use of the following linkage editor options: MAP, LET, NCAL, and SIZE. Because of its fewer options and because it can process a job in one job step, the loader reduces editing and loading time by about one-half.

Linkage editor processing is performed in a *link edit* step. The linkage editor can be used for compile-link edit-go, compile-link edit, link edit, and link edit-go jobs. Loader processing is performed in a *load* step, which is equivalent to the *link edit-go* steps. The loader can be used for compile-load and load jobs.

Details of how each language interfaces with the linkage editor can be found in the publication(s) describing that language.

PART 1. LINKAGE EDITOR

Linkage editor processing is a necessary step that follows the source program assembly or compilation of any problem program. The linkage editor is a processing program and a service program used in association with the language translators.

Every problem program is designed to fulfill a particular purpose. To achieve that purpose, the program can generally be divided into logical units that perform specific functions. A logical unit of coding that performs a function, or several related functions, is a *module*. Ordinarily, separate functions should be programmed into separate modules, a process called modular programming. Each module can be written in the symbolic language that best suits the function to be performed. (The symbolic languages are Assembler, ALGOL, COBOL, FORTRAN, PL/I, and RPG.)

Each module is separately assembled or compiled by one of the language translators. The input to a language translator is a *source module*; the output from a language translator is an *object* module. Before an object module can be executed, it must be processed by the linkage editor. The output of the linkage editor is a *load module* (Figure 1).

An object module is in relocatable format with unexecutable machine code. A load module (see Appendix G) is also relocatable, but with executable machine code. A load module is in a format that can be loaded into virtual storage and relocated by program fetch (Figure 2).

Any module is composed of one or more *control sections*. A control section is a unit of coding (instructions and data) that is, in itself, an entity. All elements of a control section are loaded and executed in a constant relationship to one another. A control section is, therefore, the smallest separately relocatable unit of a program.

Each module in the input to the linkage editor may contain symbolic references to control sections in other modules; such references are called *external references*. These references are made by means of *address constants* (adcons). The symbol referred to by an external reference must be either the name of a control section or the name of an entry point in a control section. Control section names and entry names are called *external names*. By matching an external reference with an external name, the linkage editor resolves references between modules. External references and external names are called *external symbols* (Figure 3). An external symbol is one that is defined in one module and can be referred to in another.

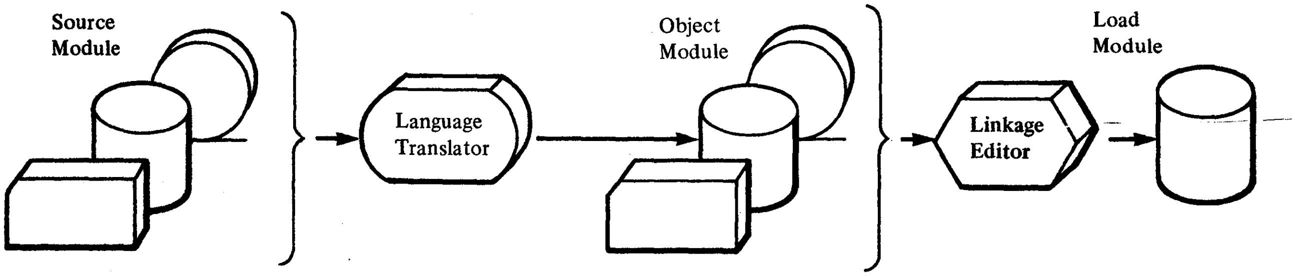


Figure 1. Preparing a Source Module for Execution

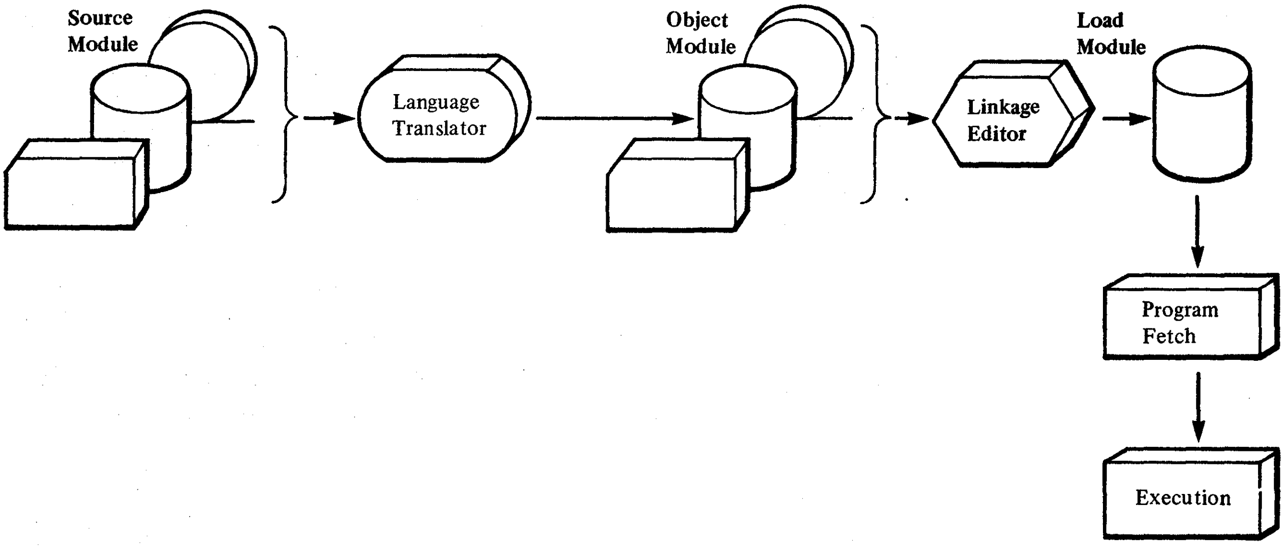


Figure 2. Preparing a Source Module for Execution and Executing the Load Module

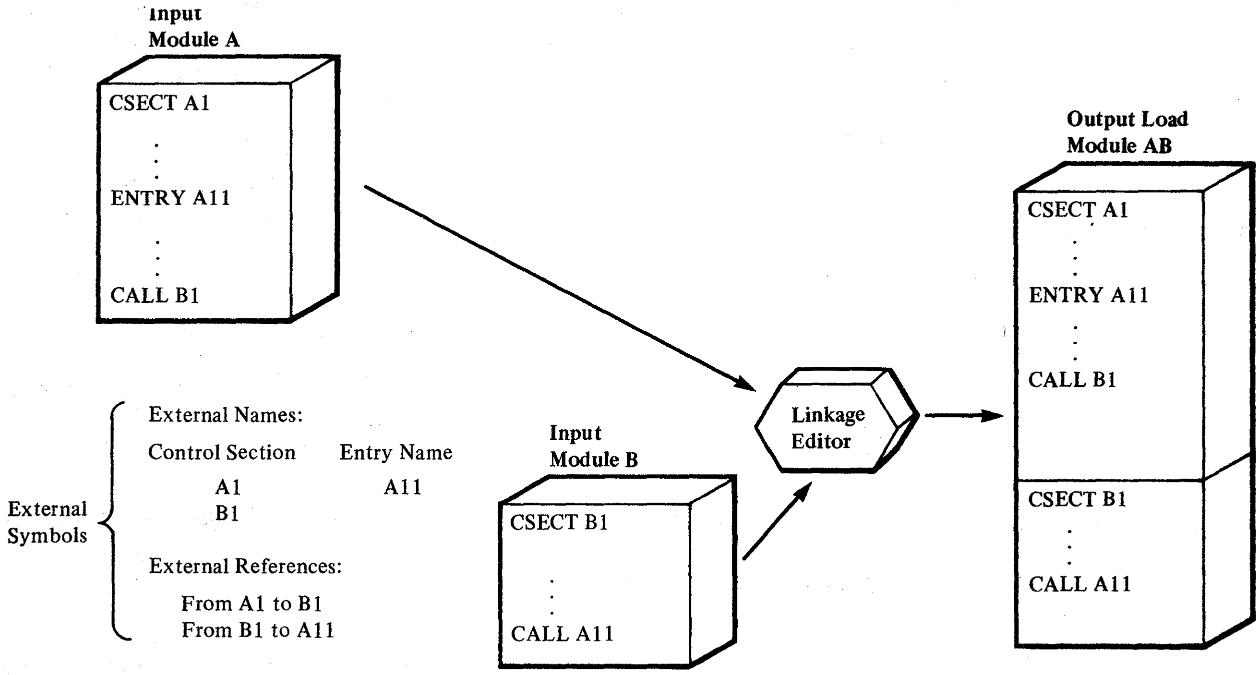


Figure 3. External Names and External References

Object and Load Modules

Object modules and load modules have the same basic logical structure. Each consists of:

- Control dictionaries, containing the information necessary to resolve symbolic cross-references between control sections of different modules, and to relocate address constants. Control dictionary entries are generated when external symbols, address constants, or control sections are processed by a language translator. Each language translator usually produces two kinds of control dictionaries: an external symbol dictionary (ESD) and a relocation dictionary (RLD).
- Text, containing the instructions and data of the program.
- An end-of-module indication: an END statement in an object module, an end-of-module indicator in a load module.

Each control dictionary, text, and end indication is described in greater detail in the following text.

Both object modules and load modules can contain data used by the linkage editor to create CSECT Identification (IDR) records. If the language translator creating an object module supports CSECT Identification, the input object module can contain translator data for Identification records on the END statement. Input load modules differ from object modules in the type of data they supply. Input load modules can also provide HMASPZAP data, linkage editor data, and user data to the Identification records that are built during linkage editor processing. During the link edit step, the optional IDENTIFY control statement is used to supply the optional user data for the CSECT Identification records.

External Symbol Dictionary

The external symbol dictionary (ESD) contains one entry for each external symbol defined or referred to within a module. The dictionary contains an entry for each external reference, pseudo register (external dummy section), entry name, named or unnamed control section, and blank or named common area. An entry name, pseudo register, or named control section can be referred to by any control section or separately processed module; an unnamed control section cannot.

Each entry identifies a symbol, or a symbol reference, and gives its location, if known, within the module. Each entry in the external symbol dictionary is classified as one of the following:

- *External reference*—a symbol that is defined as an external name in another separately processed module, but is referred to in the module being processed. The external symbol dictionary entry specifies the symbol; the location is unknown.
- *Weak external reference*—a special type of external reference that is not to be resolved by automatic library call unless an ordinary external reference to the same symbol is found. The external symbol dictionary entry specifies the symbol; the location is unknown.
- *Entry name*—a name with a control section that defines an entry point. The external symbol dictionary entry specifies the symbol and its location, and identifies the control section to which it belongs.

- *Control section name*—the symbolic name of a control section. The external symbol dictionary entry specifies the symbol, the length of the control section, and its location. In this case, the location represents the origin of the control section, which is the first byte of the control section.
- *Blank or named common area*—a control section used to reserve a virtual storage area that can be referred to by other modules. The reserved storage area can be used, for example, as a communications region within a program or to hold data supplied at execution time. The external symbol dictionary entry specifies the name, if there is one, and the length of the area. If there is no name, the name field contains blanks.
- *Private code*—an unnamed control section. The external symbol dictionary entry specifies the length of the control section and the origin. The name field contains blanks.
- *Pseudo register*—a special facility (corresponding to the external dummy section feature of Assembler F) that can be used to write re-enterable programs. A pseudo register is a dynamically obtained location in virtual storage that can be used as a pointer to dynamically acquired storage; that is, the space for such areas is not reserved in the load module but is acquired during execution. The external symbol dictionary contains the name, length, alignment, and displacement of the pseudo register.

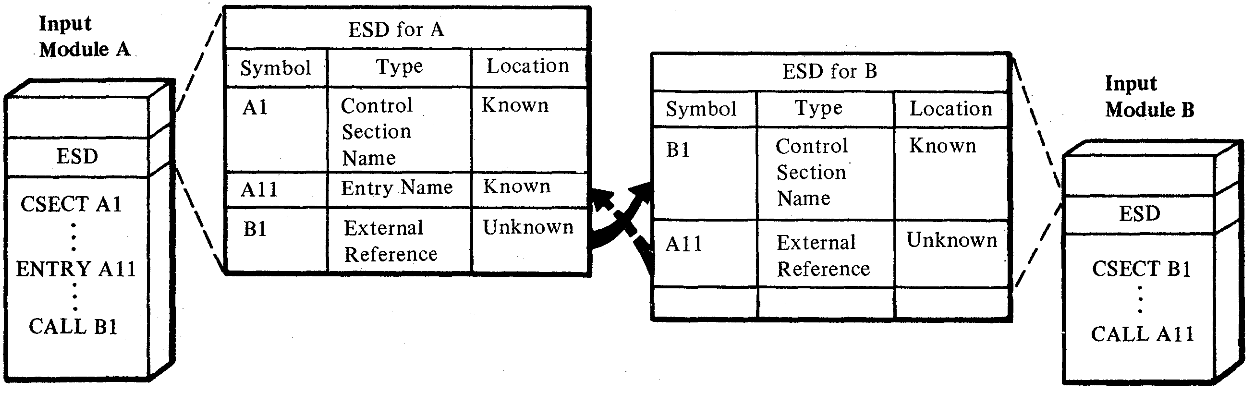
When processing input modules, the linkage editor resolves references between modules by matching the referenced symbols to defined symbols. To do this, the linkage editor searches for the external symbol definition in the external symbol dictionary of each input module. As shown in Figure 4, the linkage editor matches the external reference to B1 by locating the definition for B1 in the external symbol dictionary of Module B. In the same way, it matches the external reference to A11 by locating the definition for A11 in the external symbol dictionary of Module A.

Text

The text contains the instructions and data of the module.

Relocation Dictionary

The relocation dictionary (RLD) contains one entry for each relocatable address constant that must be modified before a module is executed. An entry identifies an address constant by indicating both its location within a control section and the external symbol whose value must be used to compute the



ESD for A

Symbol	Type	Location
A1	Control Section Name	Known
A11	Entry Name	Known
B1	External Reference	Unknown

ESD for B

Symbol	Type	Location
B1	Control Section Name	Known
A11	External Reference	Unknown

Figure 4. Use of the External Symbol Dictionary

value of the address constant. (The external symbol is defined in an external symbol dictionary entry in another control section or module.)

The linkage editor uses the relocation dictionary whenever it processes a module to adjust the address constants for references to other control sections and modules. This dictionary is also used to adjust these address constants again after program fetch reads an output load module from a library and loads it into virtual storage for execution.

End Indication

The end of a load module is marked by an *end-of-module* indicator (EOM). The EOM cannot, like the assembler END instruction, specify an entry point. Therefore, whenever a load module is reprocessed by the linkage editor, a main entry point should be specified on an ENTRY statement. If one is not specified, the linkage editor will assign the first byte of the first control section encountered as the entry point.

Linkage Editor Processing

This section discusses the input and output sources of the linkage editor, and the way in which the linkage editor produces a load module.

Input and Output Sources

The linkage editor can receive its input from several sources, as follows:

- The primary input, which can contain only object modules and linkage editor control statements (called control statements in the following text).
- Additional user-specified input, which can contain either object modules and control statements, or load modules. This input is either specified by the user as input, or incorporated automatically by the linkage editor from a call library.

During processing, the linkage editor generates *intermediate data*. Intermediate data is placed on a direct-access storage device when virtual storage allocated for input data is exhausted.

Output of the linkage editor is of two types:

- A load module, which is always placed in a library (a partitioned data set) as a named member.
- Diagnostic output, which is produced as a sequential data set.

Figure 5 shows the input, intermediate, and output sources for the linkage editor program.

Load Module Creation

In processing object and load modules, the linkage editor assigns consecutive relative addresses to all control sections and resolves all references between control sections. Object modules produced by several different language translators can be used to form one load module.

An output load module is composed of all input object modules and input load modules processed by the linkage editor. The control dictionaries of an output module are, therefore, a composite of all the control dictionaries in the linkage editor input. The control dictionaries of a load module are called the composite external symbol dictionary (CESD) and the relocation dictionary (RLD). The load module also contains all of the text from each input module, and one end-of-module indicator (Figure 6.) See Appendix G for the format of a load module.

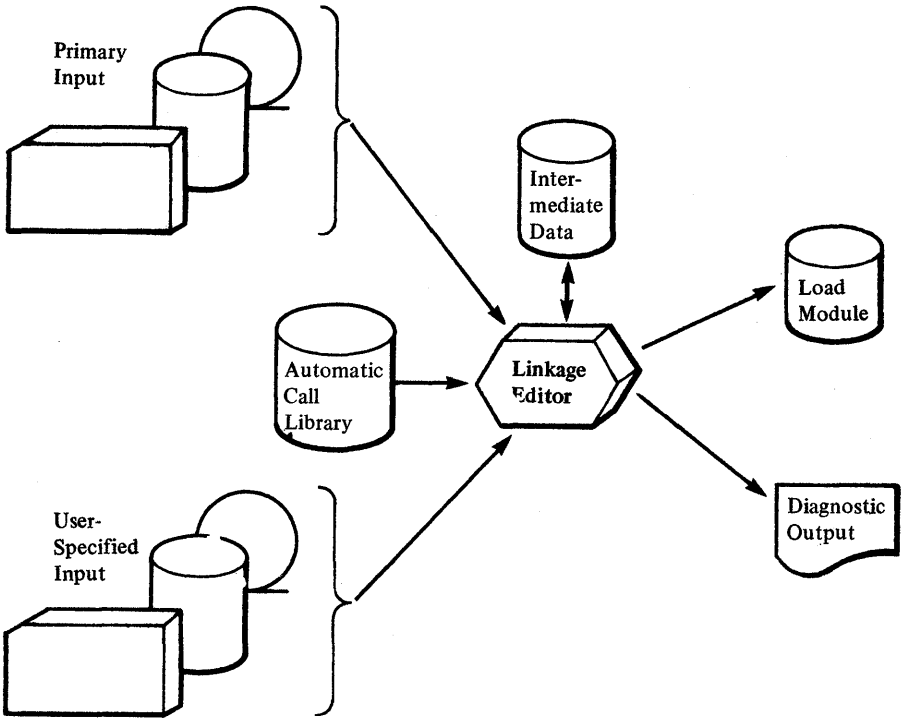


Figure 5. Input, Intermediate, and Output Sources for the Linkage Editor

Assigning Addresses

Each module to be processed by the linkage editor has an origin that was assigned during assembly, compilation, or a previous execution of the linkage editor. When several modules, each with an independently assigned origin, are to be processed by the linkage editor, the sequence of the addresses is unpredictable; two input modules may even have the same origin.

Each input module can be made up of one or more control sections. To produce an executable output load module, the linkage editor assigns relative virtual storage addresses to each control section by assigning an origin to the first control section encountered and then assigning addresses, relative to that origin, to all other control sections to be included in the output load module. The value assigned as the origin of the control section is used to relocate each address dependent item in the control section.

Although the addresses in a load module are consecutive, they are relative to zero. When a load module is to be executed, program fetch prepares the module for execution by loading it at a specific virtual storage location. The addresses in the module are then increased by this base address. Each address constant must also be readjusted, another function of program fetch.

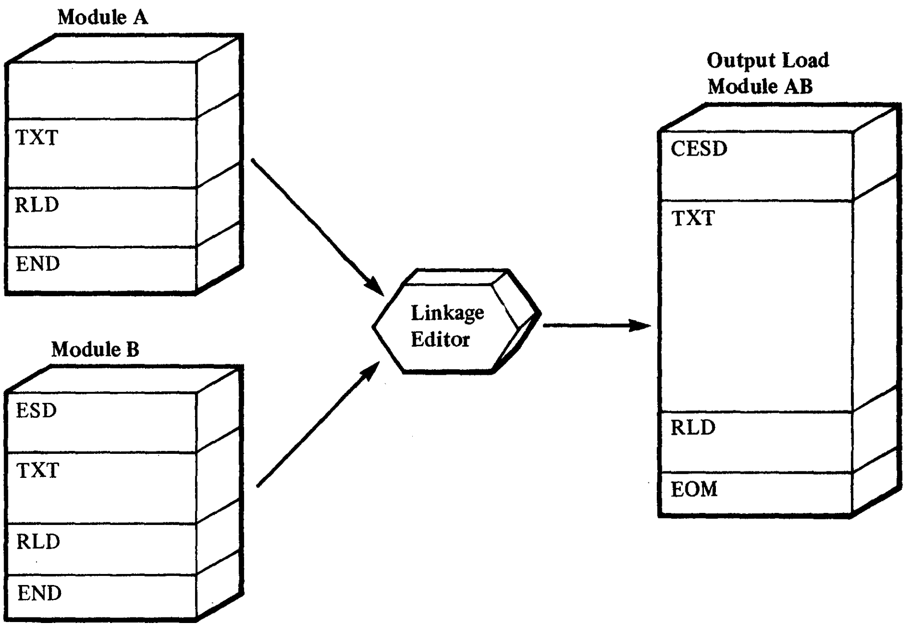


Figure 6. A Load Module Produced by the Linkage Editor

Resolving External References

The linkage editor also resolves external references in the input modules. Cross references between control sections in different modules are symbolic. They must be resolved relative to the addresses assigned to the load module. The linkage editor calculates the new address of each relocatable expression in a control section and determines the assigned origin of the item to which it refers.

Functions of the Linkage Editor

Linkage editor input may consist of a combination of object modules, load modules, and control statements. The primary function of the linkage editor is to combine these modules, in accordance with the requirements stated on control statements, into a single output load module. Although this linking or combining of modules is its primary function, the linkage editor also:

- Edits modules by replacing, deleting, rearranging, and ordering control sections as directed by control statements.
- Aligns control sections and named common areas on 2K or 4K page boundaries as directed by control statements.
- Accepts additional input modules from data sets other than the primary input data set, either automatically, or upon request.
- Reserves storage for the common control sections generated by Assembler and FORTRAN language translators, and static external areas generated by PL/I.
- Computes total length and assigns displacements for all pseudo registers (external dummy sections).
- Creates overlay programs in a structure defined by control statements.

- Creates multiple output load modules as directed by control statements.
- Provides special processing and diagnostic output options.
- Assigns module attributes that describe the structure, content, and logical format of the output load module.
- Allocates storage areas for linkage editor processing as specified by the programmer.
- Stores system status index information in the directory of the output module library (systems personnel only).
- Traces the processing history of a program.
- Allows the user to lengthen a control section or named common section without changing source code, reassembling, or recompliling.
- Allows the user to assign an authorization code to a load module that (a) makes it a restricted resouce and (b) enables it to pass control to other restricted resources.

Each of the linkage editor functions is described briefly in the following paragraphs.

Links Modules

Processing by the linkage editor makes it possible for the programmer to divide his program into several modules, each containing one or more control sections. The modules can be separately assembled or compiled. The linkage editor combines these modules into one output load module (Figure 7) with contiguous storage addresses. During processing by the linkage editor, references between modules within the input are resolved. The output module is placed in a library (partitioned data set).

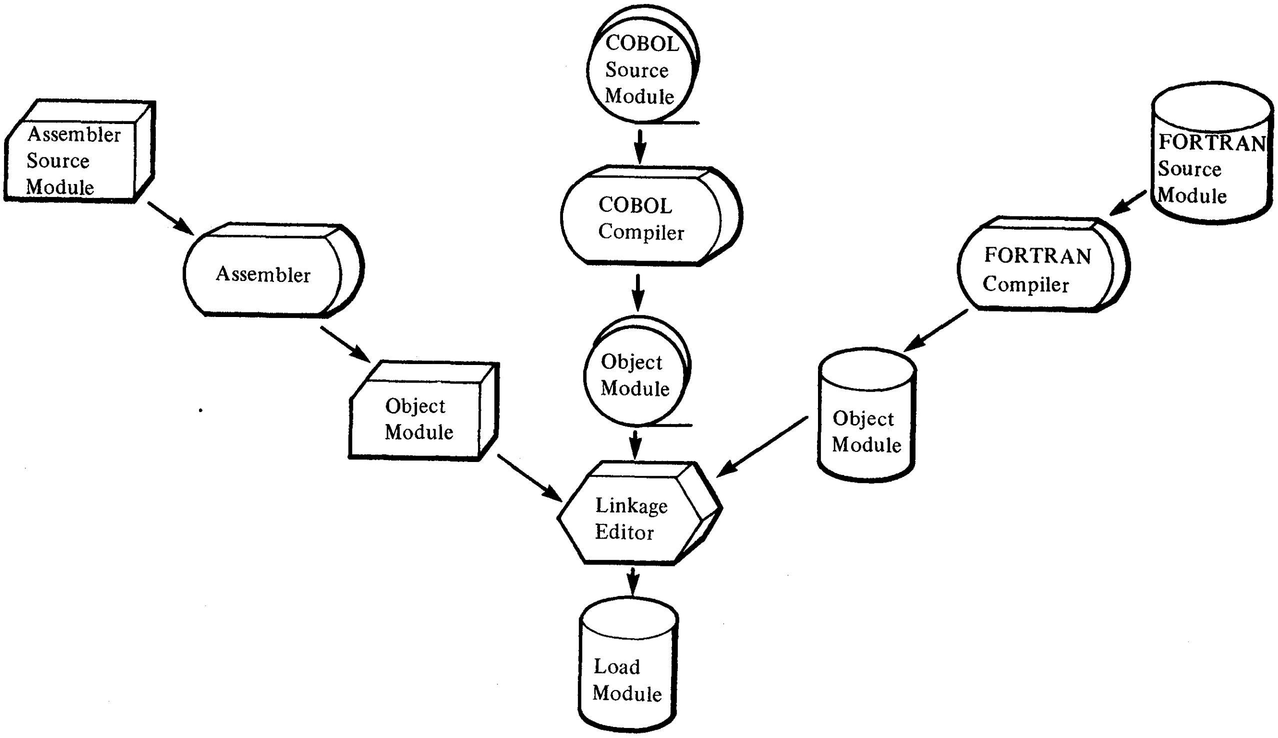


Figure 7. Linkage Editor Processing—Module Linkage

Edits Modules

Program modification is made easier by the editing functions of the linkage editor. When the functions of a program are changed, the programmer modifies, then compiles and link edits again only the affected control sections instead of the entire source module.

Control sections can be replaced, renamed, deleted, moved, or ordered as directed by control statements. Control sections can also be automatically replaced by the linkage editor. External symbols can be changed or deleted as directed by control statements.

Figure 8 illustrates the module editing function of the linkage editor.

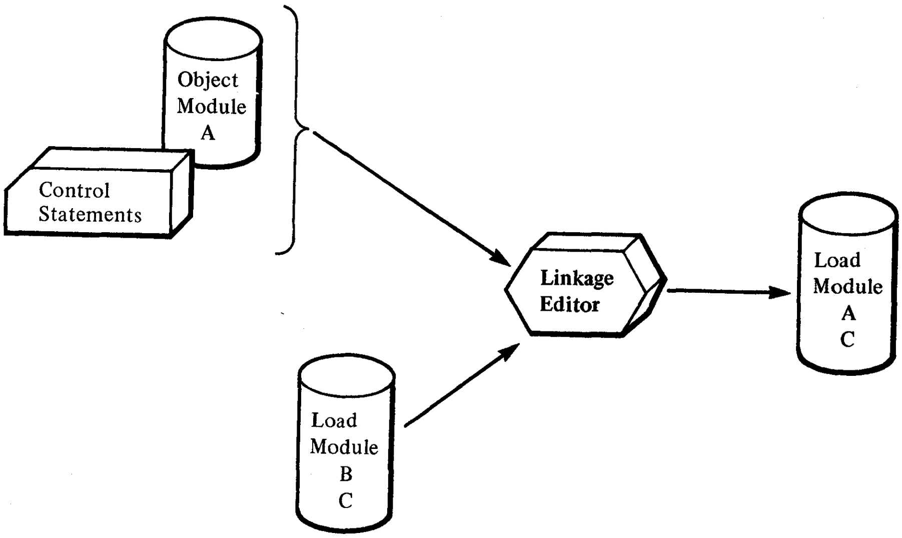


Figure 8. Linkage Editor Processing—Module Editing

Aligns Control Sections or Common Areas on Page Boundaries

Control sections or named common areas in the output load module can be aligned on either 2K or 4K page boundaries. Alignment on page boundaries enables the programmer to use real storage more efficiently and appreciably reduce the paging rate for the job.

Accepts Additional Input Sources

Standard subroutines can be included in the output module, thus reducing the work in coding programs. The programmer can specify that a subroutine be included at a particular time during the processing of his program by using a control statement. When the linkage editor processes a program that contains this statement, the module containing the subroutine is retrived from the indicated input source, and made a part of the output module (Figure 9).

Symbols that are still undefined after all input modules have been processed cause the automatic library call mechanism to search for modules that will resolve these references. When a module name is found that matches the unresolved symbol, the module is processed by the linkage editor and also becomes part of the output module (Figure 9).

Note: The level F linkage editor distinguishes a special type of external reference— the weak external reference. An unresolved weak external reference does *not* cause the linkage editor to use the automatic library call mechanism. Instead, the reference is left unresolved, and the load module is marked as executable.

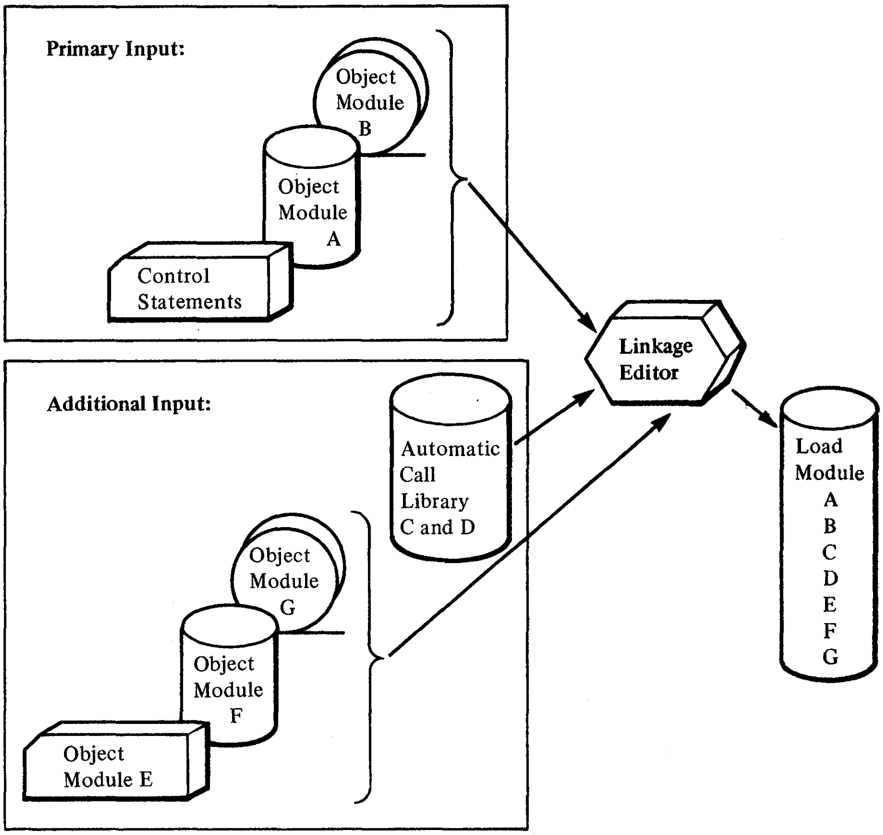


Figure 9. Linkage Editor Processing—Additional Input Sources

Reserves Storage

The linkage editor processes common control sections generated by the FORTRAN and Assember language translators. The static external storage areas generated by the PL/I compiler are processed in the same way. The common areas are collected by the linkage editor, and a reserved virtual storage area is provided within the output module.

Processes Pseudo Registers

Pseudo registers, like the external dummy sections of Assembler F, aid in generating re-enterable code. The linkage editor processes pseudo registers by accumulating the total length of storage required for all pseudo registers and recording the displacement of each. During execution, the program dynamically acquires the necessary storage.

Creates Overlay Programs

To minimize virtual storage requirements, the programmer can organize his program into an overlay structure by dividing it into segments according to the functional relationships of the control sections. Two or more segments that need not be in virtual storage at the same time can be assigned the same relative virtual storage addresses, and can be loaded at different times.

The programmer uses control statements to specify the relationship of segments within the overlay structure. The segments of the load module are placed in a library so that the control program can load them separately when the load module is executed.

Creates Multiple Load Modules

The linkage editor can also process its input to form more than one load module within a single job step. Each load module is placed in the library under a unique member name, as specified by a control statement.

Provides Special Processing and Diagnostic Output Options

The programmer can specify special processing options that negate automatic library call or the effect of minor errors. In addition, the linkage editor can produce a module map or cross-reference table that shows the arrangement of control sections in the output module and indicates how they communicate with one another. A list of the control statements processed can also be produced.

Throughout processing, errors and possible error conditions are logged. Serious errors cause the linkage editor to mark the output module not executable. Additional diagnostic data is automatically logged by the linkage editor. The data indicates the disposition of the load module in the output module library.

Assigns Load Module Attributes

When the linkage editor generates a load module, it places an entry for the module in the directory of the library. This entry contains attributes that describe the structure, content, and logical format of the load module. The control program uses these attributes to determine how a module is to be loaded, what it contains, if it is executable, whether it is executable more than once without reloading, and if it can be executed by concurrent tasks. Some module attributes can be specified by the programmer; others are specified by the linkage editor as a result of information gathered during processing.

Allocates User-Specified Virtual Storage Areas

The programmer can specify the total amount of virtual storage to be made available to the linkage editor, the amount to be used for the load module buffer, and the buffer for the output load module.

Stores System Status Index Information

The following information is intended for systems personnel responsible for maintaining IBM-supplied load modules. It is not generally applicable to non-IBM load modules.

Four bytes in the library directory entry for IBM-supplied load modules are used to store system status index information. This information, which is used for maintenance of the modules, is placed in the directory with a control statement.

Traces Processing History

Tracing the processing history of a program is simplified by the CSECT Identification (IDR) records created and maintained by the linkage editor. A CSECT Identification record can contain data that describes:

- The language translator, its level, and the translation date for each control section.

- The most recent processing by the linkage editor.
- Any modification made to the executable code of any control section.

Optionally, user-supplied data associated with the executable code of a control section can also be recorded.

Lengthens Control Sections or Named Common Sections

The user can lengthen control sections or named common sections of a program to add patch space without changing the source code, reassembling, or recompiling.

Added space, consisting of binary zeros, is put at the end of a specified control section by using the EXPAND control statement (see the "Control Statement Summary" section). Space cannot be added to a private code or blank common section.

Assigns an Authorization Code to Output Load Modules

The authorized program facility (APF) limits the use of sensitive system and (optionally) user services and resources to authorized system and user programs. Authorization is defined as accesss to those services and resources. The services and resources to which access is limited are described in the following publications: for VS1, *OS/VS1 Planning and Use Guide*; for VS2, *OS/VS2 System Programming Library: Initialization and Tuning Guide.*

Programs are authorized at the job-step level. For a job step to gain authorization initially, the first module loaded at the start of the job step must be an authorized module, and it must have been loaded from an authorized library. Otherwise, the job step is not authorized initially and cannot subsequently gain authorization.

For a job step to maintain its authorization, all subsequent modules invoked during the job step (via LINK, LOAD, ATTACH, and/or XCTL macro instructions) must be loaded from an authorized library. Otherwise, the job step loses its authorization and cannot gain authorization.

A library becomes an "authorized" library by the inclusion of its name in a list called IEAAPF00. This list is described in more detail in *OS/VS1 Planning and Use Guide* and *OS/VS2 System Programming Library: Initialization and Tuning Guide.*

In VS1 and VS2, a load module becomes "authorized" by the assignment of an authorization code to the load module during linkage-editing. This assignment is made via the PARM field parameter AC or via the control statement SETCODE, which are described in the sections that follow.

In VS1, a load module becomes "authorized" by the inclusion of its name in a list called IEFSDPPT. This list is described in more detail in *OS/VS1 Planning and Use Guide.*

Relationship to the Operating System

The linkage editor has the same relationship to the operating system as any other processing program. It can be executed either as a job step, a subprogram, or a subtask. Control is passed to the linkage editor in one of three ways:

- As a job step, when the linkage editor is specified on an EXEC job control statement in the input stream.
- As a subprogram, with the execution of a CALL macro instruction (after the execution of a LOAD macro instruction), a LINK macro instruction, or an XCTL macro instruction.
- As a subtask, in multitasking systems, with the execution of the ATTACH macro instruction.

Execution of the linkage editor and the data sets used by the linkage editor are described to the system with job control language statements. These statements describe all jobs to be performed by the system.

Note: Job control statements are not to be confused with linkage editor control statements. Job control statements are processed before the linkage editor is executed; linkage editor control statements are processed during linkage editor execution.

Time Sharing Option (TSO)

When the linkage editor is used under TSO (VS2 only), it is invoked by the linkage editor prompter program that acts as an interface between the user, operating system, and linkage editor. Under TSO, execution of the linkage editor and definition of data sets used by the linkage editor are described to the system through use of the LINK command that causes the prompter to be executed. Operands of the LINK command can also be used to specify the linkage editor options a job requires. Complete procedures for use of the LINK command are given in the *OS/VS2 TSO Terminal User's Guide*.

INPUT TO THE LINKAGE EDITOR

The linkage editor accepts input from two major sources: the primary input data set and additional data sets. The *primary input data set* is made available through job control language specifications. *Additional data sets* are made available either through the automatic library call mechanism, or through user-specified control statements. They must, however, also be defined with job control language specifications.

Primary and additional input data sets may contain the following types of data:

- One or more object modules.
- One or more load modules.
- Control statements.
- Combinations of the above (restrictions on certain combinations are noted where they apply).

Object modules and control statements may be contained in either sequential or partitioned data sets. Load modules must be contained in partitioned data sets.

This chapter describes the "linking " functions of the linkage editor only; the "editing" functions are described in the chapter "Module Editing."

Primary Input Data Set

The primary input data set is required for every linkage editor job step. It must be defined by a DD statement with the ddname SYSLIN. The primary input can be:

- A sequential data set.
- A member of a partitioned data set.
- A concatenation of sequential data sets and/or members of partitioned data sets.

The primary input data set must contain object modules and/or control statements. The modules and control statements are processed sequentially and their order determines the basic order of linkage editor processing during a given execution. However, the order of the control sections after processing does not necessarily reflect the order in which they appeared in the input.

In the examples that follow, only the statements necessary to define the input to the linkage editor are shown; complete examples are shown in Appendix A.

Object Modules

The primary input to the linkage editor may consist solely of one or more object modules. The rest of this section discusses object module input from cards, as a member of a partitioned data set, passed from a previous job step, and created in a separate job.

From Cards

Object module input to the linkage editor may be on cards. The card deck itself is treated as a sequential data set; the cards are placed in the input stream, after a DD * statement, as follows:

```
//SYSLIN      DD      *
Object Deck A
Object Deck B
/*
```

The card input is followed by a /* statement.

An example of the JCL when card decks are used in addition to other input is as follows:

```
//SYSLIN      DD      DSNAME=INPUT,...
//            DD      *
Object Deck A
Object Deck B
/*
```

By omitting the ddname on the second DD statement, the card input is concatenated to the data set described on the SYSLIN DD statement.

As a Member of a Partitioned Data Set

An object module in a partitioned data set can be used as primary input to the linkage editor by specifying its data set name and member name on the SYSLIN DD statement. In the following example, the member named TAXCOMP in the object module library LIBROUT is to be the primary input; LIBROUT is a cataloged data set:

```
//SYSLIN      DD      DSNAME=LIBROUT(TAXCOMP),
//                    DISP=(OLD,KEEP)
```

The library member is processed as if it were a sequential data set.

Members of partitioned data sets can be concatenated with other input data sets, as follows:

```
//SYSLIN      DD      DSNAME=OBJLIB,DISP=(OLD,KEEP),...
//            DD      DSNAME=LIBROUT(TAXCOMP),
//                    DISP=(OLD,KEEP)
```

Library member TAXCOMP is concatenated to data set OBJLIB; both must contain object modules since they are the primary input.

Passed from a Previous Job Step

An object module to be used as input can be passed from a previous job step to a linkage editor job step in the same job, as in a compile-link edit job. That is, the output from the compiler is direct input to the linkage editor. In the following example, an object module that was created in a previous job step (STEPA) is passed to the linkage editor job step (STEPB):

```
STEPA:

//SYSGO       DD      DSNAME=&&OBJECT,DISP=(NEW,PASS),...
               .
               .
               .
STEPB:

//SYSLIN      DD      DSNAME=&&OBJECT,DISP=(OLD,DELETE)
```

The data set name &&OBJECT, used in both job steps, identifies the object module as the output of the language processor on the SYSGO DD statement, and as the primary input to the linkage editor on the SYSLIN DD statement.

Note: The double ampersand (&&) in the data set name defines a temporary data set. These data sets exist for the duration of the job and are automatically deleted at the end of the job. If the data set is to be preserved for longer than the duration of a single job, the double ampersand is not used (DSNAME=OBJECT).

The method used in the preceding example can also be used to retrieve object modules created in previous steps. If the same data set name is used for the output of each language processor, one SYSLIN DD statement can be used to retrieve all the object modules, as follows:

STEPA:

```
//SYSGO        DD       DSNAME=&&OBJMOD,DISP=(NEW,PASS),...
                .
                .
                .
```

STEPB:

```
//SYSPUNCH     DD       DSNAME=&&OBJMOD,DISP=(MOD,PASS)
                .
                .
                .
```

STEPC:

```
//SYSLIN       DD       DSNAME=&&OBJMOD,DISP=(OLD,DELETE)
```

The two object modules from STEPA and STEPB are placed in the same sequential data set, &&OBJMOD. The SYSLIN DD statement in STEPC causes both object modules to be used as the primary input to the linkage editor.

Another method can be used to accomplish this purpose: concatenation of data sets. This method could be used if the object modules were created in previous job steps with different member names, as follows:

STEPA:

```
//SYSGO        DD       DSNAME=&&OBJLIB(MODA),DISP=(NEW,
//                      PASS),...
                .
                .
                .
```

STEPB:

```
//SYSPUNCH     DD       DSNAME=&&OBJLIB(MODB),DISP=(MOD,
//                      PASS),...
                .
                .
                .
```

STEPC:

```
//SYSLIN       DD      DSNAME=&&OBJLIB(MODA),DISP=(OLD,
//                     DELETE)
//             DD      DSNAME=&&OBJLIB(MODB),DISP=(OLD,
//                     DELETE),VOL=REF=*.STEPB.SYSPUNCH
```

The object modules created in STEPA and STEPB were placed in a partitioned data set with different member names. The two members are concatenated in STEPC as primary input. Each member is considered to be a sequential data set.

Created in a Separate Job

If the only input to the linkage editor is an object module from a previous job, the SYSLIN DD statement contains all the information necessary to locate the object module, as follows:

```
//SYSLIN      DD      DSNAME=OBJECT,DISP=(OLD,DELETE),
//                    UNIT=2314,VOLUME=SER=LIB613
```

An object module created in a separate job may also be on cards, in which case it is handled as described earlier.

Control Statements

The primary input data set may also consist solely of control statements. When the primary input is control statements, input modules are specified on INCLUDE control statements (see "Included Data Sets "). The control statements may be either placed in the input stream or stored in a permanent data set.

In the following example, the primary input consists of control statements in the input stream:

```
//SYSLIN      DD      *
Linkage Editor Control Statements
/*
```

In the next example, the primary input consists of control statements stored in the member INCLUDES in the partitioned data set CTLSTMTS:

```
//SYSLIN      DD      DSNAME=CTLSTMTS(INCLUDES),DISP=(OLD,
//                    KEEP),...
```

In either case, the control statements can be any of those described in "Linkage Editor Control Statement Summary," as long as the rules given there are followed.

Object Modules and Control Statements

The primary input to the linkage editor may contain both object modules and control statements. The object modules and control statements may be in either the same data set or different data sets. If the modules and statements are in the same data set, this data set is described on the SYSLIN DD statement as any data set is described.

If the modules and statements are in different data sets, the data sets are concatenated. The control statements may be defined either in the input stream or as a separate data set.

Control Statements in the Input Stream

Control statements can be placed in the input stream and concatenated to an object module data set, as follows:

```
//SYSLIN      DD      DSNAME=&&OBJECT,...
//            DD      *
Linkage Editor Control Statements
/*
```

Another method of handling control statements in the input stream is to use the DDNAME parameter, as follows:

```
//SYSLIN       DD       DSNAME=&&OBJECT,...
//             DD       DDNAME=SYSIN
                .
                .
                .
//SYSIN        DD        *
```

Linkage Editor Control Statements

```
/*
```

Note: The linkage editor cataloged procedures use DDNAME=SYSIN for the SYSLIN DD statement to allow the programmer to specify the primary input data set required.

Control Statements in a Separate Data Set

A separate data set that contains control statements may be concatenated to a data set that contains an object module. The control statements for a frequently used procedure (for example, a complex overlay structure or a series of INCLUDE statements) can be stored permanently. In the following example, the members of data set CTLSTMTS contain linkage editor control statements. One of the members is concatenated to data set && OBJECT.

```
//SYSLIN       DD       DSNAME=&&OBJECT,DISP=(OLD,DELETE),...
//             DD       DSNAME=CTLSTMTS(OVLY),DISP=(OLD,
//                      KEEP),...
```

The control statements in the member named OVLY of the partitioned data set CTLSTMTS are used to structure the object module.

Automatic Call Library

The automatic library call mechanism is used to resolve external references that were not resolved during primary input processing. Unresolved external references found in modules from additional data sources are also processed by this mechanism.

Note: The following discussion of automatic library call does not apply to unresolved weak external references; they are left unresolved.

The automatic library call mechanism involves a search of the directory of the automatic call library for an entry that matches the unresolved external reference. When a match is found, the entire member is processed as input to the linkage editor.

Automatic library call can resolve an external reference when the following conditions exist; the external reference must be (1) a member name or an alias of a module in the call library, and (2) defined as an external name in the external symbol dictionary of the module with that name. If the unresolved external reference is a member name or an alias in the library, but is not an external name in that member, the member is processed but the external reference remains unresolved unless subsequently defined.

The automatic library call mechanism searches the call library defined on the SYSLIB DD statement. The call library can contain either (1) object modules and control statements or (2) load modules; it must not contain both.

Modules from libraries other than the SYSLIB call library can be searched by the automatic library call mechanism as directed by the LIBRARY control statement. The library specified in the control statement is searched for member names that match specific external references that are unresolved at the end of input processing. If any unresolved references are found in the modules located by automatic library call, they are resolved by another search of the library. Any external references not specified on a LIBRARY control statement are resolved from the library defined on the SYSLIB DD statement.

In addition, two means exist to negate the automatic library call mechanism. The LIBRARY statement can be used to negate the automatic library call for *selected* external references unresolved after input processing; the NCAL option on the EXEC statement can be used to negate the automatic library call for *all* external references unresolved after input processing. Use of the LIBRARY control statement and the NCAL option are discussed after the SYSLIB DD statement that follows.

SYSLIB DD Statement

If the automatic library call mechanism is to be used, the call library must be a partitioned data set described by a DD statement with a ddname of SYSLIB. The call library may be either a system call library or a private call library; call libraries may be concatenated.

System Call Library

Most of the system processing programs have their own automatic call library (Figure 10). This library must be defined when an object module produced by that processor is to be link edited.

Processing Program	Library Name
ALGOL	SYS1.ALGLIB
COBOL	SYS1.COBLIB
FORTRAN	SYS1.FORTLIB
PL/I	SYS1.PL1LIB
Sort/Merge	SYS1.SORTLIB

Figure 10. System Automatic Call Libraries

The call library may contain input/output, data conversion, and/or other special routines that are needed to complete the module. The processor creates an external reference for these special routines and the linkage editor resolves the references from the appropriate call library.

In the following example, a FORTRAN object module created in STEPA is to be link edited in STEPB, and the FORTRAN automatic call library is used to resolve external references:

STEPA:

```
//SYSOBJ      DD      DSNAME=&&OBJMOD,DISP=(NEW,
//                    PASS),...
               .
               .
               .
```

STEPB:

```
//SYSLIN      DD      DSNAME=&&OBJMOD,DISP=(OLD,DELETE)
//SYSLIB      DD      DSNAME=SYS1.FORTLIB,DISP=SHR
```

The disposition of SHR on the SYSLIB DD statement means that other tasks which may be executing concurrently with STEPB may also use SYS1.FORTLIB.

Private Call Libraries

The SYSLIB DD statement can also describe a private, user-written library. In this case, the automatic library call mechanism searches the private library for unresolved external references. In the following example, unresolved external references are to be resolved from a private library named PVTPROG:

```
//SYSLIB      DD      DSNAME=PVTPROG,DISP=SHR,UNIT=2314,
//                    VOLUME=SER=PVT002
```

Concatenation of Call Libraries

System call libraries and private call libraries may be concatenated either to themselves, and/or to each other. When libraries are concatenated, they must all be either object module libraries or load module libraries; they may not be mixed.

