DNOS LINK EDITOR REFERENCE MANUAL

MANUAL REVISION HISTORY

DNOS Link Editor Reference Manual (2270522-9701)

Original Issue .... August 1981

Revision A .... October 1982

Revision B .... October 1982

Texas Instruments Incorporated
ATTN: Data Systems Group, M/S 2151
P.O. Box 2909
Austin, Texas 78769-2909

DNOS Software Manuals

This diagram shows the manuals supporting DNOS, arranged according to user type. Refer to the block identified by your user group and all blocks above that set to determine which manuals are most beneficial to your needs.

All DNOS Users:

DNOS Concepts and Facilities
2270501-9701

DNOS Operations Guide
2270502-9701

DNOS System Command Interpreter (SCI) Reference Manual
2270503-9701

DNOS Text Editor Reference Manual
2270504-9701

DNOS Messages and Codes Reference Manual
2270506-9701

DNOS Reference Handbook
2270505-9701

DNOS Master Index to Operating System Manuals
2270500-9701

High-Level Language Users:

COBOL Reference Manual
2270518-9701

DNOS COBOL Programmer's Guide
2270516-9701

DNOS Performance Package Documentation
2272109-9701

TI Pascal Reference Manual
2270519-9701

DNOS TI Pascal Programmer's Guide
2270517-9701

FORTRAN-78 Reference Manual
2268681-9701

DNOS FORTRAN-78 Programmer's Guide
2268680-9701

MATHSTAT-78 Programmer's Reference Manual
2268687-9701

FORTRAN-78 ISA Extensions Manual
2268696-9701

TI BASIC Reference Manual
2308769-9701

RPG II Programmer's Guide
939524-9701

Assembly Language Users:

990/99000 Assembly Language Reference Manual
2270509-9701

DNOS Assembly Language Programmer's Guide
2270508-9701

DNOS Link Editor Reference Manual
2270522-9701

DNOS Supervisor Call (SVC) Reference Manual
2270507-9701

Productivity Tools Users:

DNOS Sort/Merge User's Guide
2272060-9701

DNOS TIFORM Reference Manual
2276573-9701

DNOS Query-990 User's Guide
2276554-9701

DNOS Data Base Management System Programmer's Guide
2272058-9701

DNOS Data Base Administrator User's Guide
2272059-9701

Data Dictionary User's Guide
2276582-9701

DNOS TIPE Reference Manual
2308786-9701

DNOS TIPE Exercise Guide
2308787-9701

Communications Software Users:

DNOS DNCS/SNA User's Guide
2302663-9701

DNOS DNCS Operations Guide
2302662-9701

DNOS DNCS 914A User's Guide
2302664-9701

DNOS 3270 Interactive Communications Software (ICS) User's Guide
2302670-9701

DNOS 3780/2780 Emulator User's Guide
2270520-9701

Systems Programmers:

DNOS System Generation Reference Manual
2270511-9701

DNOS Systems Programmer's Guide
2270510-9701

DNOS Online Diagnostics and System Log Analysis Tasks User's Guide
2270532-9701

Universal ROM Loader User's Guide
2270534-9701

Source Code Users:

DNOS System Design Document
2270512-9701

DNOS SCI and Utilities Design Document
2270513-9701

DNOS Software Manuals Summary

Concepts and Facilities
Presents an overview of DNOS with topics grouped by operating system functions. All new users (or evaluators) of DNOS should read this manual.

DNOS Operations Guide
Explains how to perform daily tasks at a DNOS installation; includes step-by-step procedures for performing such tasks as operating peripherals, initializing and backing up the system, and manipulating disk files.

System Command Interpreter (SCI) Reference Manual
Describes how to use SCI in both interactive and batch jobs. Describes command procedures and primitives and gives a detailed presentation of all SCI commands in alphabetical order for easy reference.

Text Editor Reference Manual
Explains how to use the Text Editor on DNOS and describes each of the editing commands and function keys.

Messages and Codes Reference Manual
Lists the error messages, informative messages, and error codes reported by DNOS.

DNOS Reference Handbook
Provides a summary of commonly used information for quick reference.

Master Index to Operating System Manuals
Contains a composite index to topics in the DNOS operating system manuals.

Programmer's Guides and Reference Manuals for Languages
Contain information about the languages supported by DNOS. Each programmer's guide covers operating system information relevant to the use of that language on DNOS. Each reference manual covers details of the language itself, including language syntax and programming considerations.

Performance Package Documentation
Describes the enhanced capabilities that the DNOS Performance Package provides on the Model 990/12 Computer.

Link Editor Reference Manual
Describes how to use the Link Editor on DNOS to combine separately generated object modules to form a single linked output.

Supervisor Call (SVC) Reference Manual
Presents detailed information about each DNOS supervisor call and general information about DNOS services.

DNOS System Generation Reference Manual
Explains how to generate a DNOS system for your particular configuration and environment.

User's Guides for Productivity Tools
Describe the features, functions, and use of each productivity tool supported by DNOS.

User's Guides for Communications Software
Describe the features, functions, and use of the communications software available for execution under DNOS.

Systems Programmer's Guide
Discusses the DNOS subsystems at a conceptual level and describes how to modify the system for specific application environments.

Online Diagnostics and System Log Analysis Tasks User's Guide
Explains how to execute the online diagnostic tasks and the system log analysis task and how to interpret the results.

Universal ROM Loader User's Guide
Explains how to load the operating system using the ROM loader and describes the error conditions.

DNOS Design Documents
Contain design information about the DNOS system, SCI, and the utilities. This information is useful when using a source kit.

Preface

The DNOS Link Editor is a software utility that executes under the DNOS operating system. The Link Editor combines independently generated object modules as required to form a single, executable program. This capability allows programmers to divide a source program into modules and to assemble (or compile) each module separately. The resulting object modules can then be linked together in a structure suitable for the program.

This manual explains what the Link Editor does and how to use it. Parts of the manual are written as a tutorial, starting with simple linking operations and gradually working up to more complex structures. Other parts of the manual are designed for easy reference.

The manual assumes the reader is an assembly language programmer. However, high-level language programmers who need more details on linking than is provided in the appropriate programmer's guide may also find this manual useful. The manual assumes that the reader is familiar with the 990 assembly language and has a working knowledge of the DNOS operating system, specifically the application development tools and the System Command Interpreter (SCI).

This manual contains the following sections and appendixes:

Section

1 Link Editor Overview — Provides background information on how the Link Editor works.

2 Link Control Commands — Describes the commands that direct the Link Editor; explains the functions performed and gives details on syntax and use of each command.

3 Linking Single Task Segments — Explains how to link modules in a single segment structure; covers the basic linking principles and explains how to execute the Link Editor.

4 Linking Procedure Segments — Explains how to link programs using procedure segments. This type of structure enables programs to share code for efficient use of memory.

5 Linking Program Segments — Explains how to link programs using a number of program segments. This type of structure enables programs to conserve memory while keeping execution time to a minimum.

6 Linking Overlays — Explains how to link programs using overlays. This type of structure enables you to increase the amount of code that can fit into the program's logical address space.

7 Linking Absolute Memory Partitions — Explains how to link programs for absolute memory partitioning. This type of structure allows the creation of programs for execution on systems that use a combination of read-only memory (ROM) and random-access memory (RAM).

8 Partial Linking — Explains how to perform a partial link, which links only some of the modules required by a program. Partial linking reduces the amount of memory required by the Link Editor for symbol tables.

9 Error Reporting — Explains the error and warning messages that can occur during operation of the Link Editor.

Appendix

A Object Code Format — Explains the format of the output produced by the assembler, the compilers, and the Link Editor.

B High-Level Language Information — Explains how the Pascal, COBOL, and FORTRAN compilers generate tags used by the Link Editor. This information relates some of the material covered in this manual to a specific high-level language.

C Command Syntax — Lists the link control commands and their syntax for quick reference.

In addition to this manual, the DNOS software manuals shown on the support manual diagram (frontispiece) contain information related to the DNOS Link Editor.

Contents

Illustrations

Tables

1 Link Editor Overview

1.1 INTRODUCTION

The DNOS Link Editor combines (links) object modules to form an executable program. With the Link Editor, you can divide the source code of a program into separate modules and assemble (or compile) the modules independently. This is useful for the following reasons:

- Avoids long assemblies of large programs
- Allows you to modify a single module without having to reassemble the entire program
- Reduces the symbol table size during assembly
- Allows different programs to use the same object modules
- Allows structuring of programs to share code and to conserve memory during execution

Linking consists of gathering the modules required by a program, resolving symbol references among them, and arranging them in a defined structure. You must link any program that references another program or module, as well as any program that requires structuring. You must also link programs containing common segments or data segments (produced by the CSEG and DSEG assembler directives); this reorganizes each segment type into a continuous block within the program. The Link Editor performs these functions and produces a linked output that you can install and execute.

This section gives an overview of the Link Editor and its operation. The following paragraphs describe how the Link Editor performs symbol resolution, the various ways you can structure a program, and the files associated with Link Editor operation.

1.2 SYMBOL RESOLUTION

Symbol resolution consists of assigning a value to a symbol when the symbol is first defined and replacing the symbol with that value wherever the symbol is referenced. The 990 assembler (SDSMAC) resolves symbols that are local to a single module (symbols used only within that module). The Link Editor resolves symbols that are referenced in one module but defined in another module; these are referred to as *external symbols*.

The assembler provides two directives (DEF and REF) that make external symbols available to the Link Editor for resolution. The source code of each module that defines or references an external symbol must include these directives.

The DEF directive in a module specifies the symbols that are defined in that module but referenced by other modules. The DEF directive causes the assembler to include both the symbols and their locations in the object code. These symbols are then externally defined so that they are available for linking to other modules. The assembler tags the records containing these symbols as *external definitions* (DEF tags).

The REF directive in a module specifies symbols that are referenced by that module but defined in other modules. The REF directive causes the assembler to include the symbols in the object code so that the corresponding locations can be obtained from external definitions. The assembler tags the records containing these symbols as *external references* (REF tags).

To resolve the external references, the Link Editor first builds a list of symbols from the REF tag and DEF tag records in the object modules that are to be included in the link. The Link Editor then matches the REF tag symbols with the DEF tag symbols. Each time it finds a match, the Link Editor inserts the correct locations for external symbols in the linked output. Thus, every symbol specified in a REF directive must match a symbol specified in a DEF directive in one of the modules that will be included in the link.

The Link Editor has an automatic search capability that searches defined libraries for DEF tags to match unresolved REF tags. A subsequent paragraph explains this capability further.

If the linking process completes and some unresolved external references still exist, the Link Editor issues a warning message. Section 9 explains warning and error messages and how they are reported.

1.3 PROGRAM STRUCTURE OVERVIEW

The Link Editor allows you to specify the manner and order in which modules are to be linked. This allows you to design and structure programs that use memory efficiently. The different ways you can structure a program are as follows:

- A single task segment
- A task segment with one or two attached procedure segments
- A task segment with a number of program segments
- A task segment with overlays or a task segment and program segment(s) with overlays
- A task segment with absolute memory partitions

The following paragraphs give an overview of each type of structure and explain when and why each is useful. Sections 3 through 7 explain how to link modules for each type of structure. Section 3 covers the basic linking principles, using the simplest type of structure (single task segment) as an example. Since the other sections build on the material covered in Section 3, read Section 3 first.

The Link Editor also allows you to *partially link* modules by linking only some of the modules required by a program. The output of a partial link is a single module that contains external definitions and external references that could not be resolved in the partial link. You must link this output module with other modules to form an executable program.

Partial linking is useful when more than one program can use a complete set of modules or when all the modules to be linked contain too many symbols for the Link Editor to handle at one time. Section 8 explains how to perform a partial link, building on the information covered in Section 3.

1.3.1 Single Task Segments

A *task* is a program that executes under the control of the operating system. It consists of a logical address space that defines the memory accessible to the program during execution. A program cannot access more than 64K (where K equals 1,024) bytes of memory at one time. In DNOS, the logical address space of a program can consist of several segments, with each segment occupying a single block of memory.

The *task segment* contains the addresses required to initiate execution of the program (*entry vector*). The task segment may also contain code and/or data. Every program must have one and only one task segment. In a single task structure, the entire program is contained within the task segment. Thus, the entire program is loaded in one continuous block of memory.

You can structure a program as a single task segment when the entire program does not exceed 64K bytes and when it does not share any portions of its code with other programs during execution. This type of structure does not use system resources efficiently. However, you should use it for small, nonreplicatable programs. (Nonreplicatable means that only one copy of the program can be in memory at one time.)

Single tasks are simple structures that do not require a lot of consideration in linking. The Link Editor simply includes the required modules in the order specified, resolves the references among them, and produces a single linked output module. The output module is defined as a task segment. Section 3 explains how to link modules to form a single task structure.

1.3.2 Procedure Segments

In addition to the task segment, a program can have one or two *procedure segments* to allow sharing of code between programs. The Link Editor produces one linked output module for each segment. The segments are installed separately in the program file, but the procedure segments are associated with (attached to) the task segment during installation. Procedure segments can be attached to more than one task at a time. When a program is bid, both the task segment and any attached procedure segments are loaded into memory. Once in memory, a procedure segment can be shared among several different tasks and/or several copies of the same task.

You can use this type of structure when the total size of the segments does not exceed 64K bytes. Since the entire program is loaded at one time, this type of structure does not conserve memory on the execution of one task. However, it does conserve physical memory when several tasks that can share the same procedure segment are in memory at the same time. In this case, only one copy of the shared procedure needs to be in memory; the area of memory containing the procedure segment is mapped into the logical address space of each program that uses it.