If object modules from different system processors are to be link edited to form one load module, the call library for each must be defined. This is accomplished by concatenating the additional call libraries to the library defined on the SYSLIB DD statement. In the following example, a FORTRAN object module and a COBOL object module are to be link edited; the two system call libraries are concatenated as follows:

```
//SYSLIB      DD      DSNAME=SYS1.FORTLIB,DISP=SHR
//            DD      DSNAME=SYS1.COBLIB,DISP=SHR
```

System libraries are cataloged; no unit or volume information is needed.

A system call library and a private call library can also be concatenated in this way. For example, by adding the following statement to the two in the preceding example, the private call library PVTPROG, which is not cataloged, is concatenated to the two system call libraries:

```
//            DD      DSNAME=PVTPROG,DISP=SHR,UNIT=2314,
//                    VOLUME=SER=PVT002
```

Any external references not resolved from the two system libraries are resolved from the private library.

Library Control Statement

The LIBRARY control statement can be used to direct the automatic library call mechanism to a library other than that specified in the SYSLIB DD statement. Only external references listed on the LIBRARY statement are resolved in this way. All other unresolved external references are resolved from the library in the SYSLIB DD statement.

The LIBRARY statement can also be used to specify external references that are *not* to be resolved by the automatic library call mechanism. The LIBRARY statement specifies the duration of the nonresolution: either during the current linkage editor job step, called *restricted no-call*; or during this or any subsequent linkage editor job step, called *never-call.*

Examples of each use of the LIBRARY statement follow; a description of the format is given in "Linkage Editor Control Statement Summary."

Additional Call Libraries

If the additional libraries are to be used to resolve specific references, the LIBRARY statement contains the ddname of a DD statement that describes the library. The LIBRARY statement also contains, in parentheses, the external references to be resolved from the library; i.e., the names of the members to be used from the library. If the unresolved external reference is not a member name in the specified library, the reference remains unresolved unless subsequently defined.

For example, two modules (DATE and TIME) from a system call library have been rewritten. The new modules are to be tested with the calling modules before they replace the old modules. Because the automatic library call mechanism would otherwise search the system call library (which is needed for other modules), a LIBRARY statement is used, as follows:

```
//SYSLIB      DD     DSNAME=SYS1.COBLIB,DISP=SHR
//TESTLIB     DD     DSNAME=TEST,DISP=(OLD,KEEP),...
//SYSLIN      DD     DSNAME=ACCTROUT,...
//            DD     *
  LIBRARY            TESTLIB(DATE,TIME)
/*
```

Two external references, DATE and TIME, are resolved from the library described on the TESTLIB DD statement. All other unresolved external references are resolved from the library described on the SYSLIB DD statement.

Restricted No-Call Function

The programmer can use the LIBRARY statement to specify those external references in the output module for which there is to be no library search during the current linkage editor job step. This is done by specifying the external reference(s) in parentheses without specifying a ddname. The reference remains unresolved, but the linkage editor marks the module executable.

For example, a program contains references to two large modules that are called from the automatic call library. One of the modules has been tested and corrected, the other is to be tested in this job step. Rather than execute the tested module again, the restricted no-call function is used to prevent automatic library call from processing the module as follows:

```
//            EXEC   PGM=HEWL,PARM=LET
//SYSLIB      DD     DSNAME=PVTPROG,DISP=SHR,UNIT=2314,
//                   VOLUME=SER=PVT002
               .
               .
               .
//SYSLIN      DD     DSNAME=&&PAYROL,...
//            DD     *
  LIBRARY            (OVERTIME)
/*
```

As a result, the external reference to OVERTIME is not resolved by automatic library call.

Never-Call Function

The never-call function specifies those external references that are not to be resolved by automatic library call during this or any subsequent linkage editor job step. This is done by specifying an asterisk followed by the external reference(s) in parentheses. The reference remains unresolved but the linkage editor marks the module executable.

For example, a certain part of a program is never executed, but it contains an external reference to a large module (CITYTAX) which is no longer used by this program. However, the module is in a call library needed to resolve other references. Rather than take up storage for a module that is never used, the never-call function is specified, as follows:

```
//            EXEC    PGM=HEWL,PARM=LET
//SYSLIB      DD      DSNAME=PVTPROG,DISP=SHR,UNIT=2314,
//                    VOLUME=SER=PVT002
               .
               .
               .
//SYSLIN      DD      DSNAME=TAXROUT,DISP=OLD,...
//            DD      *
  LIBRARY     *(CITYTAX)
/*
```

As a result, when program TAXROUT is link edited, the external reference to CITYTAX is not resolved by automatic library call.

NCAL Option

When the NCAL option is specified, no automatic library call occurs to resolve external references that are unresolved after input processing. The NCAL option is similar to the restricted no-call function on the LIBRARY statement, except that the NCAL option negates automatic library call for all unresolved external references and restricted no-call negates automatic library call for selected unresolved external references. With NCAL, all external references that are unresolved after input processing is finished, remain unresolved. The module is however, marked executable.

The NCAL option is a special processing parameter that is specified on the EXEC statement as described in "No Automatic Library Call Option" under "Job Control Language Summary."

Included Data Sets

The INCLUDE control statement requests the linkage editor to use additional data sets as input. These can be sequential data sets containing object modules and/or control statements, or members of partitioned data sets containing object modules and/or control statements, or load modules.

The INCLUDE statement specifies the ddname of a DD statement that describes the data set to be used as additional input. If the DD statement describes a partitioned data set, the INCLUDE statement also contains the name of each member to be used. See "Linkage Editor Control Statement Summary" for a detailed description of the format of the INCLUDE statement.

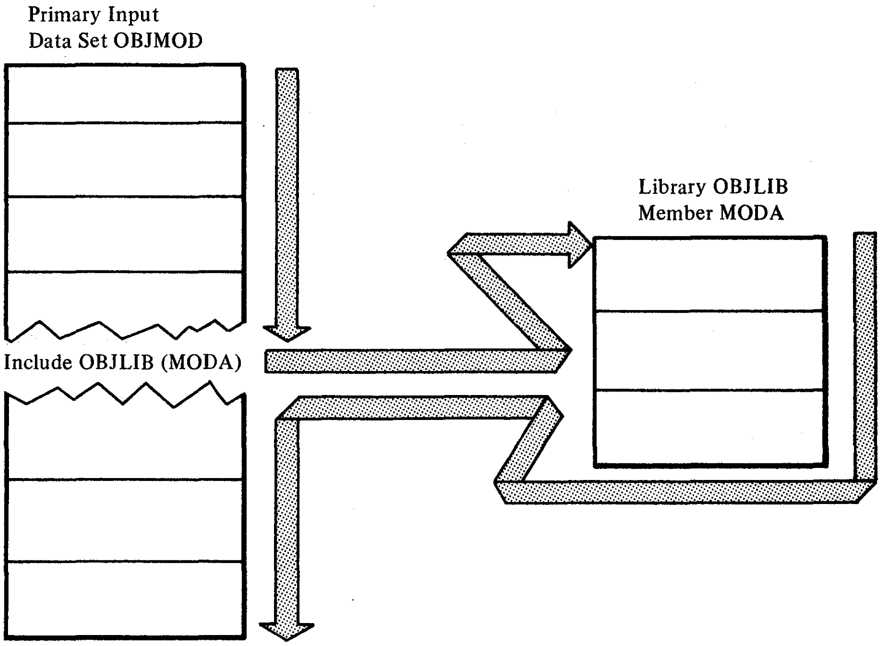


Figure 11. Processing of One INCLUDE Control Statement

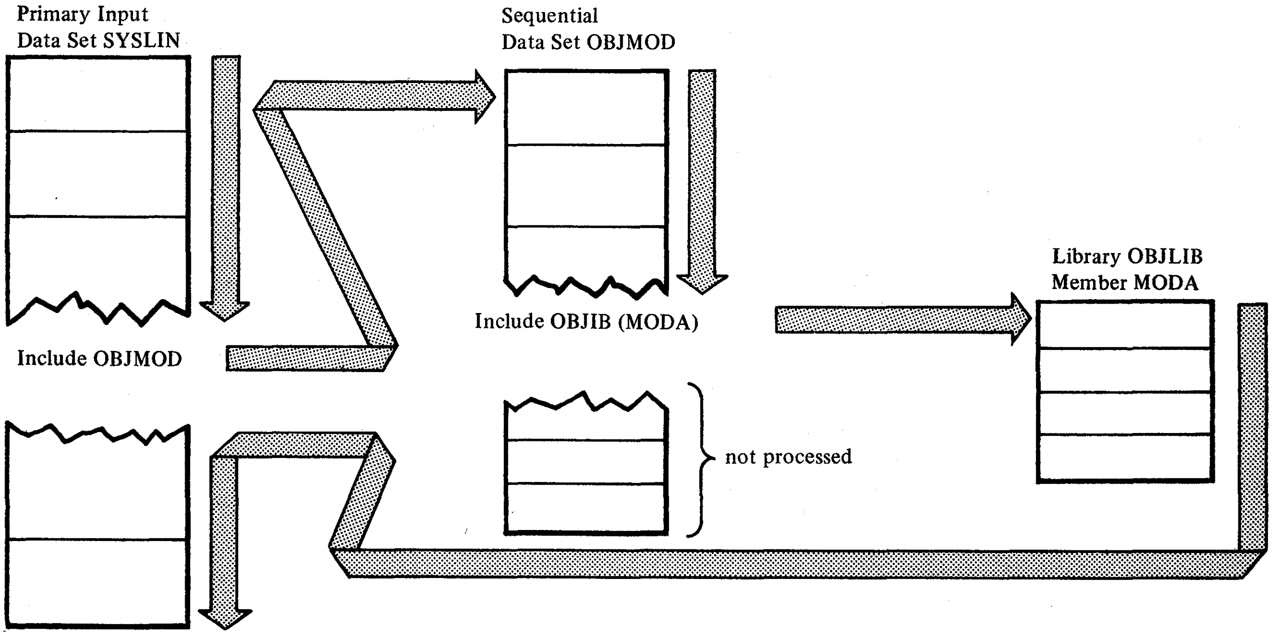


Figure 12. Processing of More than One INCLUDE Control Statement

When an INCLUDE control statement is encountered, the linkage editor processes the module or modules indicated. Figure 11 shows the processing of an INCLUDE statement. In the illustration, the primary input data set is a sequential data set named OBJMOD which contains an INCLUDE statement. After processing the included data set, the linkage editor processes the next primary input item. The arrows indicate the flow of processing.

If an included data set also contains an INCLUDE statement, this specified module is also processed. However, any data following the INCLUDE statement is not processed.

If the OBJMOD data set shown in Figure 11 is itself included, the data following the INCLUDE statement for OBJLIB is not processed. Figure 12 shows the flow of processing for this example.

Including Sequential Data Sets

Sequential data sets containing object modules and/or control statements can be specified by an INCLUDE control statement. In the following example, an INCLUDE statement specifies the ddnames of two sequential data sets to be used as additional input:

```
//ACCOUNTS   DD     DSNAME=ACCTROUT,DISP=(OLD,KEEP),...
//INVENTRY   DD     DSNAME=INVENTRY,DISP=(OLD,KEEP),...
//SYSLIN     DD     DSNAME=QTREND,...
//           DD     *
  INCLUDE ACCOUNTS,INVENTRY
/*
```

Each ddname could also have been specified on a separate INCLUDE statement; with either method, a DD statement must be specified for each ddname.

Another method of doing the preceding example is given in "Including Concatenated Data Sets."

Including Library Members

One or more members of a partitioned data set can be specified on an INCLUDE control statement. The member name must be specified on the INCLUDE statement; no member name should appear on the DD statement itself.

In the following example, one member name is specified on the INCLUDE statement:

```
//PAYROLL    DD     DSNAME=PAYROUTS,DISP=(OLD,KEEP),...
//SYSLIN     DD     DSNAME=&&CHECKS,DISP=(OLD,DELETE)
//           DD     *
  INCLUDE    PAYROLL(FICA)
/*
```

If more than one member of a partitioned data set is to be included, the INCLUDE statement specifies all the members to be used from each library. The member names are not repeated on the DD statement.

In the following example, an INCLUDE statement specifies two members from each of two libraries to be used as additional input:

```
//PAYROLL    DD     DSNAME=PAYROUTS,DISP=(OLD,KEEP),...
//ATTEND     DD     DSNAME=ATTROUTS,DISP=(OLD,KEEP),...
//SYSLIN     DD     *
  INCLUDE    PAYROLL(FICA,TAX),ATTEND(ABSENCE,OVERTIME)
/*
```

Each library could have been specified on a separate INCLUDE statement; with either method, a DD statement must be specified for each ddname.

Another method of doing this example is given in "Including Concatenated Data Sets."

Including Concatenated Data Sets

Several data sets can be designated as input with one INCLUDE statement that specifies one ddname; additional data sets are then concatenated to the data set described on the specified DD statement. When data sets are concatenated, all of the records must have the same characteristics (that is, format, record length, block size, etc.).

Sequential Data Sets: In the following example, two sequential data sets are concatenated and then specified as input with one INCLUDE statement:

```
//CONCAT      DD      DSNAME=ACCTROUT,DISP=(OLD,KEEP),...
//            DD      DSNAME=INVENTRY,DISP=(OLD,KEEP),...
//SYSLIN      DD      DSNAME=SALES,DISP=OLD,...
//            DD      *
  INCLUDE      CONCAT
/*
```

When the INCLUDE statement is recognized, the contents of the sequential data sets ACCTROUT and INVENTRY are processed.

Library Members: Members from more than one library can be designated as input with one ddname on an INCLUDE statement. In this case, all the members are listed on the INCLUDE statement; the partitioned data sets are concatenated using the ddname from the INCLUDE statement:

```
//CONCAT      DD      DSNAME=PAYROUTS,DISP=(OLD,KEEP),...
//            DD      DSNAME=ATTROUTS,DISP=(OLD,KEEP),...
//SYSLIN      DD      DSNAME=REPORT,DISP=OLD,...
//            DD      *
  INCLUDE     CONCAT(FICA,TAX,ABSENCE,OVERTIME)
/*
```

When the INCLUDE statement is recognized, the two libraries PAYROUTS and ATTROUTS are searched for the four members; the members are then processed as input.

OUTPUT FROM THE LINKAGE EDITOR

The linkage editor produces two types of output: a load module and diagnostic information. The principal output of the linkage editor is the output load module. The linkage editor always places this load module in a partitioned data set. In addition, the linkage editor issues diagnostic information. Error and/or warning messages, module disposition data, and optional diagnostic output are stored in the diagnostic output data set.

Output Load Module

The linkage editor produces one or more load modules (see Appendix G) from the input processed. When more than one load module is produced, the process is called *multiple load module processing*.

Whether or not the linkage editor produces one or more load modules, the following apply:

- The load module is stored in a partitioned data set called the *output module library*.
- The load module must have an entry point; if the programmer has not assigned one, the linkage editor does.
- The output load module is assigned an authorization code.
- During processing, the linkage editor reserves and collects common areas, as specified in the source language program.
- During processing, the linkage editor accumulates total length and individual displacements for each pseudo register (external dummy section).
- During processing, the linkage editor collects and records identification data in the CSECT Identification (IDR) records.
- During the processing of a load module, the linkage editor deletes any private code (unnamed control section) having a length of zero and any identification data associated with it.

Output Module Library

The linkage editor stores every load module it produces in the output module library. This library is a partitioned data set that must be described by a DD statement with the name SYSLMOD. The data set name of the library is also specified on this DD statement. The data set can be either temporary (defined with a double ampersand), or permanent (defined without a double ampersand). If the data set name is either SYS1.LINKLIB or SYS1.SVCLIB, it would be advisable to re-IPL the system after linkage editor processing is complete. This ensures that the corresponding Data Extent Block (DEB) is updated to reflect additional extents if secondary allocation of direct-access space was required.

Whether the data set is permanent or temporary, each module must be assigned a unique name, called the *member name*, to distinguish one load module from another. The output module can be assigned *aliases* if the programmer wants the module either identified by more than one name or entered for execution at several different points. Each member name and alias in a load module library must be unique. The library member name and aliases for each load module appear as separate entries in the library directory, along with the module attributes. (Some module attributes can be assigned on the EXEC statement for each linkage editor job step; see "Module Attributes" in "Job Control Language Summary.")

Member Name

The member name of the output load module may be specified either on the SYSLMOD DD statement, in a NAME statement, or both. If the member name is not specified, the default is TEMPNAME. If this default name has been previously assigned to a load module, using it again will cause a failure.

Assigned on SYSLMOD DD Statement: If the member name is assigned on the SYSLMOD DD statement, the name is written in parentheses following the data set name of the library. For example:

```
//SYSLMOD      DD      DSNAME=MATHLIB(SQDEV),DISP=(NEW,KEEP),
//                     UNIT=2314,SPACE=(TRK,(100,10,1)),
//                     VOLUME=SER=LIB002
```

The member name SQDEV is assigned to the load module, which is placed in the new library named MATHLIB.

Assigned on NAME Control Statement: If the member name is not specified on the SYSLMOD DD statement, it may be assigned in a NAME control statement. For example:

```
//SYSLMOD      DD      DSNAME=MATHLIB,DISP=(NEW,KEEP),...
//SYSLIN       DD      DSNAME=&&OBJECT,DISP=(OLD,DELETE)
//             DD      *
  NAME         SQDEV
/*
```

The member name SQDEV is assigned to the load module, which is placed in the library named MATHLIB.

Assigned on Both: If both the SYSLMOD DD statement and the NAME control statement specify a member name, the names should be identical. If the names are different, the name on the NAME control statement is used as the member name.

Note: If a "link-edit and go" sequence of job steps is performed and the program name in the EXEC statement of the "go" step contains a backward reference to the SYSLMOD DD statement in the "link-edit" step, the user must ensure that the member name specified in the SYSLMOD DD statement is valid and is not overridden by a NAME control statement. For example:

```
//LKED         EXEC    PGM=HEWL
               .
               .
               .
//SYSLMOD      DD      DSNAME=&&LOADST(GO),DISP=(NEW,
//                     PASS),...
//SYSLIN       DD      DSNAME=&&OBJECT,DISP=(OLD,DELETE)
//             DD      *
  NAME         READ
/*
//GO           EXEC    PGM=*.LKED.SYSLMOD
               .
               .
               .
```

The EXEC statement of the GO step specifies that the module to be executed is described in the LKED step in the SYSLMOD statement. The system tries to locate a member named GO; however, the output module was assigned the name READ.

Replacing an Identically Named Library Member: An output module can replace an identically named member in the library in either of two ways. The SYSLMOD DD statement names an existing data set, as follows:

```
//SYSLMOD     DD       DSNAME=MATHLIB(SQDEV),DISP=(OLD,
//                     KEEP),...
```

Or, the NAME control statement specifies the replace function, as follows:

```
  NAME        SQDEV(R)
```

In either case, the member named SQDEV is replaced with a new module of the same name.

Alias Names

An output module can be assigned a maximum of 16 aliases, specified with the ALIAS control statement. The aliases exist in addition to the member name of the output module. When a module is referred to by an alias, execution begins at the external name specified by the alias. If the name specified by the ALIAS statement is not an external symbol within the module, the main entry point is used.

For example, an output module is to be assigned two additional entry points, CODE1 and CODE2. In addition, due to a misunderstanding, calling modules have been written and tested using both ROUTONE and ROUT1 to refer to the output module. Rather than correct the calling modules, an alternate library member name (alias) is also assigned.

```
//SYSLMOD     DD       DSNAME=PVTLIB,DISP=OLD,UNIT=2314,
//                     VOLUME=SER=LIB001
//SYSLIN      DD       DSNAME=&&OBJECT,DISP=(OLD,DELETE)
//            DD       *
  ALIAS       CODE1,CODE2,ROUTONE
  NAME        ROUT1
/*
```

The names CODE1, CODE2, and ROUTONE appear in the library directory along with ROUT1, the member name. Because CODE1 and CODE2 are defined as external symbols within the output module, when these names are used, execution begins at these points. Control may be passed to the main entry point by using either the member name ROUT1 or the alias ROUTONE.

Entry Point

Every load module must have a main entry point. The programmer may specify the entry point in one of two ways:

- On a linkage editor ENTRY control statement.
- On an Assembler language END statement, which is the last statement in the source program. The assembler produces an object module and an END statement for the module. The assembler-produced END statement contains an entry point only if the source language END statement contained one.

From its input, the linkage editor selects the entry point for the load module as follows:

1. From the first ENTRY control statement in the input.
2. If there is no ENTRY control statement in the input, from the first assembler-procuced END statement that specifies an entry point.
3. If no ENTRY control statement or no assembler-produced END statement specifies an entry point, the first byte of the first control section of the load module is used as the entry point.

In general, the entry point should be explicitly specified because it is not always possible to predict which control section will be first in the output module.

When a load module is reprocessed by the linkage editor, it has no END statement. Therefore, if the first byte of the first control section of the load module is not a suitable entry point, the entry point must be specified in one of two ways:

- Through an ENTRY control statement.
- Through the assembler-produced END statement of another input module, which is being processed for the first time. This object module must be the first such module to be processed by the linkage editor.

Entry points other than the main entry point may be specified with an ALIAS control statement. The symbol specified on the ALIAS statement must be defined as an external symbol in the load module. Any reference to that symbol causes execution of the module to begin at that point instead of the main entry point.

In the following example, assume that CDCHECK, CODE1, and CODE2 are defined as external symbols in the output module:

```
//SYSLIN      DD      DSNAME=&&OBJECT,DISP=(OLD,DELETE)
//            DD      *
  ENTRY CDCHECK
  ALIAS CODE1,CODE2,ROUTONE
  NAME ROUT1
/*
```

As a result of the preceding control statements, CDCHECK is the main entry point; CODE1 and CODE2 are additional entry points. Any reference to ROUTONE or ROUT1 causes execution to begin at CDCHECK; any reference to CODE1 and CODE2 causes execution to begin at these points.

Authorization Code

Each load module link edited is assigned an authorization code that determines whether or not the module is allowed to use restricted system services and resources. A non-zero code allows the module to use restricted services and resources, and a zero code disallows that usage. The authorization code becomes part of the directory entry for the module in the library containing the module.

Reserving Storage in the Output Load Module

In FORTRAN, Assembler language, and PL/I, the programmer can create control sections that reserve virtual storage areas that contain no data or instructions. These control sections are called "common" or "static external" areas, and are produced in the object modules by the language translators. These common areas are used, for example, as communication regions for different parts of a program or to reserve virtual storage areas for data supplied at execution time. These common areas are either named or unnamed (blank).

Collection of Common Areas: During processing, the linkage editor collects common areas. That is, if two or more blank common areas are found in the input, the largest blank common area is used in the output module; all references to a blank common area refer to the one retained. If two or more named common areas have the same name, the largest of the identically named common areas is used in the output module; all references to the named common areas refer to the one area retained.

Identically Named Common Areas and Control Sections: If a control section (as is generated from a BLOCK DATA subprogram in FORTRAN, for example) and a named common area have the same name, the length of the control section must be greater than or equal to the length of the named common area. If the control section is smaller in length than the named common area, a diagnostic message is issued. The control section is regarded as the largest of the common areas processed with that name. All subsequent control sections and/or common areas with the same name are ignored.

Processing Pseudo Registers

In PL/I, programmers can use pseudo registers to define storage that will not be reserved in the load module but can be allocated dynamically during execution. The external dummy sections generated by Assembler F or Assembler H correspond to the pseudo registers of PL/I.

The linkage editor accumulates the total length of all pseudo registers in the input and records the displacement of each. If two or more pseudo registers have the same name, the one with the longest length and the most restrictive alignment will be retained. All other pseudo registers with the same name will be ignored; all references to the identically named pseudo registers will refer to the one retained.

Multiple Load Module Processing

The linkage editor can produce more than one load module in a single job step. A NAME control statement in the input stream is used as a delimiter for input to a load module. If additional input modules follow the NAME statement in the input stream, they are used in the formation of the next load module.

Each load module that is formed has a unique name and is placed in the same library as a separate member. When processing multiple load modules in a single job step, the options and attributes specified in the EXEC statement for that job step apply to all load modules created. If the linkage editor terminates abnormally during processing of any of the output modules, neither that module nor any of the modules yet to be processed in the job step is processed or placed in the library. Load modules processed before abnormal termination have already been placed in the library.

In the following example, two load modules are produced in one linkage editor job step:

```
//LKED         EXEC    PGM=HEWL,PARM='MAP,LIST'
                 .
                 .
                 .
//SYSLMOD      DD      DSNAME=PAYROLL(OVERTIME),DISP=OLD,
//                     UNIT=2314,VOLUME=SER=LIB002
                 .
                 .
                 .
//MODTWO       DD      DSNAME=&&OBJECT,DISP=(OLD,DELETE)
//SYSLIN       DD      DSNAME=&&OBJECT(A),DISP=(OLD,DELETE)
//             DD      *
  ENTRY        INIT
  NAME         OVERTIME
  INCLUDE      MODTWO(B)
  ENTRY        HSKEEP
  NAME         VACATION
/*
```

The first load module is produced from the object module in the data set defined on the SYSLIN DD statement. The main entry point is INIT and the member name is OVERTIME.

The second load module is produced from the object module specified by the INCLUDE statement. The main entry point is HSKEEP and the member name is VACATION.

If an INCLUDE statement specifies a member name that is different from the member name on the DD statement, the member specified on the DD statement must exist even though it is not to be included.

Both load modules are placed in the library PAYROLL, defined on the SYSLMOD statement.

The parameters on the EXEC card specify that a module map and a control statement listing is produced for each load module. The map and listing are discussed in detail in the next section.

Diagnostic Output

Diagnostic information is stored in the diagnostic output data set, which must be defined by a DD statement with the name SYSPRINT. This output is a collection of messages generated by the linkage editor, as well as any optional output requested by the programmer.

Diagnostic Messages

The linkage editor generates two types of messages: module disposition messages and error/warning messages. Descriptions of the error/warning messages can be found in *OS/VS Message Library: Linkage Editor and Loader Messages*.

Module Disposition Messages

Module disposition messages of several types are printed for each load module produced. The first message indicates the options and attributes specified for each module. Invalid options or attributes are replaced by INVALID in the output. Messages are also generated to inform the programmer that incompatible attributes have been specified.

Disposition messages also describe the handling of the load module. These messages are preceded by several asterisks, and are:

- *member name* NOW ADDED TO DATA SET.
- *member name* NOW REPLACED IN DATA SET.
- *member name* DOES NOT EXIST BUT HAS BEEN ADDED TO THE DATA SET.

  The replacement function was specified, but the member did not exist in the data set; the module is added to the data set using the member name given.
- *alias name* IS AN ALIAS FOR THIS MEMBER.
- MODULE HAS BEEN MARKED NOT EXECUTABLE.

In addition, module disposition messages are used when the re-enterable (RENT), reusable (REUS), and/or refreshable (REFR) linkage editor options have been specified for the module. When one or more of these module attributes has been indicated, a message informs the user what attribute(s) have been assigned to the module. This message indicates whether the load module has been marked re-enterable or not re-enterable, reusable or not reusable, refreshable or not refreshable, depending on the option or options used. (See "Reusablity Attributes" and "Refreshable Attribute" in the Job Control Language Summary section for more information on these options.)

The message consists of several asterisks and MODULE HAS BEEN MARKED, followed by the attribute(s) assigned as a result of the linkage editor options specified. The programmer, of course, is responsible for verifying that the module actually is re-enterable, reusable, and/or refreshable. The following messages are examples of some possible combinations:

- MODULE HAS BEEN MARKED REFRESHABLE.
- MODULE HAS BEEN MARKED NOT REFRESHABLE.
- MODULE HAS BEEN MARKED REUSABLE AND NOT REFRESHABLE.
- MODULE HAS BEEN MARKED REUSABLE AND REFRESHABLE.

When an error causes the linkage editor to mark a module not executable, only the MODULE HAS BEEN MARKED NOT EXECUTABLE message appears; no attribute messages are generated.

Error/Warning Messages

Certain conditions that are present when a module is being processed can cause an error or warning message to be printed. These messages contain a message code and message text. If an error is encountered during processing, the message code for that error is printed with the applicable symbol or record in error. After processing is completed, the diagnostic message associated with that code is printed. The error warning messages have the following format:

```
IEW0mms  message text
```

where:

`IEW0` indicates a linkage editor message

`mm` is the message number

`s` is the severity code, and may be one of the following values:

1 Indicates a condition that may cause an error during execution of the output module. A module map or cross-reference table is produced if specified by the programmer. The output module is marked executable.

2 Indicates an error that could make execution of the output module impossible. Processing continues. When possible, a module map or cross-reference table is produced if specified by the programmer. The output module is marked not executable unless the LET option is specified on the EXEC statement.

3 Indicates an error that will make execution of the output module impossible. Processing continues. When possible, a module map or cross-reference table is produced if specified by the programmer. The output module is marked not executable.

4 Indicates an error condition from which no recovery is possible. Processing terminates. The only output is diagnostic messages.

Note: A special severity code of zero is generated for each control statement printed as a result of the LIST option. Severity zero does not indicate an error or warning condition.

The highest severity code encountered during processing is multiplied by 4 to create a return code that is placed in register 15 at the end of processing. This return code can be tested to determine whether or not processing is to continue (see "Job Control Language Summary").

`message text` contains combinations of the following:

- The message classification (either error or warning).
- Cause of error.
- Identification of the symbol, segment number (when in overlay), or input item to which the message applies.
- Instructions to the programmer.
- Action taken by the linkage editor.

Optionally, error/warning messages can be sent to a separate output data set, which is defined by specifying TERM in the PARM field of the EXEC statement and including a SYSTERM DD statement. This separate SYSTERM data set consists of only numbered error/warning messages. It supplements the SYSPRINT output data set, which can also include module disposition messages and optional diagnostic output. When SYSTERM is used, the numbered error/warning messages appear in both data sets.

OS/VS Message Library: Linkage Editor and Loader Messages contains a complete list of error/warning messages.

Sample Diagnostic Output

Figure 13 shows the format of the diagnostic output for the linkage editor. No optional output was requested other than the list of control statements.

The letters indicate the disposition and error/warning messages as follows:

A Is a module disposition message that lists the options and attributes specified. Additional information is printed indicating the variable and default options used.

B Is a list of control statements used (IEW0000) and the message codes (IEW0201 and IEW0461) for error/warning conditions discovered during processing. For error/warning message codes, the symbol in error, if necessary, is also listed (CCCCCCCC and BASEDUMP).

C Is a module disposition message (****) that indicates that the output module (BBBBBBBB) has been added to the output module data set.

D Is the diagnostic message directory that contains the text of the error codes listed in item **B**.

Optional Output

In addition to error/warning and disposition messages, the linkage editor can produce diagnostic output as requested by the programmer. This optional output includes a control statement listing, a module map, and a cross-reference table.

Control Statement Listing

If the LIST option is specified on the EXEC statement, a listing of all linkage editor control statements is produced. For each control statement, the listing contains a special message code, `IEW0000`, followed by the control statement. Item **B** in Figure 13 contains an example of a control statement listing.

Module Map

If the MAP option is specified on the EXEC statement, a module map of the output load module is produced. The module map shows all control sections in the output module and all entry names in each control section. Named common areas are listed as control sections.

For each control section, the module map indicates its origin (relative to zero) and length in bytes (in hexadecimal notation). For each entry name in each control section, the module map indicates the location at which the name is defined. These locations are also relative to zero.

```
A             F64-LEVEL LINKAGE EDITOR OPTIONS SPECIFIED LET,NCAL,XREF,OVLY,LIST
                        DEFAULT OPTION(S) USED - SIZE=(65536,6144)
B             IEW0000      NAME BBBBBBBB
               IEW0201
               IEW0461   CCCCCCCC
               IEW0461   BASEDUMP
C             ****BBBBBBBB NOW ADDED TO DATA SET
                                                      DIAGNOSTIC MESSAGE DIRECTORY

D                IEW0201 WARNING - OVERLAY STRUCTURE CONTAINS ONLY ONE SEGMENT -- OVERLAY OPTION
                          CANCELED.
                  IEW0461 WARNING - SYMBOL PRINTED IS AN UNRESOLVED EXTERNAL REFERENCE, NCAL WAS
                          SPECIFIED.
```

Figure 13. Diagnostic Messages Issued by the Linkage Editor

If the module is not in an overlay structure, the control sections are arranged in ascending order according to their origins. An entry name is listed with the control section in which it is defined.

If the module is an overlay structure, the control sections are arranged by segment. The segments are listed as they appear in the overlay structure, top to bottom, left to right, and region by region. Within each segment, the control sections and their corresponding entry names are listed in ascending order according to their assigned origins. The number of the segment in which they appear is also listed.

In any module map, the following are identified by a dollar sign:

- Blank common area.
- Private code (unnamed control section).
- For overlay programs, the segment table and each entry table.

When the load module processed by the linkage editor does not have an origin of zero, the linkage editor generates a one-byte private code (unnamed control section) as the first text record. This private code is deleted in any subsequent reprocessing of the load module by the linkage editor.

Each control section that is obtained from a call library during automatic library call is identified by an asterisk after the control section name.

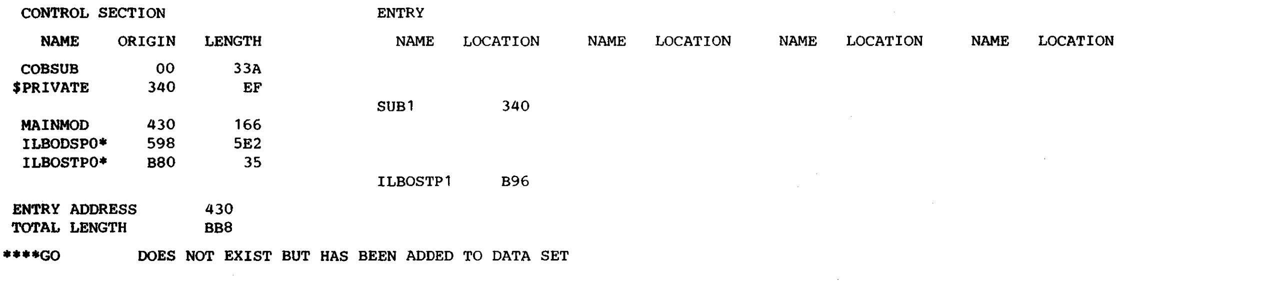
At the end of the module map is the entry address, that is, the relative address of the main entry point. The entry address is followed by the total length of the module in bytes; in the case of an overlay module, the length is that of the longest path. Pseudo registers, if used, also appear at the end of the module map; the name, length, and displacement of each pseudo register is given.

Figure 14 contains a module map with five control sections. There are two named control sections (COBSUB snd MAINMOD), one unnamed control section (designated by $PRIVATE), and two control sections obtained from a call library (ILBODSP0 and ILBOSTPO). In addition, two entry names are defined, SUB1 in the unnamed control section and ILBOSTP1 in control section ILBOSTPO.

Note: The HMBLIST service aid program described in the *OS/VS1 Service Aids* publication can also be used to obtain a module map.

Cross Reference Table

If the XREF option is specified on the EXEC statement, a cross-reference table is produced. The cross-reference table consists of a module map and a list of cross-references for each control section. Each address constant that refers to a symbol defined in another control section is listed with its assigned



```
  CONTROL SECTION                             ENTRY

    NAME    ORIGIN   LENGTH                     NAME   LOCATION      NAME   LOCATION      NAME   LOCATION      NAME   LOCATION

  COBSUB        00      33A
 $PRIVATE      340       EF
                                              SUB1        340
  MAINMOD      430      166
  ILBODSP0*    598      5E2
  ILBOSTPO*    B80       35
                                              ILBOSTP1     B96

 ENTRY ADDRESS        430
 TOTAL LENGTH         BB8

****GO        DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET
```

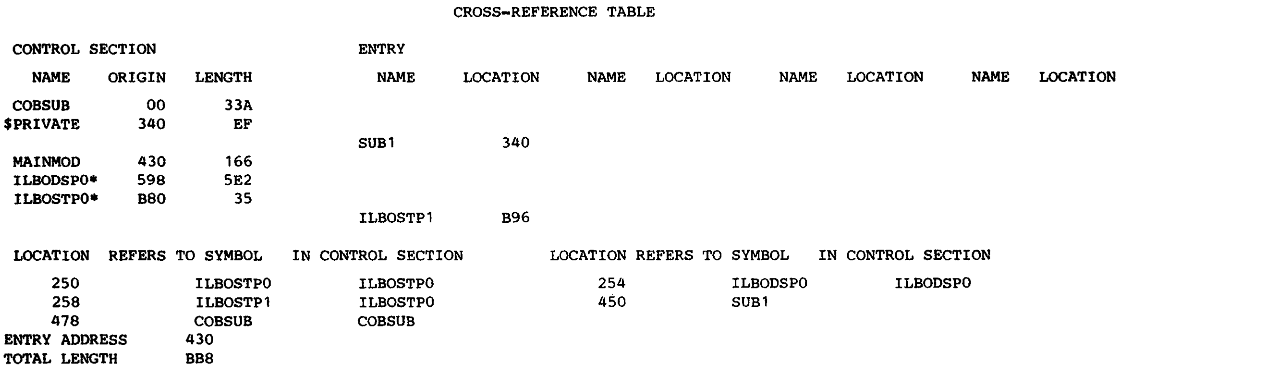
Figure 14. Module Map

another (instead of using external symbols and entry names) the control section name is listed as the symbol referred to.

For overlay programs, this information is provided for each segment; in addition, the number of the segment in which the symbol is defined is provided.

If a symbol is unresolved after processing by the linkage editor, it is identified by $UNRESOLVED in the list. However, if an unresolved symbol is marked by the never-call function (as specified on a LIBRARY control statement), it is identified by $NEVER-CALL. If an unresolved symbol is a weak external reference, it is identified by $UNRESOLVED(W).

Figure 15 contains a cross-reference table for the same program whose module map is shown in Figure 14. All of the information from the module map is present, plus a list of cross references for each control section.



```
                                              CROSS-REFERENCE TABLE

 CONTROL SECTION                    ENTRY

   NAME    ORIGIN  LENGTH             NAME     LOCATION     NAME   LOCATION     NAME   LOCATION     NAME   LOCATION

 COBSUB       00     33A
$PRIVATE     340      EF
                                    SUB1         340

 MAINMOD     430     166
 ILBODSPO*   598     5E2
 ILBOSTPO*   B80      35
                                    ILBOSTP1     B96

 LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION          LOCATION REFERS TO SYMBOL  IN CONTROL SECTION

     250            ILBOSTPO        ILBOSTPO                   254          ILBODSPO          ILBODSPO
     258            ILBOSTP1        ILBOSTPO                   450          SUB1
     478            COBSUB          COBSUB
ENTRY ADDRESS      430
TOTAL LENGTH       BB8
```

Figure 15. Cross-Reference Table

MODULE EDITING

The linkage editor performs editing functions either automatically or as directed by control statements. These editing functions provide for program modification on a control section basis. That is, they make it possible to modify a control section within an object or load module, without recompiling the entire source program.

The editing functions can modify either an entire control section or external symbols within a control section. Control sections can be deleted, replaced, or arranged in sequence; external symbols can be deleted or changed. (External symbols are control section names, entry names, external references, named common areas, or pseudo registers.)

Whatever function is used, it is requested in reference to an *input* module. The resulting output load module reflects the request. That is, no actual change, deletion, or replacement is made to an input module. The requested alterations are used to control linkage editor processing (Figure 16).

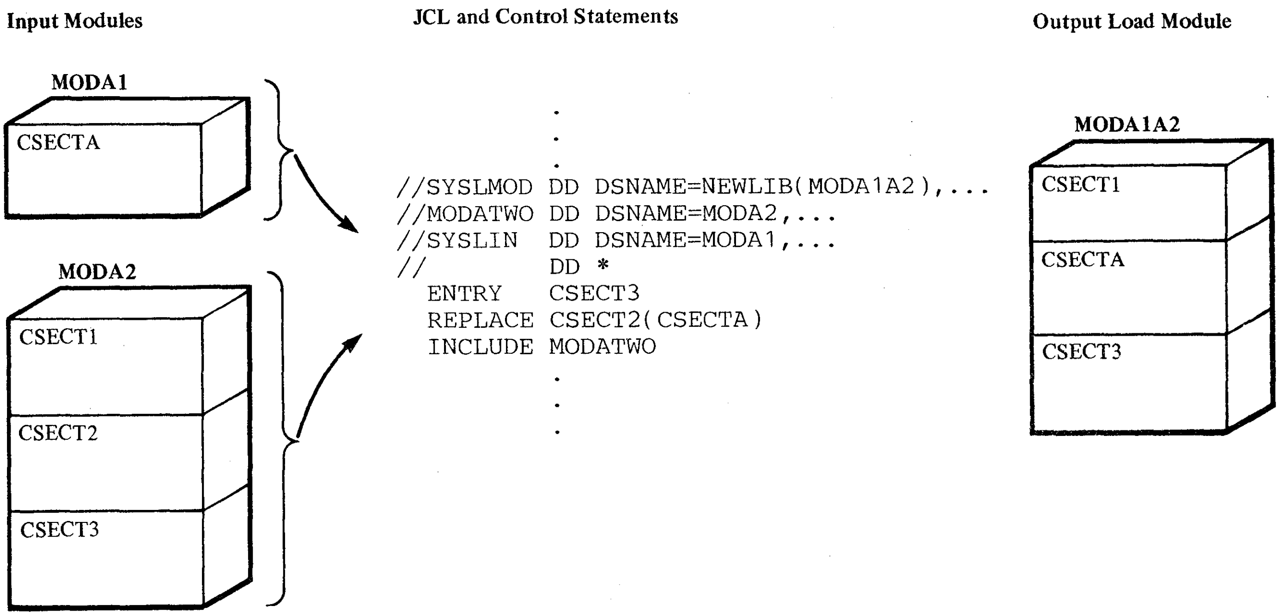
Editing Conventions

In requesting editing functions, certain conventions should be followed to ensure that the specified modification is processed correctly. These conventions concern the following items:

- Entry points for the new module.
- Placement of control statements.
- Identical old and new symbols.

Entry Points: Each time the linkage editor reprocesses a load module, the entry point for the output module should be specified in one of two ways:

- Through an ENTRY control statement.



```
                 .
                 .
                 .
//SYSLMOD DD DSNAME=NEWLIB(MODA1A2),...
//MODATWO DD DSNAME=MODA2,...
//SYSLIN  DD DSNAME=MODA1,...
//        DD *
  ENTRY   CSECT3
  REPLACE CSECT2(CSECTA)
  INCLUDE MODATWO
                 .
                 .
                 .
```

Figure 16. Editing a Module

- Through the assembler-produced END statement of an input object module, if one is present. If the entry point specified in the assembler-produced END statement is not defined in the object module, the entry name must be defined as an external reference.

The entry point assigned must be defined as an external name within the resulting load module.

Placement of Control Statements: The control statement (such as CHANGE or REPLACE) used to specify an editing function must precede either the module to be modified, or the INCLUDE statement that specifies the module. If an INCLUDE statement specifies several modules, the CHANGE or REPLACE statement applies only to the first module included.

Identical Old and New Symbols: The same symbol should not appear as both an old external symbol and a new external symbol in one linkage editor run. If a control section is to be replaced by another control section with the same name, the linkage editor handles this automatically (see "Automatic Replacement").

Changing External Symbols

The linkage editor can be directed to change an external symbol to a new symbol while processing an input module. External references and address constants within the module automatically refer to the new symbol. External references from other modules to a changed external symbol must be changed with separate control statements.

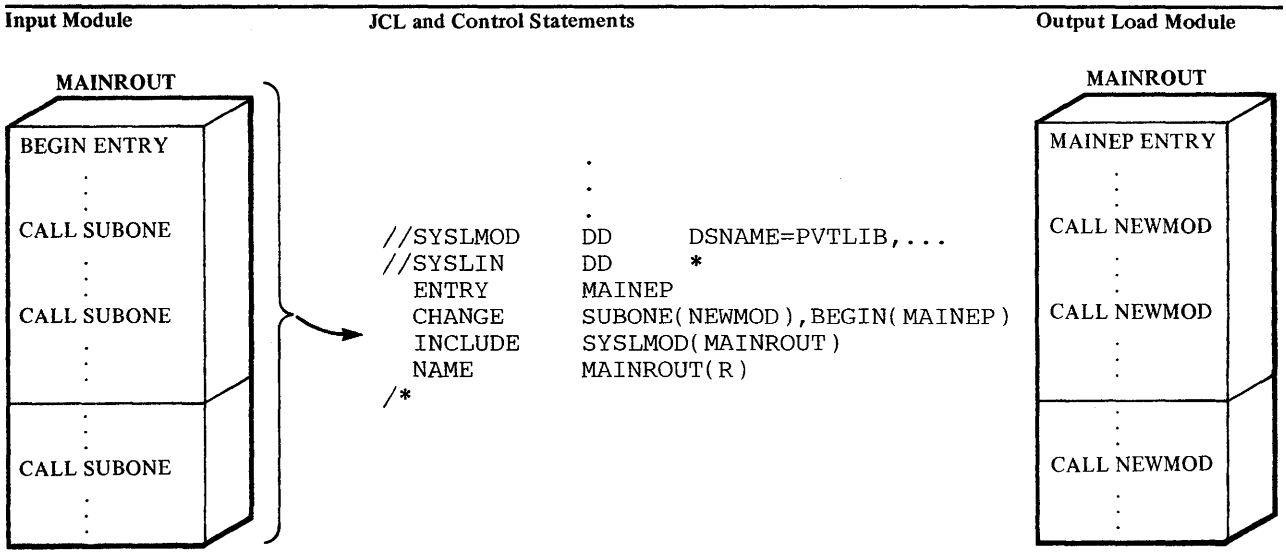
Both the old and the new symbols are specified on either a CHANGE control statement or a REPLACE control statement. The use of the old symbol within the module determines whether the new symbol becomes a control section name, an entry name, or an external reference. The old symbol appears first, followed by the new symbol in parentheses.

The CHANGE control statement changes a control section name, an entry name, or an external reference. The REPLACE statement changes or deletes an entry name; if the symbols on a REPLACE statement are control section names, the entire control section is replaced or deleted (see "Replacing Control Sections").

The CHANGE statement must immediately precede either the input module that contains the external symbol to be changed, or the INCLUDE statement that specifies the input module. The scope of the CHANGE statement is across the immediately following module (object module or load module). The END record in the immediately following object module or the end-of-module indication in the load module terminates the action of the CHANGE statement.

In the following example, assume that SUBONE is defined as an external reference in the input load module. A CHANGE statement is used to change the external reference to NEWMOD (Figure 17).

```
//SYSLMOD      DD       DSNAME=PVTLIB,DISP=OLD,UNIT=2314,
//                      VOLUME=SER=PVT002
//SYSLIN       DD       *
  ENTRY        BEGIN
  CHANGE       SUBONE(NEWMOD)
  INCLUDE      SYSLMOD(MAINROUT)
  NAME         MAINROUT(R)
/*
```



```
            .
            .
            .
//SYSLMOD   DD       DSNAME=PVTLIB,...
//SYSLIN    DD       *
  ENTRY     MAINEP
  CHANGE    SUBONE(NEWMOD),BEGIN(MAINEP)
  INCLUDE   SYSLMOD(MAINROUT)
  NAME      MAINROUT(R)
/*
```

Figure 17. Changing an External Reference and an Entry Point

In the load module MAINROUT, every reference to SUBONE is changed to NEWMOD. Note also that the INCLUDE statement specifies a ddname of SYSLMOD. This allows a library to be used both as input and as the output module library.

More than one change can be specified on the same control statement. If, in the same example, the entry point is also to be changed, the two changes can be specified at once (Figure 17).

```
//SYSLMOD   DD       DSNAME=PVTLIB,DISP=OLD,UNIT=2314,
//                   VOLUME=SER=PVT002
//SYSLIN    DD       *
  ENTRY     MAINEP
  CHANGE    SUBONE(NEWMOD),BEGIN(MAINEP)
  INCLUDE   SYSLMOD(MAINROUT)
  NAME      MAINROUT(R)
/*
```

The main entry point is now MAINEP instead of BEGIN. The ENTRY control statement specifies the new entry point because this is the entry point that is entered in the library directory entry for the load module.

Replacing Control Sections

An entire control section can be replaced with a new control section. Control sections can be replaced either automatically or with a REPLACE control statement. Automatic replacement acts upon all input modules; the REPLACE statement acts only upon the module that follows it.

Notes:

- Any CSECT Identification (IDR) records associated with a particular control section are also replaced.
- (For Assembler language programmers only.) When some but not all control sections of a separately assembled module are to be replaced, A-type address constants that refer to a deleted symbol will be incorrectly resolved unless the entry name is at the same displacement from the origin in both the old and the new control section. If all control sections of a separately assembled module are replaced, no restrictions apply.

Automatic Replacement

Control sections are automatically replaced if both the old and the new control section have the same name. The first of the identically named control sections processed by the linkage editor is made a part of the output module. All subsequent identically named control sections are ignored; external references to identically named control sections are resolved with respect to the first one processed. Therefore, to cause automatic replacement, the new control section must have the same name as the control section to be replaced, and must be processed before the old control section.

Caution: Automatic replacement applies to duplicate control section names only; if duplicate entry points exist in control sections with different names, a REPLACE control statement must be used to specify the entry point name. If a control section being automatically replaced contains unresolved external references and the control section replacing it does not, the parameter NCAL must be specified or the unresolved external references must be explicitly deleted using the REPLACE statement or marked for restricted no-call or never-call using the LIBRARY statement; otherwise, the unresolved external reference is retained.

Note on Overlay Programs: When identically named control sections appear in modules being placed in an overlay structure, the second and any subsequent control sections with that name are ignored. This occurs whether the modules are in segments in the same path or in exclusive segments. Resolution of external references may therefore cause invalid exclusive references. Invalid exclusive references cause the linkage editor to mark the output module not executable unless the XCAL option is specified on the EXEC statement.

Example 1

An object module deck contains two control sections, READ and WRITE; member INOUT of library PVTLIB also contains a control section WRITE.

```
//SYSLMOD    DD      DSNAME=PVTLIB,DISP=OLD,UNIT=2314,
//                   VOLUME=SER=PVT002
//SYSLIN     DD      *
```

Object Deck for READ
Object Deck for WRITE

```
  ENTRY      READIN
  INCLUDE    SYSLMOD(INOUT)
  NAME       INOUT(R)
/*
```

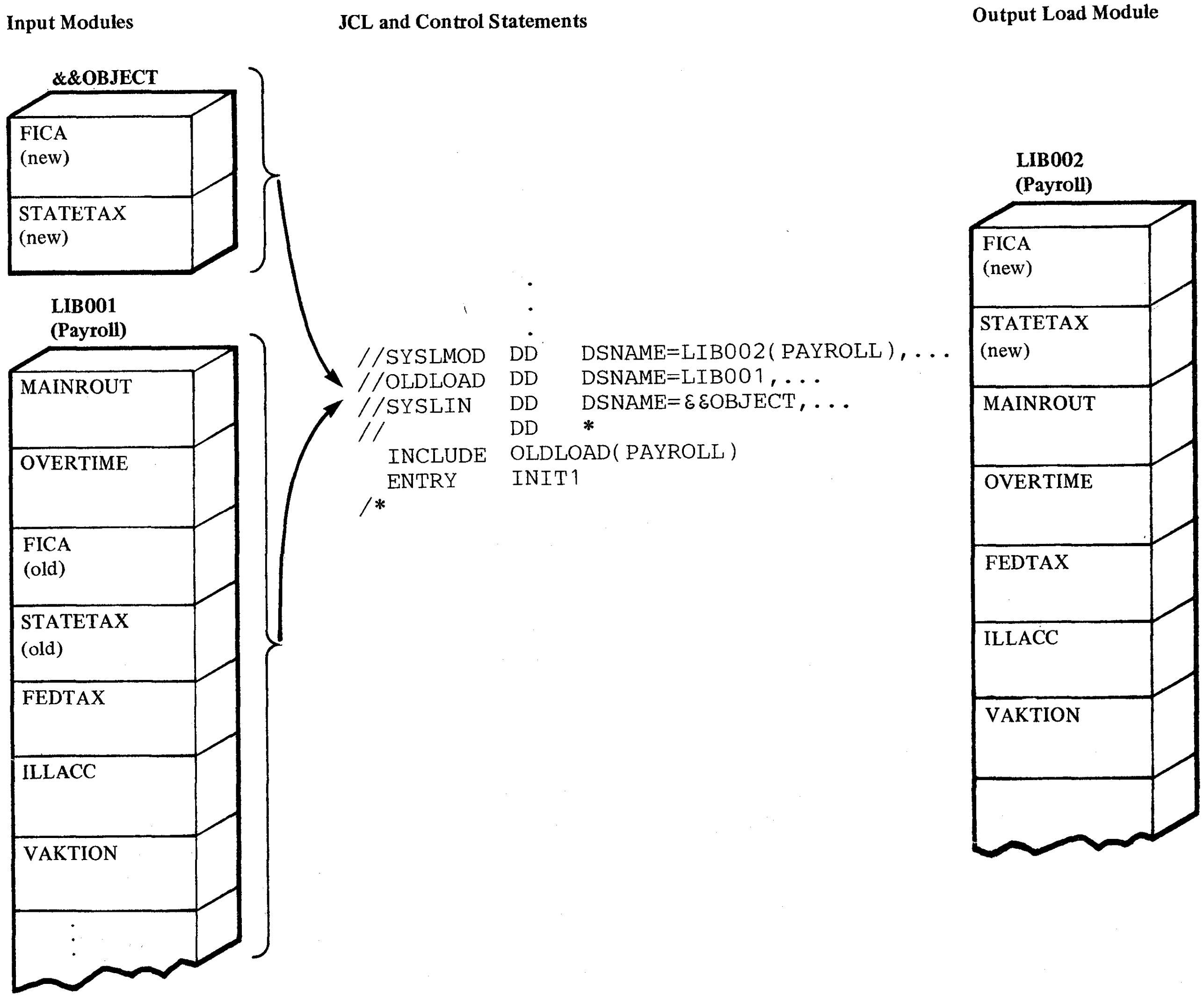
The output load module contains the new READ control section, the new WRITE control section (replacing the old WRITE control section in member INOUT), and all remaining control sections from INOUT.

Example 2

A large load module named PAYROLL, originally written in COBOL, contains many control sections. Two control sections, FICA and STATETAX, were recompiled and passed to the linkage editor job step in the && OBJECT data set. Then, by including the load module PAYROLL, a member of the partitioned data set LIB001, as well as the output of the language translator, the modified control sections automatically replace the identically named control sections (Figure 18).

```
//SYSLMOD DD DSNAME=LIB002(PAYROLL),DISP=OLD,
//           UNIT=2314,VOLUME=SER=LIB002
//SYSLIB  DD DSNAME=SYS1.COBLIB,DISP=SHR
//OLDLOAD DD DSNAME=LIB001,DISP=(OLD,DELETE),
//           UNIT=2314,VOLUME=SER=LIB001
//SYSLIN  DD DSNAME=&&OBJECT,DISP=(OLD,DELETE)
//        DD *
  INCLUDE OLDLOAD(PAYROLL)
  ENTRY   INIT1
/*
```

The output module contains the modified FICA and STATETAX control sections and the rest of the control sections from the old PAYROLL module. The main entry point is INIT1, and the output module is placed in a library named LIB002. The COBOL automatic call library is used to resolve any external references that may be unresolved after the SYSLIN data sets are processed.



```
          .
          .
          .
//SYSLMOD  DD   DSNAME=LIB002(PAYROLL),...
//OLDLOAD  DD   DSNAME=LIB001,...
//SYSLIN   DD   DSNAME=&&OBJECT,...
//         DD   *
  INCLUDE  OLDLOAD(PAYROLL)
  ENTRY    INIT1
/*
```

Figure 18. Automatic Replacement of Control Sections

REPLACE Statement

The REPLACE statement is used to replace control sections when the old and the new control sections have different names. The name of the old control section appears first, followed by the name of the new control section in parentheses. The REPLACE statement must immediately precede either the input module that contains the control section to be replaced, or the INCLUDE statement that specifies the input module. The scope of the REPLACE statement is across the immediately following module (object module or load module). The END record in the immediately following object module or the End-of-Module indication in the load module terminates the action of the REPLACE statement.

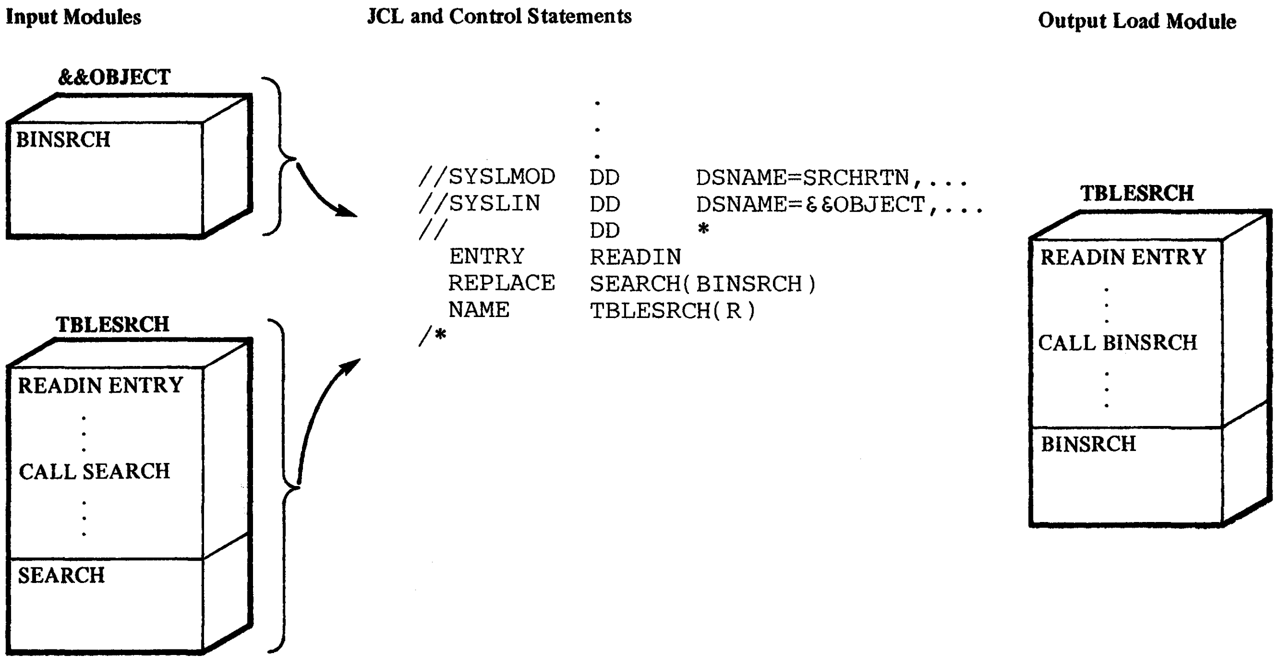
An external reference to the old control section from within the same input module is resolved to the new control section. An external reference to the old control section from any other module becomes an unresolved external reference unless one of the following occurs:

- The external reference to the old control section is changed to the new control section with a separate CHANGE control statement.
- The same entry name appears in the new control section or in some other control section in the linkage editor input.

In the following example, the REPLACE statement is used to replace one control section with another of a different name. Assume that the old control section SEARCH is in library member TBLESRCH, and that the new control section BINSRCH is in the data set && OBJECT, which was passed from a previous step (Figure 19).

```
//SYSLMOD     DD     DSNAME=SRCHRTN,DISP=OLD,UNIT=2314,
//                   VOLUME=SER=SRCHLIB
//SYSLIN      DD     DSNAME=&&OBJECT,DISP=(OLD,DELETE)
//            DD     *
  ENTRY       READIN
  REPLACE     SEARCH(BINSRCH)
  INCLUDE     SYSLMOD(TBLESRCH)
  NAME        TBLESRCH(R)
/*
```

The output module contains BINSRCH instead of SEARCH; any references to SEARCH within the module refer to BINSRCH. Any external references to SEARCH from other modules will not be resolved to BINSRCH.



```
         .
         .
         .
//SYSLMOD   DD      DSNAME=SRCHRTN,...
//SYSLIN    DD      DSNAME=&&OBJECT,...
//          DD      *
   ENTRY    READIN
   REPLACE  SEARCH(BINSRCH)
   NAME     TBLESRCH(R)
/*
```

Figure 19. Replacing a Control Section with the REPLACE Control Statement

Deleting a Control Section or Entry Name

The REPLACE statement can be used to delete a control section or an entry name. The REPLACE statement must immediately precede either the module that contains the control section or entry name to be deleted or the INCLUDE statement that specifies the module. Only one symbol appears on the REPLACE statement; the appropriate deletion is made depending on how the symbol is defined in the module.

If the symbol is a control section name, the entire control section is deleted. The control section name is deleted from the external symbol dictionary only if no address constants refer to the name from within the same input module. If an address constant does refer to it, the control section name is changed to an external record.

The preceding is also true of an entry name to be deleted. Any references to it from within the input module cause the entry name to be changed to an external reference.

These editor-supplied external references, unless resolved with other input modules, cause the automatic library call mechanism to attempt to resolve them. Also, the deletion of a control section or an entry name may cause external references from other input modules to be unresolved. Either condition can cause the output load module to be marked not executable.

If a deleted control section contains an unresolved external reference, the reference remains.

Note: When a control section is deleted, any CSECT Identification data associated with that control section is also deleted.

In the following example, control section CODER is to be deleted (Figure 20).

```
//SYSLMOD     DD     DSNAME=PVTLIB,DISP=OLD,UNIT=2314,
//                   VOLUME=SER=PVT002
//SYSLIN       DD     *
  ENTRY        START1
  REPLACE      CODER
  INCLUDE      SYSLMOD(CODEROUT)
  NAME         CODEROUT(R)
/*
```

The control section CODER is deleted. If no address constants refer to CODER from other control sections in the module, the control section name is also deleted. If address constants refer to CODER, the name is retained as an external reference.

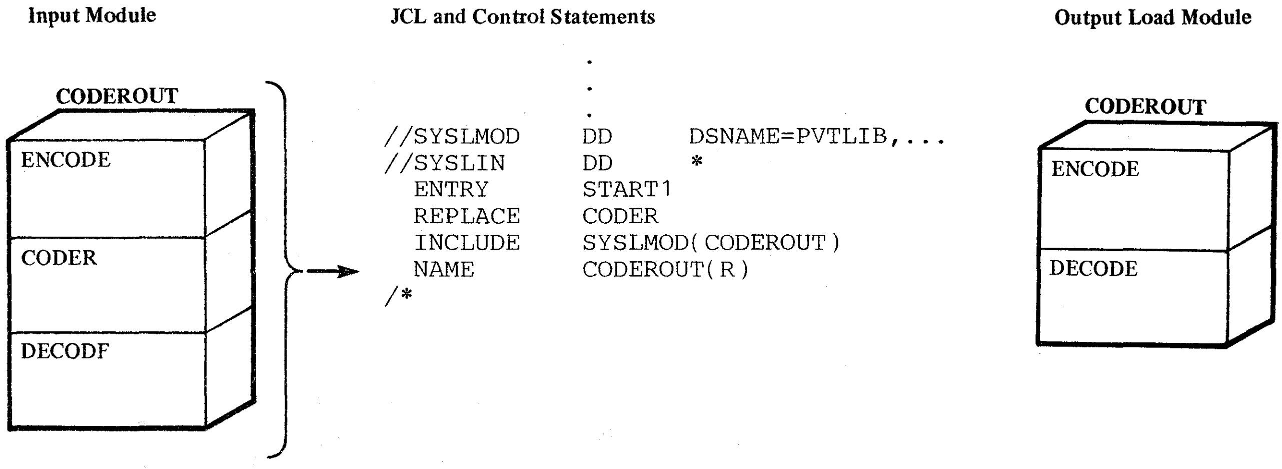
Ordering Control Sections or Named Common Areas

The sequence of control sections or named common areas in an output load module can be specified by using the ORDER control statement.

Individual control sections or named common areas are arranged in the output load module according to the sequence in which they appear on the ORDER control statement. Multiple ORDER control statements can be used in a job step. The sequence of the ORDER statements determines the sequence of the control sections or named common areas in the load module.

Any control sections or named common areas that are not specified on ORDER statements appear last in the output load module. If a control section or named common area is changed by a CHANGE or REPLACE control statement, the new name must be used on the ORDER statement.

In the following example, ORDER statements are used to specify the sequence of five of the six control sections in an output load module. A REPLACE statement is used to replace the old control section SESECTA



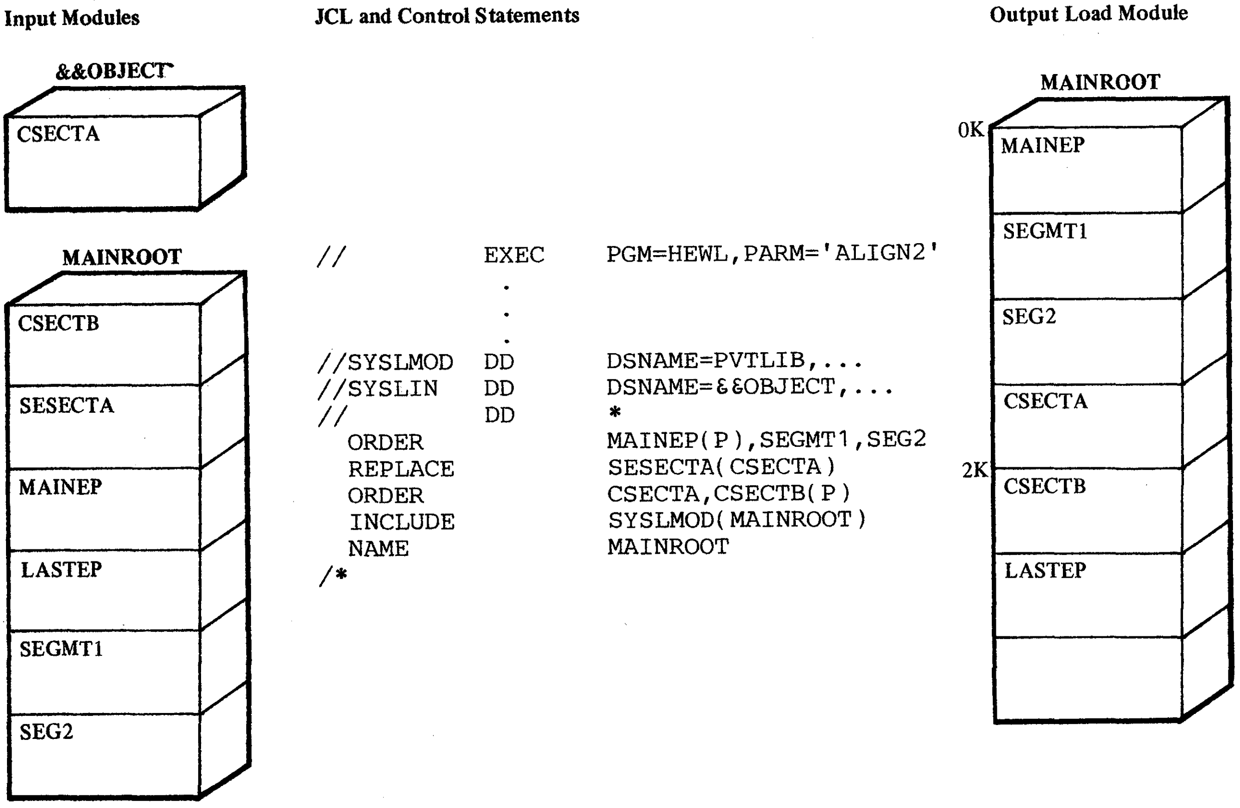
```
                .
                .
                .
//SYSLMOD     DD      DSNAME=PVTLIB,...
//SYSLIN      DD      *
  ENTRY       START1
  REPLACE     CODER
  INCLUDE     SYSLMOD(CODEROUT)
  NAME        CODEROUT(R)
/*
```

Figure 20. Deleting a Control Section

with the new control section CSECTA from the data set &&OBJECT, which was passed from a previous step. Assume that the control sections to be ordered are found in library member MAINROOT (Figure 21).

```
//SYSLMOD     DD    DSNAME=PVTLIB,DISP=OLD,
//                  UNIT=2314,VOLUME=SER=PVT002
//SYSLIN      DD    DSNAME=&&OBJECT,DISP=(OLD,DELETE)
//            DD    *
  ORDER             MAINEP(P),SEGMT1,SEG2
  REPLACE           SESECTA(CSECTA)
  ORDER             CSECTA,CSECTB(P)
  INCLUDE           SYSLMOD(MAINROOT)
  NAME              MAINROOT
/*
```

In the load module MAINROOT, the control sections MAINEP, SEGMT1, SEG2, CSECTA, CSECTB are rearranged in the output load module according to the sequence specified in the ORDER statements. A REPLACE statement is used to replace the control section SESECTA with control section CSECTA from the data set &&OBJECT, which was passed from a previous step. The ORDER statement refers to the new control section CSECTA. Control section LASTEP appears after the other control sections in the output load module because it was not included in the ORDER statement operands.



```
//          EXEC    PGM=HEWL,PARM='ALIGN2'
              .
              .
              .
//SYSLMOD   DD      DSNAME=PVTLIB,...
//SYSLIN    DD      DSNAME=&&OBJECT,...
//          DD      *
  ORDER             MAINEP(P),SEGMT1,SEG2
  REPLACE           SESECTA(CSECTA)
  ORDER             CSECTA,CSECTB(P)
  INCLUDE           SYSLMOD(MAINROOT)
  NAME              MAINROOT
/*
```

Figure 21. Ordering Control Sections

Aligning Control Sections or Named Common Areas on Page Boundaries

A control section or named common area can be placed on a page boundary by using either the ORDER statement (with the P operand) or the PAGE statement. Alignment on a page boundary can be used to effect a lower paging rate and thus make more efficient use of real storage. (Note that page boundary aligning cannot be used for VS1 overlay programs.)

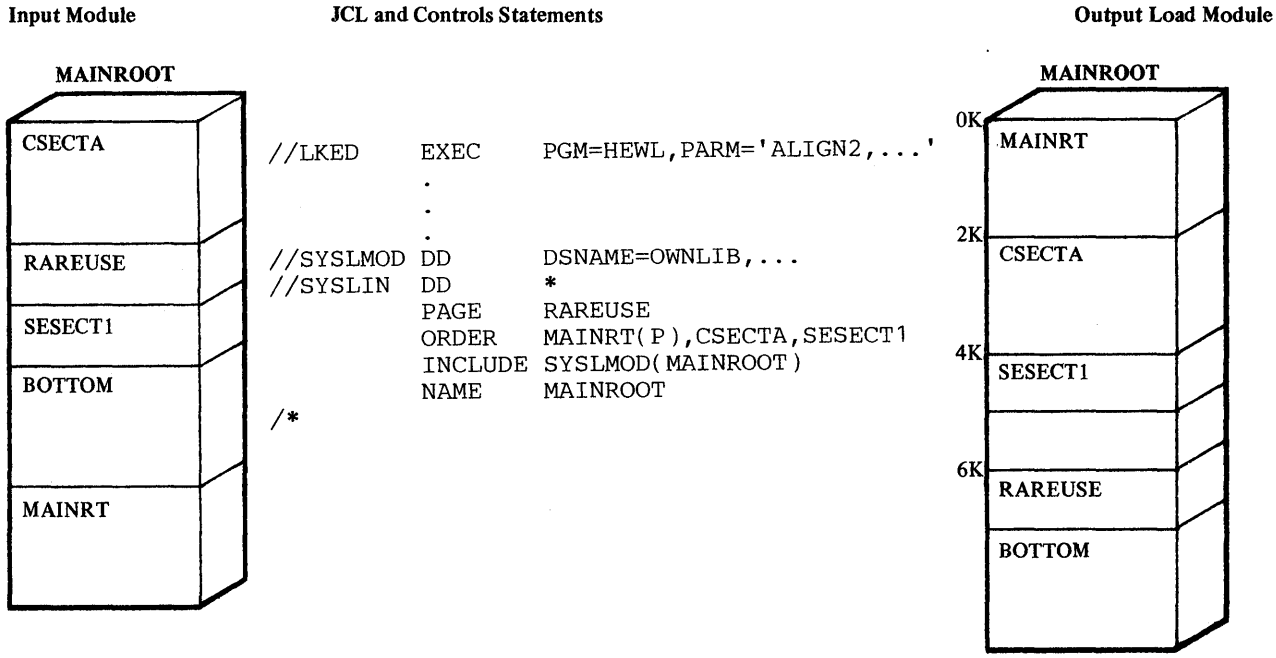
The control section or common area to be aligned is named on either the PAGE statement or the ORDER statement with the P operand. Either the PAGE statement or the ORDER statement (with the P operand) causes the linkage editor to locate the starting address of the control section or common area on a page boundary within the load module.

The default value for the page boundary is 4K. Under VS1, the ALIGN2 attribute must be specified in the PARM field of the EXEC statement to override the default. Because a module using the 2K page boundary alignment may suffer performance degradation if it is moved from a VS1 system to a VS2 system, the 2K page boundary should be used only when virtual storage is limited.

In the following example, the control sections RAREUSE and MAINRT are aligned on 2K page boundaries by PAGE and ORDER control statements used with the ALIGN2 attribute. Control sections CSECTA and SESECT1 are sequenced by the ORDER control statement. Assume that each control section is 2K in length except for SESECT1 and RAREUSE (Figure 22).

```
//LKED        EXEC     PGM=HEWL,PARM='ALIGN2,...'
              .
              .
              .
//SYSLMOD     DD       DSNAME=OWNLIB,DISP=OLD,UNIT=2314,
//                     VOLUME=SER=OWN002
//SYSLIN      DD       *
              PAGE     RAREUSE
              ORDER    MAINRT(P),CSECTA,SESECT1
              INCLUDE  SYSLMOD (MAINROOT)
              NAME     MAINROOT
/*
```

The linkage editor places the control sections MAINRT and RAREUSE on 2K page boundaries because ALIGN2 is specified on the EXEC statement. Control sections MAINRT, CSECTA, and SESECT1 are sequenced as specified in the ORDER statement. RAREUSE, while placed on a 2K page boundary, appears after the control sections specified in the ORDER statement because it was not included. The control section BOTTOM comes after RAREUSE because it appeared after RAREUSE in the input module.



```
//LKED    EXEC     PGM=HEWL,PARM='ALIGN2,...'
             .
             .
             .
//SYSLMOD DD       DSNAME=OWNLIB,...
//SYSLIN  DD       *
          PAGE     RAREUSE
          ORDER    MAINRT(P),CSECTA,SESECT1
          INCLUDE  SYSLMOD(MAINROOT)
          NAME     MAINROOT
/*
```

Figure 22. Aligning Control Sections on Page Boundaries

OVERLAY PROGRAMS

Ordinarily, when a load module produced by the linkage editor is executed, all of the control sections of the module remain in virtual storage throughout execution. The length of the load module is, therefore, the sum of the lengths of all of the control sections. When storage space is not at a premium, this is the most efficient way to execute a program. However, if a program approaches the limits of the virtual storage available, the programmer should consider using the overlay facilities of the linkage editor.

In most cases, all that is needed to convert an ordinary program to an overlay program is the addition of control statements to structure the module. The programmer chooses the overlayable portions of the program, and the system arranges to load the required portions when needed during execution of the program.

When the linkage editor overlay facility is requested, the load module is structured so that, at execution time, certain control sections are loaded only when referenced. When a reference is made from an executing control section to another, the system determines whether or not the code required is already in virtual storage. If it is not, the code is loaded dynamically and may overlay an unneeded part of the module already in storage.

The rest of this chapter is divided into three sections that describe the design, specification, and special considerations for overlay programs.

Design of an Overlay Program

The way in which an overlay module is structured depends on the relationships among the control sections within the module. Two control sections that do not have to be in storage at the same time can overlay each other. Such control sections are *independent*; that is, they do not reference each other either directly or indirectly. Independent control sections can be assigned the same load addresses and are loaded only when referenced. For example, control sections that handle error conditions or unusual data may be used infrequently, and need not be occupying storage unless in use.

Control sections are grouped into segments. A *segment* is the smallest functional unit (one or more control sections) that can be loaded as one logical entity during execution. The control sections required all of the time are grouped into a special segment called the *root segment*. This segment remains in storage throughout execution of an overlay program.

When a particular segment is to be executed, any segments between it and the root segment must also be in storage. This is a *path*. A reference from one segment to another segment lower in a path is a *downward reference*. That is, the segment contains a reference to another segment farther from the root segment. Conversely, a reference from one segment to another segment higher in a path (closer to the root segment) is an *upward reference*.

Therefore, a downward reference may cause overlay because the necessary segment may not yet be in virtual storage. An upward reference will not cause overlay because all segments between a segment and the root segment must be present in storage.

Sometimes several paths need the same control sections. This problem may be solved by placing the control sections in another region. In an overlay structure, a *region* is a contiguous area of virtual storage within which segments can be loaded independently of paths in other regions. An overlay program can be designed in single or multiple regions.

Single Region Overlay Program

To design an overlay structure, the programmer should select those control sections that will receive control at the beginning of execution, plus those that should always remain in storage; these control sections form the root segment. The rest of the structure is developed by determining the dependencies of the remaining control sections and how they can use the same virtual storage locations at different times during execution.

Besides control section dependency, other topics discussed in this section are segment dependency, the length of the overlay program, segment origin, communication between segments, and overlay processing.

Control Section Dependency

Control section dependency is determined by the requirements of a control section for a given routine in another control section. A control section is dependent upon any control section from which it receives control, or which processes its data. For example, if control section C receives control from control section B, then C is dependent upon B. That is, both control sections must be in storage before execution can continue beyond a given point in the program.

A program contains seven control sections, CSA through CSG, and exceeds the amount of storage available for its execution. Before the program is rewritten, it is examined to see whether or not it could be placed into an overlay structure. Figure 23 shows the groups of dependent control sections in the program (the arrows indicate dependencies).

Each dependent group is also a path. That is, if control section CSG is to be executed, CSB and CSA must also be in storage. Because CSA and CSB are in each path, they must be in the root segment. Control section CSC is in two groups, and therefore is a *common segment* in two different paths.

A better way to show the relationship between segments is with a tree structure. A *tree* is the graphic representation that shows how segments can use virtual storage at different times. It does not imply the order of execution, although the root segment is the first to receive control. Figure 24 shows the tree structure for the dependent groups shown in Figure 23. The structure is contained in one region, and has five segments.

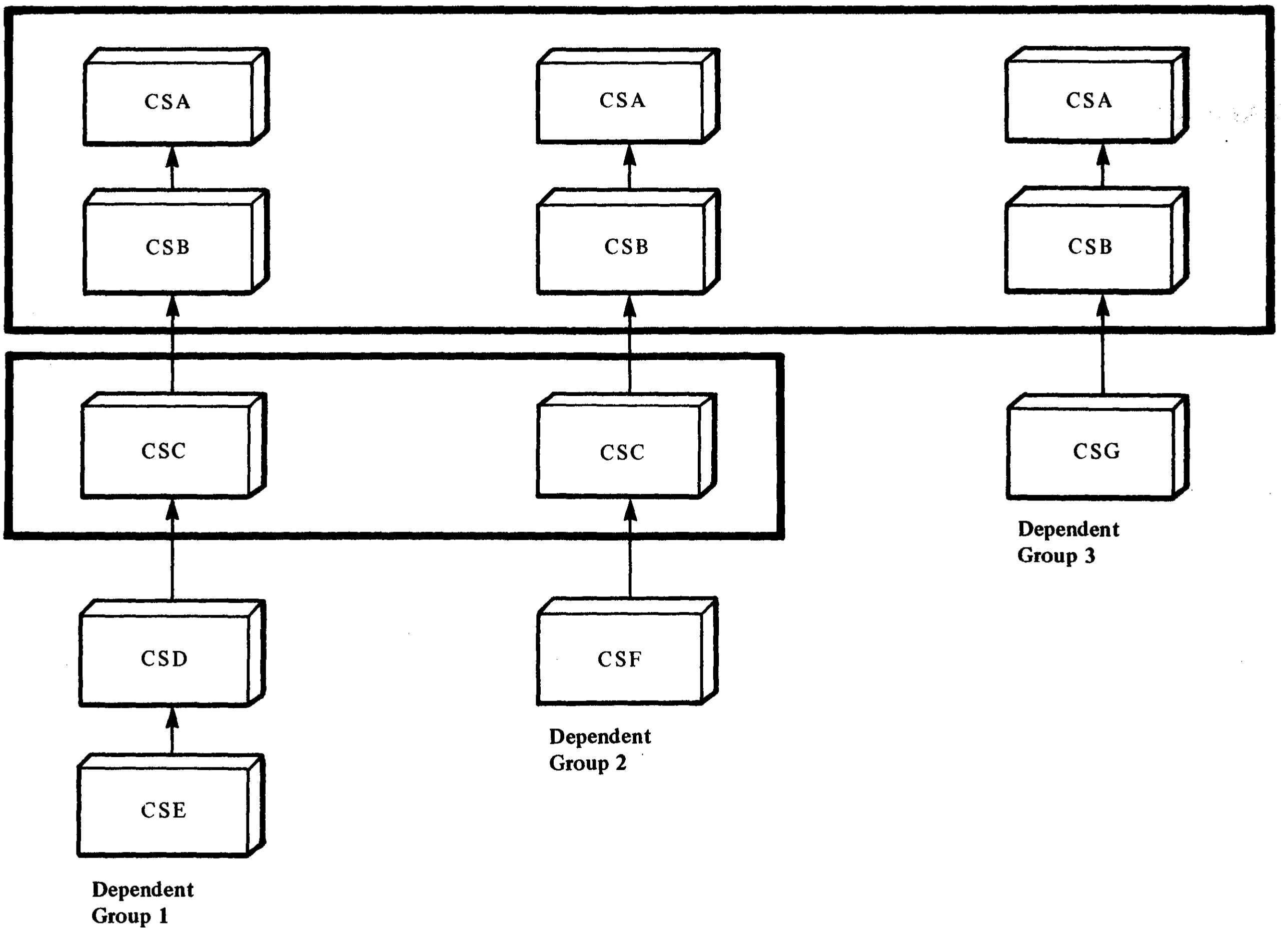


Figure 23. Control Section Dependencies

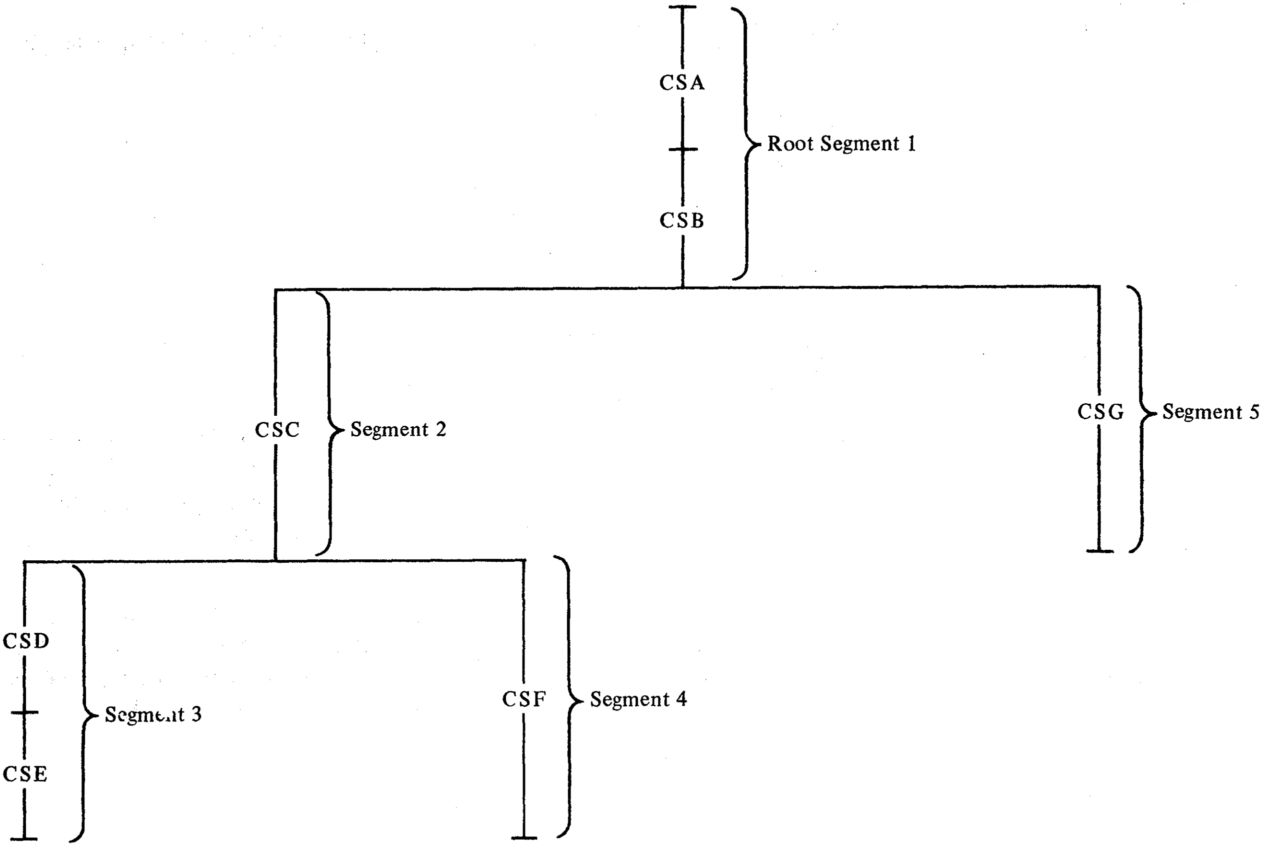


Figure 24. Single-Region Overlay Tree Structure

Segment Dependency

When a segment is in virtual storage, all segments in its path are also in virtual storage. Each time a segment is loaded, all segments in its path are loaded if they are not already in virtual storage. In Figure 24 when segment 3 is in virtual storage, segments 1 and 2 are also in virtual storage. However, if segment 2 is in storage, this does not imply that segment 3 or 4 is in virtual storage since neither segment is in the path of segment 2.

The position of the segments in an overlay tree structure does not imply the sequence in which the segments are executed. A segment can be loaded and overlaid as many times as required by the logic of the program. However, a segment will not be overlaid by itself. If a segment is modified during execution, that modification remains only until the segment is overlaid.

Length of an Overlay Program

For purposes of illustration, assume that the control sections in the sample program have the following lengths:

Control Section	Length (in bytes)
CSA	3,000
CSB	2,000
CSC	6,000
CSD	4,000
CSE	3,000
CSF	6,000
CSG	8,000

If the program were not in overlay, it would require 32,000 bytes of virtual storage. In overlay, however, the program requires the amount of storage needed for the longest path. In this structure, the longest path is formed by segments 1, 2, and 3, since, when they are all in storage, they require 18,000 bytes, as shown in Figure 25.

Note, however, that the length of the longest path is not the minimum requirement for an overlay program; when a program is in overlay, certain tables are used, and their storage requirements must also be considered. The storage required by these tables is given in the section "Special Considerations."

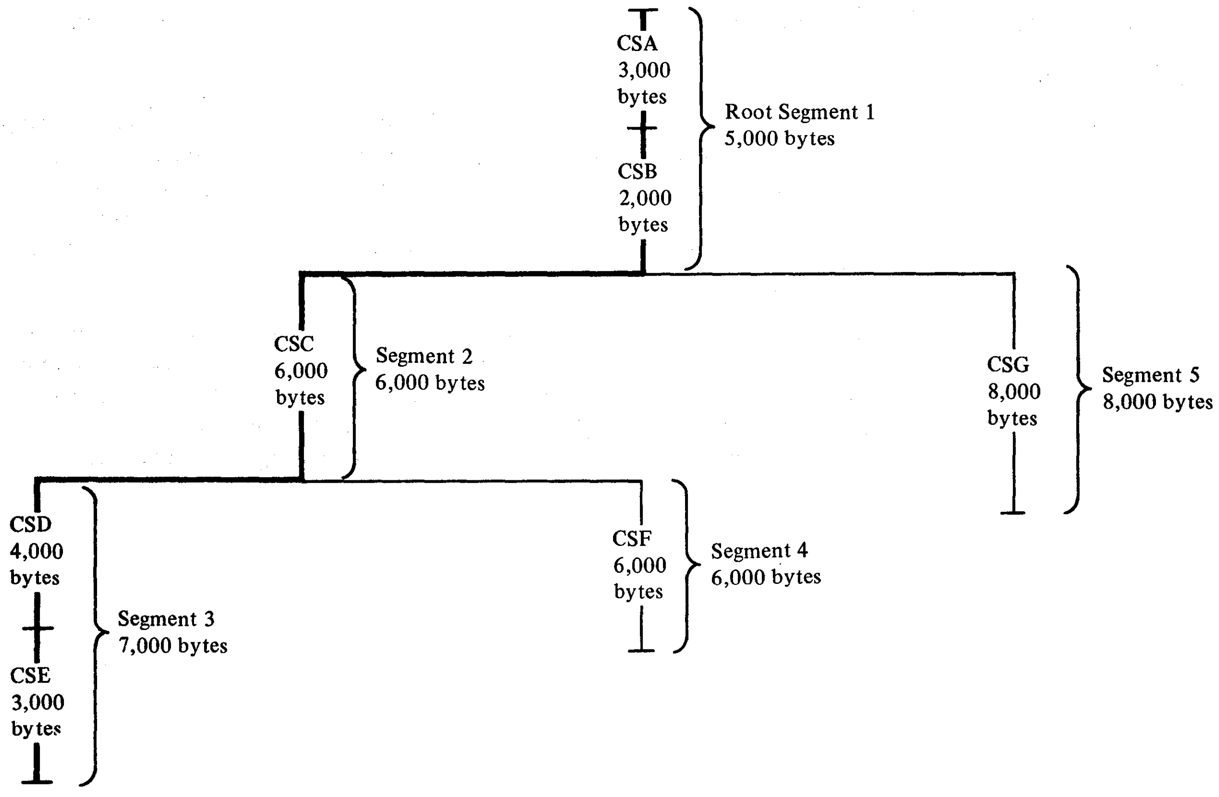


Figure 25. Length of an Overlay Module

Segment Origin

The linkage editor assigns the relocatable origin of the root segment (the origin of the program) at 0. The relative origin of each segment is determined by 0 plus the length of all segments in the path. For example, the origin of segments 3 and 4 is equal to 0 plus 6,000 (the length of segment 2) plus 5,000 (the length of the root segment), or 11,000. The origins of all the segments are as follows:

Segment	Origin
1	0
2	5,000
3	11,000
4	11,000
5	5,000

The segment origin is also called the *load point*, because it is the relative location at which the segment is loaded.

Figure 26 shows the segment origin for each segment and the way storage is used by the sample program. In the illustration, the vertical bars indicate segment origin; any two segments with the same origin may use the same storage area. Figure 25 also shows that the longest path is that of segments 1, 2, and 3.

Communication Between Segments

Segments that can be in virtual storage simultaneously are considered to be *inclusive*. Segments in the same region but not in the same path are considered to be *exclusive*; they cannot be in virtual storage simultaneously. Figure 27 shows the inclusive and exclusive segments in the sample program.

Segments upon which two or more exclusive segments are dependent are called *common segments*. A segment common to two other segments is part of

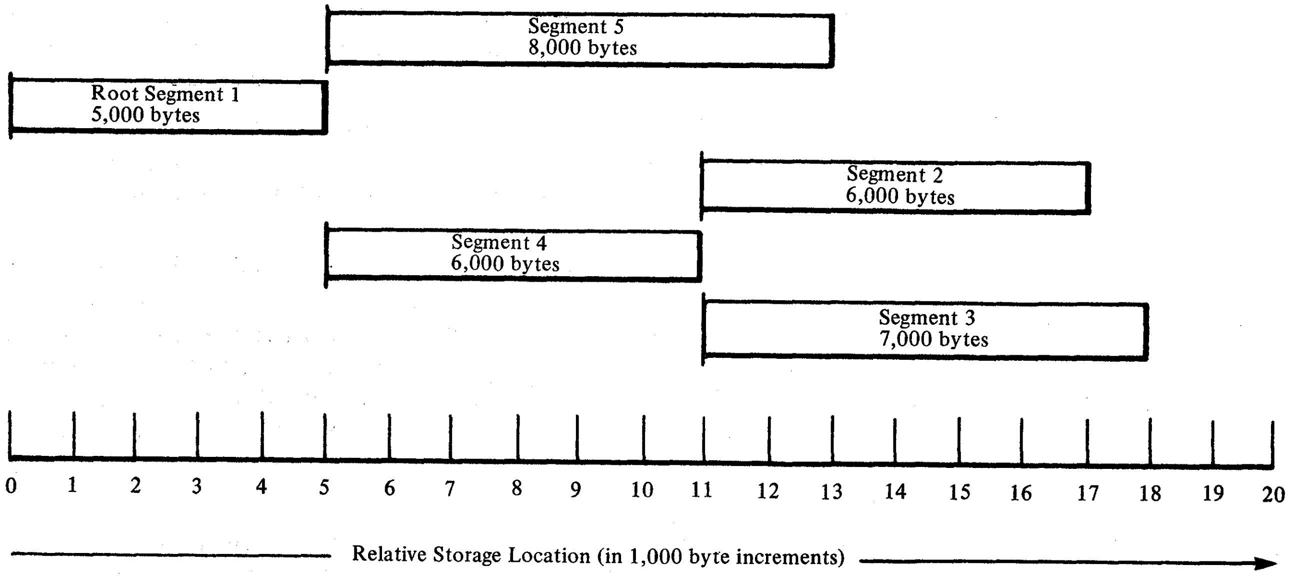


Figure 26. Segment Origin and Use of Storage

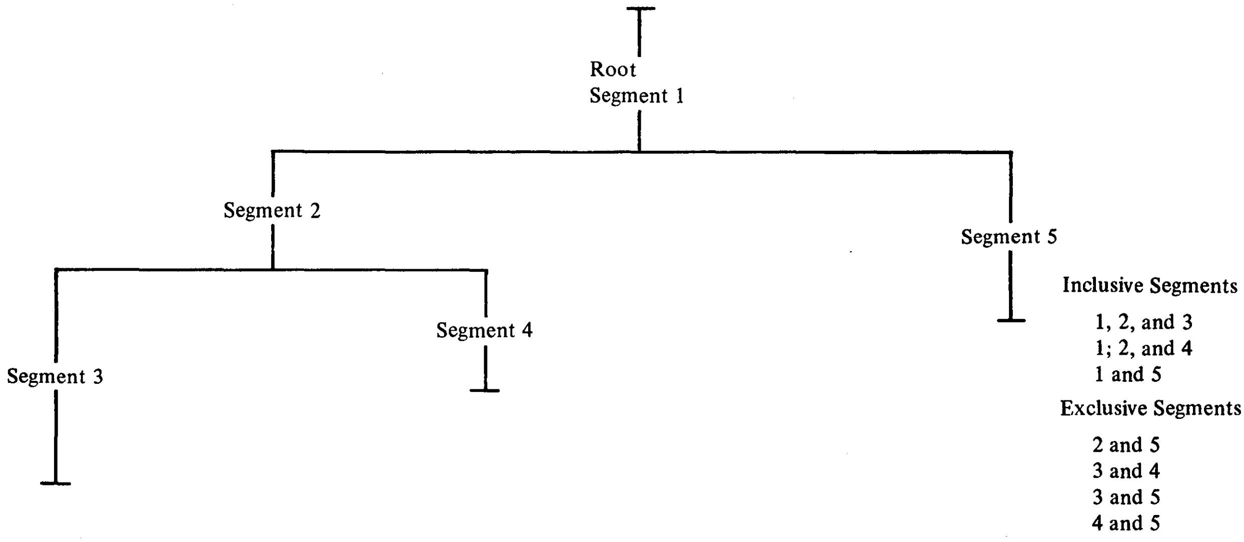


Figure 27. Inclusive and Exclusive Segments

the path of each segment. In Figure 27, segment 2 is common to segments 3 and 4, but not to segment 5.

An *inclusive reference* is a reference between inclusive segments; that is, a reference from a segment in storage to an external symbol in a segment that will not cause overlay of the calling segment. An *exclusive reference* is a reference between exclusive segments; that is, a reference from a segment in storage to an external symbol in a segment that will cause overlay of the calling segment.

Figure 28 shows the difference between an inclusive reference and an exclusive reference; the arrows indicate references between segments.

Inclusive References: Wherever possible, inclusive references should be used instead of exclusive references. Inclusive references between segments are always valid and do not require special options. When inclusive references are used, there is also less chance for error in structuring the overlay program correctly.

Exclusive References: An exclusive reference is made when the external reference in the requesting segment is to a symbol defined in a segment not in the path of the requesting segment. Exclusive references are either valid or invalid.

An exclusive reference is *valid* only if there is also a reference to the requested control section in a segment common to both the segment to be loaded and the segment to be overlaid. The same symbol must be used in both the common segment and the exclusive reference. In Figure 28, a reference from segment B to segment A is valid, because there is an inclusive reference from the common segment to segment A. (An entry table in the common segment contains the address of segment A; the overlay does not destroy this table.)