When linking this type of structure, the Link Editor arranges modules in a specified order to form the segments. You must define each segment by specifying it as a procedure segment or a task segment and assigning a name to it. Procedure segments must precede the task segment.

The procedure segment(s) usually contains only executable code and/or constant data. In order for a procedure segment to be shared, it must be reentrant; that is, one program cannot modify it so that it adversely affects the execution of another program that uses it. No more than two procedures can be linked to one task segment.

As previously stated, the task segment must contain the entry vector for the program. The task segment may also contain data or code that is not needed by other programs.

Figure 1-1 is an example of several programs (TASKA, TASKB, and TASKC) sharing one procedure (PROC1). When one of these programs is bid, both the task segment and the procedure segment are loaded into memory, each in a different area of the physical address space. When the other programs are bid, only the task segments are loaded since the procedure segment is already in memory. The appropriate memory segments are mapped into the correct positions of each program's logical address space.

Section 4 explains how to link modules to form a program with attached procedures.

1.3.3 Program Segments

As previously stated, the logical address space of a program can consist of one or more segments. In addition to the procedure and task segments, DNOS allows a number of additional *program segments*. By using program segments, you can increase the amount of code that can fit into the program's logical address space.

During execution, a program can address no more than three segments at one time. However, DNOS segment management allows a program to dynamically change the set of segments currently addressable. The program makes these changes by issuing supervisor calls (SVCs). Although the program cannot change the task segment, it can add, delete, or exchange other segments within the program's logical address space.

Program segments can contain both data and executable code. The management of program segments provides more flexibility than that of procedure segments. You can reserve a segment so that it remains in memory (if enough physical memory is available), even though it is not currently being used. This reduces execution time since the segment is already in memory and does not have to be loaded from the disk when it is needed again. Procedure and task segments are loaded into memory when the program is bid; program segments are not loaded until they are mapped in by an SVC.

The Link Editor defines the sizes and boundaries of the segments. All segments begin on 32-byte boundaries. The system maps these segments into the program's address space in one of three map positions (positions 1, 2, and 3, in that order). During linking, you must define the segments in a specific order so that each segment can be mapped into the correct position. Procedure segments must precede the task segment, and program segments must follow the task segment. You can link any number of program segments to a task, but they must all be in the last map position.

Exactly which type of segment occupies which map position depends on the type and number of segments you define. In some cases, only one or two map positions are used.

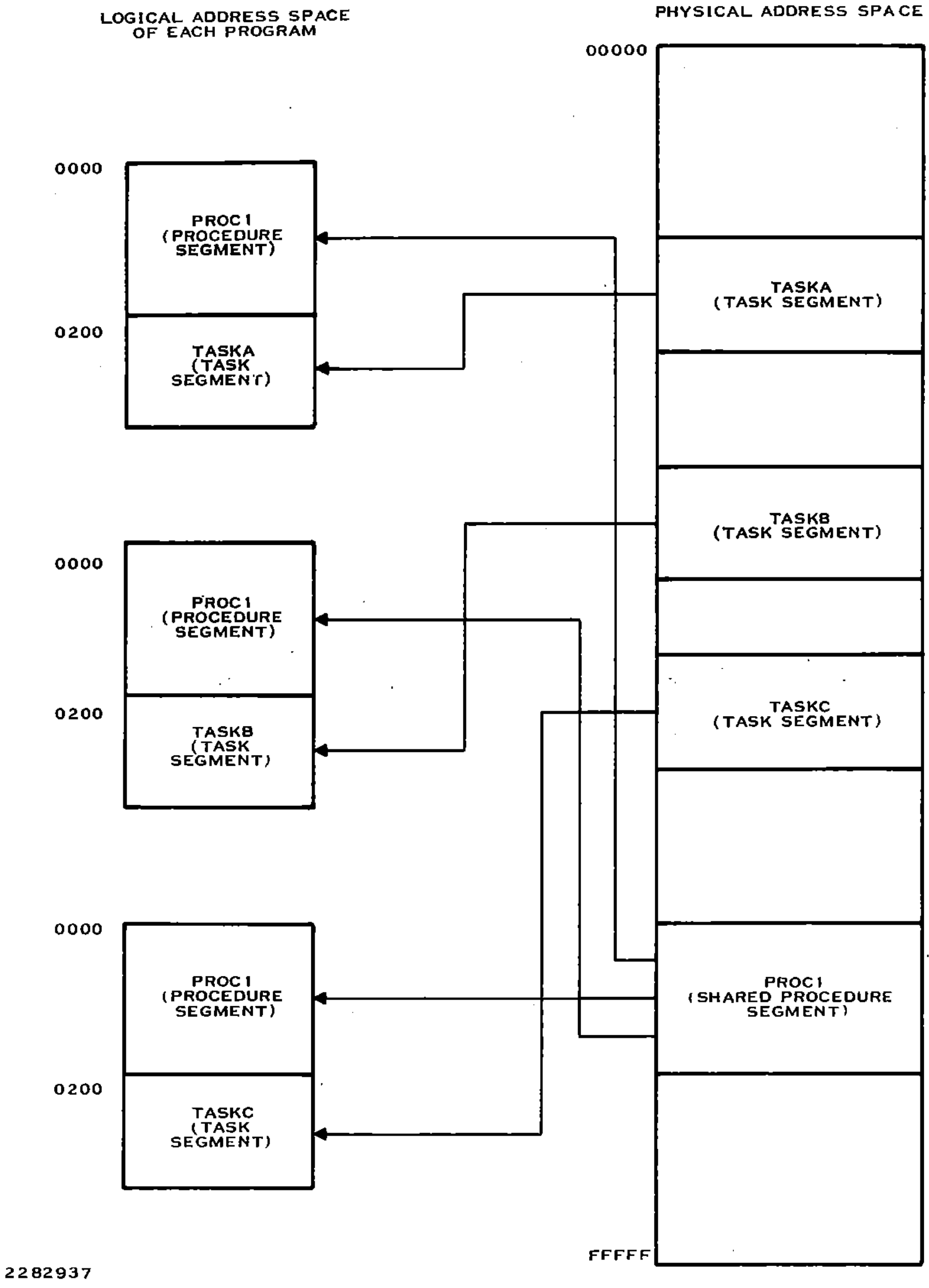


Figure 1-1. Example of Programs Sharing Procedures

In summary, you can link a program to consist of one of the following combinations of segments:

- A single task segment (map position 1)
- One procedure segment (map position 1) and one task segment (map position 2)
- Two procedure segments (one each in map positions 1 and 2) and one task segment (map position 3)
- One task segment (map position 1) and one or more program segments (all in map position 2)
- One procedure segment (map position 1), one task segment (map position 2), and one or more program segments (all in map position 3)

Figure 1-2 is an example of a program that uses program segments. During linking, one procedure segment (PROCA), one task segment (TASKA), and several program segments (SEGA, SEGB, and SEGC) were defined. When the program is bid, PROCA and TASKA are loaded into memory and mapped into positions 1 and 2, respectively, of the program's logical address space. As each program segment is called during execution, it is loaded into memory and mapped into position 3 of the program's address space. Remember that only one segment in each map position can be addressed at one time. Thus, if SEGA is exchanged with SEGB, SEGA cannot be addressed. However, if the task reserves SEGA and it remains in memory, it can be quickly exchanged so that it is mapped back into the program's address space.

Section 5 explains how to link modules in a structure that uses program segments.

1.3.4 Overlays

In an overlay structure, you can subdivide either the task segment or one or more program segments into parts called *phases*. A phase is the smallest functional unit that can be loaded as a logical entity during execution. The Link Editor produces one output module for each defined phase. The task or program segment must be designed so that it requires only certain phases to be in memory at one time. When a new phase is needed, it is loaded so that it replaces, or overlays, a phase that is currently in memory.

The Link Editor allows you to establish different levels of overlays. As you define each phase, you must assign a name and a level number to it. Only one phase can be at level 0. This phase is called the *root phase* and contains code that must remain in memory while the task is executing.

You can define any number of phases at levels 1 and higher. These phases are installed as *overlays*. Only one phase at each level can be in memory at one time; thus, phases defined at the same level must be independent of each other.

A series of phases, starting with the root phase and including a phase at each successively higher level, comprises an *overlay path*. When a specific phase is referenced, all phases in its path (that is, all phases between the referenced phase and the root phase) must be in memory. The root phase can load the overlays as they are needed by issuing SVCs. Alternatively, DNOS provides an Overlay Manager, which can be linked with a program to load overlays automatically in the task segment.

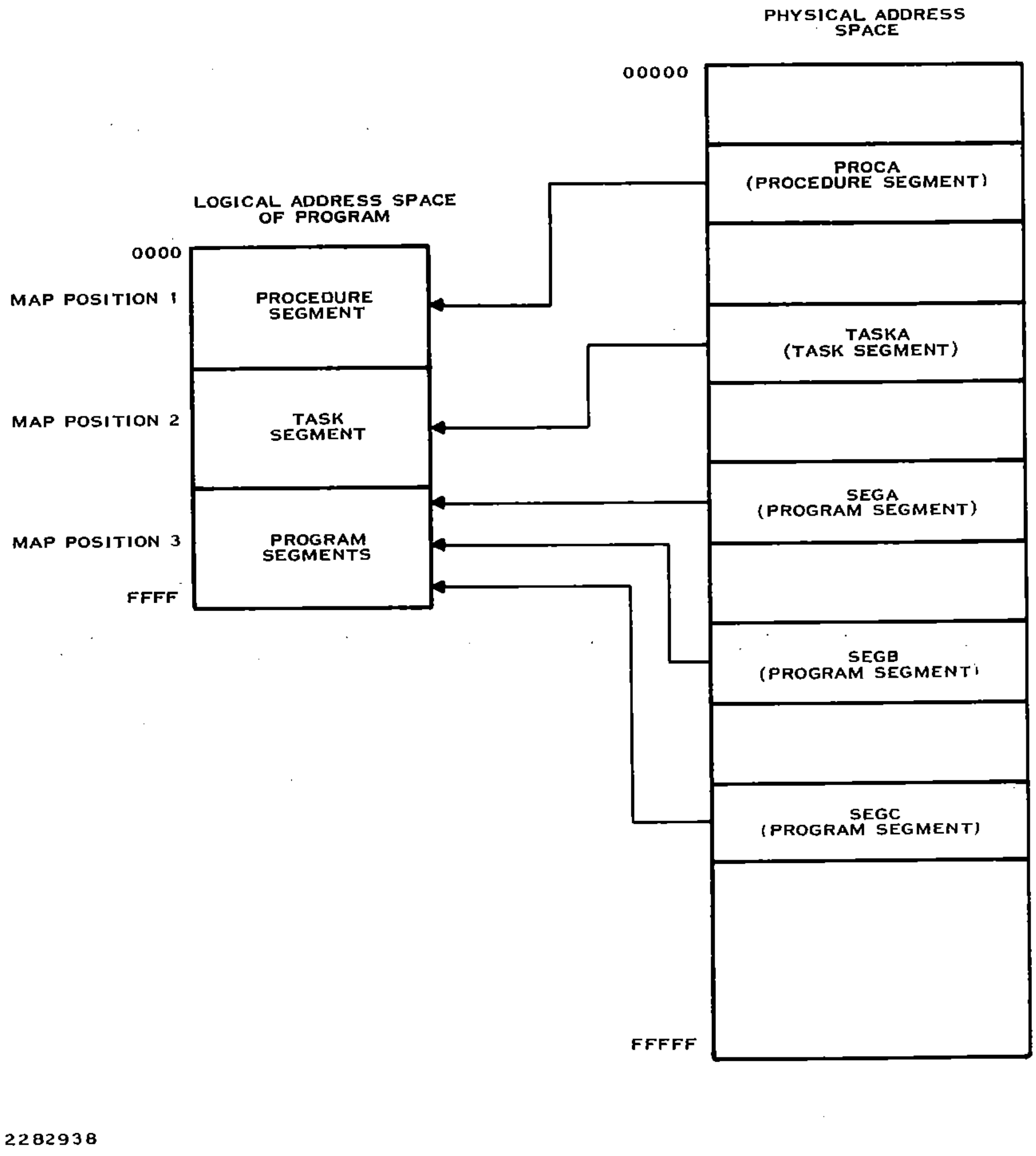


Figure 1-2. Example of a Task Using Program Segments

Figure 1-3 shows the structure of a task segment with overlays. The task segment consists of five phases at three levels. ROOTPH is the root phase at level 0 and is loaded into memory when the program is bid. ROOTPH calls both PHASEA and PHASEB; however, since they are independent of each other, only one needs to be in memory at one time. Thus, when PHASEA is no longer needed and PHASEB is required, PHASEB can be loaded into the same locations as PHASEA. In the same manner, PHASED can overlay PHASEC when it is needed. However, when either PHASEC or PHASED is in memory, PHASEB must also be in memory.

You can use overlay structures for large programs to increase the amount of code that can fit into the logical address space of a task or program segment. Each overlay is loaded into memory only when it is called. The memory allocated for the segment containing overlays is only enough to accommodate the largest overlay path. Thus, using the example in Figure 1-3, 0400 bytes of memory would be allocated for the task segment (the sum of ROOTPH, PHASEB, and PHASEC).

Section 6 further explains overlay structures and how to link modules to form such structures.