In the same illustration, a reference from segment A to segment B is *invalid* because there is no reference from the common segment to segment B. A reference from segment A to segment B can be made valid by including, in

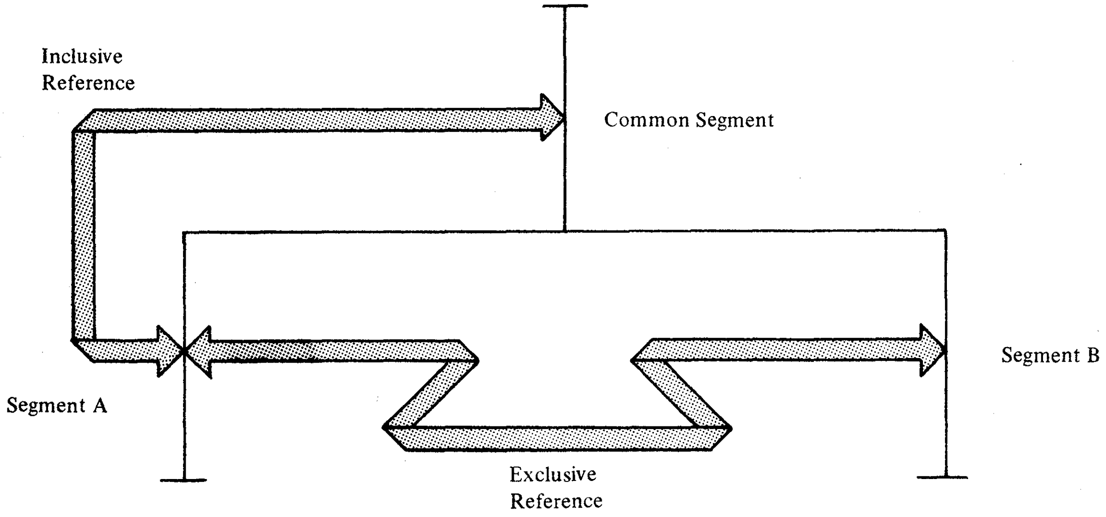


Figure 28. Inclusive and Exclusive References

the common segment, an external reference to the symbol used in the exclusive reference to segment B.

Another way to eliminate exclusive references is to arrange the program so that the references that will cause overlay are made in a higher segment. For example, the programmer could eliminate the exclusive reference shown in Figure 28 by writing a new module to be placed in the common segment; the new module's only function would be to reference segment B. He would then change the code in segment A to refer to the new module instead of to segment B. Control then would pass from segment A to the common segment, where the overlay of segment A by segment B would be initiated.

If either valid or invalid exclusive references appear in the program, the linkage editor considers them errors unless one of the special options is used. These options are described later in this section.

Notes:

- During the execution of a program written in a higher level language such as FORTRAN, COBOL, or PL/I, an exclusive call results in abnormal termination of the program if the requested segment attempts to return control directly to the invoking segment that has been overlaid.
- If a program written in COBOL includes a segment that contains a reference to a COBOL class test or TRANSFORM table, the segment containing the table must be either (1) in the root segment or (2) a segment that is higher in the same path than the segment containing the reference to the table.

Overlay Process

The overlay process is initiated during execution of a program only if a control section in virtual storage references a control section not in storage. The control program determines the segment that the referenced control section is in and, if necessary, loads the segment. When a segment is loaded, it overlays any segment in storage with the same relative origin. Any segments in storage that are lower in the path of the overlaid segment may also be overlaid. An exclusive reference can also cause segments higher in the path to be overlaid. If a control section in storage references a control section in another segment already in storage, no overlay occurs.

The portion of the control program that determines when overlay is to occur is the *overlay supervisor,* which uses special tables to determine when overlay is necessary. These tables are generated by the linkage editor, and are part of the output load module. The special tables are the segment table and the entry table(s). Figure 29 shows the location of the segment and entry tables in the sample program.

Because the tables are present in every overlay module, their size must be considered when planning the use of virtual storage. The storage requirements for the tables are given in "Special Considerations." A more detailed discussion of the segment and entry tables follows.

Segment Table: Each overlay program contains one segment table (SEGTAB); this table is the first control section in the root segment. The segment table contains information about the relationship of the segments and regions in the program. During execution, the table also indicates which segments are either in storage or being loaded, and other control information.

Entry Table: Each segment that is not the last segment in a path may contain one entry table (ENTAB); this table, when present, is the last control section in a segment.

When overlay will be required, an entry in the table is created for a symbol to which control is to be passed, provided (1) the symbol is used as an external

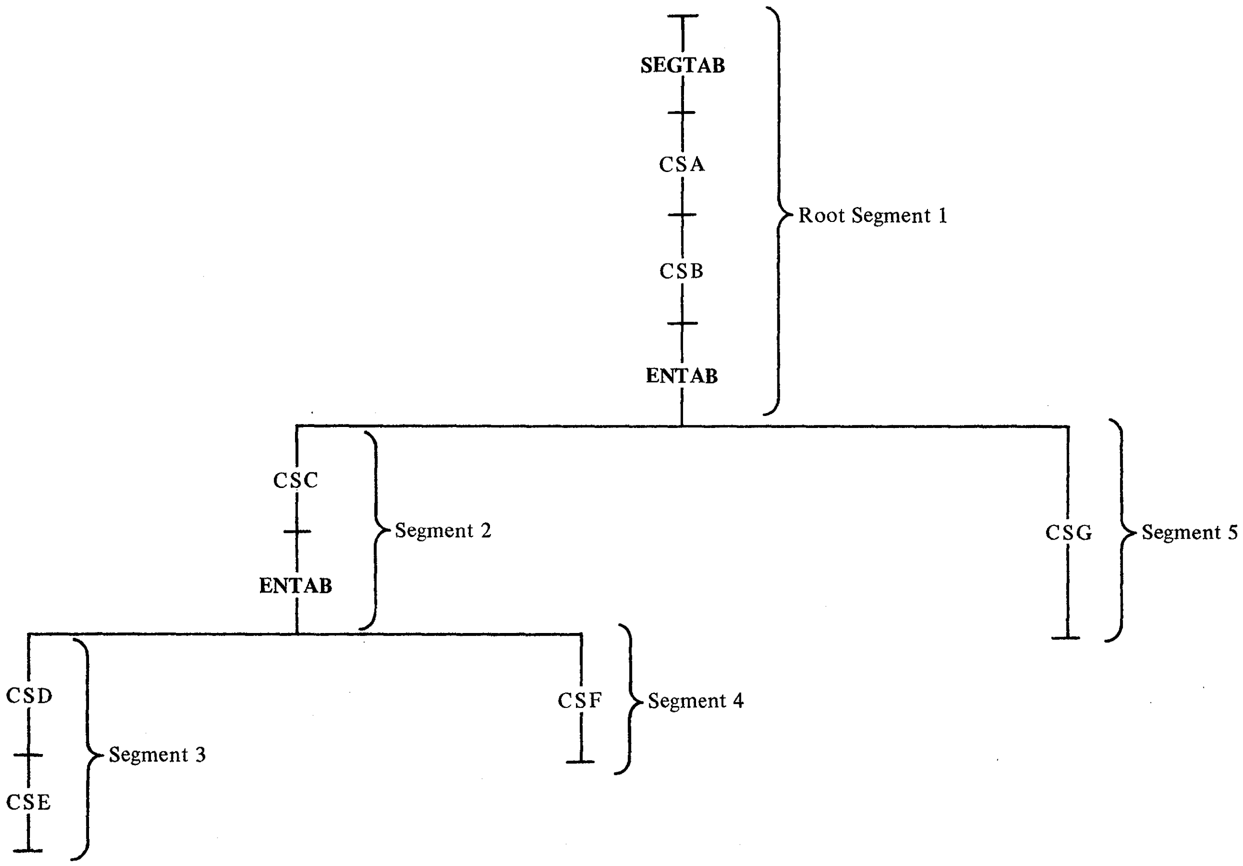


Figure 29. Location of Segment and Entry Tables in an Overlay Module

reference in the requesting segment, and (2) the symbol is defined in another segment either lower in the path of the requesting segment, or in another region. An ENTAB entry is not created for any symbol already present in an entry table closer to the root segment (higher in the path), or for a symbol defined higher in the path. (A reference to a symbol higher in the path does not have to go through the control program because no overlay is required.)

If an external reference and the symbol to which it refers are in segments not in the same path but in the same region, an exclusive reference was made. If the exclusive reference is valid, an ENTAB entry for the symbol is present in the common segment. Since the common segment is higher in the path of the requesting segment, no ENTAB entry is created in the requesting segment. When the reference is executed, control passes through the ENTAB entry in the common segment. That is, a branch to the location in the ENTAB causes the overlay supervisor to be called to load the needed segment or segments.

If the exclusive reference is invalid, no ENTAB entry is present in the common segment. If the LET option is specified, an invalid exclusive reference causes unpredictable results when the program is executed. Since no ENTAB entry exists, control is passed directly to the relative address specified in the reference, even though the requested segment may not be in virtual storage.

Multiple Region Overlay Program

If a control section is used by several segments, it is usually desirable to place that control section in the root segment. However, the root segment can get so large that the benefits of overlay are lost. If some of the control sections in the root segment could overlay each other (except for the requirement that all segments in a path must be in storage at the same time), the job may be a candidate for multiple region structure. Multiple region structures can also be used to increase segment loading efficiency: processing can continue in one region while the next path to be executed is being loaded into another region.

With multiple regions, a segment has access to segments that are not in its path. Within each region, the rules for single region overlay programs apply, but the regions are independent of each other. A maximum of four regions can be used.

Figure 30 shows the relationship between the control sections in the sample program and two new control sections, CSH and CSI. The two new control sections are each used by two other control sections in different paths. Placing CSH and CSI in the root segment makes the segment larger than necessary because CSH and CSI can overlay each other. The two control sections should not be duplicated in two paths because the linkage editor automatically deletes the second pair and an invalid exclusive reference may then result.

If however, the two control sections are placed in another region, they can be in virtual storage when needed, regardless of the path being executed in the first region. Figure 31 shows all of the control sections in a two-region structure. Either path in region 2 can be in virtual storage regardless of the path being executed in region 1; segments in region 2 can cause segments in region 1 to be loaded without being overlaid themselves.

The relative origin of a second region is determined by the length of the longest path in the first region (18,000 bytes). Region 2, therefore, begins at 0 plus 18,000 bytes. The relative origin of a third region would be determined

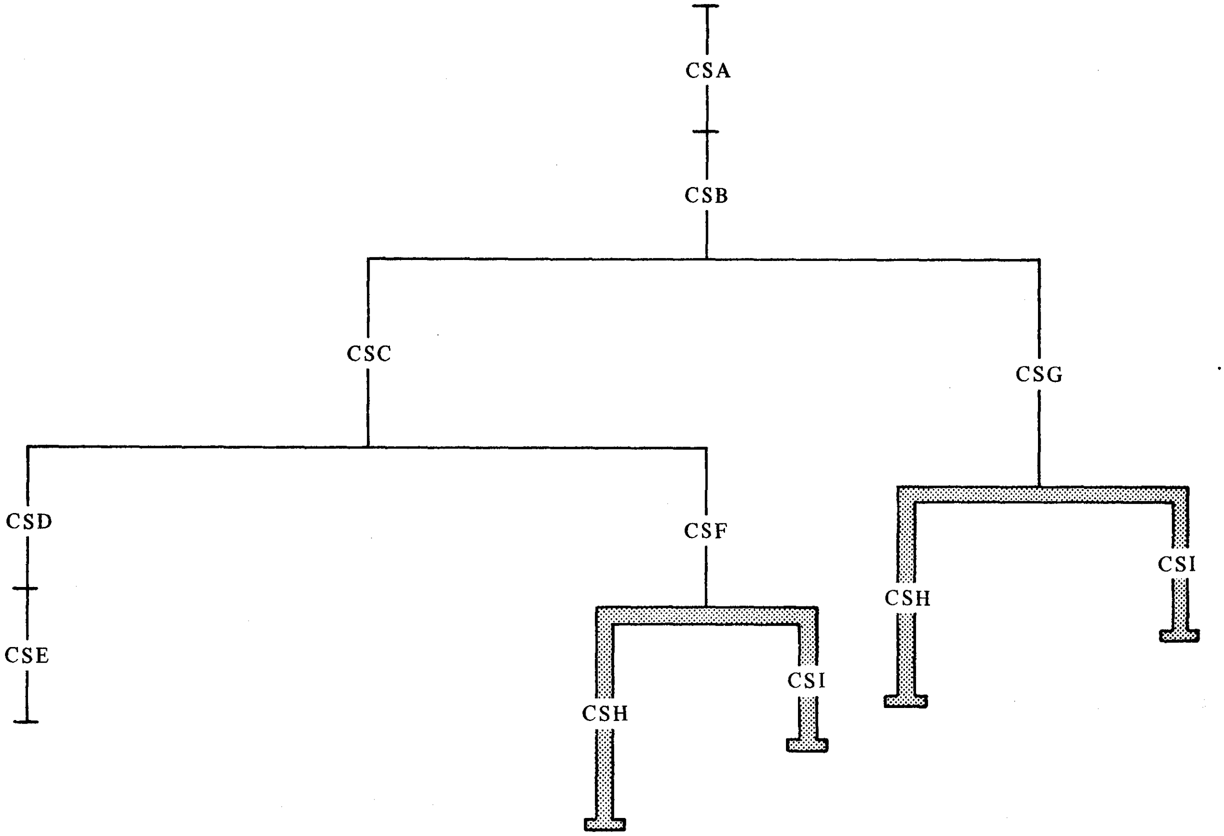


Figure 30. Control Sections Used by Several Paths

by the length of the longest path in the first region plus the longest path in the second region.

The virtual storage required for the program is determined by adding the lengths of the longest path in each region. In Figure 31, if CSH is 4,000 bytes and CSI is 3,000 bytes, the storage required is 22,000 bytes, plus the storage required by the special overlay tables. Care should be exercised when choosing multiple regions. There may be some system degradation due to the overlay supervisor being unable to optimize segment loading when multiple regions are used.

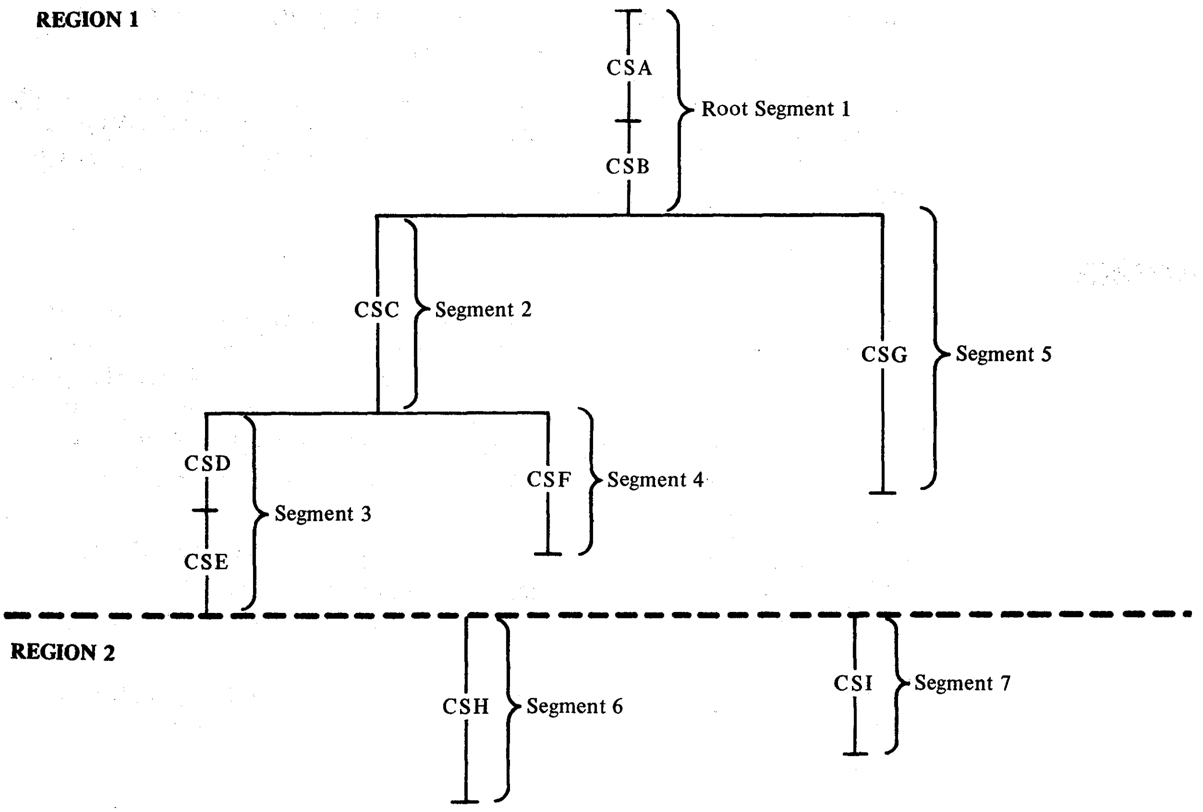


Figure 31. Overlay Tree for Multiple-Region Program

Specification of an Overlay Program

Once the programmer has designed an overlay structure, he or she must place the module in that structure by indicating to the linkage editor the relative positions of the segments and regions, and the control sections in each segment. Positioning is accomplished as follows:

- *Segments* are positioned by OVERLAY statements. Since segments are not named, the programmer identifies a segment by giving its origin (or load point) a symbolic name and then uses that name in an OVERLAY statement to specify a symbolic origin. Each OVERLAY statement begins a new segment.
- *Regions* are also positioned by OVERLAY statements. The programmer specifies the origin of the first segment of the region, followed by the word REGION in parentheses.
- *Control sections* are positioned in the segment specified by the OVERLAY statement with which they are associated in the input sequence. However, the sequence of the control sections within a segment is not necessarily the order in which the control sections are specified.

The input sequence of control statements and control sections should reflect the sequence of the segments in the overlay structure from top to bottom, left to right, and region by region. This sequence is illustrated in later examples.

In addition, several special options are used with overlay programs. These options are specified on the EXEC statement for the linkage editor job step, and are described at the end of this section.

Note: If a load module in overlay structure is to be reprocessed by the linkage editor, the OVERLAY statements and special options (such as OVLY) must be respecified. If the statements and options are not provided, the output load module will not be in overlay structure.

Segment Origin

The symbolic origin of every segment, other than the root segment, must be specified with an OVERLAY statement. The first time a symbolic origin is specified, a load point is created at the end of the previous segment. That load point is logically assigned a relative address at the doubleword boundary that follows the last byte in the preceding segment. Subsequent use of the same symbolic origin indicates that the next segment is to have its origin at the same load point.

In the sample single-region program, the symbolic origin names ONE and TWO are assigned to the two necessary load points, as shown in Figure 31. Segments 2 and 5 are at load point ONE, segments 3 and 4 are at load point TWO.

The following sequence of OVERLAY statements will result in the structure in Figure 32 (the control sections in each segment are indicated by name):

```
Control section CSA
Control section CSB
OVERLAY ONE
Control section CSC
OVERLAY TWO
Control section CSD
Control section CSE
OVERLAY TWO
Control section CSF
OVERLAY ONE
Control section CSG
```

Note that the sequence of OVERLAY statements reflects the order of segments in the structure from top to bottom and left to right.

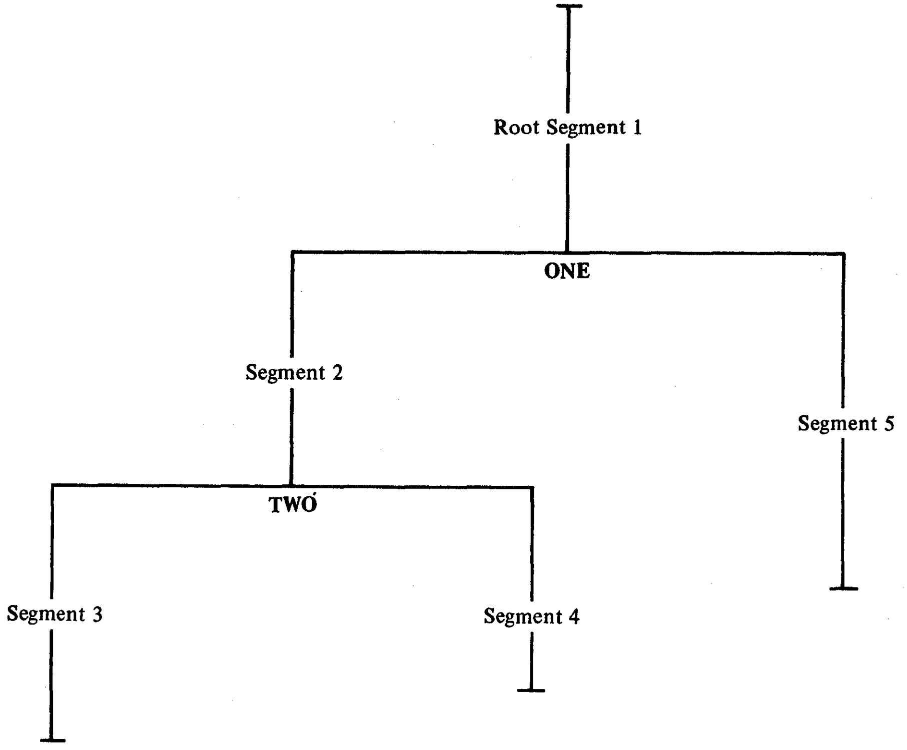


Figure 32. Symbolic Segment Origin in Single-Region Program

Region Origin

The symbolic origin of every region, other than the first, must be specified with an OVERLAY statement. Once a new region is specified, a segment origin from a previous region should not be specified.

In the sample multiple-region program, the symbolic origin THREE is assigned to region 2, as shown in Figure 33. Segments 6 and 7 are at load point THREE.

If the following is added to the sequence for the single-region program, the multiple-region structure will be produced:

```
        .
        .
        .
OVERLAY THREE(REGION)
Control section CSH
OVERLAY THREE
Control section CSI
```

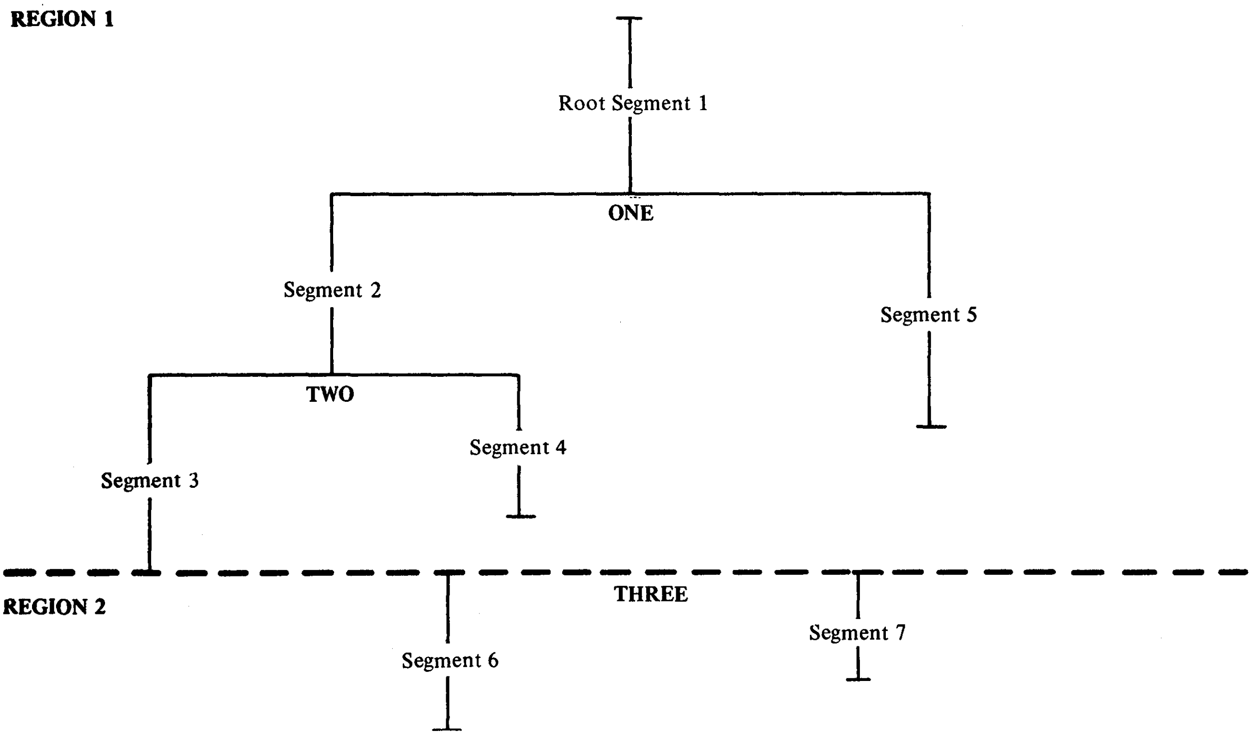


Figure 33. Symbolic Segment and Region Origin in Multiple-Region Program

Positioning Control Sections

After each OVERLAY statement, the control sections for that segment must be specified. The control sections for a segment can be specified in one of three ways:

- By placing the object decks for each segment after the appropriate OVERLAY statement.
- By using INCLUDE control statements for the modules containing the control sections for the segment.
- By using INSERT control statements to reposition a control section from its position in the input stream to a particular segment.

Any control sections that precede the first OVERLAY statement are placed in the root segment; they can be repositioned with an INSERT statement. Control sections from the automatic call library are also placed in the root segment. The INSERT statement can be used to place these control sections in another specific segment. Common areas in an overlay program are described in "Special Considerations."

An example of each of the three methods of positioning control sections follows. Each example results in the structure for the single-region sample program. An example is also given of repositioning control sections from the automatic call library.

Using Object Decks

The primary input data set for this example contains an ENTRY statement and seven object decks, separated by OVERLAY statements:

```
//LKED          EXEC   PGM=HEWL,PARM='OVLY'
                 .
                 .
                 .
//SYSLIN        DD     *
  ENTRY BEGIN
 Object deck for CSA
 Object deck for CSB
  OVERLAY ONE
 Object deck for CSC
  OVERLAY TWO
 Object deck for CSD
 Object deck for CSE
  OVERLAY TWO
 Object deck for CSF
  OVERLAY ONE
 Object deck for CSG
/*
```

The EXEC statement illustrates that the OVLY parameter must be specified for every overlay program to be processed by the linkage editor.

Using INCLU] Statements

The primary input data set for this example contains a series of control statements. The INCLUDE statements in the primary input data set direct the linkage editor to library members that contain the control sections of the program.

```
//LKED          EXEC   PGM=HEWL,PARM='OVLY'
                 .
                 .
                 .
//MODLIB        DD     DSNAME=OBJLIB,DISP=(OLD,KEEP),...
//SYSLIN        DD     *
  ENTRY BEGIN
  INCLUDE MODLIB(CSA,CSB)
  OVERLAY ONE
  INCLUDE MODLIB(CSC)
  OVERLAY TWO
  INCLUDE MODLIB(CSD,CSE)
  OVERLAY TWO
  INCLUDE MODLIB(CSF)
  OVERLAY ONE
  INCLUDE MODLIB(CSG)
/*
```

This example differs from the previous one in that the control sections of the program are not part of the primary input data set, but are represented in the primary input by the INCLUDE statements. When an INCLUDE statement is processed, the appropriate control section is retrieved from the library and processed.

Using INSERT Statements

When INSERT statements are used, the INSERT and OVERLAY statements may either follow or precede all the input modules. However, the order of the control sections in a segment is not necessarily the same as the order of the INSERT statements for each segment. An example of each is given, as well as an example of repositioning automatically called control sections.

Following All Input: The control statements can follow all the input modules, as shown in the following example:

```
//LKED         EXEC  PGM=HEWL,PARM='OVLY'
                 .
                 .
                 .
//SYSLIN       DD    DSNAME=OBJECT,DISP=(OLD,KEEP),...
//             DD    *
  ENTRY BEGIN
  INSERT CSA,CSB
  OVERLAY ONE
  INSERT CSC
  OVERLAY TWO
  INSERT CSD,CSE
  OVERLAY TWO
  INSERT CSF
  OVERLAY ONE
  INSERT CSG
/*
```

The primary input data set contains the object modules for the control sections, and the input stream is concatenated to it.

Preceding All Input: The control statements can also precede all input modules, as shown in the following example:

```
//LKED         EXEC  PGM=HEWL,PARM='OVLY'
//MODULES      DD    DSNAME=OBJSEQ,DISP=(OLD,KEEP),...
                 .
                 .
                 .
//SYSLIN       DD    *
  ENTRY BEGIN
  INSERT CSA,CSB
  OVERLAY ONE
  INSERT CSC
  OVERLAY TWO
  INSERT CSD,CSE
  OVERLAY TWO
  INSERT CSF
  OVERLAY ONE
  INSERT CSG
  INCLUDE MODULES
/*
```

The primary input data set contains all of the control statements for the overlay structure and an INCLUDE statement. The data set specified by the INCLUDE statement contains all of the object modules for the structure, and is a sequential data set.

Repositioning Automatically Called Control Sections: The INSERT statement can also be used to move automatically called control sections from the root segment to the desired segment. This is helpful when control sections from the automatic call library are used in only one segment. By moving such control sections, the root segment will contain only those control sections used by more than one segment.

When a program is written in a higher level language, special control sections are called from the automatic call library. Assume that the sample program is written in COBOL and that two control sections (ILBOVTR0 and ILBOSCH0) are called automatically from SYS1.COBLIB. Ordinarily, these control sections are placed in the root segment. However, INSERT statements are used in the following example to place these control sections in segments other than the root segment.

```
//LKED         EXEC  PGM=HEWL,PARM='OVLY'
//MODLIB       DD    DSNAME=OBJLIB,DISP=(OLD,KEEP),...
//SYSLIB       DD    DSNAME=SYS1.COBLIB,DISP=SHR
                .
                .
                .
//SYSLIN       DD     *
  ENTRY BEGIN
  INCLUDE MODLIB(CSA,CSB)
  OVERLAY ONE
  INCLUDE MODLIB(CSC)
  OVERLAY TWO
  INCLUDE MODLIB(CSD,CSE)
  INSERT ILBOVTR0
  OVERLAY TWO
  INCLUDE MODLIB(CSF)
  INSERT ILBOSCH0
  OVERLAY ONE
  INCLUDE MODLIB(CSG)
/*
```

As a result, segments 3 and 4 will also contain ILBOVTR0 and ILBOSCH0, respectively.

This example also combines two of the ways of specifying the control sections for a segment.

Special Options

The linkage editor provides three special job step options for the overlay programmer. These options are specified on the EXEC statement for the linkage editor job step. They must be specified each time a load module in overlay structure is reprocessed by the linkage editor. The three options are OVLY, LET, and XCAL.

OVLY Option

The OVLY option must be specified for every overlay program. If the option is omitted, all the OVERLAY and INSERT statements are considered invalid. The output module is marked not executable unless the LET option is specified. The output module is not in an overlay structure.

LET Option

With the LET option, the output module is marked executable even though certain error conditions were found during linkage editor processing. When LET is specified, any exclusive reference (valid or invalid) is accepted. At execution time, a valid exclusive reference is executed correctly; an invalid exclusive reference usually causes unpredictable results.

Also with the LET option, unresolved external references do not prevent the module from being marked executable. This could be helpful when part of a large program is ready for testing; the segments to be tested may contain references to segments not yet coded. If LET is specified, the program can be executed to test those parts that are finished (as long as the references to the absent segments are not executed). If the LET option is not specified, these unresolved references will cause the module to be marked not executable.

XCAL Option

With the XCAL option, a valid exclusive call is not considered an error, and the load module is marked executable. However, other errors could cause the module to be marked not executable, unless the LET option is specified; in this case, the XCAL option is not required.

Special Considerations

This section discusses several special considerations that affect overlay programs. These considerations include the handling of common areas, special storage requirements, and overlay communication.

Common Areas

When common areas (blank or named) are encountered in an overlay program, the common areas are collected as described previously (i.e., the largest blank or identically named common area is used). The final location of the common area in the output module depends on whether INSERT statements were used to structure the program.

If INSERT statements are used to structure the overlay program, a named common area should either be part of the input stream in the segment to which it belongs, or should be placed there with an INSERT statement.

Because INSERT statements cannot be used for blank common areas, a blank common area should always be part of the input stream in the segment to which it belongs.

If INSERT statements are not used, and the control sections for each segment are placed or included between OVERLAY statements, the linkage editor "promotes" the common area automatically. That is, the common area is placed in the common segment of the paths that contain references to it so that the common area is in storage when needed. The position of the promoted area in relation to other control sections within the common segment is unpredictable.

If a common area is encountered in a module from the automatic call library, automatic promotion places the common area in the root segment. In the case of a named common area, this may be overridden by use of the INSERT statement.

Assume that the sample program is written in FORTRAN and that common areas are present as shown in Figure 34. Further assume that the overlay program is structured with INCLUDE statements between the OVERLAY statements so that automatic promotion occurs.

Segments 2 and 5 contain blank common areas, segments 3 and 4 contain named common area A, and segments 4 and 5 contain named common area B. During linkage editor processing, the blank common areas are collected and the largest area is promoted to the root segment (the first common segment in the two paths); the common areas named A are collected and the largest area is promoted to segment 2; the common areas named B are collected and promoted to the root segment. Figure 35 shows the location of the common areas after processing by the linkage editor.

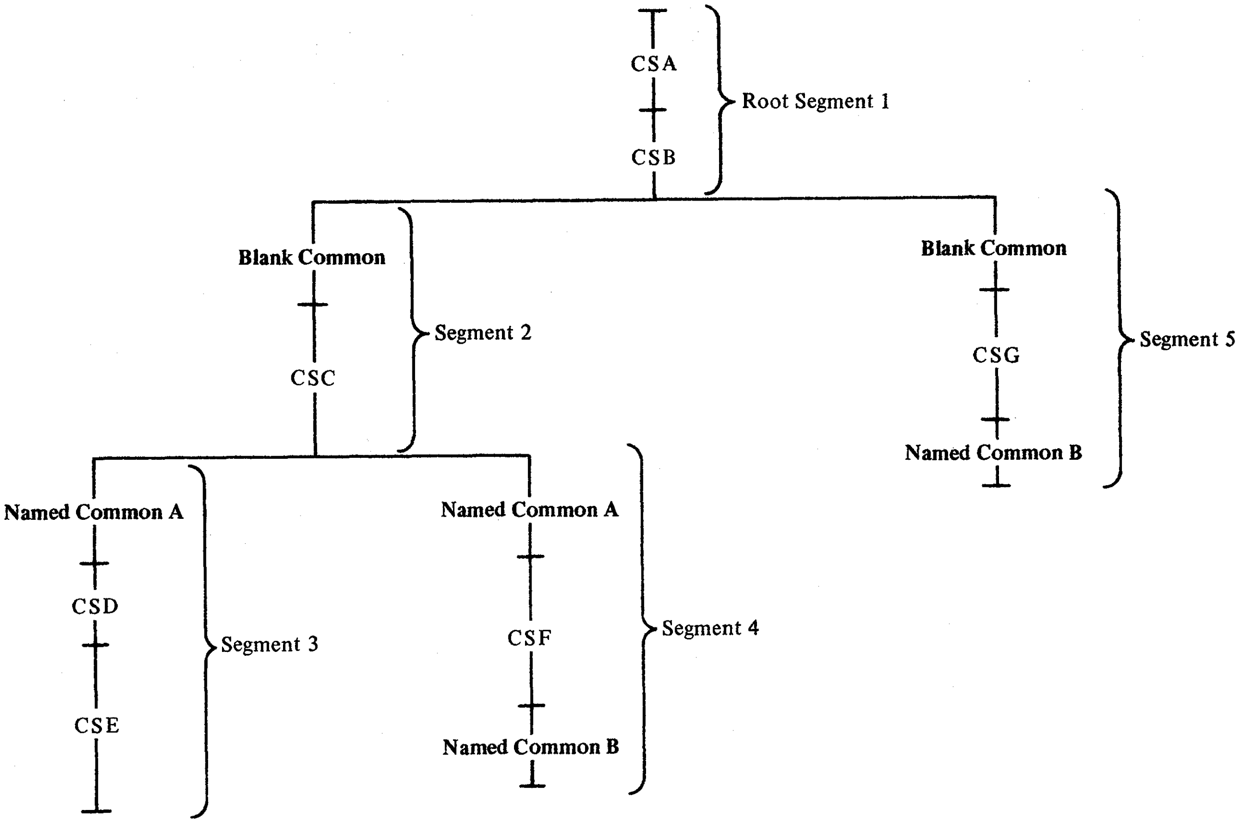


Figure 34. Common Areas before Processing

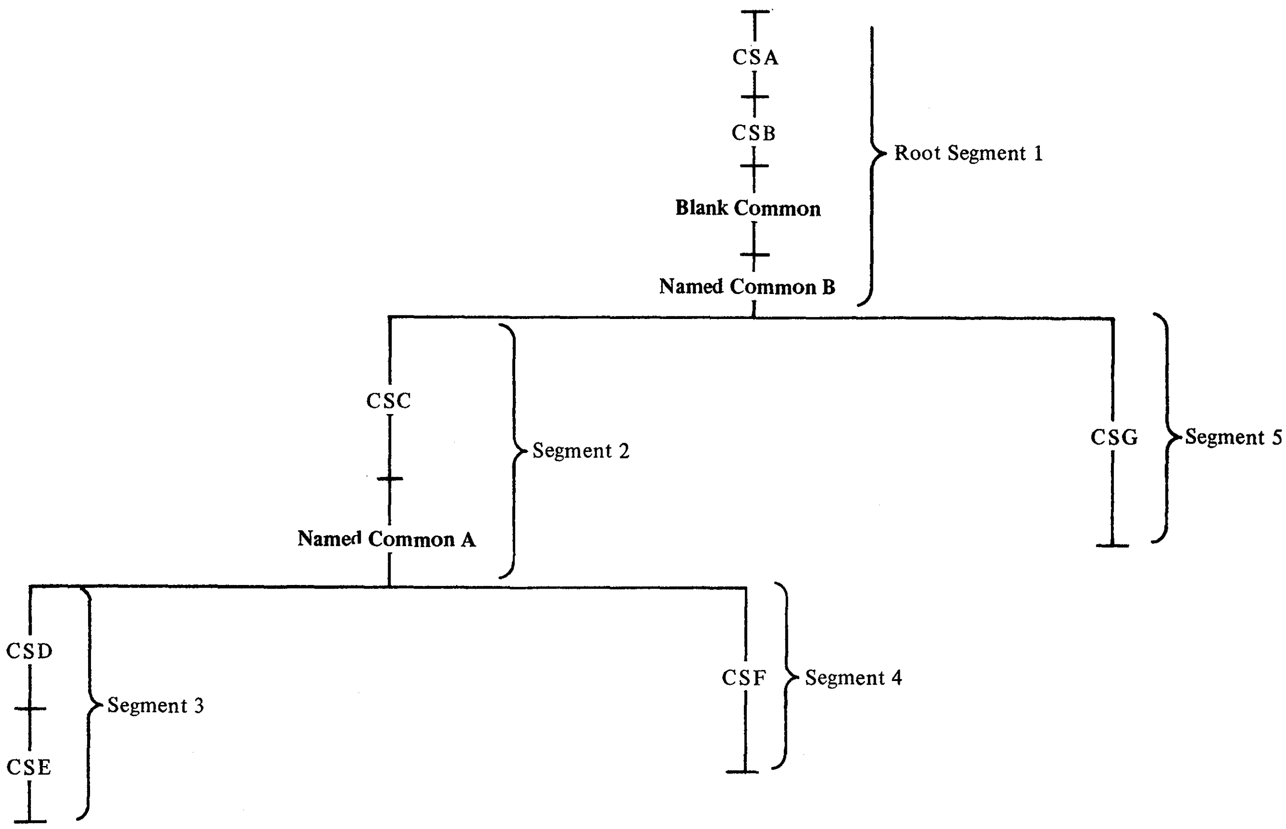


Figure 35. Common Areas after Processing

Storage Requirements

The virtual storage requirements for an overlay program include the items placed in the module by the linkage editor and the overlay supervisor necessary for execution.

Items in the Load Module: The items that the linkage editor places in an overlay load module are the segment table, entry tables, and other control information. Their size must be included in the minimum requirements for an overlay program, along with the storage required by the longest path and any control sections from the automatic call library.

Every overlay program has one segment table in the root segment. The storage requirements are:

SEGTAB = 4n + 24

where:

n = the number of segments in the program

Some segments will have an entry table. The requirements of the entry tables in the segments in the longest path must be added to the storage requirements for the program. The requirements for an entry table are:

ENTAB = 12(x + 1)

where:

x = the number of entries in the table

Finally, a NOTE list is required to execute an overlay program. The storage requirements are:

NOTELST = 4n + 8

where:

n = the number of segments in the program

Overlay Supervisor: To the minimum requirements of the load module itself must be added the requirements of the overlay supervisor. This system routine is not placed in an overlay module, but, during execution of the module, the supervisor may be called to initiate an overlay. If called, the storage allocated for the program must be large enough for the supervisor also.

Three overlay supervisor modules are furnished with the system: the basic, advanced, and asychronous modules. The basic module does does not test whether a request for overlay is valid; the other two do. Neither the basic nor advanced modules permit overlay through the SEGLD macro instruction (see "Overlay Communication"); the asynchronous module does. When the SEGLD macro instruction is used with the basic and advanced modules, it is ignored. The storage requirements for the overlay supervisor modules are:

Module	Storage Requirements (in bytes)
Basic (used with VS1)	436
Advanced (used with VS1)	512
Asynchronous (used with VS2)	992

Overlay Communication

Several ways of communicating between segments of an overlay program are discussed in this section. A higher level or Assembler language program may use a CALL statement or CALL macro instruction, respectively, to cause control to be passed to a symbol defined in another segment. The CALL may cause the segment to be loaded if it is not already present in storage. An Assembler language program may also use three additional ways to communicate between segments:

- By a branch instruction, which causes a segment to be loaded and control to be passed to a symbol defined in that segment.
- By a segment load (SEGLD) macro instruction (VS2 only), which requests loading of a segment. Processing continues in the requesting segment while the requested segment is being loaded.
- By a segment load and wait (SEGWT) macro instruction, which requests loading of a segment. Processing continues in the requesting segment only after the requested segment is loaded.

Any of the four methods may be used to make inclusive references. Only the CALL and branch may be used to make exclusive references. Neither the SEGLD nor SEGWT macro isntruction should be used to make exclusive references; since both imply that processing is to continue in the requesting segment, an exclusive reference leads to erroneous results when the program is executed.

CALL Statement or CALL Macro Instruction

A CALL statement or CALL macro instruction refers to an external name in the segment to which control is to be passed. The external name must be defined as an external reference in the requesting segment. In Assembler language, the name must be defined as a four-byte V-type address constant; the high-order byte is reserved for use by the control program, and must not be altered during execution of the program.

When a CALL is used, the requested segment and any segments in its path are loaded if they are not part of the path already in virtual storage. After the segment is loaded, control is passed to the requested segment at the location specified by the external name.

A CALL between inclusive segments is always valid. A return can be made to the requesting segment by another source language statement, such as RETURN. A CALL between exclusive segments is valid if the conditions for a valid exclusive reference are met; a return from the requested segment can be made only by another exclusive reference, because the requesting segment has been overlaid.

Branch Instruction

Any of the branching conventions shown in Figure 36 can be used to request loading and branching to a segment. As a result, the requested segment and any segments in its path are loaded if they are not part of the path already in virtual storage. Control is then passed to the requested segment at the location specified by the address constant placed in general register 15.

The address constant must be a 4-byte V-type address constant. The high-order byte is reserved for use by the control program, and must not be altered during execution of the program.

A branch between inclusive segments is always valid; a return may be made by means of the address stored in Rn. A branch betweeen exclusive segments is valid if the conditions for a valid exclusive reference are met; a return can be made only by another exclusive reference.

Example	Name[1]	Operation	Operand[2,3]
1		L	R15,=V(name)
		BALR	Rn,R15
2		L	R15,ADCON
		BALR	Rn,R15
		.	
		.	
		.	
	ADCON	DC	V(name)
3		L	R15,=V(name)
		BAL	Rn,0(0,R15)[4]
4		L	R15,=V(name)
		BAL	Rn,0(R15)[5]
5[6]		L	R15,=V(name)
		BCR	15,R15
6[6]		L	R15,=V(name)
		BC	15,0(0,R15)[4]
7[6]		L	R15,=V(name)
		BC	15,0(R15)[5]

[1] When the name field is blank, specification of a name is optional.

[2] R15 is the register into which is loaded a 4-byte address constant that is an entry name or a control section name in the requested segment. The address constant must be loaded into the standard entry point register, register 15.

[3] Rn is any other register and is used to hold the return address. This register is usually register 14.

[4] This may also be written so that the index register is loaded with the address constant; the other fields must be zero.

[5] In this format, the base register must be loaded with the address constant; the displacement must be zero.

[6] This example is an unconditional branch; other conditions are also allowed.

Figure 36. Branch Sequences for Overlay Programs

Segment Load (SEGLD) Macro Instruction

The SEGLD macro instruction is used to provide overlap between segment loading and processing within the requesting segment. As a result of using any of the examples in Figure 37, the loading of the requested segment and any segments in its path is initiated when they are not part of the path already in virtual storage. Processing then resumes at the next sequential instruction in the requesting segment while the segment or segments are being loaded. Control may be passed to the requested segment with either a CALL or a branch, as shown in examples 1 and 2, respectively. A SEGWT instruction can be used to ensure that the data in the control section specified by the external name is in virtual storage before processing begins, as shown in Example 3.

The external names specified in the SEGLD macro instruction must be defined with a 4-byte V-type address constant. The high-order byte is reserved for use by the control program and must not be altered during execution of the program.

Note: Some configurations of the control program do not have the capability of processing the SEGLD macro instruction. When used, the macro instruction is treated as a NOP (no operation) and the segment is loaded when a SEGWT macro instruction or a branch is executed. If the rules of overlay are followed, correct execution occurs.

Example	Name[1]	Operation	Operand[2,3]
1		SEGLD	external name
		CALL	external name
2		SEGLD	external name
		branch	
3		SEGLD	external name
		SEGWT	external name
		L	Rn,=A(name)

[1] When the name field is blank, specification of a name is optional.

[2] External name is an entry name or a control section name in the requested segment.

[3] Rn is any other register and is used to hold the return address. This register is usually register 14.

Figure 37. Use of the SEGLD Macro Instruction

Segment Wait (SEGWT) Macro Instruction

The SEGWT macro instruction is used to stop processing in the requesting segment until the requested segment is in virtual storage.

As a result of using any of the examples in Figure 38, no further processing takes place until the requested segment and all segments in its path are loaded when not already in virtual storage. Processing resumes at the next sequential instruction in the requesting segment after the requested segment has been loaded.

Example	Name[1]	Operation	Operand[2,3]
1		SEGLD	external name
		SEGWT	external name
		L	Rn,ADCON
		branch	
	ADCON	DC	A(name)
2		SEGWT	external name
		L	Rn,=A(name)

[1] When the name field is blank, specification of a name is optional.

[2] External name is an entry name or a control section name in the requested segment.

[3] Rn is any other register and is used to hold the return address. This register is usually register 14.

Figure 38. Use of the SEGWT Macro Instruction

If the SEGWT and SEGLD macro instructions are used together, overlap occurs between processing and segment loading; use of the SEGWT macro instruction serves as a check to see that the necessary information is in storage when it is finally needed (see Example 1 in Figure 38). In Example 2 in Figure 38, no overlap is provided; the SEGWT macro instruction initiates loading, and processing is stopped in the requesting segment until the requested segment is in virtual storage.

The external name specified in the SEGWT macro instruction must be defined with a 4-byte V-type address constant. The high-order byte is reserved for use by the control program, and must not be altered during execution of the program.

If the contents of a virtual storage location in the requested segment are to be processed, the entry name of the location must be referred to by an A-type address constant.

JOB CONTROL LANGUAGE SUMMARY

This chapter summarizes those aspects of the job control language that pertain directly to the use of the linkage editor. The major topics covered are the EXEC statement, DD statements, and cataloged procedures for the linkage editor. The reader should be familiar with the job control language as described in *OS/VS1 JCL Reference* or *OS/VS2 JCL.*

EXEC Statement—Introduction

The EXEC statement is the first statement of every job step. For the linkage editor job step, the following topics are pertinent:

- The program name of the linkage editor.
- Linkage editor options passed to the job step.
- Region requirements for the linkage editor.

For an execution job step following the linkage editor job step, the linkage editor return code is important.

The EXEC statement contains the symbolic name of the load module to be invoked for execution. The linkage editor can be invoked with the following program name:

```
  HEWL
```

LINKEDIT is an alias name for the linkage editor and can also be used to invoke it.

For example, the following EXEC statement causes the linkage editor to be invoked:

```
//LKED          EXEC    PGM=HEWL
```

PGM=LINKEDIT could also be used.

To ensure compatibility with the operating system, the linkage editor can also be invoked by any of the following alias names: IEWL, IEWLF440, IEWLF880, IEWLF128.

EXEC Statement—Job Step Options

The EXEC statement also contains a list of options or parameters to be passed to the linkage editor. These options are of four types:

- Module attributes, which describe the characteristics of the output load module.
- Special processing options, which affect linkage editor processing.
- Space allocation options, which affect the amount of storage used by the linkage editor for processing and output module library buffers.
- Output options, which specify the kind of output the linkage editor is to produce.

The rest of this section describes the options in each category. All of the options for a particular linkage editor execution are listed in the PARM parameter on the EXEC statement. They can be listed in any sequence, as long as the rules for coding parameters are followed.

Module Attributes

The module attributes describe the characteristics of the output module, or modules. (If more than one load module is produced by the same linkage editor job step, all output modules will have the attributes assigned on the EXEC statement.) The attributes for each load module are stored in the directory of the output module library along with the member name. (The format of the directory entry of a partitioned data set is given in *OS/VS1 System Data Areas* and *OS/VS2 Data Areas.*

Module attributes specify whether or not the module:

- Can ever be processed by the linkage editor.
- Can be brought into virtual storage only by the LOAD macro instruction.
- Is to be in overlay format.
- Can be reused.
- Can be placed in the link pack area; that is, is re-enterable.
- Can be replaced during execution by recovery management; that is, is refreshable.
- Is to be tested by the TSO TEST command under VS2.
- Is to have specified control sections aligned on page boundaries.
- Is or is not authorized to use the restricted system resources and functions.

After the descriptions of the module attributes, the default and incompatible attributes are discussed.

Downward Compatible Attribute

When this attribute is specified, a maximum record size of 1024 bytes is used for the output module library.

To assign the downward compatible attribute, code DC in the PARM field as follows:

```
//LKED        EXEC    PGM=IEWL,PARM='DC,...'
```

Note: If the DC attribute is specified and the output load module library is a data set created by the link-edit job step, the blocksize in the DSCB (data set control block) is set to 1024. If the DC attribute is specified and the output load module library is an existing data set, then the blocksize in the DSCB is set to 1024 only if the current blocksize in the DSCB is less than 1024; if the current blocksize in the DSCB is greater than 1024, the load module is written using a maximum record size of 1024 bytes but the blocksize in the DSCB is not changed.

Hierarchy Format Attribute

Although VS systems do not provide hierarchy support, the HIAR attribute is included in the linkage editor for compatibility with OS systems. If the HIAR attribute is specified and the module is link-edited under VS, the module will run on either OS or VS but the attribute will be ignored when fetching the load module on VS systems.

Control sections within a module with the hierarchy format attribute are suitable for either block or scatter loading into the hierarchies specified in HIARCHY control statements. Specification of hierarchy format, when main storage hierarchy support is included in the system, allows the programmer to make use of both processor storage (hierarchy 0) and IBM 2361 Core Storage (hierarchy 1). When main storage hierarchy support is not included in the system, programs with the hierarchy format attribute are block or scatter loaded into processor storage (see "Scatter Format").

When storage hierarchies are used, all control sections assigned to a hierarchy are normally block loaded. If the allocated region within the hierarchy is not large enough for block loading of the control sections, and the scatter loading feature is available, the control sections may be scatter loaded into the allocated area within the hierarchy.

The hierarchy format attribute overrides the scatter format attribute; the overlay attribute overrides the hierarchy format attribute and must be omitted if hierarchies are to be assigned.

To assign the hierarchy format attribute, code HIAR in the PARM field, as follows:

```
//LKED         EXEC   PGM=IEWL,PARM='HIAR,...'
```

See the description of the HIARCHY control statement for information on assigning control sections to a specific hierarchy.

Note: Because control sections may be scatter loaded when HIAR is specified, the programmer should ensure that the load module does not contain zero-length control sections, private code sections, or common areas. The presence of such sections in a module that is to be scatter loaded can, under certain circumstances, cause Program Fetch to terminate abnormally when the module is loaded into main storage for execution.

Scatter Format Attribute

A module with the scatter format attribute need not be loaded into a contiguous block of main storage; rather, the programmer can specify the dynamic loading of control sections into noncontiguous, or scattered, areas within his assigned main storage area. Although scatter loading can also be left to the control program, the programmer should specify the loading process himself for most effective use of available storage. If the scatter format attribute is not specified, the linkage editor produces a load module in a format suitable for block loading. That is, the control program can load the module only into one contiguous main storage area large enough to contain the complete module.

When the scatter format attribute is specified, the linkage editor produces a load module in a format suitable for either scatter or block loading. If the scatter load feature is not available in the control program, modules with the scatter format attribute are block loaded.

To assign the scatter format attribute, code SCTR in the PARM field, as follows:

```
//LKED         EXEC   PGM=IEWL,PARM='SCTR,...'
```

Notes:

- The block format attribute is assigned by the linkage editor if scatter format is not specified. (The programmer cannot specify block format.)
- If SCTR is specified, the programmer should ensure that the load module does not contain zero-length control sections, private code sections, or common areas. The presence of such sections in a module that is to be scatter loaded can, under certain circumstances, cause Program Fetch to

terminate abnormally when the module is loaded into main storage for execution.

- The SCTR attribute must be specified when the nucleus for a VS system is link-edited. In all other instances, if the SCTR attribute is specified, the linkage editor builds the output load module appropriately; however, scatter load support is not provided in the VS systems and the attribute/load module format is ignored when fetching the load module.

Not Editable Attribute

A load module which is marked NE (not editable) is not reprocessable by the linkage editor. If a module map or a cross-reference table is requested, the not editable attribute is ignored.

To assign the not editable attribute, code NE in the PARM field, as follows:

```
//LKED         EXEC    PGM=HEWL,PARM='NE,...'
```

Note: The not editable attribute disables the EXPAND function for the output load module and also limits to eighteen the number of consecutive iterations of AMASPZAP (for VS2) or HMASPZAP (for VS1). If the EXPAND function is req'ired or more than eighteen iterations of AMASPZAP/HMASPZAP are required, the load module will have to be recreated.

Only Loadable Attribute

A module with the only loadable attribute can be brought into virtual storage only with a LOAD macro instruction. Some subsets of the control program use a smaller control table when the load module is invoked with a LOAD. This reduces the overall virtual storage requirements of the module.

A module with the only loadable attribute must be entered by means of a branch instruction or a CALL macro instruction. If an attempt is made to enter the module with a LINK, XCTL, or ATTACH macro instruction, the program making the attempt is terminated abnormally by the control program.

To assign the only loadable attribute, code OL in the PARM field as follows:

```
//LKED         EXEC    PGM=HEWL,PARM='OL,...'
```

Overlay Attribute

A program with the overlay attribute is placed in an overlay structure as directed by the linkage editor OVERLAY control statements. The module is suitable only for block loading; it cannot be refreshable, re-enterable, serially reusable, or assigned to hierarchies.

If the overlay attribute is specified and no OVERLAY control statements are found in the linkage editor input, the attribute is negated. The condition is considered a recoverable error; that is, if the LET option is specified, the module is marked executable.

The overlay attribute must be specified for overlay processing. If this attribute is omitted, the OVERLAY and INSERT statements are considered invalid, and the module is not an overlay structure. This condition is also recoverable; if the LET option is specified, the module is marked executable.

To assign the overlay attribute, code OVLY in the PARM field as follows:

```
//LKED        EXEC    PGM=HEWL,PARM='OVLY,...'
```

See "Overlay Programs" for information on the design and specification of an overlay structure.

Reusability Attributes

Either one of two attributes may be specified to denote the reusability of a module. Reusability means that the same copy of a load module can be used by more than one task either concurrently or one at a time. The reusability attributes are *re-enterable* and *serially reusable*; if neither is specified, the module is not reusable and a fresh copy must be brought into virtual storage before another task can use the module.

The linkage editor only stores the attribute in the directory entry; it does not check whether the module is really re-enterable or serially reusable. A re-enterable module is automatically assigned the reusable attribute. However, a reusable module is not also defined as re-enterable; it is reusable only.

Re-enterable: A module with the re-enterable attribute can be executed by more than one task at a time; that is, a task may begin executing a re-enterable module before a previous task has finished executing it. This type of module cannot be modified by itself or by any other module during execution.

If a module is to be re-enterable, all of the control sections within the module must be re-enterable. If the re-enterable attribute is specified, and any load modules that are not re-enterable become a part of the input to the linkage editor, the attribute is negated.

To assign the re-enterable attribute, code RENT in the PARM field, as follows:

```
//LKED        EXEC    PGM=HEWL,PARM='RENT,...'
```

Serially Reusable: A module with the serially reusable attribute can be executed by only one task at a time; that is, a task may not begin executing a serially reusable module before a previous task has finished executing it. This type of module must initialize itself and/or restore any instructions or data in the module altered during execution.

If a module is to be serially reusable, all of its control sections must be either serially reusable or re-enterable. If the serially reusable attribure is specified, and any load modules that are neither serially reusable nor re-enterable become a part of the input to the linkage editor, the serially reusable attribute is negated.

To assign the serially reusable attribute, code REUS in the PARM field, as follows:

```
//LKED        EXEC    PGM=HEWL,PARM='REUS,...'
```

Refreshable Attribute

A module with the refreshable attribute can be replaced by a new copy during execution by a recovery management routine without changing either the sequence or results of processing. This type of module cannot be modified by itself or by any other module during execution. The linkage editor only stores the attribute in the directory entry; it does not check whether the module is refreshable.

If a module is to be refreshable, all of the control sections within it must be refreshable. If the refreshable attribute is specified, and any load modules that are not refreshable become a part of the input to the linkage editor, the attribute is negated.

To assign the refreshable attribute, code REFR in the PARM field, as follows:

```
//LKED        EXEC    PGM=HEWL,PARM='REFR,...'
```

Test Attribute

A module with the test attribute is to be tested and contains the testing symbol tables for the TSO TEST command. The linkage editor accepts these tables as input, and places them in the output module. The module is marked as being under test. If the test attribute is not specified, the symbol tables are ignored by the linkage editor and are not placed in the output module. If the test attribute is specified, and no symbol table input is received, the output load module will not contain symbol tables to be used by the TSO TEST command.

To assign the test attribute, code TEST in the PARM field, as follows:

```
//LKED        EXEC    PGM=HEWL,PARM='TEST,...'
```

Note: The test attribute applies to programs using TESTRAN or the TSO TEST command. Do not use the 'TEST' option unless the load module is to be executed by TSO or TESTRAN.

Page Boundary Attribute

Control sections within a load module with the page boundary attribute are aligned in storage on page boundaries. Used with the PAGE control statement or the ORDER statement with the P operand, this attribute causes alignment of specified control sections on 2K boundaries. If virtual storage is limited under VS1, alignment on 2K page boundaries reduces paging and conserves storage; however, performance degradation may result when 2K alignment is used under VS2.

To assign the 2K page boundary attribute, code ALIGN2 in the PARM field, as follows:

```
//LKED        EXEC    PGM=HEWL,PARM='ALIGN2,...'
```

Notes:

- If the ALIGN2 attribute is not coded and the PAGE statement or ORDER statement with the P operand is used, the default boundary alignment is 4K.
- Page boundary aligning cannot be used for VS1 overlay programs.

Authorization Code

The output load module is assigned an authorization code which determines whether or not the load module may use restricted system services and resources.

To assign an authorization code through the PARM field, code the AC parameter as follows:

```
//LKED        EXEC    PGM=HEWL,PARM='AC=n,...'
```

The authorization code n must be 1 to 8 decimal digits giving a value from 0 to 255.

'AC=', 'AC=,...' and 'AC= ' are equivalent to 'AC=0'. The authorization code assigned in the PARM field is overridden by an authorization code assigned through the SETCODE control statement.

Default Attributes

Unless specific module attributes are indicated by the programmer, the output module is not in an overlay structure, and it is not tested (assembler only). The module is in block format, not refreshable, not re-enterable, and not serially reusable. Its control sections are aligned on 4K page boundaries if page boundary alignment is requested.

One other attribute is specified by the linkage editor after processing is finished. If, during processing, severity 2 errors were found that would prevent the output module from being executed successfully, the linkage editor assigns the not executable attribute. The control program will not load a module with this attribute.

If the LET option is specified, the output module is marked executable even if severity 2 errors occur. The LET option is discussed later in this section.

If the AC parameter is not specified or is coded incorrectly, the default authorization code of zero (0) is assigned to the output load module.

Incompatible Attributes

Of the module attributes that the programmer may specify, several are mutually exclusive. When mutually exclusive attributes are specified for a load module, the linkage editor ignores the less significant attributes. For example, if both OVLY and RENT are specified, the module will be in an overlay structure and will not be re-enterable.

Certain attributes are also incompatible with other job step options. For convenience, all job step options are shown in Figure 41 at the end of this chapter along with those options that are incompatible.

Special Processing Options

The special processing options affect the executability of the output module and the use of the automatic library call mechanism. These options are the exclusive call option, the let execute option, and the no automatic call option.

Exclusive Call Option

When the exclusive call option is specified, the linkage editor marks the output module as executable when valid exclusive references have been made between segments. However, a warning message is given for each valid exclusive reference.

To specify the exclusive call option, code XCAL in the PARM field as follows:

```
//LKED        EXEC    PGM=HEWL,PARM='XCAL,OVLY,...'
```

The OVLY attribute must also be specified for an overlay program.

Note: Other errors may cause the module to be marked not executable unless the let execute option is specified.

Let Execute Option

When the let execute option is specified, the linkage editor marks the output module as executable even though a severity 2 error condition was found during processing. (A severity 2 error condition could make execution of the output load module impossible.) Some examples of severity 2 errors are:

- Unresolved external references.
- Valid or invalid exclusive calls in an overlay program.
- Error on a linkage editor control statement.
- A library module that cannot be found.
- No available space in the directory of the output module library.

To specify the let execute option, code LET in the PARM field as follows:

```
//LKED        EXEC    PGM=HEWL,PARM='LET,...'
```

Note: If LET is specified, XCAL need not be specified.

No Automatic Library Call Option

When the no automatic library call option is specified, the linkage editor library call mechanism does not call library members to resolve external references. The output module is marked executable even though unresolved external references are present. If this option is specified, the LIBRARY statement need not be used to negate the automatic library call for selected external references. Also, with this option, a SYSLIB DD statement need not be supplied.

To specify the no automatic library call option, code NCAL in the PARM field, as follows:

```
//LKED        EXEC    PARM=HEWL,PARM='NCAL,...'
```

Note: Other errors may cause the module to be marked not executable unless the LET option is also specified.

Space Allocation Options

These options allow the programmer to specify the storage available to the linkage editor, and to specify the blocksize for the output module.

SIZE Option

The programmer can specify, through the SIZE option, the amount of virtual storage to be used by the level F linkage editor and the portion of that storage to be used as the load module buffer.

Default values for the SIZE option are chosen during system generation. The default values are used if one or both of the values are not specified correctly, or not specified at all. These defaults should be made adequate for most link edits, relieving the programmer from having to specify the SIZE option for each link edit. For details on how to establish default values for VS1, see the EDITOR macro in *OS/VS1 System Generation Reference.* The default values for VS2 are: value1 is 192K and value2 is 64K.

Format: The format of the SIZE option is:

SIZE=(*value1* **,** *value2* **)**

SIZE=(*value1* **)**

SIZE=(*value1* **,)**

SIZE=(, *value2* **)**

SIZE=(,)

When coded in the PARM field, the expression is enclosed in single quotes, as follows:

```
//LKED          EXEC    PGM=HEWL,
//          .           PARM='SIZE=(value1,value2),...'
```

Both value1 and value2 may be expressed as integers specifying the number of bytes of virtual storage or as nK where n represents the number of 1K (1024) bytes of virtual storage.

When determining the values for the SIZE option, it is best to establish value2 first, then value1.

Value2: Value2 specifies the number of bytes of storage to be allocated as the module buffer. The allocation specified by value2 is a part of the virtual storage specified by value1.

The actual minimum for value2 is 6144 (6K) or the length of the largest input load module text record, whichever is larger. If a value less than 6144 (6K) is specified, the default value for value2 is used.

The space allocated by value2 is used as: the buffer into which the input load module text is read, the buffer from which load module text is written to the intermediate data set, the buffer into which the load module text is read from the intermediate data set, and the buffers from which the load module text is written to the output data set. Therefore the determination of value2 requires that the programmer consider the record sizes of the data sets from which any load module text records are to be read (SYSLIB, any data set referenced by an INCLUDE, any library data set), the record size for the intermediate data set (SYSUT1), and the record size for the output load module data set (SYSLMOD).

Figure 39 lists the direct access devices that may contain data sets that are the source of input load module text, the intermediate data set, and the output load module data set, and lists the maximum record size used for each device by the linkage editor. These maximum record sizes may always be used in specifying value2 or, if the programmer can determine them, exact sizes can be used.

The programmer must specify value2 so that the linkage editor has sufficient space to allocate buffers that are compatible with the record sizes for the intermediate data set and the output load module data set.

The linkage editor optimizes the record size for the device type of output load

Device	Maximum Record Size (Bytes)	Maximum Record Size (K Bytes)
2305-1	14136	13
2305-2	14660	14
2314	7294	6
2319	7294	6
3330-1	13030	12
3330-11	13030	12
3340	8368	8
3350	19069	18

Figure 39.SYSUT1 and SYSLMOD Device Types and their Maximum Record Sizes

module data set unless one of the following conditions exists.

1. The programmer has specified PARM='...DC,...', forcing the linkage editor to write records having a maximum size of 1024 (1K) bytes.
2. The programmer has specified PARM='...DCBS,...', and the SYSLMOD DD statement contains a BLKSIZE subprarmeter in the DCB parameter, forcing the linkage editor to write records having a maximum length equal to the BLKSIZE specification.
3. The output load module data set is an existing data set having a block size less than the optimum record size, forcing the linkage editor to write records no longer than that block size.
4. The programmer has specified a value2 less than twice the maximum record size for the output load module data set, forcing the linkage editor to write records having a maximum size of one-half value2.
5. The intermediate data set and the output load module data set have dissimilar record sizes, forcing the linkage editor to write records having a maximum size determined for compatibility between the two data sets.

The linkage editor optimizes the record size of the output load module data set for its device type but selects a record size compatible with the intermediate data set (see restrictions above). Therefore, use of the load module buffer is optimized if the intermediate data set and the output load module data set reside on the same device type. The performance of the linkage editor is improved if the data sets are on different units of the same type.

Figure 40 shows the record sizes used for compatibility between every combination of device types for the intermediate and output load module data sets.

Value2 is, minimally, twice the record size for the output load module data set. If value2 can be made larger than twice the record size for the output load module data set, the increase should be the larger of the record sizes for the intermediate and output load module data sets.

The maximum for value2 is 102400 (100K). The practical maximum however, is the length of the load module to be built, plus 4K if the length of the load module to be built is equal to or greater than 40960 (40K). Any space allocated to the load module buffer above this amount is not used and need not be allocated to value2.

If a value greater than the maximum for value2 is specified, the default value for value2 is used. If a value2 is specified that cannot be accomodated in the

SYSLMOD Record Size		SYSUT1 Record Size		Minimum Load Module Buffer Area (Value2)
Device Used	Maximum Record Size Produced	Device Used	Maximum Record Size Produced	
IBM 2314	6K	2305	12K[2]	12K
IBM 2319		2314,2319	6K	12K
		3330,3330-1	12K[2]	24K
		3340	6K[2]	12K
	6K	3350	18K	18K
IBM 3330	12K	2305	12K[2]	24K
IBM 3330-1		2314,2319	6K	24K
		3330,3330-1	12K	24K
		3340	6K[2]	24K
	12K	3350	12K[2]	24K
IBM 3340	7.5K	2305	7.5K[2]	15K
IBM 3344	6K[1]	2314,2319	6K	12K
	7.5K	3330,3330-1	7.5K[2]	15K
	7.5K	3340	7.5K	15K
	7.5K	3350	15K[2]	15K
IBM 2305	13K	2305	13K	26K
	12K[1]	2314,2319	6K	24K
	12K[1]	3330,3330-1	12K	24K
	12K[1]	3340	6K	24K
	13K	3350	13K[2]	26K
IBM 3350	13K[1]	2305	13K	26K
	18K	2314,2319	6K	36K
	12K[1]	3330,3330-1	12K	24K
	18K	3340	6K	36K
	18K	3350	18K	36K

Notes:

[1] The SYSLMOD record size is reduced to less than the maximum to make it compatible with the SYSUT1 record size.

[2] The SYSUT1 record size is reduced to less than the maximum to make it compatible with the SYSLMOD record size.

Figure 40. Load Module Buffer Area and SYSLMOD and SYSUT1 Record Sizes

available storage, value2 is reduced to the next lower 2K multiple of storage that is available. This reduction, however, never decreases value2 to less than the minimum, 6144 (6K).

The optimal value2 is the practical maximum, as explained above. If the entire load module is contained in storage, the performance of the linkage editor is improved and the use of the intermediate data set may be eliminated.

Examples of Value2 Determination

1. A load module of between 21K and 22K is to be built. The load module data set is a new data set on a 3330. The intermediate data set is allocated to a 2314. A SYSLIB data set is to be used, residing on a 3330. The entire load module could be contained in the load module buffer if value2 were 22K (the load module size). The minimum for value2 would be 12K (the size of the largest possible input load module text record from the SYSLIB data set). However, value2 must be at least as large as two records to be written to the load module data set (that is, 24K). There is a reconciliation necessary in this case between the two dissimilar device types for the intermediate and output load module data sets; but the record size of the output load module data set is an even multiple of the record size of the intermediate data set so no adjustment of the record sizes is made. Therefore, the minimum, as well as the maximum and optimal, value2 in this case is 24K.

2. A load module of more than 50K is to be re-link-edited; however, a maximum of 40K is available to be allocated to value2. The output load module data set is an old data set residing on a 2314, written with maximum record size. The intermediate data set is allocated to a 2305. The link-edit involves a control section in the SYSLIN data set that will replace a control section in the old load module, followed by an INCLUDE statement naming the old load module on the SYSLMOD data set. The maximum for value2 cannot be satisfied, since only 40K is available. The size of two maximum records written to a 2314 would be 12K. However, the size of one record to be written or to be read from the intermediate data set is 12K. Therefore, the minimum for value2 in this case is 12K. This is sufficient space for one input load module text record or one record written to or to be read from the intermediate data set or two records written to the output load module data set. The optimum value2 in this case is 36K; the minimum, 12K, plus two increments of the larger of the record sizes for the intermediate data set and the output load module data set, 12K.

3. The output load module data set resides on a 2305. The intermediate data set is allocated to a 3330. All load module input comes from a 3330. Value2 in this case is 24K, because the input load module text records are, at most, 12K, the records written to and read from the intermediate data set are 12K, and the records written to the output load module data set are 12K. The maximum record size of 13K for the 2305 is reduced to 12K for this link-edit in order to be compatible with the intermediate data set.

   An alternative for value2 in the above example is 12K. 12K is adequate for the input load module text records and the records written to and read from the intermediate data set. 12K forces a maximum record size of 6K to be written to the output load module data set. At 6K each, two records can be written on a 2305 track while, as in the above example, only one record of 12K can be written on a 2305 track.

4. A load module of 10K is to be link-edited. The output load module data set resides on a 2305. The input load module libraries all reside on 2314s. The intermediate data set is allocated to a 2314. The programmer has specified the linkage editor parameter DC. The minimum for value2 of 6K is adequate in this case, since 6K is sufficient for input and intermediate data set records and the output load module data set records have a maximum size of 1K.

5. The output load module data set is a new data set allocated to a 3330. The programmer has specified the linkage editor parameter DCBS and the SYSLMOD DD statement contains '...DCB=(...BLKSIZE=3072,...),...'. The only load module input comes from a data set created previously in a similar manner. The intermediate data set is allocated to a 2314. The minimum for value2 in this case is 6K; the input load module records are 3K at most, the intermediate data set records are 6K at most, and, as directed by the programmer, the linkage editor produces records having a maximum size of 3K on the output load module data set.

Value1: Value1 specifies the number of bytes of virtual storage available to the linkage editor, regardless of the region or partition size. The storage specified by value1 includes the allocation specified by value2.

The minimum for value1 is the design point of the linkage editor, 64K. If a value less than the minimum for value1 is specified, the default options for both value1 and value2 are used.

The practical minimum value1 is 65536 (64K) plus any excess in value2 over 6144 (6K), plus any additional space required to support the blocking factor for the SYSLIN, object module library, and SYSPRINT data sets.

The design point of the linkage editor provides for the minimum load module buffer— 6144 (6K) bytes of virtual storage. If a load module buffer larger than 6144 (6K) is specified in value2, value1 must be increased by the excess of that value2 over 6144 (6K).

The linkage editor supports three different blocking factors for the SYSLIN, object module library, and SYSPRINT data sets; they are 5, 10, and 40 to 1. The requirement for additional space depends upon the blocking factor that is to be supported.

The following table shows the additional space required to support each blocking factor.

Blocking Factor	
5 to 1	0 or 0K
10 to 1	18432 or 18K
40 to 1	28672 or 28K

Blocking factors of 1 through 4, 6 through 9, and 11 through 39 are treated as blocking factors of 5, 10, and 40, respectively. Blocking factors greater than 40 are invalid.

The additional space requirement is determined by the largest blocking factor among the affected data sets.

The blocking factor supported is dependent upon space available after value2 has been allocated to the load module buffer out of value1. Therefore, if the space provided in value1 is insufficient, the link-edit will be terminated with an error message to that effect.

Value1 should be as large as possible. Generally, the performance of the linkage editor is improved when additional storage is allocated by value1.

The maximum value that can be specified for value1 is 999999 or 9999K. However, the amount of virtual storage actually allocated for value1 is the *smaller* of:

- the region or partition size
- the amount specified for value1

CAUTION: The region or partition size must be at least 10K bytes larger than the storage amount specified in value1. (See *EXEC Statement—Region Parameter* later in this chapter.) If not, an abnormal termination of the link-edit could result, because some of the storage reserved for data management and other system functions had been allocated to value1.

Examples of Value1 Determination

1. An optimum value2 of 36K has already been determined for the link-edit. An appropriate value1 is 94K, since an additional 30K, above the minimum of 64K, is needed to support the allocation of 36K to value2 and no additional storage is required to support the blocking factors for SYSLIN, SYSPRINT, and any object module libraries.
2. The minimum for value2 (6K) is being used. All of the object module libraries are blocked 5-to-1, except one that is blocked 10-to-1. The SYSLIN and SYSPRINT data sets are assigned blocking factors of 5. An appropriate value1 for this link-edit is 82K, the minimum plus the 18K needed to support the blocking factor of 10-to-1 on the object module library. Minimum region size is 92K.
3. The same situation exists as in example 2. However, in this case the minimum region size is 100K. A more appropriate value1, under these circumstances, is 90K. Since extra space is available, it is possible to optimize use of the region allocated.

DCBS Option

The DCBS option allows the programmer to specify the block size for the SYSLMOD data set in the DCB parameter of SYSLMOD DD statement.

If the DCBS option is specified, the block size value in the DSCB for the SYSLMOD data set *may* be overridden. If the DCBS option is not specified, the block size value in the DSCB for the SYSLMOD data set *may not* be overridden.

If the DCBS option is specified and no block size value is provided in the DCB parameter of the SYSLMOD DD statement, the linkage editor uses the maximum track size for the device. If the DCBS option is not specified and a block size value is provided in the DCB parameter of the SYSLMOD DD statement, the block size value in the DCB parameter of the SYSLMOD DD statement is ignored by the linkage editor.

Even though the DCBS option is specified, the linkage editor will not allow the programmer to set the block size for the SYSLMOD data set to a value less than the minimum; that is, 256, or 1024 if the SCTR option is specified, or a value less than the block size in the DSCB for an existing data set.

The block size specified by the programmer will be used unless (1) it is larger than the maximum record size for the device, in which case the maximum record size is used, or (2) it is less than the minimum block size, in which case the minimum block size is used.

The following example shows the use of the DCBS option for a 2314 disk:

```
//LKED         EXEC    PGM=HEWL,PARM='XREF,DCBS'
                .
                .
                .
//SYSLMOD      DD      DSNAME=LOADMOD(TEST),DISP=(NEW,KEEP),
//                     DCB=(BLKSIZE=3072),...
```

As a result, the linkage editor uses a 3K blocksize for the output module library.

Note: When the DCBS option is used, a blocksize must be specified in the DCB parameter of the SYSLMOD DD statement.

Output Options

These options control the optional diagnostic output produced by the linkage editor. The programmer can request that the linkage editor produce a list of all control statements and a module map or cross-reference table to help in testing a program. The format of each is described in the chapter "Output from the Linkage Editor."

In addition, the programmer can request that the numbered error/warning messages generated by the linkage editor should appear on the SYSTERM data set as well as on the SYSPRINT data set.

Control Statement Listing Option

To request a control statement listing, code LIST in the PARM field, as follows:

```
//LKED         EXEC    PGM=HEWL,PARM='LIST,...'
```

When the LIST option is specified, all control statements processed by the linkage editor are listed in card-image format on the diagnostic output data set.

Module Map Option

To request a module map, code MAP in the PARM field, as follows:

```
//LKED         EXEC    PGM=HEWL,PARM='MAP,...'
```

When the MAP option is specified, the linkage editor produces a module map of the output module on the diagnostic output data set.

Cross-Reference Table Option

To request a cross-reference table, code XREF in the PARM field, as follows:

```
//LKED         EXEC    PGM=HEWL,PARM='XREF,...'
```

When the XREF option is specified, the linkage editor produces a cross-reference table of the output module on the diagnostic output data set. The cross-reference table includes a module map; therefore, both XREF and MAP need not be specified for one linkage editor job step.

Alternate Output (SYSTERM) Option

To request that the numbered linkage editor error/warning messages be generated on the data set defined by a SYSTERM DD statement, code TERM in the PARM field, as follows:

```
//LKED         EXEC    PGM=HEWL,PARM='TERM,...'
```

When the TERM option is specified, a SYSTERM DD statement must be provided. If it is not, the TERM option is negated.

Output specified by the TERM option supplements printed diagnostic information; when TERM is used, linkage editor error/warning messages appear in both output data sets.

Incompatible Job Step Options

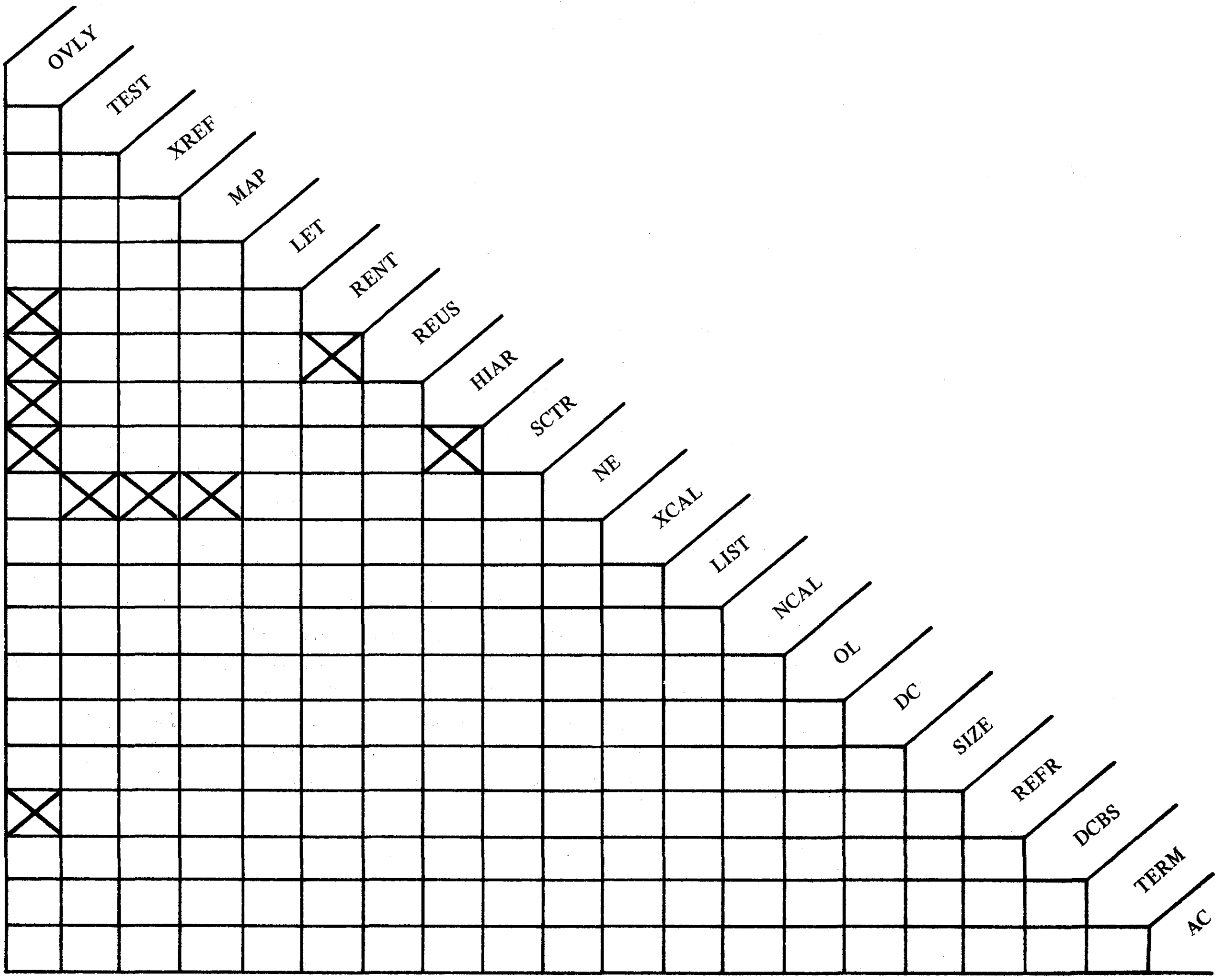
When mutually exclusive job step options are specified for a linkage editor execution, the linkage editor ignores the less significant options. Figure 41 illustrates the significance of those options that are incompatible. When an X appears at an intersection, the options are incompatible. The option that appears higher in the list is selected.

For example, to check the compatibility of XREF and NE, follow the XREF column down and the NE row across until they intersect. Since an X appears where they intersect, they are incompatible; XREF is selected, NE is negated.

If incorrect values are specified for the SIZE parameter, the default values are used. If incompatible options are detected, the message

```
*** OPTIONS INCOMPATIBLE ***
```

is printed. This message follows the standard module disposition message.



	OVLY	TEST	XREF	MAP	LET	RENT	REUS	HIAR	SCTR	NE	XCAL	LIST	NCAL	OL	DC	SIZE	REFR	DCBS	TERM
TEST																			
XREF																			
MAP																			
LET																			
RENT	X																		
REUS	X				X														
HIAR	X																		
SCTR	X						X												
NE		X	X	X															
XCAL																			
LIST																			
NCAL																			
OL																			
DC																			
SIZE																			
REFR	X																		
DCBS																			
TERM																			
AC																			

Figure 41.Incompatible Job Step Options for the Linkage Editor

EXEC Statement—Region Parameter

If the SIZE option is specified, the partition size in VS1 must be at least 10K larger than value1. If VS2 is used, the default or specified region size must be at least 10K larger than value1.

Note: Due to certain paging requirements it may be necessary to increase the 10K slightly.

For example, if SIZE=(200K,36K) is coded, the REGION specified must be 210K.

EXEC Statement—Return Code

The linkage editor passes a return code to the control program upon completion of the job step. The return code reflects the highest severity code recorded in any iteration of the linkage editor within that job step. The highest severity code encountered during processing is multiplied by 4 to create the return code; this code is placed into register 15 at the end of linkage editor processing. Figure 42 contains the return codes, the corresponding severity code, and a description of each.

Return Code	Severity Code	Description
00	0	Normal conclusion.
04	1	Warning messages have been listed, execution should be successful. For example, if the overlay option is specified and the overlay structure contains only one segment, a return code of 04 is issued.
08	2	Error messages have been listed, execution may fail. The module is marked not executable unless the LET option is specified. For example, if the blocksize of a specified library data set cannot be handled by the linkage editor, a return code of 08 is issued.
12	3	Severe errors have occurred, execution is impossible. For example, if an invalid entry point has been specified, a return code of 12 is issued.
16	4	Terminal errors have occurred, the processing has terminated. For example, if the linkage editor cannot handle the blocking factor requested for SYSPRINT, a return code of 16 is issued.

Figure 42.Linkage Editor Return Codes

The programmer may use this return code to determine whether or not the load module is to be executed by using the condition parameter (COND) on the EXEC statement for the load module. The control program compares the return code with the values specified in the COND parameter, and the results of the comparisons are used to determine subsequent action. The COND parameter may be specified either in the JOB statement or the EXEC statement (see the publication *OS/VS1 JCL Reference* or *OS/VS2 JCL).*

DD Statements

Every data set used by the linkage editor must be described with a DD statement. Each DD statement must have a name, unless data sets are concatenated. The DD statements for data sets required by the linkage editor have pre-assigned names; those for additional input data sets have user-assigned names; those for concatenated data sets (after the first) have no names.

In addition to the name, the DD statement provides the control program with information about the input/output device on which the data set resides, and a description of the data set itself. All of the job control language facilities for device description are available to the users of the linkage editor.

Besides information about the device, the DD statement also contains a data set description, which includes the data set name and its disposition. Information for the data control block (DCB) may also be given.

General information pertinent to the linkage editor on the data set name and DCB information follows; information on disposition is given in the discussion for each data set.

Data Set Name: The linkage editor uses either sequential or partitioned data sets. For sequential data sets, only the name of the data set is specified; for partitioned data sets, the member name must also be specified either on the DD statement or with a control statement.

When input data sets are passed from a previous job step, or when the output load module is being tested, a recommended practice is to use temporary data set names (that is, &&dsname). Use of temporary names ensures that there are no duplicate data sets with out-of-date modules. A data set with a temporary name is automatically deleted at the end of the job. When a module is to be stored permanently, a data set name without ampersands is used.

DCB Information: Before a data set can be used for input, information describing the data set must be placed in the data control block (DCB). If this information does not exist in the DCB or header label, or if no labels are used (magnetic tape does not require labels), the programmer must specify it in the DCB parameter on the DD statement.

Record format (RECFM), logical record size (LRECL), and blocksize (BLKSIZE) subparameters of the DCB parameter are discussed as they apply to the linkage editor. Specific information on each as it applies to the linkage editor data sets is given in the description of the data set which follows later in this section. Other DCB information (tape recording technique, density, and so forth) is described in the publication *OS/VS1 JCL Reference* or *OS/VS2 JCL.*

Record Format:

The following record formats are used with the linkage editor:

F The records are fixed length.

FB The records are fixed length and blocked.

FBA The records are fixed length, blocked, and contain ANSI control characters.

FBS The records are fixed length, blocked, and written in standard blocks.

FA The records are fixed length and contain ANSI control characters.

FS The records are fixed length and written in standard blocks.

U The records are undefined length.

UA The records are undefined length and contain ANSI control characters.

A record format of FS or FBS must be used with caution. All blocks in the data set must be the same size. This size must be equal to the specified blocksize. A truncated block can occur only as the last block in the data set.

Note: Track overflow is never used by the linkage editor. When moving or copying load modules, it is recommended that the track overflow feature not be used on the target data set as errors may occur in fetching the load modules for execution.

Logical Record and Blocksize: Blocking is allowed for input object module data sets and the diagnostic output data set. The blocking factors used to determine buffer allocations are 10 and 40. The BLKSIZE must therefore be a multiple of LRECL. See the description of blocking factors in the discussion of the SIZE opion.

Also, a blocksize should be specified for the output load module library when the DCBS option is specified (see "SYSLMOD DD Statement" later in this section).

Linkage Editor DD Statements

The linkage editor uses six data sets; of these, four are required. The DD statements for these data sets must use the preassigned ddnames given in Figure 43. The descriptions that follow give pertinent device and data set information for each linkage editor data set.

Data Set	ddname	Required
Primary input data set	SYSLIN	Yes
Automatic call library	SYSLIB	Only if the automatic library call mechanism is used
Intermediate data set	SYSUT1	Yes
Diagnostic output data set	SYSPRINT	Yes
Output module library	SYSLMOD	Yes
Alternate output data set	SYSTERM	Only if the TERM option is specified

Figure 43. Linkage Editor ddnames

SYSLIN DD Statement

The SYSLIN DD statement is always required; it describes the primary input data set which can be assigned to a direct-access device, a magnetic tape unit, or the card reader. The data set may be either sequential or partitioned; in the latter case, a member name must be specified.

If SYSLIN is assigned to a card reader or "pseudo card reader," input records must be unblocked and 80-bytes long. (A pseudo card reader is defined as input from a tape or direct-access device in card reader mode.)

This data set must contain object modules and/or control statements. Load modules used in the primary input data set are considered a severity 4 error.

The recommended disposition for the primary input data set is SHR or OLD.

The DCB requirements are shown in Figure 44.

DCB Requirements		
LRECL	BLKSIZE	RECFM
80	80	F,FS
80	400,800,3200*	FB,FBS

* These are the maximum blocksizes allowed. Which maximum is applicable depends on the value given to value1 and value2 of the SIZE option.

Figure 44. DCB Requirements for Object Module and Control Statement Input

SYSLIB DD Statement

The SYSLIB DD statement is required when the automatic library call mechanism is to be used. This DD statement describes the automatic call library, which must be assigned to a direct-access device. The data set must be partitioned, but member names should not be specified.

The recommended disposition for the call library is SHR or OLD.

If concatenated call libraries are used, object and load module libraries must not be mixed. If only object modules are used, the call library may also contain control statements.

The DCB requirements for object module call libraries are given in Figure 44. The DCB requirement for load module call libraries is a record format of U; the blocksize used for storage allocation is equal to the maximum for the device used, not the record read.

This data set must not be assigned to SYSOUT.

SYSUT1 DD Statement

The SYSUT1 DD statement is always required; it describes the intermediate data set, which is a sequential data set assigned to a direct-access device. Space must be allocated for this data set but the DCB requirements are supplied by the linkage editor.

SYSPRINT DD Statement

The SYSPRINT DD statement is always required; it describes the diagnostic output data set, which is a sequential data set assigned to a printer or an intermediate storage device. If an intermediate storage device is used, the data records contain a carriage control character as the first byte.

The usual specification for this data set is SYSOUT=A. The programmer may assign a blocksize if he is running under a VS1 or VS2 system. The record format assigned by the linkage editor depends on whether blocking is used or not.

Figure 45 shows the DCB requirements for SYSPRINT. The bold-face type represents information supplied by the linkage editor. The only information that can be supplied by the programmer is the blocksize.

DCB Requirements for SYSPRINT

LRECL	BLKSIZE	RECFM
121	**121**	**FA**
121	*n* x 121 where *n* is less than or equal to 40	**FBA**

Note: The value specified for BLKSIZE, either on the DCB parameter of the SYSPRINT DD statement or in the DSCB (data set control block) of an existing data set, must be a multiple of 121; if it is not, the linkage editor issues a message to the operator's console and terminates processing.

Figure 45. DCB Requirements for SYSPRINT

SYSLMOD DD Statement

The SYSLMOD DD statement is always required; it describes the output module library, which must be a partitioned data set assigned to a direct-access device.

A member name may be specified on the SYSLMOD DD statement. If a member name is specified, it is used only if a name was not specified on a NAME control statement. This member name must conform to the rules for the name on the NAME control statement. This would imply the replacement of an identically named member in the output load module library, if one exists.

If the member is to replace an identically named member in an existing library, the disposition should be OLD or SHR. If the member is to be added to an existing library, the disposition should be MOD, OLD, or SHR. If no library exists and the member is the first to be added to a new library, the disposition should be NEW or MOD. If the member is to be added to an existing library that may be used concurrently in another region or partition, the disposition should be SHR.

The record format U is assigned by the linkage editor. See Appendix G.

The linkage editor assigns a blocksize by:

1. Finding the smallest of the following values:
   - The maximum track size for the device
   - The value of the BLKSIZE subparameter in the DCB parameter on the SYSLMOD DD statement, if the DCBS option was specified
   - 1024, if the DC option was specified
   - The actual output buffer length (half the number specified for value2 of the SIZE option)
2. Comparing the smallest value above to the value currently in the DSCB. The greater value is assigned as the block size.

   Note: When a new data set is created at linkage editor time, without the DCBS option specified, the DSCB will reflect the maximum blocksize available for the device type.

   If the SYSLMOD DD statement is used as a source of load module input, the SYSLMOD data set is read with a record format of U in all cases.

In the following example, the SYSLMOD DD statement specifies a permanent library on an IBM 2314 Disk Storage Device:

```
//SYSLMOD     DD      DSNAME=USERLIB( TAXES ),DISP=MOD,
//                    UNIT=2314,...
```

The linkage editor assigns a record format of U, and a logical record and blocksize of 6K, the maximum for a 2314. However, consider the following example:

```
//LKED        EXEC    PGM=HEWL,PARM='XREF,DCBS'
               .
               .
               .
//SYSLMOD     DD      DSNAME=USERLIB( TAXES ),DISP=MOD,
//                    UNIT=2314,DCB=( BLKSIZE=3072 ),...
```

The linkage editor still assigns a record format of U, but the logical record and block size are now 3K rather than 6K, due to the use of the DCBS option.

SYSTERM DD Statement

The SYSTERM DD statement is optional; it describes a data set that is used only for numbered error/warning messages. Although intended to define the terminal data set when the linkage editor is being used under the Time Sharing Option (TSO) of VS2, the SYSTERM DD statement can be used in any environment to define a data set consisting of numbered error/warning messages that supplements the SYSPRINT data set.

SYSTERM output is defined by including a SYSTERM DD statement and specifying TERM in the PARM field of the EXEC statement. When SYSTERM output is defined, numbered messages are then written to both the SYSTERM and SYSPRINT data sets.

The following example shows how the SYSTERM DD statement could be used to specify the system output unit:

```
//SYSTERM     DD      SYSOUT=A
```

The DCB requirements for SYSTERM (LRECL=121,BLKSIZE=121, and RECFM=FBA) are supplied by the linkage editor. If necessary, the linkage editor will modify the DSCB (data set control block) of an existing data set to reflect these values.

Additional DD Statements

Each ddname specified on an INCLUDE or LIBRARY control statement must also be described with a DD statement. These DD statements describe sequential or partitioned data sets, assigned to magnetic tape units or direct-access devices.

The ddnames are specified by the user along with any other necessary information. The DCB requirements for these data sets are shown in Figure 46.

DCB Requirements

Data Set Contents	LRECL	BLKSIZE	RECFM
Include Control Statement			
Object modules and/or control statements	80	80	F,FS
Load modules	1K	1K	U
Library Control Statement			
Object modules and/or control statements	80	80 400,800,3200*	F,FS FB,FBS
Load Modules	maximum for device, or one-half of value2, whichever is smaller	equal to LRECL	U

* These are the maximum blocksizes allowed. Which maximum is applicable depends on the values given to value1 and value2 of the SIZE option.

Figure 46.DCB Requirements for Additional Input Data Sets

When concatenated data sets are included, each data set must contain records of the same format, record size, and blocksize. If the data sets reside on magnetic tape, the tape recording technique and density must also be identical.

If the SYSLMOD DD statement is used as a source of load module input, the SYSLMOD data set is read with a record format of U in all cases.

Cataloged Procedures

To facilitate the operation of the system, the control program allows the programmer to store EXEC and DD statements under a unique member name in a procedure library. Such a series of job control language statements is called a *cataloged procedure*. These job control language statements can be recalled at any time to specify the requirements for a job. To request this procedure, the programmer places an EXEC statement in the input stream. The EXEC statement specifies the unique member name of the procedure desired.

The specifications in a cataloged procedure can be temporarily overridden, and DD statements can be added. The information altered by the programmer is in effect only for the duration of the job step; the cataloged procedures themselves are not altered permanently. Any additional DD statements supplied by the programmer must follow those that override the cataloged procedure.

Linkage Editor Cataloged Procedures

Two linkage editor cataloged procedures are provided: a single-step procedure that link edits the input and produces a load module (procedure LKED), and a two-step procedure that link edits the input, produces a load module, and executes that module (procedure LKEDG). Many of the cataloged procedures provided for language translators also contain linkage editor steps. The EXEC and DD statement specifications in these steps are similar to the specifications in the cataloged procedures described in the following paragraphs.