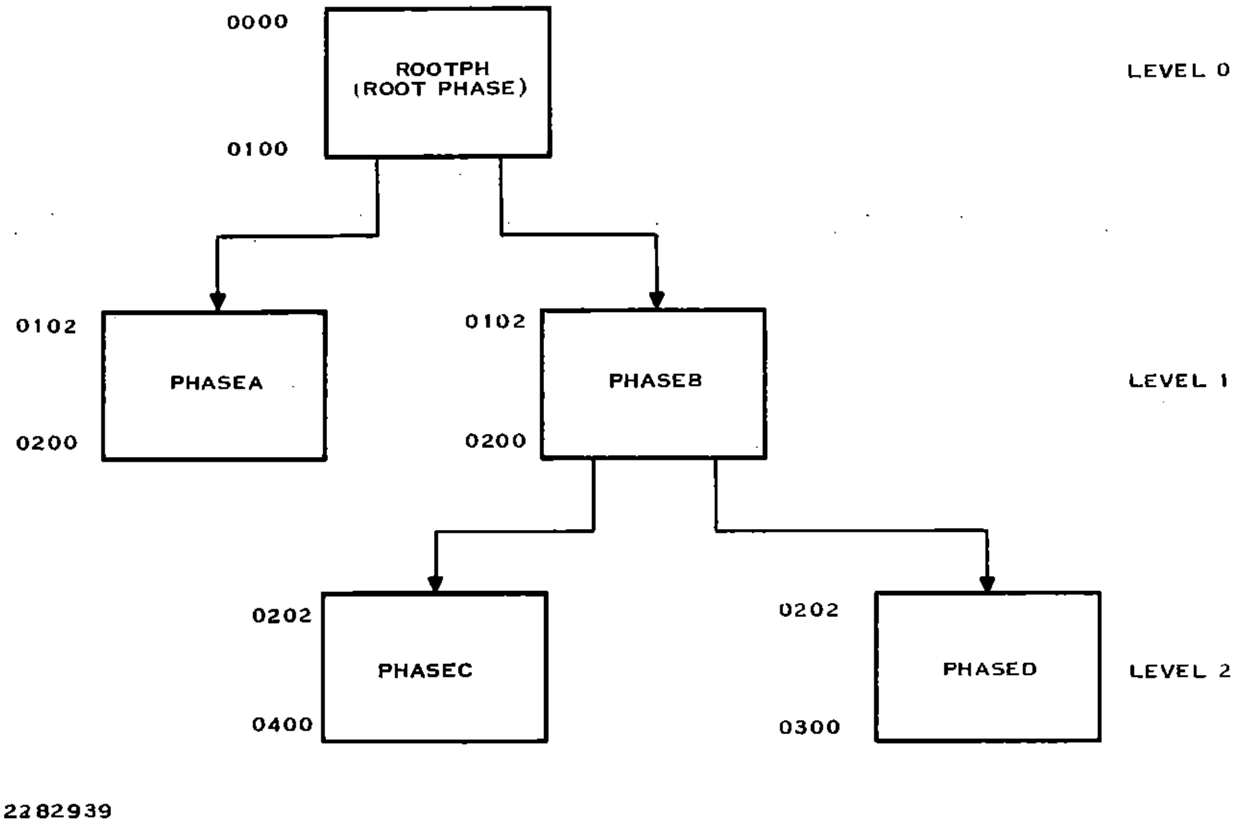


Figure 1-3. Example of an Overlay Structure

1.3.5 Absolute Memory Partitions

The Link Editor can structure programs for development systems that will eventually use a combination of read-only memory (ROM) and random-access memory (RAM). The address space of these systems is divided into two partitions: one located in the ROM area and one located in the RAM area. Programs that execute on these systems must be structured so that different portions of the code can be placed in the correct memory partition.

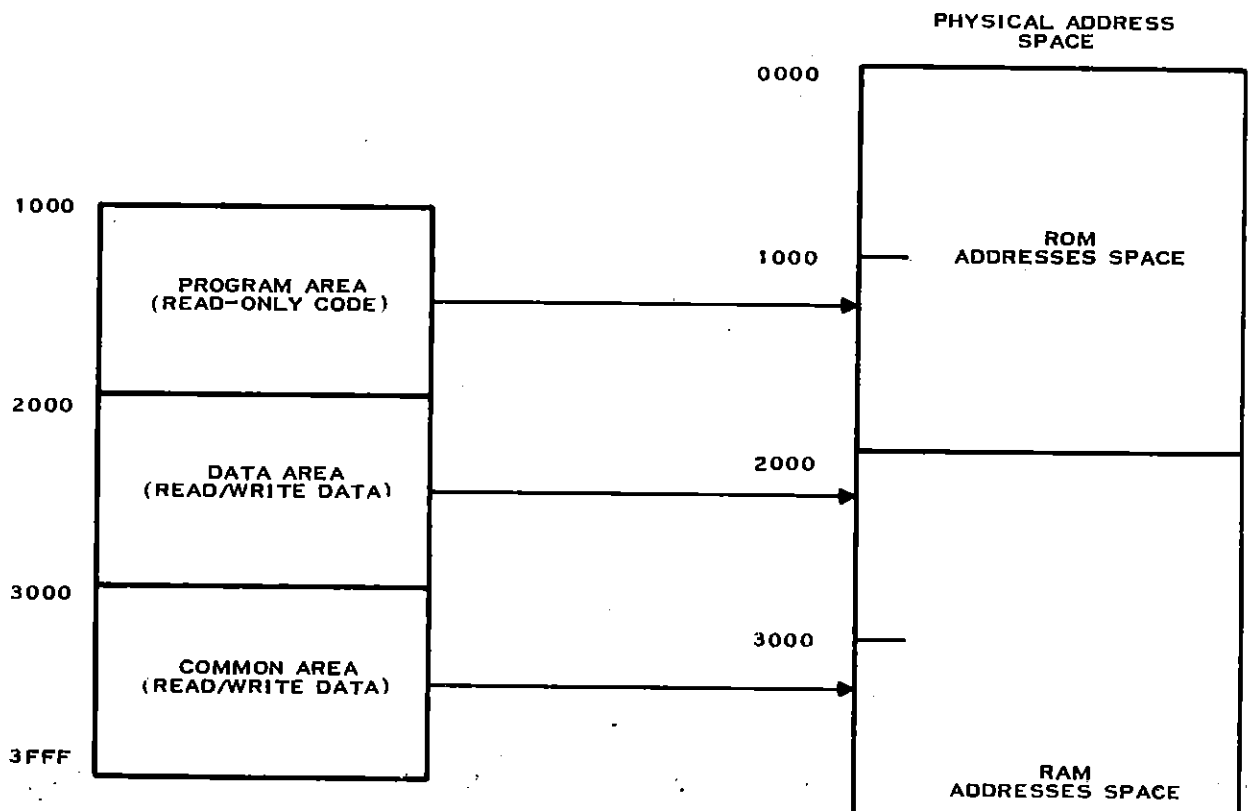
You can use this type of structure for stand-alone programs only. Since linking a program in this structure produces absolute code, the program cannot be executed under control of the operating system. Consequently, the program must handle for itself those services normally performed by the operating system. Such services include interrupt handling and device service routines (DSRs) to interface to peripherals.

To link a program with absolute memory partitions, the source code must include assembler directives to define the program, data, and common segments of each module. The PSEG directive defines program segments, which generally contain instructions and nonvariable data (read-only code). The DSEG directive defines data segments, which generally contain variable data (read/write data). The CSEG directive defines common segments, which contain read/write data that more than one module can share. If these directives are not used, the entire module is defined as a program segment. The Pascal, COBOL, and FORTRAN compilers automatically define these segments.

The Link Editor rearranges the segments from each module into three areas in the linked output. The first area contains all the program segments from each module, the second contains the data segments, and the third contains the common segments. Only one copy of a given common segment is included in the linked output. The Link Editor can then position the three areas to prescribed boundaries corresponding to absolute locations in each memory partition. Thus, the read-only code is in one area and aligned on a boundary corresponding to memory locations in ROM, and the read/write data is aligned on boundaries corresponding to memory locations in RAM.

Figure 1-4 illustrates the structure of an example program and shows how the program will eventually be loaded into a combination of ROM/RAM. The example program is divided into three areas. The program area contains read-only code from the assembler-defined program segments. This code will eventually be programmed into a ROM device; thus, it is aligned on a boundary that corresponds to ROM locations. The data and common areas both contain read/write data from the assembler-defined data and common segments. This code will eventually be loaded into RAM devices; thus, each area is aligned on a boundary that corresponds to RAM locations.

Section 7 explains how to link modules to form this type of structure.



2282940

Figure 1-4. Example of Absolute Memory Partitioning

1.4 LINK EDITOR FILES

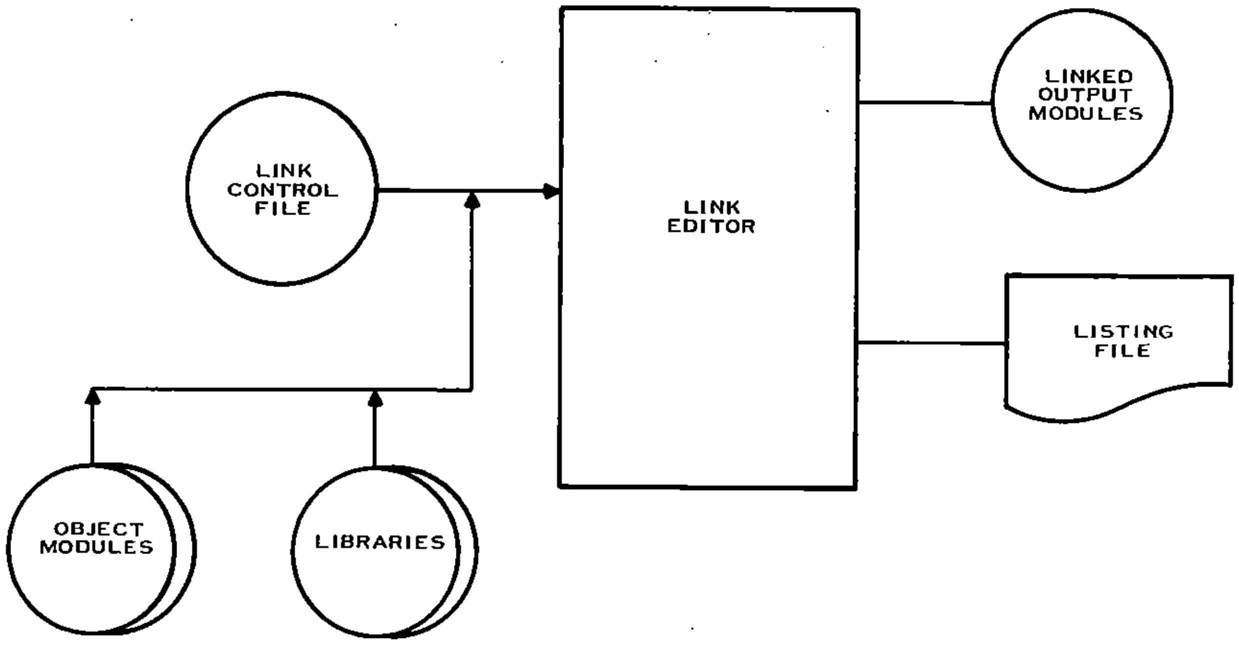
Operation of the Link Editor involves a number of different files; some are used as input and others are produced as output. Figure 1-5 shows the relationship of these files to the Link Editor.

As shown in the figure, the link control file is the primary input to the Link Editor. You must create this file using the Text Editor. The control file contains a control stream that directs the Link Editor in the linking operation. Optionally, the control file may also contain some or all of the object modules to be linked. However, the object modules are typically stored in separate files and are retrieved from the disk as required. The files or directories containing object modules may also be defined as libraries for automatic searching.

As output, the Link Editor produces one or more linked output modules (depending on the type of structure used) and a listing file. The linked output modules contain either partially linked code or completely linked code that can be installed and executed. The listing file provides a summary of the linking process and the structure produced.

When you execute the Link Editor, you must specify the pathnames of the control file and the files to which the Link Editor is to write the linked output and the listing file.

The following paragraphs further describe each of the files associated with the Link Editor.



2282941

Figure 1-5. Link Editor Files

1.4.1 Link Control File

As previously stated, you must create the link control file and build the control stream to direct the Link Editor. The *control stream* consists of a set of link control commands that specify which object modules are to be linked and define the structure of the linked output. A number of commands are provided to perform a wide variety of functions. Section 2 describes each command and its function. Subsequent sections give more details on how to build the control stream to define a particular type of structure.

1.4.2 Object Modules

The object modules are generated by the 990 assembler, a 990 compiler, or the Link Editor (partial links). They contain either standard or compressed 990 object code, which consists of ASCII tags followed by fields of data. Appendix A describes the object code format.

Object modules are included in the linking process in either of two ways. You can specifically include them by using commands in the control stream, or the Link Editor can automatically include them as the result of a search for unresolved references.

1.4.3 Libraries

You can define directories and files that contain object modules as *libraries*.The Link Editor can then use these libraries to search for modules to be included in the linking process.

At least one module must be specifically included for each segment or phase defined in the control stream. If a library that contains this module has been defined, you need to specify only the last component of the file name or the module name rather than the entire pathname. The Link Editor will search the defined libraries to obtain the specified module.

The Link Editor automatically resolves the REF and DEF tag symbols between object modules specifically included in the control stream. However, if some of the REF tags remain unresolved, the Link Editor also searches defined libraries for modules that contain corresponding DEF tags. The Link Editor automatically includes these modules in the link as they are found. If any modules included in this manner also contain unresolved REF tags, the Link Editor also includes modules to satisfy these references. The libraries are searched in the same order in which they are defined in the control stream.