Procedure LKED

The cataloged procedure named LKED is a single-step procedure that link edits the input, produces a load module, and passes the load module to another step in the same job. The statements in this procedure are shown in Figure 47; the following is a description of these statements.

Statement Numbers: The 8-digit numbers on the right-hand side of each statement are used to identify each statement and would be used, for example, when permanently modifying the cataloged procedure with the system utility program IEBUPDTE. For a description of this utility program, see the publications *OS/VS1 Utilities* and *OS/VS2 MVS Utilities.*

EXEC Statement: The PARM field specifies the XREF, LIST, LET, and NCAL options. If the automatic library call mechanism is to be used, the NCAL option must be overridden, and a SYSLIB DD statement must be added. Overriding and adding DD statements is discussed later in this section.

SYSPRINT Statement: The SYSPRINT DD statement specifies the SYSOUT class A, which is either a printer or an intermediate storage device. If an intermediate storage device is used, ANS carriage control characters

```
//LKED       EXEC   PGM=HEWL,PARM='XREF,LIST,LET,NCAL',REGION=96K 00020000
//SYSPRINT   DD     SYSOUT=A                                       00040000
//SYSLIN     DD     DDNAME=SYSIN                                   00060000
//SYSLMOD    DD     DSNAME=&&GOSET(GO),SPACE=(1024,(50,20,1)),    C00080000
//                  UNIT=SYSDA,DISP=(MOD,PASS)                     00100000
//SYSUT1     DD     UNIT=(SYSDA,SEP=(SYSLMOD,SYSLIN)),            C00120000
//                  SPACE=(1024,(200,20))                          00140000
```

Figure 47.Statements in the LKED Cataloged Procedure accompany the data to be printed.

SYSLIN Statement: The specification of DDNAME=SYSIN allows the programmer to specify any input data set as long as it fulfills the requirements for linkage editor input. The input data set must be defined with a DD statement with the ddname SYSIN. This data set may be either in the input stream or residing on a separate volume.

If the data set is in the input stream, the following SYSIN statement is used:

```
//LKED.SYSIN   DD      *
```

If this SYSIN statement is used, it may be anywhere in the job step DD statements as long as it follows all overriding DD statements. The object module decks and/or control statements should follow the SYSIN statement, with a delimiter statement (/*) at the end of the input.

If the data set resides on a separate volume, the following SYSIN statement is used:

```
//LKED.SYSIN DD        ( parameters describing the input data set )
```

If this SYSIN statement is used, it may be anywhere in the job step DD statements as long as it follows all overriding DD statements. Several data sets may be concatenated as described in the chapter "Input to the Linkage Editor."

SYSLMOD Statement: The SYSLMOD DD statement specifies a temporary data set and a general space allocation. The disposition allows the next job step to execute the load module. If the load module is to reside permanently in a library, these general specifications must be overridden.

SYSUT1 Statement: The SYSUT1 DD statement specifies that the intermediate data set is to reside on a direct-access device, but not the same device as either the SYSLMOD or the SYSLIN data sets. Again, a general space allocation is given.

SYSLIB Statement: Note that there is no SYSLIB DD statement. If the automatic library call mechanism is to be used with a cataloged procedure, a SYSLIB DD statement must be added; also, the NCAL option in the PARM field of the EXEC statement must be negated.

Invoking the LKED Procedure: To invoke the LKED procedure, code the following EXEC statement:

```
//stepname   EXEC   LKED
```

where *stepname* is optional and is the name of the job step.

The following example shows a sample JCL sequence for using the LKED procedure in one step to link-edit object modules to produce a load module, then executing the load module in a subsequent step.

```
//LESTEP       EXEC    LKED
   ( Overriding and/or additional DD statements
   for the LKED step )
//LKED.SYSIN DD        *
   ( Object module decks and/or
   control statements )
//EXSTEP       EXEC    PGM=*.LESTEP.LKED.SYSLMOD
   ( DD statements and data for
   load module execution )
/*      ( If data is supplied for the execution step )
```

Note: LESTEP invokes the LKED procedure and EXSTEP executes the load module produced by LESTEP.

Procedure LKEDG

The cataloged procedure named LKEDG is a two-step procedure that link edits the input, produces a load module, and executes that load module. The statements in this procedure are shown in Figure 48. The two steps are named LKED and GO. The specifications in the statements in the LKED step are identical to the specifications in the LKED procedure.

GO Step: The EXEC statement specifies that the program to be executed is the load module produced in the LKED step of this job. This module was stored in the data set described on the SYSLMOD DD statement in that step. (If a NAME statement was used to specify a member name other than that used on the SYSLMOD statement, use the LKED procedure.)

The condition parameter specifies that the execution step is bypassed if the return code issued by the LKED step is greater than 4.

Invoking the LKEDG Procedure: To invoke the LKEDG procedure, code the following EXEC statement:

```
//stepname    EXEC    LKEDG
```

where *stepname* is optional and is the name of the job step.

The following example shows a sample JCL sequence for using the LKEDG procedure to link-edit object modules, produce a load module, and execute that load module.

```
//LKED        EXEC   PGM=HEWL,PARM='XREF,LIST,NCAL',REGION=96K        00020000
//SYSPRINT    DD     SYSOUT=A                                         00040000
//SYSLIN      DD     DDNAME=SYSIN                                     00060000
//SYSLMOD     DD     DSNAME=&&GOSET(GO),SPACE=(1024,(50,20,1)),      C00080000
//                   UNIT=(SYSDA,DISP=(MOD,PASS)                      00100000
//SYSUT1      DD     UNIT=(SYSDA,SEP=(SYSLMOD,SYSLIN)),              C00120000
//                   SPACE=(1024,(200,20))                            00140000
//GO          EXEC   PGM=*.LKED.SYSLMOD,COND=(4,LT,LKED)              00160000
```

Figure 48.Statements in the LKEDG Cataloged Procedure

```
//TWOSTEP     EXEC    LKEDG
```

(Overriding and/or additional DD statements for the LKED step.)

```
//LKED.SYSIN DD      *
```

(Object module decks and/or control statements)

```
/*
```

(DD statements for the GO step)

```
//GO.SYSIN    DD      *
```

(Data for the GO step)

```
/*
```

Overriding Cataloged Procedures

The programmer may override any of the EXEC or DD statement specifications in a cataloged procedure. These new specifications remain in effect only for the duration of the job step. For a detailed description of overriding cataloged procedures, see the publication *OS/VS1 JCL Reference* or *OS/VS2 JCL.*

Overriding the EXEC Statement

The EXEC statement in a cataloged procedure is overridden by specifying the changes and additions on the EXEC statement that invokes the cataloged procedure. The stepname should be specified when overriding the EXEC statement parameters.

For example, the REGION parameter can be increased as follows:

```
//LESTEP      EXEC    LKED,REGION.LKED=136K
```

The rest of the specifications on the EXEC statement of procedure LKED remain in effect.

If the PARM field is to be overridden, all of the options specified in the cataloged procedure are negated. That is, if XREF, LIST, or NCAL is desired when overriding the PARM field, it must be respecified. In the following example, the OVLY option is added and the NCAL option is negated:

```
//LESTEP      EXEC    LKED,PARM.LKED='OVLY,XREF,LIST'
```

As a result, the XREF and LIST options are retained, but the NCAL option is negated; when NCAL is negated, a SYSLIB DD statement must be added.

If you use the LKEDG procedure and want to execute the load module just built, an efficient way is to specify the parameter LET in the LKED step and invoke the LKEDG procedure with the following EXEC statement:

```
//stepname    EXEC    LKEDG,PARM.LKED='XREF,LIST,NCAL,LET',
//                    COND.GO=(8,LT,LKED)
```

Overriding DD Statements

Each DD statement that is used to override a DD statement in the LKED step of either the LKED procedure or the LKEDG procedure must begin with `//LKED.ddname....`

Each DD statement that is used to override a DD statement in the GO step of the LKEDG procedure must begin with `//GO.ddname....`

Any of the DD statements in the cataloged procedures can be overridden as long as the overriding DD statements are in the same order as they appear in the procedure. If any DD statements are not overridden, or overriding DD statements are included but are not in sequence, the specifications in the cataloged procedure are used.

Only those parameters specified on the overriding DD statement are affected; the rest of the parameters remain as specified in the procedure. In the following example, the output load module is to be placed in a permanent library:

```
//LIBUPDTE   EXEC   LKED
//LKED.SYSLMOD DD   DSNAME=LOADLIB(PAYROLL),DISP=OLD
//LKED.SYSIN   DD   DSNAME=OBJMOD,DISP=(OLD,DELETE)
```

Unit and volume information should be given if these data sets are not cataloged.

As a result of the statements in the example, the LKED procedure is used to process the object module in the OBJMOD data set. The output load module is stored in the data set LOADLIB with the name PAYROLL. The SPACE parameter on the SYSLMOD DD statement and the other specifications in the procedure remain in effect.

Adding DD Statements

DD statements for additional data sets can be supplied when using cataloged procedures. These additional DD statements must follow any overriding DD statements.

Each additional DD statement for the LKED step must begin with //LKED.ddname... and for the GO step must begin with //GO.ddname....

In the following example, the automatic library call mechanism is to be used along with the LKEDG procedure:

```
//CPSTEP         EXEC LKEDG,PARM.LKED='XREF,LIST'
//LKED.SYSLMOD DD   DSNAME=LOADLIB(TESTER),DISP=OLD,...
//LKED.SYSLIB  DD   DSNAME=SYL1.PL1LIB,DISP=SHR
//LKED.SYSIN   DD   *
   ( Object module decks and/or control statements )
/*
//GO.SYSIN     DD      *
   ( Data for execution step )
/*
```

The NCAL option is negated, and a SYSLIB DD statement is added between the overriding SYSLMOD DD statement and the SYSIN DD statement.

LINKAGE EDITOR CONTROL STATEMENT SUMMARY

This chapter summarizes the linkage editor control statements. The description of each statement includes:

- What the statement does
- The format of the statement
- Placement of the statement in the input
- Notes on use, if any
- One or more examples that include job control language statements, when necessary.

The control statements are described in alphabetical order. Before using this chapter, the user should be familiar with the following information on general format, format conventions, and placement.

Note: If the control statement to specify hierarchy format (HIARCHY) is specified for VS, the linkage editor prepares the load module accordingly. However, hierarchy format is not supported by VS, and it is ignored during execution of the load module.

General Format

Each linkage editor control statement specifies an *operation* and one or more *operands*. Nothing must be written preceding the operation, which must begin in or after columm 2. The operation must be separated from the operand by one or more blanks.

A control statement can be continued on as many cards as necessary by terminating the operand at a comma, and by placing a nonblank character in column 72 of the card. Continuation must begin in column 16 of the next card. A symbol cannot be split; that is, it cannot begin on one card and be continued on the next.

Format Conventions

The following conventions are used in the formats to describe the coding of the linkage editor control statements:

- **Boldface** type indicates the exact characters to be entered. Such items must be entered exactly as illustrated (in upper case, if applicable).
- *Italic* type specifies fields to be supplied by the user.
- Other punctuation (parentheses, commas, spaces, etc.) must be entered as shown.
- Braces { } indicate a choice of entry; unless a default is indicated, you must choose one of the entries.
- Brackets [ ] indicate an optional field or parameter.
- An ellipsis (...) indicates that multiple entries of the type immediately preceding the ellipsis are allowed.
- Items separated by a vertical bar (|) represent alternative items. No more than one of the items may be selected.

Placement Information

Linkage editor control statements are placed before, between, or after modules. They can be grouped, but they cannot be placed within a module. However, specific placement restrictions may be imposed by the nature of the functions being requested by the control statement. Any placement restrictions are noted.

ALIAS Statement

The ALIAS statement specifies additional names for the output library member, and also can specify names of alternative entry points. Up to 16 names can be specified on one ALIAS statement, or separate ALIAS statements for one library member. The names are entered in the directory of the partitioned data set in addition to the member name.

Format: The format of the ALIAS statement is:

ALIAS	{*symbol* | *external name* },...

symbol
specifies an alternate name for the load module. When the module is executed, the main entry point is used as the starting point for execution.

external name
specifies a name that is defined as a control section name or entry name in the output module. When the module is called for execution, execution begins at the external name referred to.

Placement: An ALIAS statement can be placed before, between, or after object modules or other control statements. It must precede a NAME statement used to specify the member name, if one is present.

Notes:

- In an overlay program, an external name specified by the ALIAS statement must be in the root segment.
- No more than 16 alias names can be assigned to one output module.
- Each alias specified for a load module is retained in the directory entry for the module; the linkage editor does not delete an old alias. Therefore, each alias that is specified must be unique; assigning the same alias to more than one load module can cause incorrect module reference.
- Obsolete alias names should be deleted from the PDS directory using a system utility such as IEHPROGM, to avoid future name conflicts.
- If the replace option is in effect for the output load module (that is, the load module built in this link edit does or may replace an identically named load module in the output module library), the replace option is in effect for each ALIAS name for the load module as well as the primary name.

Example: An output module, ROUT1, is to be assigned two alternate entry points, CODE1 and CODE2. In addition, calling modules have been written using both ROUT1 and ROUTONE to refer to the output module. Rather than correct the calling modules, an alternative library member name is also assigned.

```
ALIAS        CODE1,CODE2,ROUTONE
NAME         ROUT1
```

Since CODE1 and CODE2 are entry names in the output module, when these names are used to call the module, execution begins at the point referred to. The modules that call the output module with the name ROUTONE now correctly refer to ROUT1 at its main entry point. The names CODE1, CODE2, and ROUTONE appear in the library directory along with ROUT1.

CHANGE Statement

The CHANGE statement causes an external symbol to be replaced by the symbol in parentheses following the external symbol. The external symbol to be changed can be a control section name, an entry name, or an external reference. More than one such substitution may be specified in one CHANGE statement.

Format: The format of the CHANGE statement is:

CHANGE	*externalsymbol*(*newsymbol*) [,*externalsymbol*(*newsymbol*)]...

externalsymbol
is the control section name, entry name, or external reference that is to be changed.

newsymbol
is the name to which the external symbol is to be changed.

Placement: The CHANGE control statement must be placed immediately before either the module containing the external symbol to be changed, or the INCLUDE control statement specifying the module. The scope of the CHANGE statement is across the immediately following module (object module or load module); the END record in the immediately following object module or the End-of-Module indication in the immediately following load module delimits the scope of the CHANGE statement.

Notes:

- External references from other modules to a changed control section name or entry name remain unresolved unless further action is taken.
- If the symbol specified on the CHANGE statement is inadvertently misspelled, the symbol will not be changed. Linkage editor output, such as the cross-reference listing or module map, can be used to verify each change.
- When a REPLACE statement that deletes a control section is followed by a CHANGE statement with the same control section name, unpredictable results will occur.

Example 1: Two control sections in different modules have the name TAXROUT. Since both modules are to be link edited together, one of the control section names must be changed. The module to be changed is defined with a DD statement named OBJMOD. The control section name could be changed as follows:

```
//OBJMOD     DD     DSNAME=TAXES,DISP=(OLD,KEEP),...
//SYSLIN     DD     *
  CHANGE  TAXROUT(STATETAX)
  INCLUDE OBJMOD
/*
```

As a result, the name of control section TAXROUT in module TAXES is changed to STATETAX. Any references to TAXROUT from other modules are not affected.

Example 2: A load module contains references to TAXROUT that must now be changed to STATETAX. This module is defined with a DD statement named LOADMOD. The external references could be changed at the same time the control section name is changed, as follows:

```
//OBJMOD      DD      DSNAME=TAXES,DISP=(OLD,DELETE),...
//LOADMOD     DD      DSNAME=LOADLIB,DISP=OLD,...
//SYSLIN      DD      *
  CHANGE TAXROUT(STATETAX)
  INCLUDE OBJMOD
  CHANGE TAXROUT(STATETAX)
  INCLUDE LOADMOD(INVENTRY)
/*
```

As a result, control section name TAXROUT in module TAXES and external reference TAXROUT in module INVENTRY are both changed to STATETAX. Any references to TAXROUT from other modules are not affected.

ENTRY Statement

The ENTRY statement specifies the symbolic name of the first instruction to be executed when the program is called by its module name for execution. An ENTRY statement should be used whenever a module is reprocessed by the linkage editor. If more than one ENTRY statement is encountered, the first statement specifies the main entry point; all other ENTRY statements are ignored.

Format: The format of the ENTRY statement is:

ENTRY	*externalname*

external name
is defined as either a control section name or an entry name in a linkage editor input module.

Placement: An ENTRY statement can be placed before, between, or after object modules or other control statements. It must precede the NAME statement for the module, if one is present.

Notes:

- In an overlay program, the first instruction to be executed must be in the root segment.
- The external name specified must be the name of an instruction, not a data name, if the module is to be executed.

Example: In the following example, the main entry point is INIT1:

```
//LOADLIB    DD     DSNAME=LOADLIB,DISP=OLD,...
//SYSLIN     DD     *
  ENTRY INIT1
  INCLUDE LOADLIB(READ,WRITE)
      .
      .
      .
  ENTRY READIN
/*
```

INIT1 must be either a control section name or an entry name in the linkage editor input. The entry point specification of READIN is ignored.

EXPAND Statement

The EXPAND statement lengthens control sections or named common sections by a specified number of bytes.

Format: The format of an EXPAND statement is

EXPAND	*name* (*xxxx*) [,*name* (*xxxx*)]...

name
is the symbolic name of a common section or control section whose length is to be increased.

xxxx
is the decimal number of bytes to be added to the length of a common section. Binary zeros will be added for an expanded control section. The maximum is 4095 for each section indicated.

The EXPAND statement is followed by a message, IEW0740, that indicates the number of bytes added to the control section and the offset, relative to the start of the control section, at which the expansion begins. The *effective* length of the expansion is given in hexadecimal and may be greater than the *specified* length if, after the specified expansion, padding bytes must be added for alignment of the next control section or named common section.

Placement: An EXPAND statement can be placed before, between, or after other control statements or object modules. However, the statement must follow the module containing the control or named common section to which it refers. If the control section or named common section is entered as the result of an INCLUDE statement, the EXPAND statement must follow the INCLUDE statement.

Note: EXPAND should be used with caution so as not to increase the length of a program beyond its own design limitations. For example, if space is added to a control section beyond the range of its base register addressability, that space is unusable.

Example: In the following example EXPAND statements add a 250-byte patch area (initialized to zeros) at the end of control section CSECT1 and increase the length of named common section COM1 by 400 bytes.

```
//LKED        EXEC   PGM=HEWL
//SYSPRINT    DD     SYSOUT=A
//SYSUT1      DD     UNIT=SYSDA,SPACE=(TRK,(10,4))
//SYSLMOD     DD     DSNAME=PDSX,DISP=OLD
//SYSLIN      DD     DSNAME=&&LOADSET,DISP=(OLD,PASS),
//                   UNIT=SYSDA
//            DD     *
  EXPAND             CSECT1(250)
  EXPAND             COM1(400)
  NAME               MOD1(R)
/*
```

IDENTIFY Statement

The IDENTIFY statement specifies any data supplied by the user to be entered into the CSECT Identification (IDR) records for a particular control section. The statement can be used either to supply descriptive data for a control section or to provide a means of associating system-supplied data with executable code.

Format: The format of the IDENTIFY statement is:

IDENTIFY	*csectname* ('*data*')[,*csectname* ('*data*')]...

csectname
is the symbolic name of the control section to be identified.

data
specifies up to 40 EBCDIC characters of identifying information. The user may supply any information desired for identificaionn purposes.

The rules of syntax for the operand field are:

1. No blanks or characters may appear between the left parenthesis and the leading quote nor between the trailing quote and the right parenthesis.
2. The data field consists of from 1 to 40 characters; therefore, a null entry must be represented, minimally, by a single blank.
3. Blanks may appear between the leading quote and the trailing quote. Each blank counts as 1 character toward the 40 character limit.
4. A single quote between the leading quote and the trailing quote is represented by 2 consecutive quotes. The pair of quotes counts as 1 character toward the 40 character limit.
5. Any EBCDIC character may appear between the leading quote and the trailing quote. Each character counts as 1 character toward the 40 character limit.
6. The IDENTIFY statement may be continued; however, a whole operand must appear on a single card image and at least 1 whole operand must appear on each card image of the continued statement.
7. If a leading quote is found, all characters are absorbed until a trailing quote is found or the 40 character limit is exhausted.
8. Blanks may not appear between the CSECT name and the left parenthesis.
9. A blank following a left parenthesis terminates the operand field; a blank following a comma that terminates an operand terminates the operand field of that card image.

Placement: An IDENTIFY statement can be placed before, between, or after other control statements or object modules. The IDENTIFY statement must follow the module containing the control section to be identified or the INCLUDE statement specifying the module.

Note: When two or more IDENTIFY statements specify the same CSECT name, only the last statement is effective.

Example: In the following example, IDENTIFY statements are used to identify the source level of a control section, a PTF application to a control section, and the functions of several control sections.

```
//LKED         EXEC    PGM=HEWL
//SYSPRINT     DD      SYSOUT=A
//SYSUT1       DD      UNIT=SYSDA,SPACE=(TRK,(10,5))
//SYSLMOD      DD      DSNAME=LOADSET,DISP=OLD
//OLDMOD       DD      DSNAME=OLD.LOADSET,DISP=OLD
//PTFMOD       DD      DSNAME=PTF.OBJECT,DISP=OLD
//SYSLIN       DD      *
 (input object deck for a control section named FORT)
  IDENTIFY     FORT('LEVEL 03')
  INCLUDE      PTFMOD(CSECT4)
  IDENTIFY     CSECT4('PTF99999')
  INCLUDE      OLDMOD(PROG1)
  IDENTIFY     CSECT1('I/O ROUTINE'),                                  X
               CSECT2('SORT ROUTINE'),                                 X
               CSECT3('SCAN ROUTINE')
/*
```

Execution of this example produces IDR records containing the following identification data:

- The name of the linkage editor that produced the load module, the linkage editor version and modification level, and the date of the current linkage editor processing of the module. This information is provided automatically.
- User-supplied data describing the functions of several control sections in the module, as indicated on the third IDENTIFY statement.
- If the language translator used supports IDR, the Identification records produced by the linkage editor also contain the name of the translator that produced the object module, its version and modification level, and the data of compilation.

The IDR records created by the linkage editor can be referenced by using the LISTIDR function of the service aid program HMBLIST for VS1 or AMBLIST for VS2. For instructions on how to use HMBLIST, see *OS/VS1 Service Aids*. For instructions on how to use AMBLIST, see *OS/VS2 System Programming Library: Service Aids*.

INCLUDE Statement

The INCLUDE statement specifies sequential data sets and/or libraries that are to be sources of additional input for the linkage editor. INCLUDE statements are processed in the order in which they appear in the input. However, the sequence of data sets and modules within the output load module does not necessarily follow the order of the INCLUDE statements.

Format: The format of the INCLUDE statement is:

INCLUDE	*ddname* [(*membername* [,...])] [,*ddname* [(*membername* [,...])]]...

ddname
is the name of a DD statement that describes either a sequential or a partitioned data set to be used as additional input to the linkage editor. For a sequential data set, ddname is all that must be specified. For a partitioned data set, at least one member name must also be specified.

membername
is the name of or an alias for a member of the library defined in the specified DD statement. The membername must not be specified again on the DD statement.

Placement: An INCLUDE statement can be placed before, between, or after object modules or other control statements.

Note: A NAME statement in any data set specified in an INCLUDE statement is invalid; the NAME statement is ignored. All other control statements are processed.

Example 1: In the following example, an INCLUDE statement specifies two data sets to be the input to the linkage editor:

```
//OBJMOD      DD      DSNAME=&&OBJECT,DISP=(OLD,DELETE)
//LOADMOD     DD      DSNAME=LOADLIB,DISP=SHR,...
                .
                .
                .
//SYSLIN      DD       *
  INCLUDE OBJMOD,LOADMOD(TESTMOD,READMOD)
/*
```

Note that a DD statement must be supplied for every ddname specified in an INCLUDE statement.

Example 2: Two separate INCLUDE statements could have been used in the preceding example, as follows:

```
  INCLUDE OBJMOD
  INCLUDE LOADMOD(TESTMOD,READMOD)
```

INSERT Statement

The INSERT statement repositions a control section from its position in the input sequence to a segment in an overlay structure. However, the sequence of control sections within a segment is not necessarily the order of the INSERT statements.

If a symbol specified in the operand field of an INSERT statement is not present in the external symbol dictionary, it is entered as an external reference. If the reference has not been resolved at the end of primary input processing, the automatic library call mechanism attempts to resolve it.

Format: The format of the INSERT statement is:

INSERT	*csectname*,...

csectname
is the name of the control section to be repositioned. A particular control section can appear only once within a load module.

Placement: The INSERT statement must be placed in the input sequence following the OVERLAY statement that specifies the origin of the segment in which the control section is to be positioned. If the control section is to be positioned in the root segment, the INSERT statement must be placed before the first OVERLAY statement.

Note: Control sections that are positioned in a segment must contain all address constants to be used during execution unless:

- The A-type address constants are located in a segment in the path.
- The V-type address constants used to pass control to another segment are located in the path. If an exclusive reference is made, the V-type address constant must be in a common segment.
- The V-type address constants used with the SEGLD and SEGWT macro instructions are located in the segment.

Example: The following INSERT (and OVERLAY) statements specify the overlay structure shown in Figure 49:

```
//            EXEC   PGM=HEWL,PARM='OVLY,XREF,LIST'
               .
               .
               .
//SYSLIN      DD     *
  INSERT CSA
  INSERT CSB
  OVERLAY ALPHA
  INSERT CSC,CSD
  OVERLAY ALPHA
  INSERT CSE
/*
```

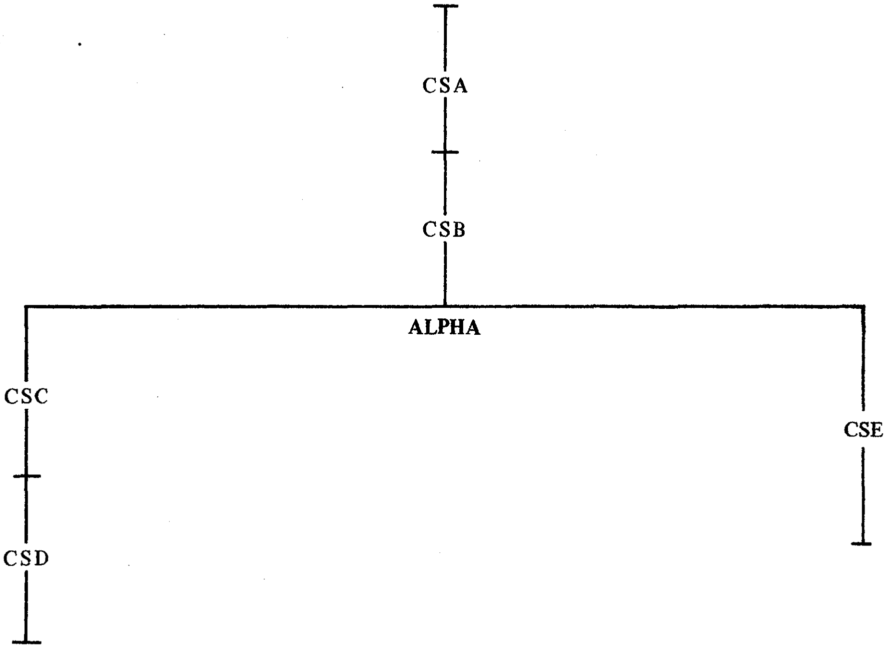


Figure 49.Overlay Structure for INSERT Statement Example

LIBRARY Statement

The LIBRARY statement can be used to specify:

- Additional automatic call libraries, which contain modules used to resolve external references found in the program.
- Restricted no-call function: External references that are not to be resolved by the automatic library call mechanism during the current linkage editor job step.
- Never-call function: External references that are not to be resolved by the automatic library call mechanism during any linkage editor job step.

Combinations of these functions can be written in the same LIBRARY statement.

Format: The format of the LIBRARY statement is:

LIBRARY	{*ddname* (*membername* [,...]) | (*externalreference* [,...]) | *(*externalreference* [,...])},...

ddname
is the name of a DD statement that defines a library.

membername
is the name of or an alias for a member of the specified library. Only those members specified are used to resolve references.

externalreference
is an external reference that may be unresolved after primary input processing. The external reference is not to be resolved by automatic library call.

*

indicates that the external reference is never to be resolved; if the *(asterisk) is missing, the reference is left unresolved only during the current linkage editor run.

Placement: A LIBRARY statement can be placed before, between, or after object modules or other control statements.

Notes:

- If the unresolved external symbol is not a member name in the library specified, the external reference remains unresolved unless defined in another input module.
- If the NCAL option is specified, the LIBRARY statement cannot be used to specify additional call libraries.
- Members called by automatic library call are placed in the root segment of an overlay program, unless they are repositioned with an INSERT statement.
- Specifying an external reference for restricted no-call or never-call by means of the LIBRARY statement prevents the external reference from being resolved by automatic inclusion of the necessary module from an automatic call library; it does not prevent the external reference from being resolved if the module necessary to resolve the reference is specifically included or is included as part of an input module.

Example: The following example shows all three uses of the LIBRARY statement:

```
//            EXEC    PGM=HEWL,PARM='LET,XREF,LIST'
//TESTLIB     DD      DSNAME=TEST,DISP=SHR,...
              .
              .
              .
//SYSLIN      DD      *
  LIBRARY TESTLIB(DATA,TIME),(FICACOMP),*(STATETAX)
/*
```

As a result, members DATE and TIME from the additional library TEST are used to resolve external references. FICACOMP and STATETAX are not resolved; however, because the references remain unresolved, the LET option must be specified on the EXEC statement if the module is to be marked executable. In addition, STATETAX will not be resolved in any subsequent reprocessing by the linkage editor.

NAME Statement

The NAME statement specifies the name of the load module created from the preceding input modules, and serves as a delimiter for input to the load module. As a delimiter, the NAME statement allows multiple load module processing in one linkage editor job step. The NAME statement can also indicate that the load module replaces an identically named module in the output module library.

Format: The format of the NAME statement is:

NAME	*membername* [**(R)**]

membername
is the name to be assigned to the load module that is created from the preceding input modules.

(R)
indicates that this load module replaces an identically named module in the output module library. If the module is not a replacement, the parenthesized value **(R)** should not be specified.

Placement: The NAME statement is placed after the last input module or control statement that is to be used for the output module.

Notes:

- Any ALIAS statement used must precede the NAME statement.
- A NAME statement found in a data set other than the primary input data set is invalid. The statement is ignored.

Example: In the following example, two load modules, RDMOD and WRTMOD, are produced by the linkage editor in one job step:

```
//SYSLMOD     DD      DSNAME=AUXMODS,DISP=MOD,...
//NEWMOD      DD      DSNAME=&&WRTMOD,DISP=OLD
//SYSLIN      DD      DSNAME=&&RDMOD,DISP=OLD
//            DD      *
  NAME RDMOD(R)
  INCLUDE NEWMOD
  NAME WRTMOD
/*
```

As a result, the first module is named RDMOD and replaces an identically named module in the output module library AUXMODS; the second module is named WRTMOD and is added to the library.

ORDER Statement

The ORDER statement indicates the sequence in which control sections or named common areas appear in the output load module. The control sections or named common areas appear in the sequence in which they are specified on the ORDER statement. When multiple ORDER statements are used, their sequence further determines the sequence of the control sections or named common areas in the output load module; those named on the first statement appear first, and so forth.

Format: The format of the ORDER statement is:

ORDER	{*common area name* [**(P)**] | *csectname* [**(P)**]},...

common area name
is the name of the common area to be sequenced.

csectname
is the name of the control section to be sequenced.

(P)
indicates that the starting address of the control section or named common area is to be on a page boundary within the load module. The control sections or common areas are aligned on 4K page boundaries unless the ALIGN2 attribute is specified on the EXEC statement.

Placement: An ORDER statement can be placed before, between, or after object modules or other control statements.

Notes:

- A control section or common area can be named on only one ORDER statement. If the same name is used more than once, except when it is the last operand on one ORDER statement and the first operand on the next, the name is ignored, as is the balance of the control statement on which it appears.
- The control sections and common areas named as operands can appear in either the primary input or the automatic call library, or both.
- If a control section or named common area is changed by a CHANGE or REPLACE control statement and sequencing is desired, specify the new name on the ORDER statement.

Example: In this example, the control sections in the load module LDMOD are arranged by the linkage editor according to the sequence specified on ORDER statements. The page boundary alignments and the control section sequence made as a result of these statements are shown in Figure 50. Assume each control section is 1K in length.

Note: The control section name PART1 is changed by a CHANGE statement to FSTPART. The ORDER statement refers to the control section by its new name.

JCL and Control Statements

Output Load Module

```
                .
                .
                .
//SYSLMOD      DD       DSNAME=PVTLIB,DISP=OLD,...
//SYSLIN       DD       *
  ORDER        ROOTSEG(P),MAINSEG,SEG1,SEG2
  ORDER        SEG3(P),ENTRY1
  CHANGE       PART1(FSTPART)
  ORDER        FSTPART,SESECTA,SESECTB(P)
  INCLUDE      SYSLMOD(LDMOD)
/*
```

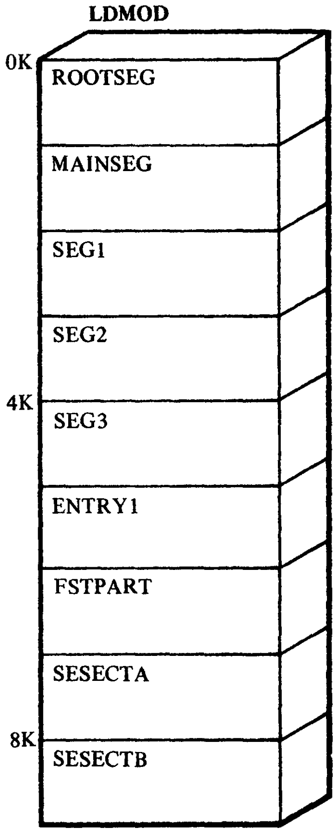


Figure 50.Output Load Module for ORDER Statement Example

OVERLAY Statement

The OVERLAY statement indicates either the beginning of an overlay segment, or the beginning of an overlay region. Since a segment or a region is not named, the programmer identifies it by giving its origin (or load point) a symbolic name. This name is then used on a OVERLAY statement to signify the start of a new segment or region.

Format: The format of the OVERLAY statement is:

OVERLAY	*symbol* **[(REGION)]**

symbol
is the symbolic name assigned to the origin of a segment. This symbol is not related to external symbols in a module.

(REGION)
specifies the origin of a new region.

Placement: The OVERLAY statement must precede the first module of the next segment, the INCLUDE statement specifying the first module of the segment, or the INSERT statement specifying the control sections to be positioned in the segment.

Notes:

- The OVLY option must be specified on the EXEC statement when OVERLAY statements are to be used.
- The sequence of OVERLAY statements should reflect the order of the segments in the overlay structure from top to bottom, left to right, and region by region.
- No OVERLAY statement should precede the root segment.

Example: The following OVERLAY and INSERT statements specify the overlay structure in Figure 51.

```
//            EXEC    PGM=HEWL,PARM='OVLY,XREF,LIST'
               .
               .
               .
//SYSLIN      DD      DSNAME=&&OBJ,...
//            DD      *
  INSERT CSA
  OVERLAY ONE
  INSERT CSB
  OVERLAY TWO
  INSERT CSC
  OVERLAY TWO
  INSERT CSD
  OVERLAY ONE
  INSERT CSE,CSF
  OVERLAY THREE(REGION)
  INSERT CSH
  OVERLAY THREE
  INSERT CSI
/*
```

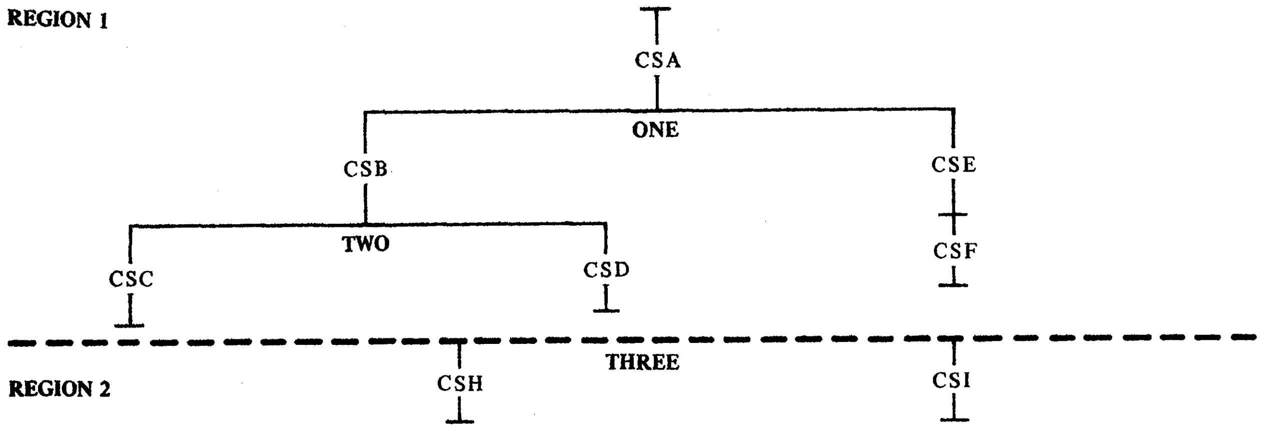


Figure 51.Overlay Structure for OVERLAY Statement Example

PAGE Statement

The PAGE statement aligns a control section or named common area on a 4K page boundary in the load module. If the ALIGN2 attribute is specified on the EXEC statement for the linkage editor job step, use of the PAGE statement aligns the specified control sections or common areas on 2K page boundaries within the load module. However, page boundary alignment in the executing module can occur only when the operating system supervisor includes support for fetch on a page boundary.

Format: The format of the PAGE statement is:

PAGE	{*common area name* | *csectname* },...

common area name
is the name of the common area to be aligned on a page boundary.

csectname
is the name of the control section to be aligned on a page boundary.

Placement: The PAGE statement can be placed before, between, or after object modules or other control statements.

Notes:

- If a control section or named common area is changed by a CHANGE or REPLACE control statement and page alignment is desired, specify the new name in the PAGE statement.
- The control sections and common areas named as operands can appear in either the primary input or the automatic call library, or both.
- Page boundary aligning cannot be used for VS1 overlay programs.

Example: In this example, the control sections in the load module LDMOD are aligned on page boundaries as specified in the following PAGE statement:

```
PAGE ALIGN,BNDRY4K,EIGHTK
```

The job control statements and control statements as well as the output load module are shown in Figure 52. Assume each control section is 3K bytes in length.

JCL AND CONTROL STATEMENTS

```
//LKED         EXEC    PGM=HEWL,PARM='ALIGN2,...'
               .
               .
               .
//SYSLMOD      DD      DSNAME=PVTLIB,DISP=OLD,...
//SYSLIN       DD      *
  PAGE         ALIGN,BNDRY4K,EIGHTK
  INCLUDE      SYSLMOD( LDMOD )
/*
```

OUTPUT LOAD MODULE

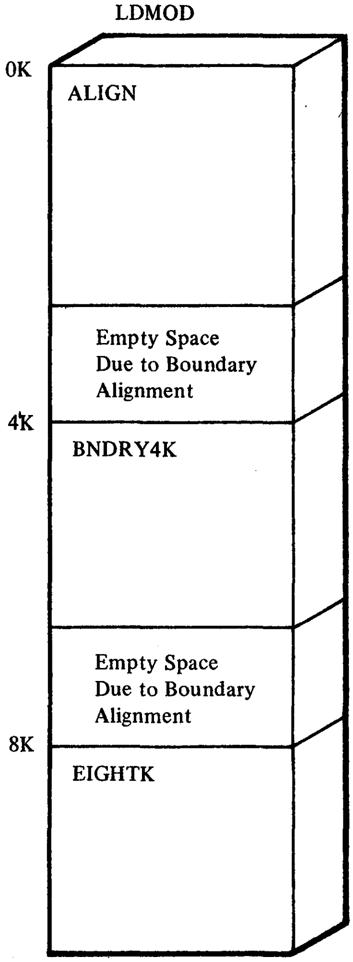

LDMOD
0K
ALIGN
Empty Space Due to Boundary Alignment
4K
BNDRY4K
Empty Space Due to Boundary Alignment
8K
EIGHTK

Figure 52.Output Load Module for PAGE Statement Example

REPLACE Statement

The REPLACE statement specifies one of the following:

- The replacement of one control section with another.
- The deletion of a control section.
- The deletion of an entry name.

A REPLACE statement can specify more than one function.

When a control section is replaced, all references within the input module to the old control section are changed to the new control section. Any external references to the old control section from other modules are unresolved unless changed.

When a control section is deleted, the control section name is also deleted from the external symbol dictionary unless references are made to the control section from within the input module. If there are any such references, the control section name is changed to an external reference. External references from other modules to a deleted control section also remain unresolved.

When deleting an entry name, the entry name is changed to an external reference if there are any references to it within the same input module.

Format: The format of the REPLACE statement is:

REPLACE	{*csectname*-1[(*csectname*-2)] | *entryname*},...

csectname
is the name of a control section. If only csectname-1 is used, the control section is deleted; if csectname-2 is also used, the first control section is replaced with the second.

entryname
is the entry name to be deleted.

Placement: The REPLACE statement must immediately precede either (1) the module containing the control section or entry name to be replaced or deleted, or (2) the INCLUDE statement specifying the module. The scope of the REPLACE statement is across the immediately following module (object module or load module). The END record in the immediately following object module or the end-of-module indication in the load module terminates the action of the REPLACE statement.

Notes:

- Unresolved external references are not deleted from the output module even though a deleted control section contains the only reference to a symbol.
- When some but not all control sections of a separately assembled module are to be replaced, A-type address constants that refer to a deleted symbol will be incorrectly resolved, unless the entry name is at the same displacement from the origin in both the old and the new control sections.
- If the control section specified on the REPLACE statement is inadvertently misspelled, the control section will not be replaced or deleted. Linkage editor output, such as the cross-reference listing and module map, can be used to verify each change.

Example: In the following example, assume that control section INT7 is in member LOANCOMP and that control section INT8, which is to replace

INT7, is in data set && NEWINT. Also assume that control section PRIME in member LOANCOMP is to be deleted.

```
//NEWMOD      DD      DSNAME=&&NEWINT,DISP=(OLD,DELETE)
//OLDMOD      DD      DSNAME=PVTLIB,DISP=OLD,...
//SYSLIN      DD      *
  ENTRY MAINENT
  INCLUDE NEWMOD
  REPLACE INT7(INT8),PRIME
  INCLUDE OLDMOD(LOANCOMP)
/*
```

As a result, INT7 is removed from the input module described by the OLDMOD DD statement, and INT8 replaces INT7. All references to INT7 in the input module now refer to INT8. Any references to INT7 from other modules remain unresolved. Control section PRIME is deleted; the control section name is also deleted from the external symbol dictionary if there are no references to PRIME in LOANCOMP.

SETCODE Statement

The SETCODE statement assigns the specified authorization code to the output load module. The authorization code is placed in the directory entry for the output load module.

The format of the SETCODE statement is as follows:

SETCODE	**AC(*authorizationcode*)**

authorizationcode
is 1 to 8 decimal digits specifying a value from 0 to 255.

Placement: A SETCODE statement can be placed before, between, or after object modules or other control statements. It must precede the NAME statement for the module if one is present.

Notes: The authorization code assigned by the SETCODE statement overrides the authorization code assigned by the AC parameter in the PARM field of the EXEC statement.

If more than one SETCODE statement is encountered in the link edit of a load module, the last valid authorization code assigned is used.

The operand 'AC( )' results in an authorization code of zero.

Example: In the following example, an authorization code of 1 is assigned to the output load module MOD1.

```
//LKED        EXEC    PGM=HEWL
//SYSPRINT    DD      SYSOUT=A
//SYSUT1      DD      UNIT=SYSDA,SPACE=(TRK,(10,5))
//SYSLMOD     DD      DSNAME=SYS1.LINKLIB,DISP=OLD
//SYSLIN      DD      DSNAME=&&LOADSET,DISP=(OLD,PASS)
//                    UNIT=SYSDA
//            DD      *
  SETCODE             AC(1)
  NAME                MOD1(R)
/*
```

SETSSI Statement

The SETSSI statement specifies hexadecimal information to be placed in the system status index of the directory entry for the output module.

Format: The format for the SETSSI statement is:

SETSSI	*xxxxxxxx*

xxxxxxxx

represents eight hexadecimal characters (0 through 9 and A through F) to be placed in the 4-byte system status index of the output module library directory entry.

Placement: The SETSSI statement can be placed before, between, or after object modules or other control statements. It must precede the NAME statement for the module, if one is present.

Note: A SETSSI statement must be provided whenever an IBM-supplied load module is reprocessed by the linkage editor. If the statement is omitted, no system status index information is present.

APPENDIX A: SAMPLE PROGRAMS

This appendix contains sample linkage editor programs. The material presented for each program includes a description of the program, the job control language necessary for the linkage editor job step, linkage editor control statements (if any), and the linkage editor output. The sample programs are:

- Link editing a COBOL and a FORTRAN object module (COBFORT).
- Replacing one control section with another by using the REPLACE statement (RPLACJOB).
- Creating a multiple-region overlay program (REGNOVLY).
- Placing the control statements for the multiple region overlay program in a partitioned data set, and using them (PARTDS).

The output for each program includes a cross-reference table and module map, and a control statement listing and diagnostic messages, if any.

Sample Program COBFORT

Sample program COBFORT link edits a COBOL object module and a FORTRAN object module to form one load module. The source programs were compiled in two steps previous to the linkage editor job step, and the output from each compilation was placed in data set &&OBJMOD.

Job Control Language

The job control language for the linkage editor job step of this sample program is:

```
//LKED         EXEC    PGM=HEWL,PARM='XREF'
//SYSUT1       DD      DSNAME=&&UT1,UNIT=SYSDA,SPACE=(TRK,
//                     (100,10))
//SYSLIB       DD      DSNAME=SYS1.COBLIB,DISP=SHR
//             DD      DSNAME=SYS1.FORTLIB,DISP=SHR
//SYSLMOD      DD      DSNAME=&&LOADMD(GO),UNIT=SYSDA,
//                     DISP=(NEW,PASS),SPACE=(TRK,
//                     (100,10,1))
//SYSPRINT     DD      SYSOUT=A
//SYSLIN       DD      DSNAME=&&OBJMOD,DISP=(OLD,DELETE)
/*
```

Statement	Explanation
EXEC	Causes the execution of the linkage editor. The PARM field option requests a cross-reference table and a module map to be produced on the diagnostic output data set.
SYSUT1	Defines a temporary direct-access data set to be used as the intermediate data set.
SYSLIB	Defines the automatic call library; the call libraries for COBOL and FORTRAN are concatenated; both are used to resolve external references.
SYSLMOD	Defines a temporary data set to be used as the output module library; the load module is assigned a member name of GO, and is passed to a subsequent step for execution.
SYSPRINT	Defines the diagnostic output data set, which is assigned to output class A.
SYSLIN	Defines the primary input data set, &&OBJMOD, which contains both input object modules; this data set was passed from a previous job step and is to be deleted at the end of this job step.

Linkage Editor Output

Figure 53 shows the linkage editor output for COBFORT. The *listing header* indicates the options specified (XREF,LIST), and the SIZE option values used in decimal (196608 for value1 and 65536 for value2). Because XREF is specified, the heading CROSS REFERENCE TABLE precedes the rest of the output.

Part 1 of Figure 53 shows the *module map* for COBFORT. IPCT30 and TX652F are the names of the input control sections. The rest of the control sections are either from the COBOL automatic call library or from the FORTRAN automatic call library. (They can be distinguished by the initial three letters; ILB indicates a COBOL control section, IHC a FORTRAN control section.) The origin and length (in hexadecimal) of each control section follow the name.

To the right of each control section is a list of the entry names defined in each control section. The location (in hexadecimal) of each entry name is also given. For example, in control section IHCCOMH2 (the asterisk is not a part of the name; it indicates that the control section is from the automatic call library), entry name SEQDASD is defined at location 154A.

Part 2 of Figure 53 shows the *cross reference table* for COBFORT. The table contains the location of any address constant that refers to a symbol defined in another control section. The symbol that the address constant refers to is also listed, along with the control section in which the symbol is defined. For example, at location 1F0 in control section IPCT30 (determined by using the module map, 1F0 falls between origin 00 and origin 360), an address constant refers to symbol IHDFDISP, defined in control section IHDFDISP.

The *entry address* is 00 and the *total length* of the load module is 4AE8. Note that the length of the module is rounded up to a doubleword boundary.

The *disposition message* at the end of the output in Figure 53 indicates that the load module GO has been added to the output module library. The library did not contain any other module with that name. The four asterisks identify the message.

```
F64-LEVEL LINKAGE EDITOR OPTIONS SPECIFIED XREF
         DEFAULT OPTIONS(S) USED - SIZE=(196608,65536)

                                                 CROSS REFERENCE TABLE

CONTROL SECTION                           ENTRY

  NAME     ORIGIN   LENGTH                  NAME   LOCATION     NAME   LOCATION     NAME   LOCATION     NAME   LOCATION

IPCT30        00       360
TX652F       360       1E0
IHCFCOMH*    540       CD9
                                          IBCOM#       540     FDIOCS#      5FC     INTSWTCH     11FE
IHCCOMH2*   1220       434
                                          SEQDASD     154A
IHDFDISP*   1658       626
IHCFCVTH*   1C80      119D
                                          ADCON#      1C80     FCVAOUTP    1D2A     FCVLOUTP     1DBA     FCVZOUTP    1F0A
                                          FCVIOUTP    22B8     FCVEOUTP    27BA     FCVCOUTP     29D4     INT6SWCH    2CBB
IHCFINTH*   2E20       39E
                                          ARITH#      2E20     ADJSWTCH    30D8
IHCFIOSH*   31C0      100E
                                          FIOCS#      31C0
IHCUOPT *   41D0         8
IHCTRCH *   41D8       2D4
                                          IHCERRM     41D8
IHCUATBL*   44B0       638
```

Figure 53 (Part 1 of 2). Linkage Editor Output for Sample Program COBFORT

```
LOCATION   REFERS TO SYMBOL    IN CONTROL SECTION          LOCATION   REFERS TO SYMBOL    IN CONTROL SECTION
   1F0          IHDFDISP          IHDFDISP                    1F4          TX652F            TX652F
   410          IBCOM#            IHCFCOMH                    5FC          SEQDASD           IHCCOMH2
  1108          ADCON#            IHCFCVTH                   1100          FIOCS#            IHCFIOSH
  110C          ARITH#            IHCFINTH                   112C          ADJSWTCH          IHCFINTH
  1128          IHCUOPT           IHCUOPT                    1110          FCVEOUTP          IHCFCVTH
  1114          FCVLOUTP          IHCFCVTH                   1118          FCVIOUTP          IHCFCVTH
  111C          FCVCOUTP          IHCFCVTH                   1120          FCVAOUTP          IHCFCVTH
  1124          FCVZOUTP          IHCFCVTH                   10E0          IHCCOMH2          IHCCOMH2
  10E4          IHCERRM           IHCTRCH                    14A9          IHCFCOMH          IHCFCOMH
  14AC          IHCFCOMH          IHCFCOMH                   1268          IHCERRM           IHCTRCH
  1264          IBCOM#            IHCFCOMH                   2C7C          IBCOM#            IHCFCOMH
  2C78          IHCERRM           IHCTRCH                    311C          IBCOM#            IHCFCOMH
  3120          INTSWTCH          IHCFCOMH                   30D4          INT6SWCH          IHCFCVTH
  30D0          IHCUOPT           IHCUOPT                    3128          ADCON#            IHCFCVTH
  3124          FIOCS#            IHCFIOSH                   32F8          IHCERRM           IHCTRCH
  3FF8          IHCUATBL          IHCUATBL                   4004          IBCOM#            IHCFCOMH
  43D0          IBCOM#            IHCFCOMH                   43D4          ADCON#            IHCFCVTH
  43D8          FIOCS#            IHCFIOSH
ENTRY ADDRESS        00
TOTAL LENGTH       4AE8

****GO           DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET
AUTHORIZATION CODE IS             0.
```

Figure 53 (Part 2 of 2). Linkage Editor Output for Sample Program COBFORT

Sample Program RPLACJOB

Sample program RPLACJOB shows the use of the REPLACE statement to replace one control section with another. The source program for the new control section (NEWMOD) is processed in a previous job step and passed to the linkage editor job step. The control section (SUBONE) to be replaced is in an existing load module. Figure 54 shows the linkage editor output for the job step that created this load module. Note that the entry address is F0, which is the location of the entry point MAINMOD (specified on the ENTRY control statement).

```
  F64-LEVEL LINKAGE EDITOR OPTIONS SPECIFIED XREF,LIST
          DEFAULT OPTION(S) USED -  SIZE=(196608,65536)
 IEW0000             ENTRY  MAINMOD

                                                                    CROSS REFERENCE TABLE

    CONTROL SECTION                        ENTRY

      NAME    ORIGIN  LENGTH                 NAME    LOCATION     NAME    LOCATION      NAME    LOCATION      NAME    LOCATION
    SUBONE        00      EF
                                           SUB1         00
    MAINMOD       F0     146

   LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION            LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION
        11C         SUBONE         SUBONE
  ENTRY ADDRESS       F0

  TOTAL LENGTH       238
 ****GO        DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET
 AUTHORIZATION CODE IS        0.
```

Figure 54. Linkage Editor Output for Job Step that Created SUBONE

Job Control Language

The job control language for the replacement job step of this sample program is:

```
//LKED        EXEC    PGM=HEWL,PARM='XREF,LIST'
//SYSUT1      DD      UNIT=SYSDA,SPACE=(TRK,(100,10))
//INPUTX      DD      DSNAME=LOADLIB,DISP=OLD,UNIT=SYSDA,
//                    VOL=SER=SCRTCH
//SYSLMOD     DD      DSNAME=LOADLIB(GO),DISP=OLD,UNIT=SYSDA,
//                    VOL=SER=SCRTCH
//SYSPRINT    DD      SYSOUT=A
//SYSLIN      DD      DSNAME=&&OBJMOD,DISP=(OLD,DELETE),
//                    UNIT=SYSDA
//            DD      *
```

Linkage Editor Control Statements

```
/*
```

Statement **Explanation**

EXEC Causes the execution of the linkage editor. The PARM field options request a cross-reference table and a module map (XREF), and a control statement listing (LIST) to be produced on the diagnostic output data set.

SYSUT1 Defines a temporary direct-access data set to be used as the intermediate data set.

INPUTX Defines a permanent data set, used later as additional linkage editor input.

SYSLMOD Defines a permanent data set to be used as the output module library. Note that it is the same data set that was described on the INPUTX DD statement. The output load module is added to the data set, under the member name GO.

SYSPRINT Defines the diagnostic output data set, which is assigned to output class A.

SYSLIN Defines the primary input data set, &&OBJMOD, which contains the object module for the replacement control section. This data set is temporary and was passed from a previous job step; it is to be deleted at the end of this job. This statement also concatenates the input stream to the primary input data set. The input stream contains linkage editor control statements that may be followed by a /* statement.

Linkage Editor Control Statements

The input stream contains the linkage editor control statements that are necessary for the replacement of SUBONE with NEWMOD. The control statements are:

```
  ENTRY     MAINMOD
  REPLACE   SUBONE(NEWMOD)
  INCLUDE   INPUTX(GO)
```

Statement **Explanation**

ENTRY Specifies that the entry point is to be MAINMOD.

REPLACE Specifies that control section SUBONE in the module that follows the REPLACE statement is to be replaced by control section NEWMOD.

INCLUDE Specifies additional input: member GO of the data set described on the INPUTX DD statement. This library member contains the control section to be replaced. Since this member name is identical to that specified on the SYSLMOD DD statement, the output load module replaces the existing library member.

Linkage Editor Output

Figure 55 shows the linkage editor output for sample program RPLACJOB. The *listing header* indicates the options specified (XREF and LIST), and the SIZE option values used (196608 for value1 and 65536 for value2).

Because the LIST option is specified, a *control statement listing* is produced. Each control statement is preceded by a special message number, IEW0000. Because XREF is specified, the heading CROSS REFERENCE TABLE precedes the rest of the output.

The *module map* shows that control section NEWMOD is now part of the load module, and that control section SUBONE has been deleted. The new *entry address* is F8, because NEWMOD is longer than SUBONE. The *total length* of the load module is 240 bytes.

The *cross reference table* indicates that at location 124 in MAINMOD, an address constant refers to symbol NEWMOD, defined in control section NEWMOD. Note that before the replacement occurred, the address constant in MAINMOD referred to SUBONE, defined in control section SUBONE (Figure 54). When the REPLACE statement is used to replace a control section, references to the old control section from within the same input module are also changed.

The *disposition message* indicates that the output load module (GO) has been added to the output module library.

Sample Program REGNOVLY

Sample program REGNOVLY creates a multiple-region overlay structure. The structure produced is shown in Figure 56. In this program, some of the references between control sections are:

CSA to CSE

CSB to CSE

CSB to CSD

CSD to CSC

The reference from CSB to CSE is a valid exclusive call because there is a reference to CSE in the segment common to both CSB and CSE; the reference from CSD to CSC is invalid because there is no reference to CSC in the common segment.

```
 F64-LEVEL LINKAGE EDITOR OPTIONS SPECIFIED XREF,LIST
          DEFAULT OPTION(S) USED -  SIZE=(196608,65536)
IEW0000             ENTRY MAINMOD
IEW0000             REPLACE SUBONE(NEWMOD)
IEW0000             INCLUDE INPUTX(GO)

                                                CROSS REFERENCE TABLE

  CONTROL SECTION                       ENTRY

    NAME    ORIGIN  LENGTH                NAME   LOCATION     NAME  LOCATION     NAME  LOCATION     NAME   LOCATION
  NEWMOD        00      F1
  MAINMOD       F8     146

  LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION             LOCATION   REFERS TO SYMBOL   IN CONTROL SECTION
      124            NEWMOD          NEWMOD
 ENTRY ADDRESS       F8

 TOTAL LENGTH       240
****GO        NOW REPLACED IN DATA SET
AUTHORIZATION CODE IS        0.
```

Figure 55. Linkage Editor Output for Sample Program RPLACJOB

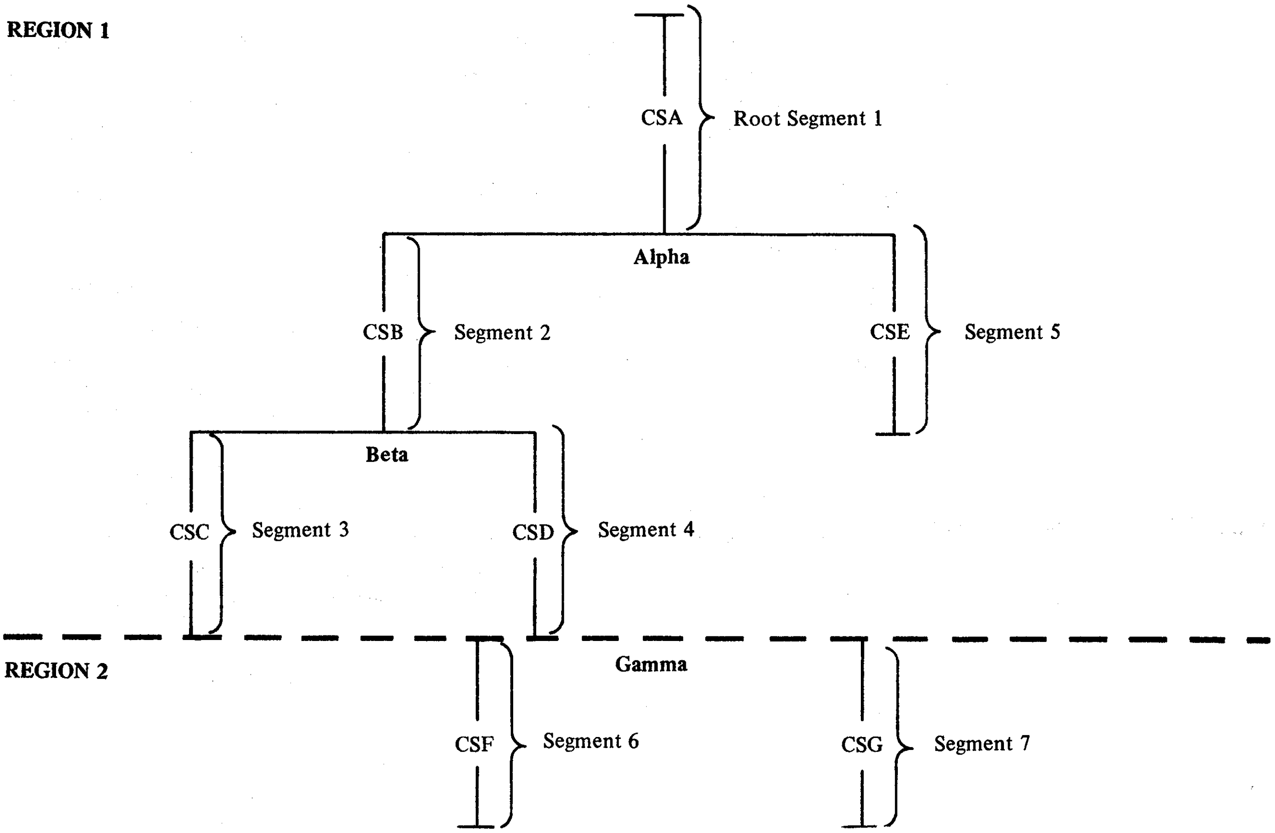


Figure 56. Overlay Tree for Multiple-Region Sample Program REGNOVLY

The source programs for all the control sections were compiled in previous job steps. All of the object modules were placed in the same data set, which was passed to the linkage editor job step.

Job Control Language

The job control language for the linkage editor job step of this sample program is:

```
//LKED        EXEC    PGM=HEWL,PARM='XREF,LIST,OVLY,LET'
//SYSUT1      DD      DSNAME=&&UT1,UNIT=SYSDA,SPACE=(TRK,
//                    (100,10))
//SYSLIB      DD      DSNAME=SYS1.COBLIB,DISP=SHR
//SYSLMOD     DD      DSNAME=&&OVLYJB(GO),UNIT=SYSDA,
//                    DISP=(NEW,PASS),SPACE=(TRK,(100,10,1))
//SYSPRINT    DD      SYSOUT=A
//SYSLIN      DD      DSNAME=&&OBJMOD,DISP=(OLD,DELETE)
//            DD      *
Linkage Editor Control statements
/*
```

Statement	Explanation
EXEC	Causes the execution of the linkage editor. The PARM field options request a cross reference table and a module map (XREF), and a control statement listing (LIST) to be produced on the diagnostic output data set. The module is to be assigned the overlay attribute (OVLY), and marked executable in spite of severity 2 errors (LET). The LET option is specified to permit testing of the output module, even though an invalid exclusive call is present. The XCAL option allows only valid exclusive calls.

Statement	Explanation
SYSUT1	Defines a temporary direct-access data set to be used as the intermediate data set.
SYSLIB	Defines the automatic call library (SYS1.COBLIB) to be used to resolve external references. All control sections from this library are placed in the root segment; they remain there unless they are repositioned.
SYSLMOD	Defines a temporary data set to be used as the output module library; the load module is assigned the member name GO and is passed to a subsequent step for execution.
SYSPRINT	Defines the diagnostic output data set, which is assigned to output class A.
SYSLIN	Defines the primary input data set, &&OBJMOD, which contains the object modules for the overlay structure. This data set is temporary and was passed from a previous job step; it is to be deleted at the end of this job. This statement also concatenates the input stream to the primary input data set. The input stream contains linkage editor control statements, which must be delimited by a /* statement.

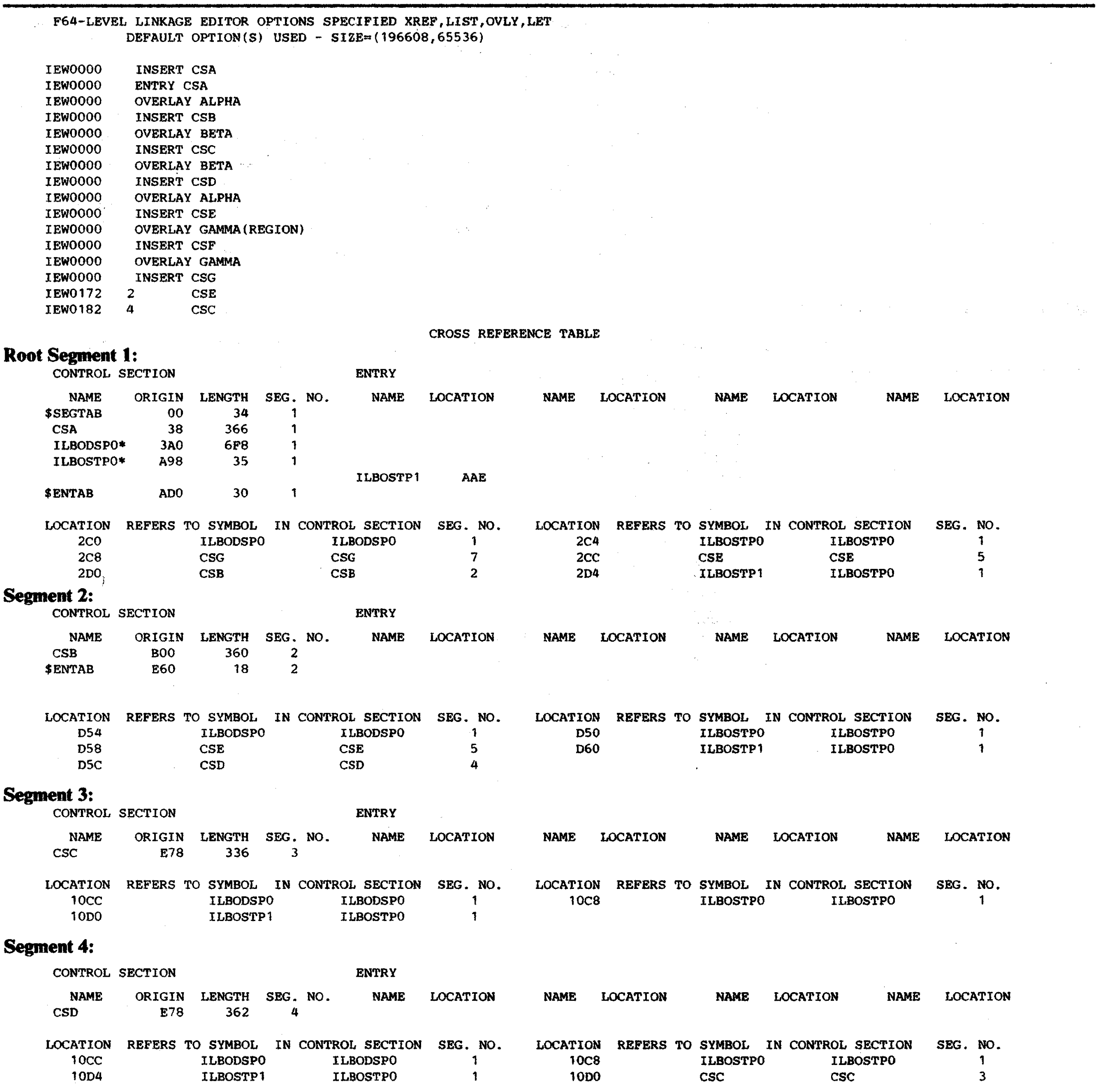
Linkage Editor Control Statements

The input stream contains the linkage editor control statements that structure the overlay program. The control statements are:

```
INSERT CSA
ENTRY CSA
OVERLAY ALPHA
INSERT CSB
OVERLAY BETA
INSERT CSC
OVERLAY BETA
INSERT CSD
OVERLAY ALPHA
INSERT CSE
OVERLAY GAMMA( REGION )
INSERT CSF
OVERLAY GAMMA
INSERT CSG
```

Linkage Editor Output

Figure 57 shows the linkage editor output for sample program REGNOVLY. The *list header* indicates the options specified (XREF, LIST, OVLY, and LET), and the SIZE option values used (196608 for value1 and 65536 for value2).



```
  F64-LEVEL LINKAGE EDITOR OPTIONS SPECIFIED XREF,LIST,OVLY,LET
            DEFAULT OPTION(S) USED - SIZE=(196608,65536)

 IEW0000     INSERT CSA
 IEW0000     ENTRY CSA
 IEW0000     OVERLAY ALPHA
 IEW0000     INSERT CSB
 IEW0000     OVERLAY BETA
 IEW0000     INSERT CSC
 IEW0000     OVERLAY BETA
 IEW0000     INSERT CSD
 IEW0000     OVERLAY ALPHA
 IEW0000     INSERT CSE
 IEW0000     OVERLAY GAMMA(REGION)
 IEW0000     INSERT CSF
 IEW0000     OVERLAY GAMMA
 IEW0000     INSERT CSG
 IEW0172   2       CSE
 IEW0182   4       CSC
                                          CROSS REFERENCE TABLE
```

Root Segment 1:

```
  CONTROL SECTION                      ENTRY

    NAME    ORIGIN  LENGTH  SEG. NO.     NAME    LOCATION      NAME    LOCATION      NAME    LOCATION      NAME    LOCATION
  $SEGTAB       00      34     1
   CSA          38     366     1
   ILBODSPO*   3A0     6F8     1
   ILBOSTPO*   A98      35     1
                                       ILBOSTP1      AAE
  $ENTAB       AD0      30     1

 LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.    LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.
     2C0          ILBODSPO         ILBODSPO          1           2C4           ILBOSTPO        ILBOSTPO          1
     2C8          CSG              CSG               7           2CC           CSE             CSE               5
     2D0          CSB              CSB               2           2D4           ILBOSTP1        ILBOSTPO          1
```

Segment 2:

```
  CONTROL SECTION                      ENTRY

    NAME    ORIGIN  LENGTH  SEG. NO.     NAME    LOCATION      NAME    LOCATION      NAME    LOCATION      NAME    LOCATION
  CSB          B00     360     2
  $ENTAB       E60      18     2


 LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.    LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.
     D54          ILBODSPO         ILBODSPO          1           D50           ILBOSTPO        ILBOSTPO          1
     D58          CSE              CSE               5           D60           ILBOSTP1        ILBOSTPO          1
     D5C          CSD              CSD               4
```

Segment 3:

```
  CONTROL SECTION                      ENTRY

    NAME    ORIGIN  LENGTH  SEG. NO.     NAME    LOCATION      NAME    LOCATION      NAME    LOCATION      NAME    LOCATION
  CSC          E78     336     3

 LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.    LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.
    10CC           ILBODSPO        ILBODSPO          1          10C8           ILBOSTPO        ILBOSTPO          1
    10D0           ILBOSTP1        ILBOSTPO          1
```

Segment 4:

```
  CONTROL SECTION                      ENTRY

    NAME    ORIGIN  LENGTH  SEG. NO.     NAME    LOCATION      NAME    LOCATION      NAME    LOCATION      NAME    LOCATION
  CSD          E78     362     4

 LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.    LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.
    10CC          ILBODSPO         ILBODSPO          1          10C8           ILBOSTPO        ILBOSTPO          1
    10D4          ILBOSTP1         ILBOSTPO          1          10D0           CSC             CSC               3
```

Figure 57 (Part 1 of 2). Linkage Editor Output for Sample Program REGNOVLY

```
                                                CROSS REFERENCE TABLE
```

Segment 5:

```
   CONTROL SECTION                        ENTRY

     NAME    ORIGIN  LENGTH  SEG. NO.      NAME    LOCATION        NAME   LOCATION        NAME    LOCATION        NAME    LOCATION
   CSE          B00     336     5

  LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.     LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.
     D54            ILBODSPO         ILBODSPO          1           D50             ILBOSTPO         ILBOSTPO           1
     D58            ILBOSTP1         ILBOSTPO          1
```

Segment 6:

```
   CONTROL SECTION                        ENTRY

     NAME    ORIGIN  LENGTH  SEG. NO.      NAME    LOCATION        NAME   LOCATION        NAME    LOCATION        NAME    LOCATION
   CSF         11E0     2FA     6

  LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.     LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.
     1430           ILBOSTPO         ILBOSTPO          1          1434             ILBOSTP1         ILBOSTPO           1
```

Segment 7:

```
   CONTROL SECTION                        ENTRY

     NAME    ORIGIN  LENGTH  SEG. NO.      NAME    LOCATION        NAME   LOCATION        NAME    LOCATION        NAME    LOCATION
   CSG         11E0     336     7

  LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.     LOCATION  REFERS TO SYMBOL  IN CONTROL SECTION  SEG. NO.
     1434           ILBODSPO         ILBODSPO          1          1430             ILBOSTPO         ILBOSTPO          1
     1438           ILBOSTP1         ILBOSTPO          1
  ENTRY ADDRESS     38

  TOTAL LENGTH     1518
 ****GO       DOES NOT EXIST BUT HAS BEEN ADDED TO DATA SET
 AUTHORIZATION CODE IS        0.


                                             DIAGNOSTIC MESSAGE DIRECTORY
   IEW0172 ERROR - EXCLUSIVE CALL FROM SEGMENT NUMBER PRINTED TO SYMBOL PRINTED.
   IEW0182 ERROR - INVALID EXCLUSIVE CALL FROM SEGMENT NUMBER PRINTED TO SYMBOL PRINTED.
```

Figure 57 (Part 2 of 2). Linkage Editor Output for Sample Program REGNOVLY

Because the LIST option was specified, the *control statement listing* is produced. Each control statement is preceded by a special message number, IEW0000.

The control statement listing is followed by two diagnostic message numbers (IEW0172 and IEW0182). The explanation of the messages and the information following each message is given at the end of the output in the diagnostic message directory.

The output for each segment contains a module map and a cross-reference table. The segments are listed as they appear in the overlay structure, top to bottom, left to right, and region by region. (Note that this is also the sequence in which the OVERLAY and INSERT statements must be given.)

Within each segment, a *module map* lists the control sections in ascending sequence according to their assigned origin. The origin, length, and segment number is listed for each control section, along with any entry names and the location where each entry name is defined. For example, the root segment has five control sections: $SEGTAB, which is always the first control section in the root segment; CSA, which is from the object module input; ILBODSPO and ILBOSTPO, which are from the automatic call library and were not repositioned; and $ENTAB, which, when present, is always the last control section in any segment (as also in segment 2). One entry name is defined, ILBOSTP1 at location D58 in control section ILBOSTPO.

The *cross reference table* for each segment contains all of the address constants that refer to symbols defined in other control sections. The location of the address constant is followed by the symbol referred to, the control section in which the symbol is defined, and the segment in which the control section is located. For example, in the root segment, an address constant at location 11E0 refers to symbol CSG, which is defined in control section CSG in segment 7. Although the region is not given, the overlay tree in Figure 56 shows that segment 7 is in region 2.

At the end of the output for all the segments is the entry address and total length. The entry address is 38, which is the origin of CSA, the specified entry point. The total length given refers to main storage used, not device storage. The length given, therefore, is that of the longest path. The longest path is that formed by the root segment and segments 2, 4, and 7; the length given is 1518.

However, if the given lengths of the control sections in each segment are added, the result is 14D3. The discrepancy exists because the given lengths do not include the padding bytes necessary to make control sections begin on a doubleword address (multiple of 8). For example, in the root segment, the length of $SEGTAB is 34; however, the origin of CSA which follows $SEGTAB is 38 (decimal 56). Four additional bytes are needed so that the origin of CSA is a multiple of 8.

The *disposition message* indicates that the load module GO has been added to the output module library. The library did not contain any other module by that name. The four asterisks identify the message.

The last item in the output for this sample program is the *diagnostic message directory*. The directory contains the text for the message numbers listed after the control statement listing. The directory must be correlated to the information following the number to interpret the message.

For example, message IEW0172 is an error message which indicates that an exclusive call was made *from* the segment number printed (2) following the message number *to* the symbol printed (CSE). The output for segment 2 indicates that this call is at location D58 in control section CSB, and the symbol is defined in control section CSE in segment 5. This is the valid exclusive call from CSB to CSE described earlier. (If XCAL were specified, a *warning* message would be issued instead of an error message.)

If an invalid exclusive call is detected, message IEW0182 appears as shown. This is also an error message; it indicates that an invalid exclusive call was made from segment 4 to symbol CSC. This call is at location E78 in control section CSD, and the symbol is defined in control section CSC in segment 3. This is the invalid exclusive call from CSD to CSC, also described earlier.

Sample Program PARTDS

Sample program PARTDS illustrates that linkage editor control statements can be placed in a separate data set and then used as input. For convenience, the control statements are those for sample progarm REGNOVLY, described previously. These control statements are placed in a partitioned data set. When the member that contains the control statements is referenced, the linkage editor uses the control statements to produce the overlay structure shown earlier in Figure 56.

Figure 58 shows the input statements for the IEBUPDTE utility program used to place the control statements in a partitioned data set.

The source programs for all the control sections were compiled in previous job steps. All the object modules were placed in the same data set, which was passed to the linkage editor job step. The input modules are those used for sample program REGNOVLY.

Job Control Language

The job control language for the overlay program job step of this sample program is:

```
//LKED        EXEC   PGM=HEWL,PARM='XREF,LIST,OVLY,LET'
//SYSUT1      DD     DSNAME=&&UT1,UNIT=SYSDA,SPACE=(TRK,
//                   (100,10))
//OVLYCDS     DD     DSNAME=OVLYLIB,UNIT=SYSDA,
//                   VOL=SER=SCRTCH,DISP=OLD
//SYSLIB      DD     DSNAME=SYS1.COBLIB,DISP=SHR
//SYSLMOD     DD     DSNAME=&&OVLYJB(GO),UNIT=SYSDA,
//                   DISP=(NEW,PASS),SPACE=(TRK,(100,10,1))
//SYSPRINT    DD     SYSOUT=A
//SYSLIN      DD     DSNAME=&&OBJMOD,DISP=(OLD,DELETE)
//            DD     *
   ( Linkage Editor Control Statements )
/*
```

```
//PARTDS     JOB     (accounting information)
//CTLG       EXEC    PGM=IEBUPDTE,PARM=(NEW)
//SYSUT2     DD      DSNAME=OVLYLIB,UNIT=2314,VOL=SER=DA028,DISP=(NEW,KEEP),
//                   SPACE=(TRK,(10,5,2)),DCB=(LRECL=80,BLKSIZE=80,RECFM=F)
//SYSPRINT   DD      SYSOUT=A
//SYSIN      DD      *
./         ADD       NAME=OVLY,LEVEL=00,SOURCE=00,LIST=ALL
./         NUMBER        NEW1=10,INCR=5
  INSERT CSA
  ENTRY CSA
  OVERLAY ALPHA
  INSERT CSB
  OVERLAY BETA
  INSERT CSC
  OVERLAY BETA
  INSERT CSD
  OVERLAY ALPHA
  INSERT CSE
  OVERLAY GAMMA(REGION)
  INSERT CSF
  OVERLAY GAMMA
  INSERT CSG
./       ENDUP
/*
```

Figure 58. Input Statements for IEBUPDTE Utility Program

Statement	Explanation
EXEC	Causes the execution of the linkage editor. The PARM field options request a cross-reference table and a module map (XREF), and a control statement listing (LIST) to be produced on the diagnostic output data set. The output load module is to be assigned the overlay attribute (OVLY), and is to be marked executable despite severity 2 errors (LET).
SYSUT1	Defines a temporary direct-access data set to be used as the intermediate data set.
OVLYCDS	Defines a permanent data set to be used later as additional input; this is the partitioned data set which was created by IEBUPDTE and contains the control statements for structuring the overlay program.
SYSLIB	Defines the automatic call library (SYS1.COBLIB) to be used to resolve external references. All control sections from this library are placed in the root segment; they remain there unless they are repositioned.
SYSLMOD	Defines a temporary data set to be used as the output module library; the load module is to be assigned the member name GO, and is passed to a subsequent step for execution.
SYSPRINT	Defines the diagnostic output data set, which is assigned to output class A.
SYSLIN	Defines the primary input data set, &&OBJMOD, which contains the object modules for the overlay structure. This data set is temporary and was passed from a previous job step; it is to be deleted at the end of this job. This statement also concatenates the input stream to the primary input data set. The input stream contains linkage editor control statements that must be delimited by a /* statement.

Linkage Editor Control Statements

The input stream contains an INCLUDE statement, as follows:

```
INCLUDE   OVLYCDS(OVLY)
```

This statement causes the control statements to be read from the partitioned data set described on the OVLYCDS DD statement. The member name of the statements is OVLY, the same name used in the ADD statement for the utility program.

Linkage Editor Output

The output for this sample program is identical to the output from the REGNOVLY sample program, with one execption. The list of control statements begins with the statement

```
IEW0000     INCLUD   OVLYCDS(OVLY)
```

This statement is followed by a list of the control statements read from the additional input data set specified in this INCLUDE statement. The rest of the output is identical to that shown in Figure 57.

APPENDIX B: INVOKING THE LINKAGE EDITOR

The linkage editor can be invoked by a problem program at execution time through the use of one of the following macro instructions.

[*symbol*]	[LINK]	**EP=** *linkeditname* **PARAM=(***optionlist* **[,***ddname list* **]),** **VL=1**

[*symbol*]	[ATTACH]	**EP=** *linkeditname* **PARAM=(***optionlist* **[,***ddname list* **]),** **VL=1**

[*symbol*]	[LOAD]	**EP=** *linkeditname*

[*symbol*]	[XCTL]	**EP=** *linkeditname*

EP= *linkeditname*
specifies the symbolic name of the linkage editor. The entry point at which execution is to begin is determined by the control program (from the library directory entry).

PARAM=(*optionlist* **[,** *ddname list* **])**
specifies, as a sublist, address parameters to be passed from the problem program to the linkage editor. The first fullword in the address parameter list contains the address of the option and attribute list for the load module. The second fullword contains the address of the ddname list. If standard ddnames are to be used, this list may be omitted.

optionlist
specifies the address of a variable length list containing the options and attributes. This address must be written even though no list is provided.

The option list must begin on a halfword boundary. The two high-order bytes contain a count of the number of bytes in the remainder of the list. If no options or attributes are specified, the count must be zero. The option list is free form with each field separated by a comma. No blanks or zeros should appear in the list.

ddnamelist
specifies the address of a variable length list containing alternative ddnames for the data sets used during linkage editor processing. If standard ddnames are used, this operand may be omitted.

The ddname list must begin on a halfword boundary. The two high-order bytes contain a count of the number of bytes in the remainder of the list. Each name of less than 8 bytes must be left justified and padded with blanks. If an alternate ddname is omitted from the list, the standard name will be assumed. If the name is omitted within the list, the 8-byte entry must contain binary zeros. Names can be omitted from the end by merely shortening the list.

The sequence of the 8-byte entries in the ddname list is as follows:

Entry	Alternate Name For:
1	SYSLIN
2	member name (the name under which the output load module is stored in the SYSLMOD data set; this entry is used if the name is not specified on the SYSLMOD DD statement or if there is no NAME control statement)
3	SYSLMOD
4	SYSLIB
5	not applicable
6	SYSPRINT
7	not applicable
8	SYSUT1
9-11	not applicable
12	SYSTERM

VL

specifies that the sign bit is to be set to 1 in the last fullword of the address parameter list.

When the linkage editor completes processing, a condition code is returned in register 15 (see "Linkage Editor Return Code").

APPENDIX C: STORAGE REQUIREMENTS AND CAPACITIES

This appendix describes the record-processing capacities of the linkage editor, the types of devices that can be used for the intermediate data set (SYSUT1), and the amount of virtual storage that the linkage editor requires.

Capacities

The minimum storage requirement and processing capacities for the linkage editor program are described in Figure 59. To increase the capacity for processing external symbol dictionary records, intermediate text records, relocation dictionary records, and identification records, increase value1 and/or value2 of the SIZE option. Output text record length can be increased by increasing the SIZE option values, but in no case can the record length ever exceed the track length for the device. The number of overlay segments and regions that can be processed is not affected by increasing the storage available.

Function	Capacity
Virtual storage allocated (in bytes)	64K
Maximum number of entries in composite external symbol dictionary (CESD)	558
Maximum number of intermediate test records	372
Maximum number of relocation dictionary (RLD) records	192
Maximum number of segments per program	255
Maximum number of overlay regions per program	4
Maximum blocking factor for input object modules (number of 80-column card images per physical record)	10[1]
Maximum blocking factor for SYSPRINT output (number of 121-character logical records per physical record)	10[1]
Output text record length (in bytes):	
On IBM 2314, 2319 Storage Facility	3072[2]
On IBM 2305 Fixed Head Storage Facility	3072[2]
On IBM 3330 Disk Storage Facility	3072[2]
On IBM 3340 Disk Storage Facility	3072[2]
On IBM 3344 Direct Access Storage Device	3072[2]
On IBM 3350 Direct Access Storage	3072[2]

[1] From 74K to 9999K for value1 of the SIZE option, the blocking factor for input object modules and SYSPRINT output is 40.

[2] The maximum output text record length is achieved when value2 of the SIZE parameter is at least twice the record length size. For example, on a 3330, 12288 byte records are written when value2 is at least 24576.

Figure 59. Linkage Editor Capacities for Minimal SIZE Values (64K,6K)

For the *composite external symbol dictionary,* the number of entries permitted can be computed by subtracting, from the maximum number given in Figure 59, one entry for each of the following:

- A data definition name (ddname) specified in LIBRARY statements.
- A data definition name (ddname) specified in INCLUDE statements.
- An ALIAS statement.
- A symbol in REPLACE or CHANGE statements that are in the largest group of such statements preceding a single object module in the input to the linkage editor.
- The segment table (SEGTAB) in an overlay program.
- An entry table (ENTAB) in an overlay program.

To compute the number of *intermediate text records* that will be produced during processing of either program, add one record for each group of x bytes within each control section, where x is the record size for the intermediate data set. The minimum value for x is 1024; a maximum is chosen depending on the amount of storage available to the linkage editor and the devices allocated for the intermediate and output data sets.

The number of text records that can be handled by a linkage editor program is less than the maximums given in Figure 59 if the text of one or more control sections is not in sequence by address in the input to the linkage editor.

The total length of the data fields of the *CSECT Identification* records associated with a load module cannot exceed 32K (32,768) bytes. To determine the number of bytes of identification data contained in a particular load module, use the following formula:

$$\text{SIZE} = 269 + 16A + 31B + 2C + I(n + 6)$$

where:

A = the number of compilations or assemblies by a processor supporting CSECT Identification that produced the object code for the module.

B = the number of pre-processor compiler compilations by a processor supporting CSECT Identification that produced the object code for the module.

C = the number of control sections in the module with END statements that contain identification data.

I = the number of control sections in the module that contain user-supplied data supplied during link editing by the optional IDENTIFY control statement.

n = the average number of characters in the data specified by IDENTIFY control statements.

Notes:

- The size computed by the formula includes space for recording up to 19 HMASPZAP modifications. When 75% of this space has been used, a new 251-byte record is created the next time the module is reprocessed by the linkage editor.
- To determine the approximate number of records involved, divide the computed size of the identification data by 256.

Example: A module contains 100 control sections produced by 20 unique compilations. Each control section is identified during link editing by 8 characters of user data specified by the IDENTIFY control statement. The size of the identification data is computed as follows:

A = 20
I = 100
n = 8

269 + 320 + 1400 = 1989 bytes

If the optional user data specified on the IDENTIFY control statements is omitted, the size can be reduced considerably, as computed below:

269 + 320 = 589 bytes

If maximum number of *downward calls* made from a segment to other segments lower in its path can never exceed 340. To compute the maximum number of downward calls allowed, subtract 12 from the SYSLMOD record size and then divide the difference by 12. Examples of maximum downward calls are 84 for a SYSLMOD record size of 1024 bytes and 340 for a SYSLMOD record size of 6144 bytes.

Intermediate Data Set

The intermediate data set (SYSUT1) is used by the linkage editor to hold intermediate data records during processing. The linkage editor places intermediate data in this data set when storage allocated for input data or certain forms of out-of-sequence text is exhausted.

The following direct-access devices, if supported by the system, can be used for this data set:

IBM 2314	Storage Facility
IBM 2319	Storage Facility
IBM 2305	Fixed Head Storage Facility
IBM 3330	Disk Storage Facility
IBM 3330-1	Disk Storage Facility
IBM 3340	Disk Storage Facility
IBM 3344	Direct Access Storage Device
IBM 3350	Direct Access Storage

Linkage Editor Storage Requirements

The linkage editor requires a minimum of 74K of storage for execution.

The linkage editor program is in overlay format and uses the overlay supervisor. For VS1, the storage required by the overlay supervisor must be added to the minimum real storage requirement for the linkage editor. The storage requirement for the overlay supervisor is 512 bytes.

The storage requirement given above is for VS1 and includes the storage required by the access method modules used by the linkage editor. The linkage editor uses the basic sequential and basic partitioned access methods (BSAM and BPAM, respectively).

Since the overlay supervisor is in the link pack area in VS2, the storage requirements for the overlay supervisor should not be included when determining the size of the editor's region.

PART 2. LOADER

The Loader is a processing program. It combines basic editing and loading functions of the linkage editor and program fetch in one job step. Therefore, the *load* function is equivalent to the *link edit-go* function. The loader can be used for compile-load and load jobs.

The loader will load object modules produced by a language processor and load modules produced by the linkage editor into virtual storage for execution. Optionally, it will search a call library (SYSLIB) or a resident link pack area, or both, to resolve external references. The loader does not produce load modules for program libraries.

The functional characteristics, compatibility and restrictions, performance considerations, and storage considerations of the loader are described in the following sections.

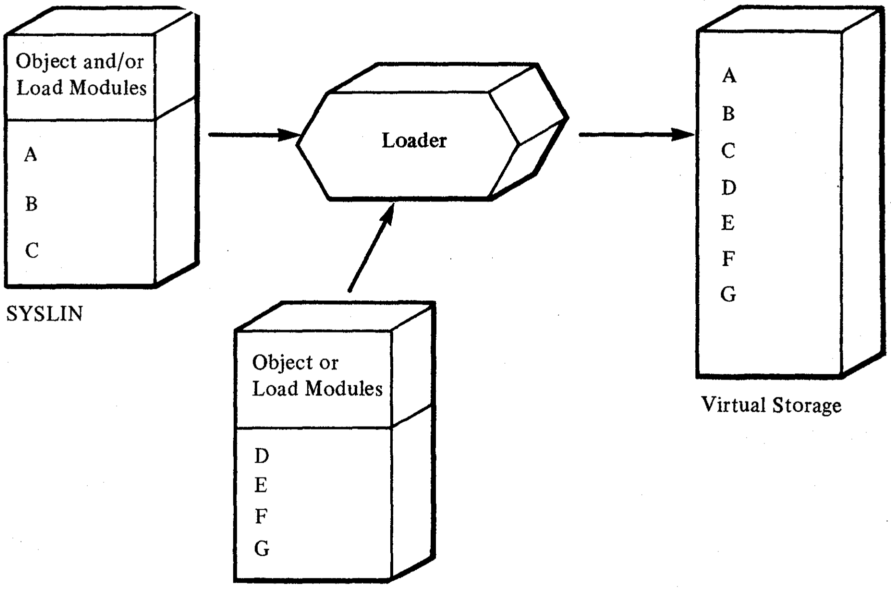
Functional Characteristics

The loader can be used with VS1 and VS2. The loader is re-enterable and, therefore, can reside in the resident link pack area.

The loader combines the following basic functions of the linkage editor and program fetch:

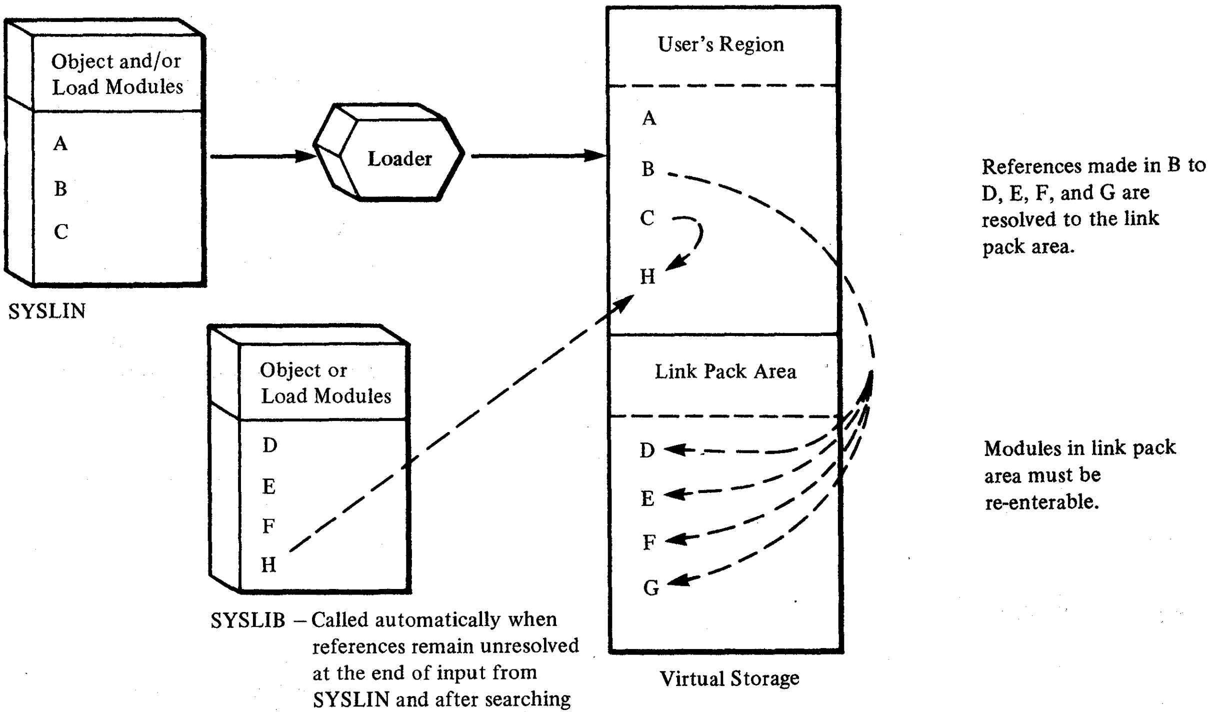
1. Resolution of external references between program modules.
2. Optional inclusion of modules from a call library (SYSLIB) or from a link pack area, or from both (Figures 60 and 61). (Inclusion of modules from a call library or the link pack area is performed, if requested, when external references remain unresolved after processing the primary input to the loader. If both are requested, the link pack area is searched first.)
3. Automatic deletion of duplicate copies of program modules (Figure 62). (The first copy is loaded and all succeeding requests use that copy.)
4. Relocation of all address constants so that control may be passed directly to the assigned entry point in virtual storage.

The diagnostics produced by the loader are similar to those of the linkage editor.



SYSLIB – called automatically when references were unresolved at the end of input from SYSLIN.

Figure 60. Loader Processing—SYSLIB Resolution


Object and/or
Load Modules
A
B
C
SYSLIN
Loader
Object or
Load Modules
D
E
F
H
User's Region
A
B
C
H
Link Pack Area
D
E
F
G
Virtual Storage
References made in B to D, E, F, and G are resolved to the link pack area.
Modules in link pack area must be re-enterable.

SYSLIB – Called automatically when references remain unresolved at the end of input from SYSLIN and after searching the link pack area.

Figure 61. Loader Processing—Link Pack Area and SYSLIB Resolution

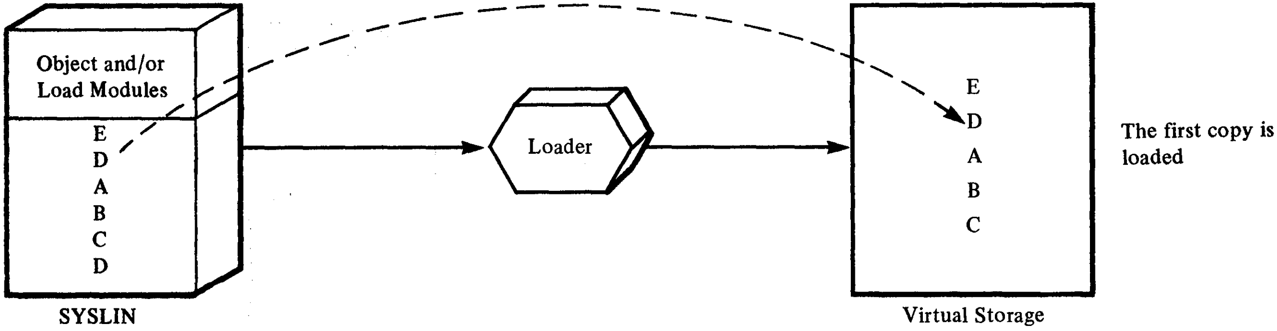


Figure 62. Loader Processing—Automatic Editing

Compatibility and Restrictions

The loader accepts the same basic input as the linkage editor:

1. All object modules that can be processed by the linkage editor can be input to the loader.
2. All load modules produced by the linkage editor can be input to the loader (except load modules edited with the NE option).