By using the appropriate commands, you can define specific points in the linking process at which search operations are to occur. Otherwise, the Link Editor automatically performs the search operation at the end of the control stream.

The Link Editor supports two types of library structures: directories of object modules and sequential files of partially linked object modules.

1.4.3.1 Directory Libraries. A *directory library* is simply a directory of files, with each file containing a separate object module. If directory libraries are to be searched for unresolved REF tag symbols, the file names of modules containing DEF tag symbols must be the same as these symbols. (Alternatively, you can assign an alias to the file for each DEF tag symbol.) In this case, the Link Editor searches for a file with the same name as the symbol referenced by a module included in the link. For example, if an included module references the symbol ABC, the Link Editor searches defined directory libraries for a file named ABC. All modules in file ABC are included in the link. The reference is resolved if a DEF tag for ABC is in that file.

When searching for a specifically included module, the Link Editor simply searches directory libraries for a file with the same name as that specified in the control stream.

1.4.3.2 Sequential Libraries. A *sequential library* is a sequential file that contains one or more object modules. Because of the way the Link Editor searches sequential libraries, you must use partial links to build them. The output of several partial links can be concatenated to form one sequential library.

The Link Editor searches sequential libraries in a different manner from directory libraries. When searching for unresolved REF tag symbols, the Link Editor searches each DEF tag in each module of the library for a value corresponding to that referenced by the included module. When a reference is resolved, all modules of a previous single partial link are included in the current linking process.

When searching for a specifically included module, the Link Editor searches sequential libraries for a module with the same name as that specified in the control stream. The module name is assigned in the partial link.

1.4.4 Linked Output Modules

The Link Editor produces one linked output module for each segment or phase defined in the control stream. The output modules must be written to a data file or a program file, depending on the linking format selected. You can select the format of the output modules by using a command in the control stream. The three formats supported are as follows:

- Standard 990 object code. This is identical to that produced by the assembler. It consists of ASCII tags followed by fields of data (further described in Appendix A).
- Compressed 990 object code. This is similar to standard 990 object code except that the data fields are expressed in binary instead of ASCII. Compared to standard code, compressed code conserves disk space. It must be written to a data file that supports binary data.
- Memory Image. In this format, the output appears exactly as the program appears in memory. When you select this format, the output modules are installed directly into a program file or written to an image file. This allows you to bypass the actual installation of segments and overlays.

If you select either standard or compressed 990 object code, you must install the linked output modules yourself, using the appropriate SCI commands. This can be advantageous since the commands to install segments and overlays allow more options in installation than the Link Editor allows. Using standard or compressed object code also allows you to retain the symbol tables in the linked output for symbolic debugging.

1.4.5 Listing File

The Link Editor produces the listing file to facilitate debugging. The file includes the following:

- A listing of the control stream
- A list of parameters entered at Link Editor execution
- A link map, which lists the modules included in the link, along with their origins and lengths

Subsequent sections further explain the link map and provide examples of it.

2

Link Control Commands

2.1 INTRODUCTION

The link control commands specify which modules are to be linked and how they are to be linked. A particular set of commands forms a control stream, which must be contained in a link control file. You can build a control stream by using the Text Editor, as described in the *DNOS Text Editor Reference Manual.*

The commands you use and the order in which you use them determine the structure of the linked output. Subsequent sections of this manual explain which commands to use for a particular structure. This section discusses the general functions of the commands and then describes the commands individually. Review this section to become familiar with the general functions performed and the syntax of the commands. Once you understand how to build a control stream for a particular structure, you can refer back to this section for details on specific commands.

2.2 COMMAND FUNCTIONS

A number of commands are provided to perform a wide variety of functions. For convenience, the commands can be broken into functional groups. Table 2-1 lists the commands by group and briefly describes the function each command performs.

Basic commands are fundamental to the operation of the Link Editor. They define segments and overlays, specifically include modules in the link, select the format of the linked output, and specify the end of the control stream. Some of the commands in this group are required; every control stream (regardless of the structure desired) must contain either a TASK or PHASE 0 command, at least one INCLUDE command for each defined segment or phase, and an END command.

Symbol resolution commands aid in the symbol resolution function of the Link Editor. They define libraries and specify the points in the control stream at which search operations are to occur for symbol resolution. You can also use the NOAUTO command to inhibit automatic searching of libraries at the end of the control stream.

Partial linking commands are used only for partial links. The PARTIAL command defines the link as a partial link. The other commands in this group specify the scope of symbols externally defined within the modules to be linked. Symbols within a partial link can be defined as either global or local (not global). *Global* symbols are externally defined in the output of the partial link so that they can be referenced by other modules in subsequent links. *Local* symbols are not externally defined in the output and, thus, can only be referenced by modules included in the current partial link. To conserve space in the symbol table, you should define as local any symbols not required in subsequent links.

Output listing commands define options for the Link Editor listing file. These commands allow you to control the listing of symbols and the page ejects of the link map in the listing file.

Shared procedure commands aid in the structuring of programs that use shared procedures. These commands allow you to control positioning of assembler-defined segments and to suppress generation of linked output.

Symbol processing commands define how symbols contained in the object modules are to be handled. If the local symbol table for a module is included in the output during assembly, it can also be included in the linked output. Including symbol tables allows for symbolic debugging of the program. However, once the program is thoroughly debugged, omitting the symbol tables conserves space.

Special function commands perform miscellaneous functions, which are not necessarily related to each other.

Absolute memory partitioning commands are used only for programs that will be executed on stand-alone systems. These commands position assembler-defined segments on prescribed boundaries for eventual ROM/RAM partitioning. The commands in this group cause the Link Editor to produce absolute locations for the code, which cannot be executed under the control of the operating system.

Table 2-1. Summary of Link Control Commands

Functional Group/Command	Function
Basic Commands:	
TASK	Defines the beginning of a task segment; assigns a name to the segment.
PROCEDURE	Defines the beginning of a procedure segment; assigns a name to the segment.
SEGMENT	Defines the beginning of a program segment; assigns a name and specifies the map position for the segment.
PHASE	Defines the beginning of a new phase in an overlay structure; assigns a level and name to the phase.
INCLUDE	Explicitly defines one or more object modules to be included in a segment or phase in the linked output.
FORMAT	Selects the format of the linked output modules.
END	Signifies the end of the control stream.

Table 2-1. Summary of Link Control Commands (Continued)

Functional Group/Command	Function
Symbol Resolution Commands:	
LIBRARY	Defines a directory or a sequential file of object modules as a library.
AUTO	Specifies an automatic search for symbol resolution at the end of the control stream. (This is the default condition.)
NOAUTO	Inhibits automatic search for symbol resolution.
SEARCH	Specifies that a search of defined libraries for symbol resolution is to occur at this point in the control stream.
FIND	Synonymous with the SEARCH command. Provided for compatibility with earlier operating systems.
Partial Linking Commands:	
PARTIAL	Specifies a partial link.
NOTGLOBAL	Declares all DEF tag symbols or specified DEF tag symbols as local (not global) to the partial link.
GLOBAL	Used in conjunction with the NOTGLOBAL command to declare specified DEF tag symbols as global.
ALLGLOBAL	Declares all DEF tag symbols from included modules as global. (This is the default condition.)
Output Listing Commands:	
MAP	Allows you to control the listing of symbols in the link map.
NOMAP	Suppresses generation of the link map in the listing file.
PAGE	Specifies page ejects between the link map of phases in the listing file. (This is the default condition.)
NOPAGE	Suppresses page ejects between the link map of phases in the listing file.

Table 2-1. Summary of Link Control Commands (Continued)

Functional Group/Command	Function
Shared Procedure Commands:	
ALLOCATE	Allocates space for assembler-defined data and common segments already included in the link.
DUMMY	Suppresses generation of all or part of the linked output.
Symbol Processing Commands:	
SYMT	Causes the Link Editor to include the symbol table in the linked output. (This is the default condition when standard or compressed object code format is used.)
NOSYMT	Causes the Link Editor to omit the symbol table in the linked output.
Special Function Commands:	
ADJUST	Aligns a phase or module within a phase on a specified boundary.
SHARE	Specifies modules that are to share the same data area.
ERROR	Allows the Link Editor to continue processing when an error occurs.
NOERROR	Terminates processing of the control stream when an error occurs. (This is the default condition.)
LOAD	Causes the Overlay Manager to be included in the linked output.
NOLOAD	Causes the Overlay Manager to be omitted from the linked output. (This is the default condition.)
Absolute Memory Partitioning Commands:	
PROGRAM	Defines the starting location of the program area.
DATA	Defines the starting location of the data area.
COMMON	Defines the starting location of the common area.
ABSOLUTE	Specifies a link of absolute memory locations; specifies a special syntax definition for the PHASE command.

2.3 COMMAND DESCRIPTIONS

You can enter commands in the control stream by typing the command name, followed by its operands if any are required. When entering commands, you can type either the entire command name or just the first four characters of the command name. (This also applies to the PROGRAM operand in the PHASE, SEGMENT, and TASK commands.) You can begin the command in any column, but each command must be on a separate line (record). The command name must be followed by at least one blank and then any required operands. If you enter more than one operand, you must separate them with commas. You can enter comments in the control stream either following the command and its operand(s) or on a separate line; however, a semicolon (;) must precede each comment.

The following paragraphs describe each command individually. For ease of reference, the commands are presented in alphabetical order. Most of the command descriptions consist of a functional statement, a syntax definition, notes on command usage, and one or two examples. In some cases, the notes and examples are omitted because the command requires little explanation. The examples in this section show only partial control streams; subsequent sections show complete control streams.

The syntax definitions use the following notations:

- Items in uppercase must be entered exactly as shown except for command names, which you can enter in the four-character abbreviated form.
- Items in lowercase italics indicate a type of operand. Replace this with a specific operand of the appropriate type.
- Items in square brackets ([ ]) indicate optional operands; items not enclosed in square brackets are required.
- Items in braces ({}) indicate a choice of enclosed operands. The choices are separated by slashes (/). You can enter only one of the choices.
- An ellipsis (...) indicates that you can repeat the preceding operand as many times as necessary. You must separate the operands with commas.

You can use synonyms and/or logical names in the control stream to specify pathnames. Some of the operands can be expressed as hexadecimal numbers by preceding the number with either 0 or >.

2.3.1 ABSOLUTE Command

The ABSOLUTE command directs the Link Editor to perform a link with absolute memory locations. This command specifies a special syntax definition for the PHASE command, allowing multiple output modules to be produced in an absolute link.

Syntax Definition

```
ABSOLUTE
```

Usage

The ABSOLUTE command is primarily intended for special microprocessor applications. You can use it when separate modules (phases) are required to load object code for memory-paging hardware or to program programmable read-only memory (PROM) devices.

2.3.2 ADJUST Command

The ADJUST command adjusts the location of a phase or a module within a phase so that it is aligned on a specified boundary. Adjustment on a boundary is useful in debugging for ease of address calculation. You can also adjust modules to leave space for future patches to the program.

Syntax Definition

```
ADJUST [n]
```

The *n* operand is a decimal number less than 16 that specifies the adjustment in bytes as a power of two. For example, an operand of 4 causes the Link Editor to align the phase or module on the next 16-byte boundary. A value greater than 15 causes an error. When the operand is omitted or equal to zero, alignment is on the next word boundary.

Usage

When an ADJUST command appears immediately before a PHASE command, the next phase and all subsequent phases of the same level and with the same parent node are aligned on the specified boundary, relative to the beginning of the program.

When an ADJUST command follows a PHASE command but precedes an INCLUDE command, the next module in that phase is aligned on the specified boundary, relative to the beginning of the phase. If an ADJUST command follows a PHASE command but precedes all INCLUDE commands in the phase, the effect is the same as when the ADJUST command precedes a PHASE command.

You should adjust all phases at the same level with the same parent node so that they are aligned on the same boundary. Otherwise, phases that overlay each other could be assigned different load points and unpredictable results could occur.

Examples

```
ADJUST   5
PHASE    1, PHASEA
INCL     VOL1.OBJ.MOD2
PHASE    1, PHASEB
INCL     VOL1.OBJ.MOD3
```

This example aligns both PHASEA and PHASEB on the next 32-byte boundary relative to the beginning of the program.

```
PHASE    1, PHASEA
INCL     VOL1.OBJ.MOD2
ADJUST   6
INCL     VOL1.OBJ.MOD3
```

This example aligns only MOD3 of PHASEA on the next 64-byte boundary relative to the beginning of PHASEA.

2.3.3 ALLGLOBAL Command

The ALLGLOBAL command declares all external definitions (DEF tag symbols) included in a partial link to be global symbols. This is the default condition; thus, the ALLGLOBAL command is optional. Global symbols are externally defined in the linked output module and can be referenced by modules in subsequent links.

Syntax Definition

```
ALLGLOBAL
```

Usage

The ALLGLOBAL command can only be used in partial links. The command has the same effect as a GLOBAL command with all the DEF tag symbols as operands or with no operands. By using the NOTGLOBAL command, you can make certain symbols or all symbols exempt from global definition.

2.3.4 ALLOCATE Command

The ALLOCATE command allocates space for the assembler-defined data and common segments (DSEGs and CSEGs) from modules already included in the link (preceding the ALLOCATE command). The Link Editor places the DSEGs and CSEGs from procedure modules in the task segment. The ALLOCATE command helps ensure that the DSEGs and CSEGs for a shared procedure are placed at the same locations in each task.