The loader supports the following linkage editor options: MAP, LET, NCAL, SIZE, and TERM. All other linkage editor options and attributes are not supported, but, if used, they will not be considered as errors. A message will be listed on SYSLOUT indicating that they are not supported. The supported options are specified in the PARM field of the EXEC statement, or with the LINK, ATTACH, LOAD, or XCTL macro instruction. In addition to the supported linkage editor options, the loader provides several other options. All loader options are described under "EXEC Statement" in the section "Using the Loader."

The loader does not process linkage editor control statements (for example, INCLUDE, NAME, OVERLAY, etc.). If they are used, they will not be treated as errors and a message will be listed on SYSLOUT indicating that the control statements are not supported.

The loader and the linkage editor are bound by the same input conventions. (These conventions are discussed in Part 1 of this publication.) In addition, the loader can accept load modules in the SYSLIN data set and object modules from a data area in virtual storage.

The loader does not use auxiliary storage space for work areas; that is, there is no loader function corresponding to the linkage editor's creation of intermediate work data sets or output load modules.

Time Sharing Option (TSO)

When the loader is used under TSO (VS2 only), it is invoked by the loader prompter, a program that acts as an interface between the user and the operating system and the loader. Under TSO, execution of the loader and definition of the data sets used by the loader are described to the system through use of the LOADGO command that causes the prompter to be executed. Operands of the LOADGO command can also be used to specify the loader options a job requires.

Complete procedures for using the LOADGO command to load and execute an object module are given in the *OS/VS2 TSO Terminal User's Guide*.

Processing Object Modules in Virtual Storage

The loader can act as an interface with a compiler that has the ability to construct a data area of one or more object modules in virtual storage as an alternative to a data set on a secondary storage volume (such as a tape or disk). Such a compiler passes the loader a description of the internal data area, which the loader then processes as primary input. This internal data area replaces external SYSLIN data set input to the loader.

Instead of placing text records for the object module in the internal data area, the compiler can pass pointers to preloaded text. The loader can then perform its relocation and linkage functions on the preloaded text itself; text is not moved during processing.

Loaded Program Restrictions

Any loaded program that issues an XCTL macro instruction or an IDENTIFY macro instruction in a VS1 environment will not execute properly. It is recommended that any such program be processed by the linkage editor.

If an IDENTIFY macro instruction is issued by the loaded program, IDENTIFY returns a 'OC' code in register 15. This code means that the entry point address is not within an eligible load module and that the entry point was not added.

In a VS1 environment, any data set opened by a loaded program should be closed by the program before execution is complete.

USING THE LOADER

This section discusses how to prepare an input deck for the loader and how to invoke the loader; it also describes the output from the loader.

Input for the Loader

The input deck for the loader must contain job control language statements for the loader and, optionally, for the loaded program (Figure 63).

```
//name        JOB     parameters              (optional)
//name        EXEC    PGM=LOADER,
                      PARM=(parameters)
//SYSLIN      DD      parameters
//SYSLIB      DD      parameters              (optional)
//SYSLOUT     DD      parameters              (optional)
//SYSTERM     DD      parameters              (optional)

//    (optional DD statements and data
//    required for loaded program)
```

Figure 63. Input Deck for the Loader—Basic Format

Only the EXEC statement and the SYSLIN DD statement are required for a loader step. The JOB statement is required if the loader is the first step in the job.

EXEC Statement

The EXEC statement is used to call the loader and to specify options for the loader and for the loaded program. The loader is called by specifying PGM=IEWLDRGO or PGM=LOADER (see "Invoking the Loader"). Loader and loaded program options are specified in the PARM field of the EXEC statement. The PARM field must have the following format:

,PARM=' [*loaderoption* [,...][/programoption[,...]]'

Note that the loaded program options, if any, must be separated from the loader options by a slash (/). If there are no loader options, the program options must begin with a slash. The entire PARM field may be omitted if there are no loader or loaded program options.

Parameters must be enclosed in single quotes when special characters (/ and =) are used.

The loader options are:

MAP
The loader produces a map of the loaded program that lists external names and their absolute storage addresses on the SYSLOUT data set. (If the SYSLOUT DD statement is not used in the input deck, this option is ignored.) The module map is described in "Loader Output" in this section.

NOMAP
A map is not produced.

RES
An automatic search of the link pack area queue is to be made. This search is always made after processing the primary input (SYSLIN), and before searching the SYSLIB data set. When this option is specified, the CALL option is automatically set.

NORES
No automatic search of the link pack area queue is to be made.

CALL
An automatic search of the SYSLIB data set is to be made. (If the SYSLIB DD statement is not included in the input deck, this option is ignored.)

NOCALL | NCAL
An automatic search of the SYSLIB data set will not be made. When this option is specified, the NORES option is automatically set.

LET
The loader will try to execute the object program even though a severity 2 error condition is found. (A severity 2 error condition is one that could make execution of the loaded program impossible.)

NOLET
The loader will not try to execute the loaded program if a severity 2 error condition is found.

SIZE=*size*
specifies the size, in bytes, of dynamic virtual storage that can be used by the loader (*see* Appendix F).

EP=*name*
specifies the external name to be assigned as the entry point of the loaded program. This parameter must be specified if the entry point of the loaded program is in an input load module. For FORTRAN, ALGOL, and PL/I, these entry points must be MAIN, IHIFSAIN, and IHENTRY, respectively, unless changed by compiler options.

NAME=*name*
specifies the name to be used to identify the loaded program to the system. If this parameter is not used, the loaded program will be named **GO.

PRINT
Informational and diagnostic messages are produced on the SYSLOUT data set.

NOPRINT
Informational and diagnostic messages are not produced on the SYSLOUT data set. SYSLOUT is not opened.

TERM
Numbered diagnostic messages are to be sent to the SYSTERM data set. Although intended to be used when operating under the Time Sharing Option (TSO), the SYSTERM data set can be used to replace or supplement the SYSLOUT data set at any time. (If the SYSTERM DD statement is not included in the input deck, this option is ignored.)

NOTERM
Numbered diagnostic messages are not to be sent to the SYSTERM data set.

The default options are: NOMAP, RES, CALL, NOLET, SIZE=100K, PRINT, NAME=**GO and NOTERM. For VS1, the default options

NOMAP, RES, CALL, NOLET, SIZE=100K, and PRINT may be changed during system generation by using the LOADER macro instruction.

The following are examples of the EXEC statement. In these examples, X and Y are parameters required by the loaded program.

```
//LOAD        EXEC    PGM=LOADER
//LOAD        EXEC    PGM=HEWLDRGO,
//                    PARM='MAP,EP=FIRST/X,Y'
//LOAD        EXEC    PGM=LOADER,PARM='/X,Y'
//LOAD        EXEC    PGM=LOADER,PARM=NOPRINT
//LOAD        EXEC    PGM=LOADER,PARM=(MAP,LET)
//LOAD        EXEC    PGM=LOADER,
//                    PARM='NAME=NEWPROG.,TERM,NOPRINT'
```

For further details in coding the EXEC statement refer to *OS/VS1 JCL Reference* and *OS/VS2 JCL.*

DD Statements

The loader uses four DD statements named SYSLIN, SYSLIB, SYSLOUT, and SYSTERM. (For VS1, these ddnames can be changed during system generation with the LOADER macro instruction.) The SYSLIN DD statement must be used in every loader job. The other three are optional.

The following considerations apply to the DCB parameter of SYSLIN, SYSLIB, and SYSLOUT.

- For better performance, BLKSIZE and BUFNO can be specified.
- If BUFNO is omitted, BUFNO=2 is assumed.
- Any value given to BUFNO is assumed for NCP (number of channel programs).
- If RECFM=U is specified, BUFNO=2 is assumed, and BLKSIZE and LRECL are ignored.
- RECFM=V is not accepted.
- RECFM=FBSA is always assumed for SYSLOUT.
- If RECFM is omitted, RECFM=F is assumed for SYSLIN and SYSLIB.
- If BLKSIZE is omitted, the value given to LRECL is assumed.
- LRECL=121 is assumed for SYSLOUT unless the loader is operating under the Time Sharing Option (TSO), when LRECL=81 is assumed.
- If LRECL is omitted, LRECL=80 is assumed for SYSLIN and SYSLIB.
- If OPTCD=C is used to specify chained scheduling, an additional 2K (2048 bytes) of virtual storage is needed in the user's region if the necessary data management routines are not resident.

Note: The SYSTERM data set will always consist of unblocked 81-character records with BUFNO=2 and RECFM=FSA. Because these values are fixed, the DCB parameter need not be used.

In addition to the DD statements used by the loader, any DD statements and data required by the loaded program must be included in the input deck.

SYSLIN DD Statement

The SYSLIN DD statement defines the input data for the loader. This input can be either object modules produced by a language translator, or load modules produced by the linkage editor, or both. The data sets defined by the SYSLIN DD statement can be either sequential data sets or members of a partitioned data set, or both. The DSNAME parameter for a partitioned data set must indicate the member name, that is, DSNAME=dsname(membername). Concatenation can be used to include more than one module in SYSLIN.

The following are examples of the SYSLIN DD statement. The first example defines a member of a previously cataloged partitioned data set:

```
//SYSLIN      DD      DSNAME=OUTPUT.FORT(MOD12),
//                    DISP=OLD,DCB=BLKSIZE=3200
```

The second example defines a sequential data set on magnetic tape:

```
//SYSLIN      DD      DSNAME=PROG15,UNIT=2400,DISP=(OLD,
//                    KEEP),VOLUME=(PRIVATE,RETAIN,
//                    SER=MCS167)
```

The third example defines a data set which was the output of a previous step in the same job:

```
//SYSLIN      DD      DSNAME=*.COBOL.SYSLIN,DISP=(OLD,
//                    DELETE)
```

The fourth example shows the concatenation of three data sets. The first two data sets are members of different partitioned data sets; the first is an object module and the second is a load module. The third data set is in the input stream following a SYSLIN DD statement (see "Loaded Program Data" in this section).

```
//SYSLIN      DD      DSNAME=PGMLIB.SET1(RFS1),DISP=OLD,
//                    DCB=(BLKSIZE=3200,RECFM=FB)
//            DD      DSNAME=PGMLIB.SET2(ABC5),DISP=OLD,
//                    DCB=RECFM=U
//            DD      DDNAME=SYSIN
```

SYSLIB DD Statement

The SYSLIB data set contains IBM-supplied or user-written library routines to be included in the loaded program. The data set is searched when unresolved references remain after processing SYSLIN and optionally searching the link pack area.

The SYSLIB data set is used to resolve an external reference when the following conditions exist: the external reference must be (1) a member name or an alias of a module in the data set, and (2) defined as an external name in the external symbol dictionary of the module with that name. If the unresolved external reference is a member name or an alias in the library, but is not an external name in that member, the member is processed but the external reference remains unresolved unless subsequently defined.

The data set defined by the SYSLIB DD statement must be a partitioned data set that contains either object modules or load modules, but not both. Concatenation may be used to include more partitioned data sets in SYSLIB. All concatenated data sets must contain the same type of modules (object or load).

The following are examples of the SYSLIB DD statement. The first example defines a cataloged partitioned data set that can be shared by other steps:

```
//SYSLIB      DD      DSNAME=SYS1.ALGLIB,DISP=SHR
```

The second example shows the concatenation of two data sets:

```
//SYSLIB      DD      DSNAME=SYS1.PL1LIB,DISP=SHR
//            DD      DSNAME=LIBMOD.MATH,DISP=OLD
```

SYSLOUT DD Statement

The SYSLOUT DD statement is used for error and warning messages and for an optional map of external references (see "Loader Output" in this section). The data set defined by this DD statement must be a sequential data set. The DCB parameter can be used to specify the blocking factor (BLKSIZE) of this data set. For better performance, the number of buffers (BUFNO) to be allocated to SYSLOUT can also be specified.

The following are examples of the SYSLOUT DD statement. The first example specifies the system output unit:

```
//SYSLOUT     DD      SYSOUT=A
```

The second example defines a sequential data set on a 1443 printer:

```
//SYSLOUT     DD      UNIT=1443,DCB=(BLKSIZE=121,
//                    BUFNO=4)
```

SYSTERM DD Statement

The SYSTERM DD statement defines a data set that is used for numbered diagnostic messages only. When the loader is being used under the Time Sharing Option (TSO) (VS2 only) of the operating system, the SYSTERM DD statement defines the terminal output data set. However, SYSTERM can also be used at any time to replace or supplement the SYSLOUT data set. Because the SYSTERM data set is not opened unless the loader must issue a diagnostic message, using SYSTERM instead of SYSLOUT can reduce loader processing time.

When the SYSTERM data set replaces the SYSLOUT data set, the numbered messages in the SYSTERM data set are the only diagnostic output; when SYSTERM supplements the SYSLOUT data set, the numbered messages appear in both data sets, and optional diagnostic and informational output, such as a list of options or a module map, can be obtained on SYSLOUT.

The DCB parmeters for SYSTERM are fixed and need not be specified. The SYSTERM data set always consists of unblocked 81-character records with BUFNO=2 and RECFM=FSA.

The following example shows the SYSTERM DD statement when used to specify the system output unit:

```
//SYSTERM     DD      SYSOUT=A
```

Loaded Program Data

Loaded program data and loader data can both be specified in the input reader in VS1 and VS2. Loaded program data can be defined by a DD statement following the loader data.

Figure 64 shows the loading of a previously compiled FORTRAN problem program. The program to be loaded (loader data) follows the SYSLIN DD statement. The loaded program data follows the FT05F001 DD statement.

```
//LOAD        JOB     MSGLEVEL=1
//LDR         EXEC    PGM=LOADER,PARM=MAP
//SYSLIB      DD      DSNAME=SYS1.FORTLIB,DISP=SHR
//SYSLOUT     DD      SYSOUT=A
//FT06F001    DD      SYSOUT=A
//SYSLIN      DD      *

     ( Loader data )

/*
//FT05F001    DD      *

     ( Loaded program data )

/*
```

Figure 64. Loader and Loaded Program Data in VS1 or VS2 Input Stream

Invoking The Loader

The loader can be referred to by either its program name, IEWLDRGO, or its alias, LOADER. The loader can be invoked through the EXEC statement, as described in "Input for the Loader," or through one of the following macro instructions.

[*symbol*]	**LINK**	**EP=** *loadername*, **PARAM=(***optionlist* [,*ddname list*]), **VL=1**

[*symbol*]	**ATTACH**	**EP=** *loadername*, **PARAM=(***optionlist* [,*ddname list*]), **VL=1**

[*symbol*]	**LOAD**	**EP=** *loadername*

[*symbol*]	**XCTL**	**EP=** *loadername*

EP= *loadername*

specifies the symbolic name of the loader. The entry point at which execution is to begin is determined by the control program from the library directory entry.

PARAM=(*optionlist* [, *ddname list*])

specifies, as a sublist, address parameters to be passed to the loader. The first fullword in the address parameter list contains the address of the option list for the loader and/or loaded program. The second fullword contains the address of the ddname list. If standard ddnames are to be used, this list may be omitted.

optionlist

specifies the address of a variable length list containing the loader and loaded program options. This address must be written even though no list is provided.

The option list must begin on a halfword boundary. The two high-order bytes contain a count of the number of bytes in the remainder of the list. If no options are specified, the count must be zero.

The option list is free form, with the loader and loaded program options separated by a slash (/), and with each option separated by a comma. No blanks or zeros should appear in the list.

ddname list

specifies the address of a variable length list containing alternative ddnames for the data sets used during loader processing. If the standard ddnames are used, this operand may be omitted.

The format of the ddname list is identical to the format of the ddname list for invoking the linkage editor; the 8-byte entries in the list are as follows:

Entry	**Alternate Name for:**
1	SYSLIN
2	not applicable
3	not applicable
4	SYSLIB
5	not applicable
6	SYSLOUT
7-11	not applicable
12	SYSTERM

VL

specifies that the sign bit is to be set to 1 in the last fullword of the address parameter list.

Figure 65 shows an Assembler language program that uses the LINK macro instruction to refer to the loader.

```
          SAVE    (14,12)                  initialize-save
           .                               registers and point
           .                               to new save area
           .
          LA      13,SAVEAREA
           .
           .
           .
          LINK    EP=LOADER,PARAM=(PARM),VL=1
           .
           .
           .
          L       13,4(13)
          RETURN  (14,12),T
           .
           .
           .
          DS OH
PARM      DC  AL2(LENGTH)                  length of options
OPTIONS   DC  C'NOPRINT,CALL/X,Y,Z'        loader and loaded
LENGTH    EQU *-OPTIONS                    program options
SAVEAREA  DS  18F                          Save area
           .
           .
           .
          END
```

Figure 65. Using the LINK Macro Instruction to Refer to the Loader

If desired, the loader may be used to process a program but not execute it. To invoke just the portion of the loader that processes input data, specify either the name HEWLOAD or the name HEWLOADR with a LOAD and CALL macro instruction.

HEWLOAD, which is used with VS2 only, will both load and identify the program. HEWLOAD returns the address of an 8-character name in register 1. This name can be used with an ATTACH, LINK, LOAD, or XCTL macro instruction to invoke the loaded program. A user program that is going to attach a loaded program, should avoid specifying SZERO=NO in its ATTACH macro. If SZERO=NO must be specified, the user program should issue a LOAD for the loaded program before performing the ATTACH and a DELETE for the loaded program after the ATTACH.

HEWLOADR, which can be used with VS1 or VS2, will load the program but will not identify it. HEWLOADR returns the entry point of the loaded program in register 0. Register 1 points to two full words: the first points to the begining of storage occupied by the loaded program; the second contains the size of the loaded program. This location and size can then be used in a FREEMAIN macro instruction to free the storage occupied by the loaded program when it is no longer needed.

Figure 66 shows an Assembler language program that uses the LOAD and CALL macro instructions to refer to HEWLOADR. Figure 67 shows an Assembler language program that uses the LOAD and CALL macro instructions to refer to HEWLOAD.

For further information on the use of these macro instructions, refer to *OS/VS1 Supervisor Services and Macro Instructions* or *OS/VS2 Supervisor Services and Macro Instructions.*

```
              SAVE      (14,12),T              initialize-save registers and
                .                                point to new save area
                .
                .
              ST        13,SAVEAREA+4
              LA        13,SAVEAREA
                .
                .
                .
              LOAD      EP=HEWLOADR            load the loader
              LR        15,0                   get its entry point address
              CALL      (15),(PARM1),VL        invoke the loader
                .
                .
                .
              LR        7,15                   save return code
              LR        5,0                    save entry to loaded program
              LR        6,1                    save pointer to list containing
*                                              start address and length
              DELETE    EP=HEWLOADR            delete loader
              CH        7,=H'4'                verify successful loading
              BH        FREE                   negative branch
              LR        15,5                   loading successful-get entry
*                                              point address for CALL
              CALL      (15),(PARM2),VL        invoke program
                .
                .
                .
FREE          L         0,4(6)                 get length into register 0
              L         1,0(6)                 get start address
              FREEMAIN R,LV=(0),A=(1)          delete loaded program
                .
                .
                .
              L         13,4(13)
              RETURN    (14,12),T
              DS   OH
PARM1         DC   AL2(LENGTH1)                length of loader options
OPTIONS1      DC   C'NOPRINT,CALL'             loader options
LENGTH1       EQU  *-OPTIONS1
              DS   OH
PARM2         DC   AL2(LENGTH2)                length of loaded program options
OPTIONS2      DC   C'X,Y,Z'                    loaded program options
LENGTH2       EQU  *-OPTIONS2
SAVEAREA      DS      18F                      save area
                .
                .
                .
              END
```

Figure 66. Using the LOAD and CALL Macro Instructions to Refer to HEWLOADR (Loading Without Identification)

```
           SAVE     (14,12),T               initialize-save registers and
            .                               point to new save area
            .
            .
           ST       13,SAVEAREA+4
           LA       13,SAVEAREA
            .
            .
            .
           LOAD     EP=HEWLOADR             load the loader
           LR       15,0                    get its entry point address
           CALL     (15),(PARM1),VL         invoke the loader
           LR       7,15                    save the return code
           MVC      PGMNAM(8),0(1)          save program name
           DELETE   EP=HEWLOAD              delete the loader
           CH       7,=H'4'                 verify successful loading
           BH       ERROR                   negative branch
           LINK     EPLOC=PGMNAM,PARM=(PARM2),VL=1
            .                               loading successful,
            .                               invoke program
            .
           L        13,4(13)
           RETURN   (14,12),T
           DS       0H
PARM1      DC       AL2(LENGTH1)            length of loader options
OPTIONS1   DC       C'MAP'                  loader options
LENGTH1    EQU      *-OPTIONS1
           DS       0H
PARM2      DC       AL2(LENGTH2)            length of loaded program options
OPTIONS2   DC       C'X,Y,Z'                loaded program options
LENGTH2    EQU      *-OPTIONS2
SAVEAREA   DS       18F                     save area
PGMNAM     DS       2F                      program name
            .
            .
            .
           END
```

Figure 67. Using the LOAD and CALL Macro Instructions to Refer to HEWLOAD (Loading With Identification)

Loader Output

Loader output consists of a collection of diagnostics and error messages, and of an optional storage map of the loaded program. This output is produced in the data set defined by the SYSLOUT DD and SYSTERM DD statements. If these are omitted, no loader output is produced.

SYSLOUT output includes a loader heading, and the list of options and defaults requested through the PARM field of the EXEC statement. The SIZE stated is the size obtained, and not necessarily the size requested in the PARM field. Error messages are written when the errors are detected. After processing is complete an explanation of the error is written. Loader error messages are similar to those of the linkage editor and are listed in the *OS/VS Message Library: Linkage Editor and Loader Messages.*

SYSTERM output includes only numbered warning and error messages. These messages are written when the errors are detected. After processing is complete, an explanation of each error is written.

The storage map includes the name and absolute address of each control section and entry point defined in the loaded program. Each map entry marked with an asterisk (*) comes from the data set specified on the SYSLIB DD statement. Two asterisks (**) indicate the entry was found in the link pack area; three asterisks (***) indicate the entry comes from text that was preloaded by a compiler. The TYPE column indicates what each entry on the map is used for: SD-control section, LR-label reference, and PR-pseudo register.

The map is written as the input to the loader is processed, so all map entries appear in the same sequence in which the input ESD items are defined. The total size and storage extent of the loaded program are also included. For PL/I programs, a list is written showing pseudo-registers with their addresses assigned relative to zero. Figure 68 shows an example of a module map. In a VS2 environment, the loader issues an informational message when the loaded program terminates abnormally.

```
                                                OS/VS LOADER
OPTIONS USED-PRINT,MAP,NOLET,CALL,NORES,SIZE=424176

  NAME      TYPE   ADDR  NAME       TYPE  ADDR  NAME       TYPE  ADDR  NAME       TYPE  ADDR    NAME    TYPE   ADDR

SAMPL2B      SD   161E0 SAMPL2BA     SD  16EC8  IHEMAIN     SD   17CF8  IHENTRY     SD   17D00  IHESPRT   SD    17D10
SYSIN        SD   17D48 IHEVQC  *    SD  17D80  IHEVQCA *   LR   17D80  IHEVQB  *   SD   17FD8  IHEVQBA* LT    17FD8
IHEDIA  *    SD   183C0 IHEDIAA *    LR  183C0  IHEDIAB *   LR   183C2  IHEVPE  *   SD   18608  IHEVPEA* LR    18608
IHEVPA  *    SD   18870 IHEVPAA *    LR  18870  IHEVFC  *   SD   189D0  IHEVFCA *   LR   189D0  IHEVPC * SD    189F8
IHEVPCA *    LR   189F8 IHEVFE  *    SD  18BE8  IHEVFEA *   LR   18BE8  IHEVSC  *   SD   18C08  IHEVSCA* LR    18C08
IHEDNC  *    SD   18CB8 IHEDNCA *    LR  18CB8  IHEDOA  *   SD   18F30  IHEDOAA *   LR   18F30  IHEDOAB* LR    18F32
IHEDMA  *    SD   19010 IHEDMAA *    LR  19010  IHEVFD  *   SD   19108  IHEVFDA *   LR   19108  IHEVFA * SD    19160
IHEVFAA *    LR   19160 IHEVPB  *    SD  19248  IHEVPBA *   LR   19248  IHEXIS  *   SD   193F0  IHEXISO* LR    193F0
IHEIOB  *    SD   19488 IHEIOBA *    LR  19488  IHEIOBB *   LR   19490  IHEIOBC *   LR   19498  IHEIOBD* LR    194A0
IHESARC *    LR   1A9CB IHESADD *    LR  1A9DE  IHESAFF *   LR   1AA18  IHEPRT  *   SD   1AB70  IHEPRTA* LR    1AB70
IHEBEGA *    LR   1AE28 IHEERR  *    SD  1AE68  IHEERRD *   LR   1AE68  IHEERRC *   LR   1AE72  IHEERRB* LR    1AE7C
IHEERRA *    LR   1AE86 IHEERRE *    LR  1B4E2  IHEIOF  *   SD   1B580  IHEIOFB *   LR   1B580  IHEIOFA* LR    1B582
IHEITAZ *    LR   1B81E IHEITAX *    LR  1B82A  IHEITAA *   LR   1B83E  IHEDCN  *   SD   1B860  IHEDCNA* LR    1B860
IHEDCNB *    LR   1B862 IHEIOD  *    SD  1BA50  IHEIODG *   LR   1BA50  IHEIODP *   LR   1BA52  IHEIODT* LR    1BB4A
IHEVTB  *    SD   1BCF0 IHEVTBA *    LR  1BCF0  IHEVQA  *   SD   1BD78  IHEVQAA *   LR   1BD78

IHEQINV      PR      00 IHEGERR      PR      4  SAMPL2BB    PR       8  SAMPL2BC    PR       C  IHEQSPR  PR       10
SYSIN        PR      14 IHEQLSA      PR     18  IHEQLWO     PR      1C  IHEQLW1     PR      20  IHEQLW2  PR       24
IHEQLW3      PR      28 IHEQLW4      PR     2C  IHEQLWE     PR      30  IHEQLCA     PR      34  IHEQVDA  PR       38
IHEQFVD      PR      3C IHEQCFL      PR     40  IHEQFOP     PR      48  IHEQADC     PR      4C  IHEQXLV  PR       50
IHEQEVT      PR      58 IHEQSLA      PR     60  IHEQSAR     PR      64  IHEQLWF     PR      68  IHEQRTC  PR       6C
IHEQSFC      PR      70

IEW1001  IHEUPBA
IEW1001  IHEUPAA
IEW1001  IHETERA
IEW1001  IHEM91C
IEW1001  IHEM91B
IEW1001  IHEM91A
IEW1001  IHEDDOD
IEW1001  IHEVPFA
IEW1001  IHEVPDA
IEW1001  IHEDBNA
IEW1001  IHEVSFA
IEW1001  IHEVSBA
IEW1001  IHEVCAA
IEW1001  IHEVSAA
IEW1001  IHEDNBA
IEW1001  IHEUPBB
IEW1001  IHEUPAB
IEW1001  IHEVSEB

  TOTAL LENGTH      5068
  ENTRY ADDRESS     17D00

IEW1001  WARNING - UNRESOLVED EXTERNAL REFERENCE (NOCALL SPECIFIED)
```

Figure 68. Module Map Format Example

APPENDIX D: SAMPLE INPUT FOR THE LOADER

Figure 69 shows an input deck for a load job. A previously assembled program, MASTER, is to be loaded. The SYSLOUT, SYSLIB, and SYSTERM DD statements are not used.

```
//LOAD        JOB     MSGLEVEL=1
//            EXEC    PGM=LOADER
//SYSLIN      DD      DSNAME=MASTER,DISP=OLD

   (DD statements and data required for execution of MASTER)

/*
```

Figure 69. Input Deck for a Load Job

Figure 70 shows an input deck for a compile-load job. The COBOL F (IEQCBL00) compiler is used for the compile step. The loaded program requires the SYSOUT, TAXRATE, and SYSIN DD statements.

```
//JOB         JOB     22,MCS,MSGLEVEL=1
//COBOL       EXEC    PGM=IEQCBL00,PARM=MAP,REGION=86K,RD=R
//SYSPRINT    DD      SYSOUT=A
//SYSPUNCH    DD      UNIT=SYSCP
//SYSUT1      DD      UNIT=SYSDA,SPACE=(TRK,(100,10))
//SYSUT2      DD      UNIT=SYSDA,SPACE=(TRK,(100,10))
//SYSUT3      DD      UNIT=SYSDA,SPACE=(TRK,(100,10))
//SYSUT4      DD      UNIT=SYSDA,SPACE=(TRK,(100,10))
//SYSLIN      DD      DSNAME=&&LOADSET,DISP=(MOD,PASS),
//                    UNIT=SYSSQ,SPACE=(TRK,(30,10))
//SYSIN       DD      *

   (source program)

//LOAD        EXEC    PGM=LOADER,PARM='MAP,LET',COND=(5,LT,
//                    COBOL)
//SYSLIN      DD      DSNAME=*.COBOL.SYSLIN,DISP=(OLD,
//                    DELETE)
//SYSLOUT     DD      SYSOUT=A
//SYSLIB      DD      DSNAME=SYS1.COBLIB,DISP=SHR
//SYSOUT      DD      SYSOUT=A
//TAXRATE     DD      DSNAME=TAXRATE,DISP=OLD
//SYSIN       DD      *

   (Data for Loaded Program)

/*
```

Figure 70. Input Deck for a Compile-Load Job

Figure 71 shows the compilation and loading of three modules. In the first three steps, the FORTRAN H (IEKAA00) compiler is used to compile a main program, MAIN, and two subprograms, SUB1 and SUB2. Each of the object modules is placed in a sequential data set by the compiler and passed to the loader job step. In addition to the FORTRAN library, a private library, MYLIB, is used to resolve external references. In the loader job step, MYLIB is concatenated with the SYSLIB DD statement. SUB1 and SUB2 are included in the program to be loaded by concatenating them with the SYSLIN DD statement. The SYSTERM statement is used to define the diagnostic output data set. The loaded program requires the FT01F001 and FT10F001 DD statements for execution, and it does not require data in the input stream.

```
//JOBX      JOB
//STEP1     EXEC PGM=IEKAA00,PARM='NAME=MAIN,LOAD'
             .
             .
             .
//SYSLIN    DD   DSNAME=&&GOFILE,DISP=(,PASS),UNIT=SYSSQ
//SYSIN     DD   *
```

(Source module for MAIN)

```
/*
//STEP2     EXEC PGM=IEKAA00,PARM='NAME=SUB1,LOAD'
             .
             .
             .
//SYSLIN    DD   DSNAME=&&SUBPROG1,DISP=(,PASS),UNIT=SYSSQ
//SYSIN     DD   *
```

(Source module for SUB1)

```
/*
//STEP3     EXEC PGM=IEKAA00,PARM='NAME=SUB2,LOAD'
             .
             .
             .
//SYSLIN    DD   DSNAME=&&SUBPROG2,DISP=(,PASS),UNIT=SYSSQ
//SYSIN     DD   *
```

(Source module for SUB2)

```
/*
//STEP4     EXEC PGM=LOADER
//SYSTERM   DD   SYSOUT=A
//SYSLIB    DD   DSNAME=SYS1.FORTLIB,DISP=OLD
//          DD   DSNAME=MYLIB,DISP=OLD
//SYSLIN    DD   DSNAME=*.STEP1.SYSLIN,DISP=OLD
//          DD   DSNAME=*.STEP2.SYSLIN,DISP=OLD
//          DD   DSNAME=*.STEP3.SYSLIN,DISP=OLD
//FT01F001 DD   DSNAME=PARAMS,DISP=OLD
//FT10F001 DD   SYSOUT=A
/*
```

Figure 71. Input Deck for Compilation and Loading of the Three Modules

APPENDIX E: LOADER RETURN CODES

The return code of a loader step is determined by the return codes resulting from loader processing *and* from loaded program processing.

The return code indicates whether errors occurred during the execution of the loader or of the loaded program. The return code can be tested through the COND parameter of the JOB statement specified for this job and/or the COND parameter of the EXEC statement specified in any succeeding job step. (For details, see the publication *OS/VS1 JCL Reference* or *OS/VS2 JCL.* Figure 72 shows the return codes.

Return Code	Loader Return Code[1]	Loaded Program Return Code	Conclusion or Meaning
0	0	0	Program loaded successfully, and execution of the loaded program was successful.
	4 8 (LET)	0 0	The loader found a condition that may cause an error during execution, but no error occurred during execution of the loaded program.
4	0	4	Program loaded successfully, and an error occurred during execution of the loaded program.
	4 8 (LET)	4 4	The loader found a condition that may cause an error during execution, and an error did occur during execution of the loaded program.
8	0	8	Program loaded successfully, and an error occurred during execution of the loaded program.
	4 8 (LET)	8 8	The loader found a condition that may cause an error during execution, and an error did occur during execution of the loaded program.
	8		The loader found a condition that could make execution impossible. The loaded program was not executed.
12	0	12	Program loaded successfully, and an error occurred during execution of the loaded program.
	4 8 (LET)	12 12	The loader found a condition that may cause an error during execution, and an error did occur during execution of the loaded program.
	12		The loader could not load the program successfully, execution impossible.
16	0	16	Program loaded successfully, and the loaded program found a terminating error.
	4 8 (LET)	16 16	The loader found a condition that may cause an error during execution, and a terminating error was found during execution of the loaded program.
	16		The loader could not load program, execution impossible.

[1] Error diagnostics (SYSLOUT and/or SYSTERM data set) for the loader will show the severity of errors found by the loader.

Figure 72. Return Codes

APPENDIX F: STORAGE CONSIDERATIONS

The loader requires virtual storage space for the following items:

- Loader code.
- Data management access methods.
- Buffers and tables used by the loader (dynamic storage).
- Loaded program (dynamic storage).

Region size includes all four of the above items; the SIZE option refers to the last two items.

For the SIZE option, the minimum required virtual storage is 4K plus the size of the loaded program. This minimum requirement grows to accommodate the extra table entries needed by the program being loaded. For example: FORTRAN requires at least 3K plus the size of the loaded program, and PL/I needs at least 8K plus the size of the loaded program. Buffer number (BUFNO) and blocksize (BLKSIZE) could also increase this minimum size. Figure 73 shows the appropriate storage requirements in bytes.

The maximum virtual storage that can be used is whatever virtual storage is available up to 8192K.

All or part of the storage required is obtained from user storage. If the access methods are made resident at IPL time, they are allocated in system storage. However, 6K is always reserved for system use.

In a VS2 environment the loader code could also be made resident in the link pack area. If so, it requires the following space: HEWLDRGO, the control/interface module (alias LOADER), approximately 700 bytes; HEWLOADR, the loader processing portion, approximately 13,664 bytes.

The size of the loaded program is the same as if the program had been processed by the linkage editor and program fetch.

The loader does not use auxiliary storage space for work areas.

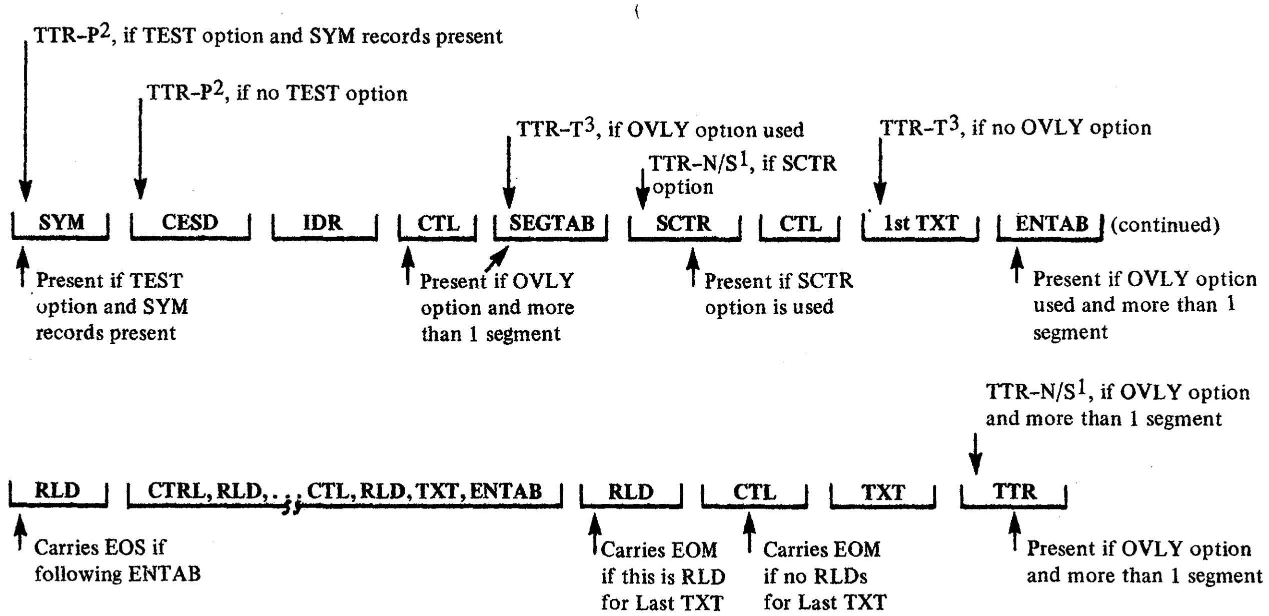
Consideration	Approximate Virtual Storage Requirements (in bytes)	Comments
Loader Code Control	700 VS1 2000 VS2	— —
Loader Code Processing	13664 VS1 14000 VS2	
Data Management	6K	BSAM
Object Module Buffers and DECBs	BUFNO(BLKSIZE + 24)	Concatenation of different BLKSIZE and BUFNO must be considered. (Minimum BUFNO=2)
Load Module Buffer and DECBs	304	—
SYSTERM DCB Buffers and DECBs	312	Allocated if TERM option is specified
SYSLOUT Buffers and DECBs	BUFNO(BLKSIZE + 24)	Buffer size rounded up to integral number of double words. (Minimum BUFNO=2)
Size of program being loaded	Program Size	Program size is restricted only by available virtual storage
Each external relocation dictionary entry	8	—
Each external symbol	20	—
Largest ESD number	4n *n* is the largest ESD number in any input module	Allocated in increments of 32 entries
Fixed Loader Table Size	1260	Subtract 88 if NOPRINT is specified
Condensed Symbol Table	12n *n* is the total number of control sections and common areas in the loaded program	Built only if TSO is operating and space is available
System Requirements	1600 VS1 4000 VS2	—

Figure 73. Virtual Storage Requirements

APPENDIX G: LOAD MODULE FORMAT

The format of a load module built by the linkage editor is shown in Figure 74.

In writing the output load module to the SYSLMOD data set, the linkage editor does not use the track overflow feature. When moving or copying load modules, it is recommended that the track overflow feature not be used on the target data set, as errors may occur in fetching the load modules for execution.


TTR-P2, if TEST option and SYM records present
TTR-P2, if no TEST option
TTR-T3, if OVLY option used
TTR-N/S1, if SCTR option
TTR-T3, if no OVLY option
SYM
CESD
IDR
CTL
SEGTAB
SCTR
CTL
1st TXT
ENTAB
(continued)
Present if TEST option and SYM records present
Present if OVLY option and more than 1 segment
Present if SCTR option is used
Present if OVLY option used and more than 1 segment
TTR-N/S1, if OVLY option and more than 1 segment
RLD
CTRL, RLD,. . . CTL, RLD, TXT, ENTAB
RLD
CTL
TXT
TTR
Carries EOS if following ENTAB
Carries EOM if this is RLD for Last TXT
Carries EOM if no RLDs for Last TXT
Present if OVLY option and more than 1 segment

[1] TTR-N/S: TTR of the note list or scatter/translation table. Used for modules in scatter load format or overlav structure only.

[2] TTR-P: TTR of the first block of the named member (load module).

[3] TTR-T: TTR of the first block of text.

Figure 74. Load Module Format

APPENDIX H: SIZE AND REGION PARAMETER GUIDELINES

This appendix gives guidelines for determining an appropriate REGION parameter value and SIZE parameter values for a linkage editor job step.

First—determine Value2 of the SIZE parameter.

$\text{Value2}=[6\text{K} \mid 6144 \mid f \mid g \mid (a+b) \mid (c{*}d) \mid (c{*}e)]$

where:

a is the length of the load module to be built

b is 0, if the length of the load module to be built is $< 40\text{K}$

b is 4K, if the length of the load module to be built is $\geq 40\text{K}$

c is an integer equal to or greater than 2, such that $c{*}d$ or $c{*}e$ is $\leq 100\text{K}$

d is the track capacity of the SYSLMOD device

e is the block size of the SYSLMOD data set

f is the length of the largest text record in load module input

g is the track capacity of the SYSUT1 device

Second—determine Value1 of the SIZE parameter.

$\text{Value1} = h + j + k$

Value1 must range between h and 9999K or 999999

where:

h is the design point of the Linkage Editor being used:

$h = 64\text{K}$

j is the excess of Value2 over 6K

$j = \text{Value2} - 6\text{K}$

k is the additional storage required to support the blocking factor for SYSLIN, object module libraries, and SYSPRINT:

Blocking Factor	**k (bytes)**
5 to 1	0K
10 to 1	18K
40 to 1	28K

Third—determine the REGION parameter.

$\text{REGION} = \text{Value1} + 10\text{K}$

GLOSSARY

This glossary includes definitions developed by the American National Standards Institute (ANSI). This material is reproduced from the American National Dictionary for Information Processing, copyright 1977 by the Computer and Business Equipment Manufacturers Association, copies of which may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. ANS definitions are preceded by an asterisk(*).

***address:** An identification, as represented by a name, label, or number, for a register, location in storage, or any other data source or destination such as the location of a station in a communication network; any part of an instruction that specifies the location of an operand for the instruction.

address constant: A value, or an expression representing a value, used in the calculation of storage addresses; can be used for branching or retrieving data.

address translation: The process of changing the address of a data item or an instruction from its virtual address to the real storage address of the location where it will reside. See also dynamic address translation.

alias name: An alternate name or entry point for a load module that is also entered in the output module library directory entry along with the member name.

automatic library call mechanism: The process whereby control sections are processed by the linkage editor or loader to resolve external references to members of partitioned data sets not resolved by primary input processing.

auxiliary storage: Data storage other than virtual storage; for example, storage on magnetic tape or direct-access devices.

common area: A control section used to reserve a virtual storage area that can be referred to by other modules; may be either named or unnamed (blank).

common segment: A segment upon which two exclusive segments are dependent.

control section: That part of a program (instructions and data) specified by the programmer to be a relocatable unit, all elements of which are to be loaded into adjoining storage locations for execution. Abbreviated CSECT.

control section name: The symbolic name of a control section.

demand paging: Transfer of a page from external page storage to real storage at the time it is needed for execution.

downward reference: A reference made from a segment to another segment lower in the same path; i.e., farther from the root segment.

dynamic address translation (DAT): (1) The change of a virtual storage address to a real storage address during execution of an instruction. See also address translation. (2) A hardware feature that performs the translation.

entry name: A name within a control section that defines an entry point, and can be referred to for execution by any control section.

exclusive reference: A reference between exclusive segments; that is, a reference from a segment in storage to an external symbol in a segment that will cause overlay of the calling segment.

exclusive segments: Segments in the same region of an overlay program, neither of which is in the path of the other; they cannot be in virtual storage simultaneously.

external name: A name that can be referred to by any control section or separately assembled or compiled module; i.e., a control section name or an entry name.

external page storage: The portion of auxiliary storage that is used to contain pages.

external reference: (1) A reference to a symbol that is defined as an external name in another module. (2) An external symbol that is defined in another module; that which is defined in the Assembler language by an EXTRN statement or by a V-type address constant, and is resolved during linkage editing. See also weak external reference.

external symbol: A control section name, entry point name, or external reference that is defined or referred to in a particular module. A symbol contained in the external symbol dictionary.

inclusive reference: A reference between inclusive segments; that is, a reference from a segment in storage to an external symbol in a segment that will not cause overlay of the calling segment.

inclusive segments: Segments in the same region of an overlay program that are in the same path; they can be in virtual storage simultaneously.

invalid exclusive reference: An exclusive reference in which a common segment does not contain a reference to the symbol used in the exclusive reference.

library: In this publication, it is a partitioned data set that always contains named members.

load module: The output of the linkage editor; a program in a format suitable for loading into virtual storage for execution.

load module buffer: An entity of virtual storage used by the linkage editor to read input load module text records and possibly to retain the text information in storage for subsequent writing of the output load module text records.

***module:** A program unit that is discreet and identifiable with respect to compiling, combining with other units, and loading, for example, the input to, or output from, an assembler, compiler, linkage editor, or executive routine.

multiple load module processing: A method of processing whereby two or more load modules can be produced in a single linkage editor job step.

***object module:** A module that is the output of an assembler or compiler and is input to a linkage editor.

overlay program: A program in which certain control sections can use the same storage locations at different times during execution.

***overlay supervisor:** A routine that controls the proper sequencing and positioning of segments of computer programs in limited storage during their exectuion.

overlay tree: A graphic representation showing the relationships of segments of an overlay program and how the segments are arranged to use the same main storage area at different times.

page: (1) A fixed-length block of instructions, data, or both, that can be transferred between real storage and external page storage. (2) To transfer instructions, data, or both between real storage and external page storage.

page fault: A program interruption that occurs when a page that is marked "not in real storage" is referred to by an active page.

paging: The process of transferring pages between real storage and external page storage.

path: All of the segments in an overlay tree between a given segment and the root segment, inclusive.

private code: An unnamed control section.

program: A logically self-contained sequence of operations or instructions that, when followed in some predetermined sequence, will produce a specified result; a sequence of instructions to be performed by an electronic computer; one or more modules, in source language or relocatable object code, or one module in executable code, that are a logically self-contained process.

program fetch: A program that prepares load modules for execution by loading them at specific storage locations; it also readjusts each address constant.

pseudo-register: In PL/I, a location in virtual storage that is used as a pointer to dynamically acquired virtual storage. It enables the PL/I compiler to generate re-enterable code. External dummy sections give the programmer using Assembler F or Assembler H the same facility.

real storage: The storage of System/370 from which the central processing unit can directly obtain instructions and data, and to which it can directly return results.

re-enterable load module: A module that can be used concurrently by more than one task.

refreshable load module: A load module that cannot be modified by itself or by any other module during execution; can be replaced by a new copy during execution by a recovery management routine without changing either the sequence or results of processing.

region: In an overlay structure, it is a contiguous area of virtual storage within which segments can be loaded independently of paths in other regions. Only one path within a region can be in virtual storage at any one time.

relocation: The modification of address constants required to compensate for a change of origin of a module, program, or control section.

root segment: That segment of an overlay program that remains in virtual storage at all times during the execution of the overlay program; the first segment in an overlay program.

scatter format: A load module attribute that permits the programmer or the control program to dynamically load control sections into noncontiguous areas of virtual storage.

segment: The smallest functional unit (one or more control sections) that can be loaded as one logical entity during execution of an overlay program.

serially reusable load module: A module that cannot be used by a second task until the first task has finished using it.

source module: The source statements that constitute the input to a language translator for a particular translation.

storage block: A 2K block of real storage to which a storage key can be assigned.

upward reference: A reference made from a segment to another segment higher in the same path; i.e., closer to the root segment.

valid exclusive reference: An exclusive reference in which a common segment contains a reference to the symbol used in the exclusive reference.

virtual address: An address that refers to virtual storage and must, therefore, be translated into a real storage address when it is used.

virtual storage: Addressable space that appears to the user as real storage, from which instructions and data are mapped into real storage locations. The size of virtual storage is limited by the addressing scheme of the computing system and the amount of auxiliary storage available, rather than by the actual number of real storage locations.

weak external reference: An external reference that does not have to be resolved during linkage editing. If it is not resolved, it appears as though its value was resolved to zero. Abbreviated WXTRN.

INDEX

GC26-3813-5

IBM®

OS/VS Linkage Editor and Loader
GC26-3813-5

Reader's Comment Form

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ____________________

Previous TNL ________________

Previous TNL ________________

Fold on two lines, tape, and mail. No postage necessary if mailed in the U.S.A. (Elsewhere, any IBM representative will be happy to forward your comments.) Thank you for your cooperation.

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

Reader's Comment Form

Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and Tape

IBM®

OS/VS Linkage Editor and Loader
GC26-3813-5

Reader's Comment Form

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

List TNLs here:

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL ____________________

Previous TNL ________________

Previous TNL ________________

Fold on two lines, tape, and mail. No postage necessary if mailed in the U.S.A. (Elsewhere, any IBM representative will be happy to forward your comments.) Thank you for your cooperation.

Please use pressure sensitive or other gummed tape to seal this form.

GC26-3813-5

Reader's Comment Form

Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and Tape

IBM®