Syntax Definition

```
ALLOCATE
```

Usage

The ALLOCATE command cannot be used in partial links. It must appear in the task segment (after a TASK or PHASE 0 command and before a PHASE 1 command).

Normally, the Link Editor places all the program segments (PSEGs) from included modules first, followed by all the DSEGs and then all the CSEGs. The ALLOCATE command directs the Link Editor to allocate space for these segments as if no more object modules were to be included in the link. Thus, the PSEGs, DSEGs, and CSEGs for modules included after the ALLOCATE command are placed after the last CSEG from the modules included before the ALLOCATE command.

The Link Editor no longer collects (groups) PSEGs and DSEGs for modules included after the ALLOCATE command. Any new CSEGs are still placed after all the DSEGs from modules included after the ALLOCATE command.

The ALLOCATE command works properly only when all read/write data is contained in DSEGs or CSEGs. The Pascal, COBOL, and FORTRAN compilers automatically generate code segmented in this manner.

When using the ALLOCATE command, observe the following:

- Be careful when using CSEGs with this command. CSEGs referenced before the ALLOCATE command must not have elements added to them by modules included after the ALLOCATE command.
- The procedure to be shared must not reference any symbols occurring in modules included after the ALLOCATE command.

Examples

```
PROC       PROC1
INCL       VOL1.OBJ.MOD1
TASK       TSK1
INCL       VOL1.OBJ.MOD2
INCL       VOL1.OBJ.MOD3
ALLOCATE
INCL       VOL1.OBJ.MOD4
```

In this example, the Link Editor allocates space for all PSEGs, DSEGs, and CSEGs from MOD1, MOD2, and MOD3 first. Then it allocates space for the PSEGs, DSEGs, and CSEGs from MOD4.

2.3.5 AUTO Command

The AUTO command specifies automatic searching of defined libraries for symbol resolution. This is the default condition; thus, the AUTO command is optional.

Syntax Definition

```
AUTO
```

Usage

Automatic searching occurs at the end of the control stream if any unresolved references remain (regardless of whether SEARCH and FIND commands are also used). You must define one or more libraries by using the LIBRARY command before automatic searching can take place. Refer to the description of the LIBRARY command for an explanation of how the two types of library structures are searched.

2.3.6 COMMON Command

The COMMON command defines the starting address for a common area in the linked output. This area contains specified assembler-defined common segments (CSEGs) from modules included in the link.

Syntax Definition

COMMON *base,name* [,*name* . . .,*name*]

The *base* operand specifies the starting address of a common area and can be expressed as either a decimal or hexadecimal number up to five digits long. Alternatively, you can specify the name of a CSEG that was specified in a previous COMMON command. This signifies the continuation of a previously defined common area.

Each *name* operand specifies the name of a CSEG. The CSEGs are placed in the order specified in the command.

Usage

The COMMON command can only be used for programs with absolute memory partitions; it cannot be used in partial links. In addition, the COMMON command is valid only when used in conjunction with the PROGRAM command and is ignored if used alone.

You must specifically identify within the command any CSEGs that are to be loaded at the specified address. Otherwise, they are placed after the assembler-defined data segment (DSEG) from the last included module. You can use more than one COMMON command in the control stream, and you can perform a continuation by repeating the command using a previously named CSEG instead of a starting location.

Examples

COMMON	>1000,COMA	Places CSEG COMA at location >1000.
COMMON	>1000,COMA,COMB	Places CSEG COMA at location >1000, followed by CSEG COMB.
COMMON	COMB,COMC	Continues the common area defined in the previous example by placing CSEG COMC after CSEG COMB.

2.3.7 DATA Command

The DATA command defines the starting address for a data area in the linked output. Each data area contains the assembler-defined data segments (DSEGs) from modules included after one DATA command but before a subsequent DATA command.

Syntax Definition

DATA *base*

The *base* operand is the starting address of the data area and can be expressed as either a decimal or hexadecimal number up to five digits long.

Usage

The DATA command can only be used for programs with absolute memory partitions; it cannot be used in partial links.

The DATA command can appear more than once in the control stream. The first DATA command should appear before the first INCLUDE command in the control stream; otherwise, unpredictable results can occur. If you omit the DATA command, the starting address for each data area defaults to the end of the corresponding program area.

Examples

DATA	01000	Begins data area at location >1000.
DATA	4096	Same as preceding example.

2.3.8 DUMMY Command

The DUMMY command causes the Link Editor to suppress the linked output for the segment in which it appears. If the DUMMY command precedes the first PROCEDURE, TASK, or PHASE command in the control stream, the Link Editor does not produce any linked output. The DUMMY command is useful when linking shared procedure segments or when only a link map is required.

Syntax Definition

```
DUMMY
```

Usage

The DUMMY command cannot be used in partial links.

You can use the DUMMY command to suppress the linked output of a shared procedure in the current link. When using image format, you cannot dummy a procedure segment that has not been previously installed on either the specified program file or the .S$SHARED program file.

You must dummy individual segments in the order defined in the control stream. For example, when linking two procedures, the second procedure can be dummied only if the first procedure is dummied. Overlays cannot be dummied.

No warning messages for unresolved references are generated for a dummied segment. You should compare the origin and length (from the link map) of any dummied procedures with the origin and length of the actual installed version of the procedure.

Examples

```
DUMMY
PROC     PROC1
INCL     VOL1.OBJ.MOD1
TASK     TSK1
INCL     VOL1.OBJ.MOD2
```

This example uses the DUMMY command to suppress generation of the linked output for all segments.

```
FORM     IMAGE
PROC     PROC1
DUMMY
INCL     VOL1.OBJ.MOD1
TASK     TSK2
INCL     VOl1.OBJ.MOD2
```

This example uses the DUMMY command to link a previously installed procedure (PROC1) to a new task (TSK2). No linked output is generated for PROC1 but all references are resolved.

2.3.9 END Command

The END command signifies the end of the control stream.

Syntax Definition

```
END
```

Usage

The END command must be the last command in every control stream.

2.3.10 ERROR Command

The ERROR command allows the Link Editor to continue processing the link control commands when an error occurs. You can use this command to identify all the errors in the control stream at one time. When errors occur, you must correct the errors and relink the program before you can install or execute it.

Syntax Definition

```
ERROR
```

Usage

When the ERROR command is used and an error is encountered, the Link Editor attempts to recover from the error and to complete the link operation by not processing the line in which the error occurs. Error messages are generated for all errors encountered. If the Link Editor is unable to process an INCLUDE command, processing always terminates.

The ERROR command should be the first command in the control stream so that it is processed before any errors occur.

2.3.11 FIND Command

The FIND command directs the Link Editor to search defined libraries for unresolved references at a particular point in the control stream. The search operation occurs at the point in the control stream where the FIND command appears rather than at the end of the control stream.

Syntax Definition

FIND [*name*. . ., *name*]

The *name* operands are the pathnames of the libraries that are to be searched for unresolved references. The order of operands specified determines the order in which libraries are to be searched. If you do not specify any operands, the order in which you define the libraries, using the LIBRARY command, determines the order of the search.

Usage

The FIND command functions the same as the SEARCH command and is listed as a SEARCH command in the listing file. It is provided for compatibility with earlier operating systems.

Refer to the description of the LIBRARY command for an explanation of how the two types of library structures are searched.

2.3.12 FORMAT Command

The FORMAT command defines the format of the linked output. You can select one of three format options.

Syntax Definition

```
FORMAT {ASCII/COMPRESSED/IMAGE[,REPLACE][, priority]}
```

The ASCII operand specifies standard 990 object code. Linked output in this format must be written to a data file and installed later in a program file. This is the default format selected if you do not use the FORMAT command or its operand.

The COMPRESSED operand specifies compressed 990 object code. Linked output in this format must be written to a data file that supports binary data. The linked output can subsequently be installed in a program file. This format conserves space compared to standard 990 object code.

The IMAGE operand specifies memory image format. Linked output in this format must be written directly to a program file or an image file.

When you select the IMAGE option, you can also enter the REPLACE and *priority* operands. REPLACE causes new segments and overlays to replace any existing ones of the same names in the specified program file. The *priority* operand defines the priority (0, 1, 2, 3, or 4) at which the task is to execute. The default for the priority is 4.

Usage

You can use the IMAGE option to install segments and overlays in a program file. You can also use it to write the linked output to an image file. You cannot use the IMAGE option to install privileged, system, or memory-resident tasks in a program file. Once a task is installed, you can modify it to be a privileged, system, or memory-resident task through use of the Modify Task Segment Entry (MTE) command. (Refer to the *DNOS System Command Interpreter (SCI) Reference Manual* for a description of this command.)

If you do not select the IMAGE option, you must write the linked output to a data file and then install the output module(s) in a program file, using the appropriate SCI command(s). The SCI commands allow more options in the installation process than the Link Editor allows.

Examples

FORMAT COMPRESSED	Produces compressed 990 object code in the linked output.
FORMAT IMAGE,REPLACE,3	Produces memory image code, replacing all segments and overlays defined in this link with those of the same name in the specified program file. It also assigns the task a priority of 3.

2.3.13 GLOBAL Command

The GLOBAL command identifies symbols that are to be externally defined (made global) in the output of a partial link. Externally defined symbols can be referenced by modules in subsequent links.

Syntax Definition

GLOBAL [*symbol*. . ., *symbol*]

Each *symbol* operand specifies a symbol that is to be processed as a global symbol. The command may include several operands, limited only by the maximum size of the record. If you do not specify any operands, the command functions as an ALLGLOBAL command.

Usage

The GLOBAL command can only be used in partial links. Normally, you should use it in conjunction with the NOTGLOBAL command to exempt certain symbols from the not global definition.

Examples

```
NOTGLOBAL
GLOBAL       VALA,VALB,VALC
```

This example first declares all symbols in the link as local (not global). Then, the GLOBAL command declares symbols VALA, VALB, and VALC as global. Thus, modules in subsequent links can reference symbols VALA, VALB, and VALC. No other symbols are externally defined in the linked output.

2.3.14 INCLUDE Command

The INCLUDE command explicitly defines object modules that are to be included in the link.

Syntax Definition

INCLUDE [*name*. . ., *name*]

Each *name* operand specifies an object module to be included in the link. If an object module is not in a defined library, the *name* operand must be the complete pathname of the file containing the object module. If the object module is in a defined library, the *name* operand can be just the file name (directory library) or module name (sequential library). In this last case, the *name* operand must be enclosed in parentheses; the Link Editor searches the defined libraries for the correct file or module (see LIBRARY command).

If you do not specify a *name* operand, the object modules to be included must be in the control file, immediately following the INCLUDE command. In this case, each object module must be terminated with a record that has a colon (:) in the first character position. The last module must be followed by an end-of-record marker (/*).

Usage

At least one INCLUDE command must follow each PROCEDURE, TASK, SEGMENT, and PHASE command in the control stream. The INCLUDE command defines the object module(s) for that segment or phase.

If a procedure segment is being linked to a task and the object modules included in the procedure segment contain assembler-defined data or common segments (DSEGs and CSEGs), the INCLUDE command is not required after the TASK command. In this case, the Link Editor includes the DSEGs and CSEGs from the procedure modules in the task segment of the linked output.

Examples

PROC INCL	PROC1 VOL1.OBJ.MOD1	This example defines a procedure segment named PROC1. The module(s) included in PROC1 is obtained from the file MOD1 in directory VOL1.OBJ.
LIBR TASK INCL	VOL1.OBJ TSK1 (MOD2)	This example defines a task segment named TSK1. The module(s) included in TSK1 is obtained by searching a directory library (VOL1.OBJ) for a file named MOD2 or a sequential library for a module named MOD2.

2.3.15 LIBRARY Command

The LIBRARY command defines directories or sequential files of object modules as libraries. Directory libraries can consist of any level of directory with files containing object modules produced by the assembler, a compiler, or the Link Editor (from partial links). Sequential libraries must consist of partially linked output modules. You can concatenate the output of several partial links to form one sequential library.

Once defined, a library can be used for automatic searching of unresolved references or in conjunction with the INCLUDE command. (See INCLUDE command.)

Syntax Definition

LIBRARY *name*[, *name*. . ., *name*]

Each *name* operand is the pathname of a directory or file to be defined as a library.

Usage

When searching libraries for unresolved references, the Link Editor searches the two types of library structures in different ways:

- For directory libraries, the Link Editor searches for a file with the same name as that referenced by an included module.
- For sequential libraries, the Link Editor searches each DEF tag in each module of the file for a value corresponding to that referenced by an included module.

Certain constraints apply to the use of sequential libraries. Only one pass is made through a sequential library to resolve references during a single search operation. Therefore, if one module in the library references a symbol in a previous module in the library, that reference is not resolved unless the referenced module has already been included in the link. In this case, you must use the SEARCH command to force another search operation to resolve the reference. (See SEARCH command.)

Examples

LIBR	VOL1.APPL.SEQLIB	Defines file SEQLIB as a sequential library.
LIBR	VOL1.AOBJ,VOL1.BOBJ	Defines directories AOBJ and BOBJ as directory libraries.

2.3.16 LOAD Command

The LOAD command causes the Link Editor to include the Overlay Manager in the linked output. The Overlay Manager performs automatic overlay loading during execution of a program with overlays. You can use the Overlay Manager for loading overlays only in the task segment, not in a program segment. When the Overlay Manager is not included in the link, the program must issue supervisor calls (SVCs) to load the overlays.

Syntax Definition

```
LOAD
```

Usage

The LOAD command is valid only when you select memory image format. You cannot use it in partial links.

Use of the Overlay Manager requires certain considerations in the structure and coding of overlays. (Refer to Section 6.)

2.3.17 MAP Command

The MAP command allows you to control the listing of symbols in the link map. You can specify that only referenced symbols be listed, or that only symbols that do not begin with a specified character string be listed. Using the MAP command, you can suppress the listing of external symbols in run-time library subroutines.

Syntax Definition

MAP {REFS/NO'*string*'[,NO'*string*'. . .,NO'*string*']}

The REFS operand specifies that only referenced symbols are to be listed.

In the NO'*string*' operands, *string* specifies the beginning character string of the symbols that are not to be listed.

Examples

MAP REFS	Lists only referenced symbols.
MAP NO'S$',NO'CXS'	Does not list symbols that begin with S$ or CXS.

2.3.18 NOAUTO Command

The NOAUTO command inhibits all automatic searching of defined libraries for symbol resolution. You should use the NOAUTO command when automatic searching is not needed. This saves time in executing the Link Editor.

Using the NOAUTO command also allows you to explicitly control library searching for unresolved references through use of SEARCH and FIND commands. (See SEARCH command.)

Syntax Definition

```
NOAUTO
```

2.3.19 NOERROR Command

The NOERROR command causes the Link Editor to terminate processing of the control stream when an error occurs. This is the default condition; thus, the NOERROR command is optional. An error message is generated for the first error encountered.

Syntax Definition

```
NOERROR
```

2.3.20 NOLOAD Command

The NOLOAD command inhibits the inclusion of the Overlay Manager in the linked output. This is the default condition; thus, the NOLOAD command is optional. When the Overlay Manager is not included in the linked output, the program must issue SVCs to load overlays.

Syntax Definition

```
NOLOAD
```

2.3.21 NOMAP Command

The NOMAP command causes the Link Editor to suppress generation of part of the listing file. You can use it when a program is completely debugged and the information in the link map is not needed.

Syntax Definition

```
NOMAP
```

Usage

When you use the NOMAP command, the Link Editor still writes the following information to the listing file:

- Length of the segments and phases
- Unresolved references
- Release number of the Link Editor
- Number of output records if memory image format is selected and the linked output is written to a program file

2.3.22 NOPAGE Command

The NOPAGE command specifies that page ejects do not separate the link maps for each defined segment or phase. You can use the NOPAGE command to save paper on hard copies of the listing file.

Syntax Definition

```
NOPAGE
```

2.3.23 NOSYMT Command

The NOSYMT command causes the Link Editor to omit the symbol tables from the included modules in the linked output. Use of the NOSYMT command provides for more compact object code but does not allow symbolic debugging.

Syntax Definition

```
NOSYMT
```

Usage

The NOSYMT command can appear anywhere in the control stream except when overlays are used. In this case, the NOSYMT command must appear in the root phase (following the TASK, PHASE 0, or SEGMENT command but before a PHASE 1 command).

2.3.24 NOTGLOBAL Command

The NOTGLOBAL command declares that either all externally defined (DEF tag) symbols or just specified DEF tag symbols are to be processed as local (not global) symbols in a partial link. Local symbols are not externally defined in the partially linked output; they can be referenced only by modules included in the current partial link. You can use the NOTGLOBAL command to reduce the symbol table size for subsequent links.

Syntax Definition

NOTGLOBAL [*symbol. . .,symbol*]

Each *symbol* operand identifies a symbol that is to be processed as a local symbol. The command can include several operands, limited only by the maximum size of the record. If you do not specify any operands, all symbols are processed as local.

Usage

The NOTGLOBAL command can be used only in partial links. The control stream can have more than one NOTGLOBAL command.

When you use the NOTGLOBAL command to declare all symbols as local, you can then use the GLOBAL command to exempt certain symbols from the local definition.

Examples

NOTGLOBAL	Processes all externally defined symbols as not global (except those specified in a GLOBAL command).
NOTG ABC,EFG	Processes symbols ABC and EFG as not global. All other externally defined symbols are processed as global symbols.

2.3.25 PAGE Command

The PAGE command causes page ejects to separate the beginnings of the link maps for each phase. This is the default condition; thus, the command is optional.

Syntax Definition

```
PAGE
```

2.3.26 PARTIAL Command

The PARTIAL command directs the Link Editor to perform a partial link. Partial linking allows you to link a set of modules so that the entire set can be used in subsequent links. You can also use partial links to build sequential libraries or reduce the symbol table size for subsequent links.

Syntax Definition

```
PARTIAL
```

Usage

The output of a partial link is not executable and must be linked again without the PARTIAL command before the program can be loaded and executed.

The PARTIAL command causes the Link Editor to do the following:

- Resolve all external references (REF tags) externally defined (matching DEF tags) by modules included in the partial link
- Retain all external definitions (DEF tags) in the partial link as an external definition in the partially linked output, subject to GLOBAL, NOTGLOBAL, and ALLGLOBAL commands

- Collect all the data segments (DSEGs) from the included modules and retain the common segment (CSEG) tags
- Output one data area, which is the total of all input DSEGs, subject to the SHARE command
- Resolve all SHARE references

Partial linking is allowed for a single task segment only. If partial linking of procedure segments, program segments, or overlays is required, each segment or phase must be linked separately and defined as a task segment in the partial link. You can include the output of the partial link in any segment or phase in subsequent links. The PARTIAL command must appear before the TASK or PHASE 0 command in the control stream.

2.3.27 PHASE Command

The PHASE command defines the beginning of a new phase in an overlay structure and assigns a level number and name to the phase. Optionally, the PHASE command can also assign a specific load point to the phase.

When used in conjunction with the ABSOLUTE command, the PHASE command requires a special syntax definition. This is given following the normal syntax definition.

Syntax Definition

PHASE *level*, *name* [,PROGRAM *base*][,ID *n*]

The *level* operand defines the level of the phase. The PHASE 0 command defines the root (memory-resident) phase. The level numbers for subsequent PHASE commands must increase in unit steps within a path, with alternate phases that load at the same point having the same level number.

The *name* operand specifies the name of the phase. This name can consist of from one to eight alphanumeric characters, the first of which must be alphabetic.

The *base* operand following PROGRAM specifies the load point of the phase. If you omit this operand, the Link Editor assigns a load point to the phase. The assigned load point is at the next word past the end of the preceding phase in the same path. If the phase is defined at level 0, the assigned load point is at the next 32-byte boundary past the preceding segment. You should use the same base for all phases at the same level and with the same parent node.

The *n* operand following ID assigns an ID to the phase. This operand is used only for overlays and with image format; otherwise, it is ignored. If you omit this operand, the Link Editor assigns an available ID from the specified program file.

Syntax Definition with ABSOLUTE Command

PHASE *level*,*name*,PROGRAM *base*[,DATA *base*]

When used with the ABSOLUTE command, the *name*, *level*, and PROGRAM *base* operands are the same as in the normal syntax definition; however, the PROGRAM *base* operand is required. The DATA *base* operand specifies the starting address (load point) of the data area for the phase. If you omit this operand, the Link Editor assigns a load point to the data area. The assigned load point is at the next word past the end of the program area for that phase.

Usage

You must follow each PHASE command with an INCLUDE command to define the module(s) in that phase. You must also define a complete path before defining another path. For example, if you are going to define two phases (PHASA and PHASB) at level 1, you must define all the phases that include PHASA in their path before you define PHASB in the control stream.

PHASE 0 and TASK commands are semantically identical; thus, one and only one of these two commands must appear in every control stream.

Examples

```
PHASE    0,MAIN
INCL     VOL1.OBJ.ROOT
PHASE    1,PHASA
INCL     VOL1.OBJ.MODA
PHASE    2,PHASA1
INCL     VOL1.OBJ.MODA1
PHASE    2,PHASEA2
INCL     VOl1.OBJ.MODA2
PHASE    1,PHASB
INCL     VOL1.OBJ.MODB
```

This example shows an overlay structure that includes a root phase (MAIN) and four overlays. PHASA and PHASB are at the same level (1) and will be assigned the same load point. PHASA also calls two overlays at level 2 (PHASA1 and PHASA2).

2.3.28 PROCEDURE Command

The PROCEDURE command defines the beginning of a procedure segment and assigns a name to the segment.

Syntax Definition

PROCEDURE *name*

The *name* operand specifies the name of the procedure.

Usage

Each PROCEDURE command must be followed by an INCLUDE command to define the module(s) in that procedure segment. No more than two PROCEDURE commands can appear in the control stream, and they must precede the TASK or PHASE 0 command.

Procedure segments usually contain only executable code. Reentrant procedures can be shared among several tasks. When linking a previously installed procedure to a task, you can use the DUMMY command to suppress generation of linked output for the procedure segment. (See DUMMY command.)

Examples

PROCEDURE FORLIB INCL VOL1.OBJ.MODS		Defines procedure FORLIB.
PROC RUNLIB INCL VOL1.OBJ.MODS		Defines procedure RUNLIB.

2.3.29 PROGRAM Command

The PROGRAM command defines the starting address for a program area in the linked output. Each program area contains the assembler-defined program segments (PSEGs) from modules included after one PROGRAM command but before a subsequent PROGRAM command.

Syntax Definition

PROGRAM *base*

The *base* operand is the starting address of the program area and can be expressed as a decimal or hexadecimal number up to five digits long.

Usage

The PROGRAM command can only be used for programs with absolute memory partitions; it cannot be used in partial links.

The PROGRAM command may appear more than once in the control stream. The first PROGRAM command should appear before the first INCLUDE command in the control stream; otherwise, unpredictable results can occur.

Use of the PROGRAM command without the DATA and COMMON commands causes a linked output that is to be loaded at the specified address (*base*).

Examples

PROGRAM	01F00	Begins program area at location >1F00.
PROG	7936	Same as preceding example.

2.3.30 SEARCH Command

The SEARCH command directs the Link Editor to search defined libraries for unresolved references at a particular point in the control stream. The search operation occurs at the point in the control stream where the SEARCH command appears rather than at the end of the control stream. Thus, using the SEARCH command gives you more control of search operations.

Syntax Definition

```
SEARCH  [name...,name]
```

The *name* operands are the pathnames of the libraries that are to be searched for unresolved references. The order of operands specified determines the order of the search. If you do not specify any operands, the order in which libraries are defined by use of the LIBRARY command determines the order of the search.

Usage

The SEARCH command can appear anywhere in the control stream and can appear more than once; however, it does not work within a procedure segment. The SEARCH command applies only to the segment or phase in which it appears; that is, the Link Editor searches only for unresolved references from modules included between the SEARCH command and the immediately preceding TASK, SEGMENT, or PHASE command. The Link Editor also searches for unresolved references caused by modules brought in by a search operation.

When modules are included as the result of a search operation, they are included at the point at which the search operation occurs. Use of the SEARCH command allows you to control the order in which these modules are included. For example, if a program requires a particular module to be the last module in the linked output, a SEARCH command can be placed just before the last INCLUDE command that defines this module. Automatic searching still occurs at the end of the control stream if unresolved references remain and you do not use the NOAUTO command.

You can also use the SEARCH command to obtain the correct module for multiply-defined external symbols. For example, if several libraries have been defined and each contains a module that defines a particular symbol, you can specify which library is to be searched for the definition you want. Alternatively, you can specify a search of libraries prior to explicitly including a module that also contains a definition for a particular symbol.

Refer to the description of the LIBRARY command for an explanation of how the two types of library structures are searched.

Examples

SEARCH	VOL1.OBJ	Searches library VOL1.OBJ for unresolved references.
SEARCH		Searches defined libraries for unresolved references.

2.3.31 SEGMENT Command

The SEGMENT command defines the beginning of a program segment in the linked output. The task segment must issue the appropriate SVCs to load the program segments.

Syntax Definition

SEGMENT *map,name*[,PROGRAM *base*][,ID *n*]

The *map* operand specifies the map position of the program segment. This must always be the last map position of the program (either 2 or 3). Thus, if the program does not include a procedure segment, the map position for the program segment(s) must be 2 (the task segment occupies map postion 1). If the program includes a procedure segment, the map position for the program segment(s) must be 3 (the procedure and task segments occupy map positions 1 and 2, respectively).

The *name* operand specifies a name for the segment. The name can consist of from one to eight alphanumeric characters, the first of which must be alphabetic.

The *base* operand following PROGRAM specifies the load point of the segment. If you omit this operand, the Link Editor assigns a load point, which is at the next 32-byte boundary past the end of the task segment. If you use this operand for one program segment, you should use it with the same base for all program segments defined. Otherwise, the Link Editor may assign different load points for each program segment and unpredictable results can occur.

The *n* operand following ID assigns an ID to the segment. This operand is used only with image format; otherwise, it is ignored. If you omit this operand, the Link Editor assigns an available ID from the specified program file.

Usage

The SEGMENT command must be followed by an INCLUDE command to define the modules in the program segment. The SEGMENT command cannot be used if two procedure segments are defined in the control stream.

Any number of SEGMENT commands can appear in the control file. However, they must all appear after the TASK or PHASE 0 command, and they must all be defined at the same map position.

Examples

```
SEGMENT   2,SEG1
INCL      VOL1.OBJ.MOD1
SEGMENT   2,SEG2
INCL      VOL1.OBJ.MOD2
```

This example defines two program segments at map position 2.

2.3.32 SHARE Command

The SHARE command specifies modules that are to share the same data area. This data area contains the assembler-defined data segments (DSEGs) in the modules. In some cases, you can use a shared data area to save space.

Syntax Definition

SHARE *name,name*[,*name. . .,name*]

The *name* operands specify the IDs of the modules that are to share the same data area. The module IDs are defined by the IDT assembler directive.

Usage

The size of the DSEG from the first module specified defines the maximum size of the shared data area.

Several SHARE commands can appear in the control stream. You can continue SHARE commands by repeating any module name specified in a previous SHARE command. The SHARE command applies only within a phase, and it cannot cross a phase boundary.

Examples

SHARE	MOD1,MOD2,MOD3	Specifies that modules MOD1, MOD2, and MOD3 will share the same data area.
SHARE	MOD3,MOD4	Continues from the preceding example.

2.3.33 SYMT Command

The SYMT command causes the Link Editor to include the symbol tables for the object modules in the linked output to allow for symbolic debugging. This is the default condition when image format is not used; thus, the command is optional. Object modules contain local symbol tables only if you selected the SYMT assembler option during assembly.

Syntax Definition

SYMT

Usage

The SYMT command cannot be used if memory image format is selected. If you want the symbol tables, you must specify standard or compressed object format and install the linked output modules using the appropriate SCI commands.

The object modules generated by the assembler may include symbol tables consisting of G, H, and J tag-character fields. The Link Editor includes these tags in the linked output unless you use the NOSYMT command in the control stream. To identify the module in which the symbol occurred, the Link Editor inserts an I tag followed by a four-character hexadecimal field and an eight-character ASCII field. These fields contain the program relocatable address and the module name, respectively. (Refer to Appendix A for further explanation.)

2.3.34 TASK Command

The TASK command defines the beginning of a task segment in the linked output and assigns a name to the task. The first module included in the task segment must contain the entry vector for the program.

Syntax Definition

```
TASK [name][,PROGRAM base]
```

The *name* operand is the name of the task segment. The name can consist of from one to eight alphanumeric characters, the first of which must be alphabetic. If you omit this operand, the IDT name of the first included module is used as the task name.

The *base* operand following PROGRAM specifies the load point for the task segment. If you omit this operand, the Link Editor assigns a load point, which is at the next 32-byte boundary past the end of the preceding procedure segment (if used).

Usage

Normally, the TASK command must be followed by an INCLUDE command to define the modules in the task segment. The INCLUDE command is not required if the task is linked to a procedure segment containing assembler-defined data segments (DSEGs) or common segments (CSEGs). In this case, the Link Editor includes the DSEGs and CSEGs from the procedure modules in the task segment. However, the DSEG from the first included module must contain the entry vector for the program.

The TASK command must follow all PROCEDURE commands and precede all PHASE and SEGMENT commands in the control stream.

The TASK command is semantically identical to the PHASE 0 command and is listed as a PHASE 0 command in the link map. One and only one of these two commands must appear in every control stream.

Examples

TASK FORPRG INCL VOL1.OBJ.MODS	Defines a task segment named FORPRG.
TASK INCL VOL1.OBJ.MODS	Defines the task segment and assigns the IDT name of the first included module to the task segment.

Linking Single Task Segments

3.1 INTRODUCTION

In a single task structure, the program consists of a single segment, defined as a task segment. You can structure a program as a single task if it does not require more than 64K bytes of memory and does not share any portions of code with other programs.

Linking single task structures is relatively simple. The basic steps involved are as follows:

1. Build the control stream.
2. Execute the Link Editor.
3. Read the link map and examine it for errors.

These steps are the same for linking all types of structures. However, the control streams and link maps become more complex as the structure increases in complexity.

This section covers the basic linking process using a single task structure as an example. First, it explains the single task structure and provides an example program to be linked. Then, it covers each step in the linking process, using the same example throughout the section. This section assumes you are familiar with the basic functions and syntax of the link control commands, as described in Section 2.

3.2 SINGLE TASK STRUCTURE

A single task structure consists of one or more object modules linked together to form a task segment, which contains the entire program. The Link Editor produces one linked output module for the program. The output module must be installed in a program file as a task segment. When the program is bid, the system loads the task segment into a single, continuous block of memory.

You can link any number of object modules together to form the task segment as long as the total length of the modules does not exceed 64K bytes. As an example, this section uses three separate object modules to form a single task. Figure 3-1 through Figure 3-3 contain assembly listings of the modules, named MODA, MODB, and MODC (from the IDT directives in each module). The example assumes that the object modules are on a disk named VOL1 under a user directory named OBJ.TASK. The file names are MODA, MODB, and MODC.

Module MODA contains the entry vector for the program. The entry vector defines the locations of the initial workspace, the first instruction to be executed, and the end-action routine for the program.

Module MODA also references external symbols MODB and MODC. Thus, the symbols MODB and MODC are specified in a REF directive in module MODA. Modules MODB and MODC contain the definitions for symbols MODB and MODC, respectively. Thus, the symbols MODB and MODC are specified in DEF directives in the respective modules. The REF and DEF directives make the external symbols (both references and definitions) available to the Link Editor for symbol resolution.

Each of the three modules also contains PSEG, DSEG, and CSEG directives. The PSEG directive defines a program segment in the object module, the DSEG directive defines a data segment, and the CSEG directive defines a common segment. If these directives are not used, the assembler defines the entire object module as a program segment. The COBOL, FORTRAN, and Pascal compilers also produce object code that defines these segments. These segments are referred to as *assembler-defined segments* in this manual to distinguish them from segments produced by the Link Editor.

During linking, the Link Editor automatically reorganizes the code so that each type of assembler-defined segment is placed in one continuous block in the linked output. The Link Editor places the program segments (PSEGs) from all the included modules first, followed by the data segments (DSEGs) and then the common segments (CSEGs). The Link Editor produces only one copy of a given CSEG in the linked output. In this example, all three modules use the same name for the CSEG; consequently, the Link Editor produces one common area named COM1.

Although only one copy of a given CSEG is produced in the linked output, each module using the CSEG can add elements to it. In the example, module MODA defines one word for CSEG COM1. However, modules MODB and MODC define four words for CSEG COM1. Thus, CSEG COM1 will be four words long in the linked output. If the modules define different initial values for the CSEG, the Link Editor uses the values defined by the last module included in the link.

```
 0001                        IDT   'MODA'
 0002                        REF   MODB,MODC
 0003                *
 0004 0000                   DSEG
 0005 0000           WP      BSS   32
 0006                *
 0007 0000                   PSEG
 0008 0000 0000"             DATA  WP,START,ENDACT
      0002 0006'
      0004 0012'
 0009 0006 04C0      START   CLR   R0
 0010 0008 06A0              BL    @MODB
      000A 0000
 0011 000C 0580              INC   R0
 0012 000E 06A0              BL    @MODC
      0010 0000
 0013 0012 2FE0      ENDACT  XOP   @END,15
      0014 0000+
 0014                *
 0015 0000                   CSEG  'COM1'
 0016 0000 0400      END     DATA  >0400
 0017                        END
NO ERRORS,      NO WARNINGS
```

Figure 3-1. MODA Assembly Listing — Single Task Structure

```
0001                         IDT   'MODB'
0002                         DEF   MODB
0003 0000                    PSEG
0004 0000 C040     MODB      MOV   R0,R1
0005 0002 C801               MOV   R1,@END+2
     0004 0002+
0006 0006 3860               MPY   @MULT,R1
     0008 0000"
0007 000A C802               MOV   R2,@END+4
     000C 0004+
0008 000E A0A0               A     @BASE,R2
     0010 0002"
0009 0012 C802               MOV   R2,@END+6
     0014 0006+
0010 0016 045B               RT
0011 0000                    DSEG
0012 0000 0005     MULT      DATA  5
0013 0002 3000     BASE      DATA  >3000
0014 0000                    CSEG  'COM1'
0015 0000          END       BSS   8
0016                         END
NO ERRORS,         NO WARNINGS
```

Figure 3-2. MODB Assembly Listing — Single Task Structure

```
0001                         IDT   'MODC'
0002                         DEF   MODC
0003 0000                    PSEG
0004 0000 C040     MODC      MOV   R0,R1
0005 0002 C801               MOV   R1,@END+2
     0004 0002+
0006 0006 3860               MPY   @MULT,R1
     0008 0000"
0007 000A C802               MOV   R2,@END+4
     000C 0004+
0008 000E A0A0               A     @BASE,R2
     0010 0002"
0009 0012 C802               MOV   R2,@END+6
     0014 0006+
0010 0016 045B               RT
0011 0000                    DSEG
0012 0000 000A     MULT      DATA  10
0013 0002 2000     BASE      DATA  >2000
0014 0000                    CSEG  'COM1'
0015 0000          END       BSS   8
0016                         END
NO ERRORS,         NO WARNINGS
```

Figure 3-3. MODC Assembly Listing — Single Task Structure

3.3 BUILDING THE CONTROL STREAM

The control stream defines the structure of the program and directs the Link Editor in the linking process. Every control stream must contain the following commands:

- A TASK or PHASE 0 command
- At least one INCLUDE command
- An END command

These are basic commands and are the only commands that you must use in the control stream to link a single task. The following is the control stream for linking the example program:

```
TASK       PROG1
INCLUDE    VOL1.OBJ.TASK.MODA
INCLUDE    VOL1.OBJ.TASK.MODB
INCLUDE    VOL1.OBJ.TASK.MODC
END
```

The TASK command defines the beginning of a task segment and assigns a name to the task, PROG1 in this example. (The TASK command is semantically identical to the PHASE 0 command; one and only one of these commands must appear in the control stream.) The INCLUDE commands specify the object modules to be included in the link. Alternatively, you can specify all three modules in one INCLUDE command by separating the pathnames of the modules with commas. The END command terminates the control stream.

The order in which the modules are included in the link is defined by the order in which they are listed in the INCLUDE commands. The module containing the entry vector for the program must be the first module included in the task segment. Thus, in this example, MODA must be included first, but the order of the remaining two modules is not important. Also, remember that the Link Editor reorganizes the PSEGs, DSEGs, and CSEGs from each module. The following shows the placement of assembler-defined segments using the previous control stream:

```
MODA    PSEG
MODB    PSEG
MODC    PSEG
MODA    DSEG
MODB    DSEG
MODC    DSEG
        CSEG COM1 (one copy for all modules)
```

Since no other commands are in this control stream, the Link Editor assumes some default conditions. For example, the default for the format of the linked output is standard 990 object code. When standard or compressed format is used, the Link Editor also includes the symbol tables from the object modules in the linked output, allowing you to symbolically debug the program. In this case, you must write the linked output to a data file and then use the SCI Install Task (IT) command to install the linked output in a program file. If desired, you can omit the symbol tables from the linked output by placing the NOSYMT command anywhere in the control stream prior to the END command.

Once a program is completely debugged, you can write the linked output directly to a program file. To do this, use the FORMAT command with the IMAGE option as follows:

```
FORMAT     IMAGE,REPLACE
TASK       PROG1
INCLUDE    VOL1.OBJ.TASK.MODA,VOL1.OBJ.TASK.MODB,VOL1.OBJ.TASK.MODC
END
```

Using this control stream, the Link Editor resolves the references to MODA, MODB, and MODC and produces a single linked output module in image format. The linked output must be written to a program file. Since the REPLACE option is also selected in the FORMAT command, the Link Editor replaces any task segment named PROG1 currently installed in the specified program file.

In addition to the basic commands, some of the commands from other functional groups are useful in linking single tasks. The following paragraphs describe these commands.

3.3.1 Symbol Resolution Commands

The symbol resolution commands perform functions that aid in symbol resolution. These commands are LIBRARY, SEARCH, FIND, and NOAUTO.

The LIBRARY command defines a directory or sequential file containing object modules as a library. In the example, the directory VOL1.OBJ.TASK can be defined as a library. When you use a defined library in conjunction with the INCLUDE command, you have to specify only the file name (directory library) or module name (sequential library) instead of the entire pathname. You must enclose the names in parentheses, as follows:

```
FORMAT     IMAGE,REPLACE
LIBRARY    VOL1.OBJ.TASK
TASK       PROG1
INCLUDE    (MODA),(MODB),(MODC)
END
```

In this example, all the modules required by the program are specified in the INCLUDE command. The Link Editor resolves symbol references among these modules. However, if any unresolved references remain at the end of the control stream, the Link Editor also automatically searches the defined library for modules to satisfy these references. Since MODA in the example references both MODB and MODC, you do not have to specify MODB or MODC in the INCLUDE command. The Link Editor searches the directory VOL1.OBJ and includes MODB and MODC as the result of the automatic search.

In a directory library structure, the Link Editor searches the directory for a file with the same name as the referenced symbol. For this reason, the file names must match the external definitions (DEF tags). If a module contains more than one external definition, you should assign an alias to the file for each additional definition; use the SCI Add Alias to Pathname (AA) command. This ensures access by the Link Editor to all external definitions.

Sequential libraries are searched in a different manner. In this case, the Link Editor searches each module in the library for a DEF tag to match the reference. Only one pass is made through a sequential library to resolve references. Therefore, if one module references a previous module in the library, that reference is not resolved unless the referenced module has already been included in the link.

All automatic searching occurs at the end of the control stream. However, you can force a search operation at a specific point in the control stream by using the SEARCH or FIND command. (The FIND command is synonymous with SEARCH and is listed as SEARCH in the link map.) For example, if MODB references other modules from a different library that need to be included in the link before MODC, you can use the following control stream:

```
FORMAT     IMAGE,REPLACE
TASK       PROG1
LIBRARY    VOL1.OBJ.TASK
INCLUDE    (MODA),(MODB)
SEARCH     VOL1.OBJ.LIB
INCLUDE    (MODC)
END
```

You can also use the SEARCH command (or multiple SEARCH commands) to ensure that all required modules from a sequential library are included in the link.

Automatic searching still occurs at the end of the control stream unless you use the NOAUTO command. If you do not need automatic searching, as in this example, use the NOAUTO command. This conserves time in the execution of the Link Editor. You can place the NOAUTO command anywhere in the control stream prior to the END command.

3.3.2 Special Function Commands

The special function commands that are useful in linking single tasks are ERROR, SHARE, and ADJUST.

Normally, the Link Editor terminates when it encounters an error. The ERROR command allows the Link Editor to continue processing the control stream when an error occurs. You might want to use this command the first time you link a program to identify all the errors at one time. However, you must correct the errors and relink the program before you can install and execute it.

The ERROR command should appear at the beginning of the control stream so that it is processed before any errors occur. With the ERROR command in use, the Link Editor does not process a line in which an error occurs but continues to the next line. For example, if an error occurs in the FORMAT command, the Link Editor does not process that command and the linked output defaults to standard 990 object code. The Link Editor always terminates when an error occurs in an INCLUDE command.

The SHARE command allows you to specify modules that share the same data area in the linked output. You can use a shared data area in some cases to conserve memory. Normally, the Link Editor allocates a separate data area for each DSEG defined in the modules. When you use the SHARE command, the Link Editor allocates one data area for all the DSEGs from the modules specified in the SHARE command. The maximum size of this data area is defined by the DSEG from the first module included in the link.

You can use the ADJUST command to adjust the starting location of a module so that it is aligned on a specified boundary. The boundary is expressed in bytes as a power of two. Adjustment on a boundary makes address calculations easier when debugging. However, adjustment also uses more memory.

The following is an example of a control stream that uses the ERROR, SHARE, and ADJUST commands:

```
ERROR
FORMAT     IMAGE,REPLACE
LIBRARY    VOL1.OBJ.TASK
TASK       PROG1
INCLUDE    (MODA)
ADJUST     5
INCLUDE    (MODB),(MODC)
SHARE      MODB,MODC
END
```

The ADJUST command in this example aligns MODB on the next 32-byte boundary following MODA. The SHARE command causes MODB and MODC to share the same data area. The amount of space allocated for this data area is equal to the size of the DSEG defined in MODB. The Link Editor allocates a separate data area for MODA.

3.3.3 Output Listing Commands

The output listing commands useful in linking single tasks are MAP and NOMAP.

The MAP command allows you to control the listing of certain symbols in the link map. Normally, the Link Editor lists all the external definitions (DEF tags) contained in the object modules included in the link. However, some of these may not be referenced by the included modules and, therefore, are of no interest. In this case, you can use the MAP command to specify the listing of only referenced symbols, as follows:

```
FORMAT     IMAGE,REPLACE
LIBRARY    VOL1.OBJ.TASK
MAP        REFS
TASK       PROG1
INCLUDE    (MODA),(MODB),(MODC)
END
```

You can also use the MAP command to suppress the listing of symbols that begin with a specified character string. For example, the following command suppresses the listing of symbols that begin with CXS:

```
MAP   NO'CXS'
```

This is useful when you want to list all the external definitions in the object modules you specifically included but you also want to suppress those from a run-time library.

If you do not need a link map (for example, the program is completely debugged and the information in the link map is not useful), you can use the NOMAP command to suppress generation of part of the listing file. You can place the NOMAP command anywhere in the control stream prior to the END command.

3.4 EXECUTING THE LINK EDITOR

The Link Editor is activated by the Execute Link Editor (XLE) command to SCI. The following explains the prompts and appropriate responses for this command.

Prompts

```
EXECUTE LINK EDITOR
         CONTROL ACCESS NAME:  pathname@     (*)
   LINKED OUTPUT ACCESS NAME:  [pathname@]   (*)
         LISTING ACCESS NAME:  [pathname@]   (*)
        PRINT WIDTH (CHARS):   integer       (80)
                PAGE LENGTH:   integer       (59)
```

Prompt Details

CONTROL ACCESS NAME

Enter the pathname of the device or file containing the control stream. The control stream can be read from a disk file or any sequential device such as a tape unit or card reader. You can enter the control stream directly from a terminal, but it should not be the same terminal from which you entered the XLE command.

LINKED OUTPUT ACCESS NAME

Enter either DUMY, a blank line (null response), or the pathname of the file to which the Link Editor is to write the linked output. DUMY or a blank line suppresses generation of the linked output. This allows you to make a trial run to ensure that no errors occur.

If you enter a pathname and you used the IMAGE option in the FORMAT command, you must specify either a program file, an image file, or a nonexisting file. If the file specified does not exist, the Link Editor creates a program file for the linked output. This program file contains only enough room to hold the segments and overlays defined in the control stream.

If you enter a pathname and you did not use the IMAGE option in the FORMAT command, you must specify a sequential data file (except for COBOL programs) or a nonexisting file. If the file specified does not exist, the Link Editor creates a sequential file for the linked output. Before you can execute the program, you must install the linked output in a program file or image file.

LISTING ACCESS NAME
: Enter either DUMY, a blank line, or the pathname of the file or device to which the Link Editor is to write the listing file. DUMY or a blank line suppresses generation of the listing file.

PRINT WIDTH (CHARS)
: Enter an integer from 60 through 132. The integer value specifies the maximum number of characters per line that the Link Editor writes to the listing file or device.

PAGE LENGTH
: Enter any positive integer. The integer value specifies the maximum number of lines per page that the Link Editor writes to the listing file or device. The Link Editor adds three lines to the specified value for header information.

Once you have entered responses to all the prompts, the Link Editor begins execution in the background. The amount of time required for it to complete execution depends on the size and structure of the program and the system load during execution. A large program may require several hours to link. When the Link Editor completes execution, a message is displayed on the terminal screen. The message indicates if any errors occurred during linking. Refer to Section 9 for an explanation of error messages.

The maximum amount of memory available to the Link Editor during execution is 64K bytes. The Link Editor code uses 16K bytes; the rest is used as table area for the modules and external symbols in the program being linked. To determine the amount of memory required for a particular program, use the following guidelines:

allow	16K bytes for the Link Editor code
plus	16 bytes for each external reference
plus	40 bytes for each included module
plus	10% of the sum of the above

If the total of this exceeds 64K bytes, you can use partial links to reduce the size required. (Refer to Section 8.)

Example

```
EXECUTE LINK EDITOR
          CONTROL ACCESS NAME:  VOL1.CNTRL.TASK
   LINKED OUTPUT ACCESS NAME:  VOL1.PROGRAM
          LISTING ACCESS NAME:  VOL1.MAP.TASK
          PRINT WIDTH (CHARS):  80
                 PAGE LENGTH:  59
```

This example reads the control stream from a disk file called VOL1.CNTRL.TASK. It writes the linked output to a program file called VOL1.PROGRAM and the listing file to a disk file called VOL1.MAP.TASK. A maximum of 80 characters per line and 59 lines per page is written to the listing file.

3.5 READING THE LINK MAP

In addition to the linked output, the Link Editor produces a listing file, which summarizes the linking process and the structure produced. In a single task structure, the listing file consists of three pages. Figure 3-4 shows the listing file produced for the example program. The first line of each page contains a header consisting of the word LINKER followed by the release number, the Julian date of the release, the date and time the listing file was created, and the page number. (The headers for the example listings show only the date, time, and page number.)

Page 1 has an additional header entitled COMMAND LIST. This page contains a copy of the control stream used.

Page 2 also has an additional header entitled LINK MAP. This page lists the the parameters entered for the XLE command, the number of output records produced (if image format is used), the format selected for the linked output, and the libraries defined in the control stream. The libraries are listed under three columns as follows:

```
    NO       ORGANIZATION       PATHNAME
```

where:

NO indicates the order in which the library was defined. Other parts of the link map refer to this number.

ORGANIZATION indicates the type of library structure used (RANDOM or SEQUENTIAL). RANDOM refers to a directory structure.

PATHNAME specifies the pathname of that library.

Page 3 begins the actual link map, which shows the structure of the linked output. The first line of the link map is as follows:

```
    PHASE 0,      PROG1      ORIGIN = 0000      LENGTH = 0076      (TASK ID = 1)
```

This line defines the first segment or phase in the link. In this example, it is a task segment, which is listed as PHASE 0. (PHASE 0 and TASK commands are semantically identical and are always listed as PHASE 0 in the link map.) PROG1 is the name assigned to the task. The ORIGIN entry shows the starting location of the task segment, relative to the beginning of the program. Since this example consists of a single task, the starting location of the task segment is 0000. The LENGTH entry specifies the actual number of bytes of memory required to hold the segment. When image format is used, the last entry on this line shows the installed ID assigned to the segment or phase. When standard or compressed format is used, the phrase ENTRY = *xxxx* (where *xxxx* is an address) may appear at the end of the line. Read-only memory (ROM) loaders use this address to determine the entry point at which execution of the program starts when loading is complete.

The modules linked to form the segment are listed after the segment definition. These modules are listed under eight columns as follows:

```
    MODULE    NO    ORIGIN    LENGTH    TYPE    DATE    TIME    CREATOR
```

where:

MODULE lists the name of each included module. The data area for each module is listed as $DATA and follows immediately after the module in which it is defined.

NO indicates the order in which the module was included in the link. Other parts of the link map refer to this number.

ORIGIN shows the starting location of each module, relative to the beginning of the program.

LENGTH shows the length in bytes of each module.

TYPE shows the command that caused the module to be included in the link. This is either INCLUDE, SEARCH, or LIBRARY if the module was included as the result of an automatic search. If a number follows the command name, it indicates the library in which the module was found. You can map this number back to the library definitions on page 2 of the listing file.

DATE shows the date the module was originally created.

TIME shows the time the module was created.

CREATOR specifies the utility that generated the module. This is either the assembler (SDSMAC), a compiler (for example, DXPSCL for the Pascal compiler), or the Link Editor (LINKER) if the module was generated by a partial link.

If the object modules being linked contain CSEGs, the following appears after the module listings:

```
COMMON      NO      ORIGIN      LENGTH
```

where:

COMMON lists the name of each CSEG.

NO indicates the number of the last module in which the CSEG is included.

ORIGIN shows the beginning location of the CSEG, relative to the beginning of the program.

LENGTH shows the length in bytes of the CSEG.

Following the CSEG listings, the link map lists and describes all the external definitions from the modules included in the link. These are listed in columns under the header DEFINITIONS, as follows:

```
NAME   VALUE   NO   NAME   VALUE   NO   NAME   VALUE   NO
```

where:

NAME lists each symbol externally defined (DEF tag symbol) in the modules. An asterisk (*) preceding the name indicates a symbol that is externally defined but not referenced within the program.

VALUE specifies the address within the link associated with the symbol. An asterisk (*) following a value indicates that the value is absolute (not relocatable).

NO indicates the number of the module in which the symbol is defined. You can map this number back to the module listings.

If any unresolved references remain at the end of the link, the link map lists the unresolved references. These are listed in columns under the header UNRESOLVED REFERENCES, as follows:

```
NAME    COUNT    NO    NAME    COUNT    NO    NAME    COUNT    NO
```

where:

NAME lists each symbol that could not be resolved.

COUNT specifies the number of times the symbol is referenced.

NO indicates the number of the module in which the symbol is referenced. You can map this number back to the module listings.

Unresolved references cause the following warning message to appear at the end of the link map:

n   WARNINGS

where:

n is the number of unresolved references.

If errors occurred, the following appears at the end of the link map:

n   ERRORS

where:

n is the number of errors.

Error messages are also distributed throughout the listing file. Refer to Section 9 for an explanation of error messages.

The last line of the link map contains a completion message, as follows:

```
****  LINKING COMPLETED
```

```
                                                 04/15/82  15:58:00         PAGE    1
COMMAND LIST

 FORMAT     IMAGE,REPLACE
 LIBRARY    VOL1.OBJ.TASK
 TASK       PROG1
 INCLUDE    (MODA),(MODB),(MODC)
 END
                                                 04/15/82  15:58:00         PAGE    2
LINK MAP

CONTROL FILE = VOL1.CNTRL.TASK

LINKED OUTPUT FILE = VOL1.PROGRAM

LIST FILE = VOL1.MAP.TASK

NUMBER OF OUTPUT RECORDS = 1

OUTPUT FORMAT = IMAGE

LIBRARIES

NO   ORGANIZATION   PATHNAME

1    RANDOM         VOL1.OBJ.TASK
                                                 04/15/82  15:58:00         PAGE    3


PHASE 0, PROG1    ORIGIN = 0000  LENGTH = 0076   (TASK ID = 1)

MODULE     NO   ORIGIN    LENGTH     TYPE        DATE       TIME       CREATOR

MODA       1     0000      0016    INCLUDE,1   04/15/82   15:45:44    SDSMAC
$DATA      1     0046      0020
MODB       2     0016      0018    INCLUDE,1   04/15/82   15:46:50    SDSMAC
$DATA      2     0066      0004
MODC       3     002E      0018    INCLUDE,1   04/15/82   15:50:42    SDSMAC
$DATA      3     006A      0004


COMMON     NO   ORIGIN    LENGTH

COM1       3     006E      0008
                                   D E F I N I T I O N S

 NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO

 MODB      0016  2      MODC      002E  3

****  LINKING COMPLETED
```

Figure 3-4. Example Listing File — Single Task Structure

4

Linking Procedure Segments

4.1 INTRODUCTION

The use of procedure segments allows you to share code among several different programs or among several copies of the same program. This saves physical memory when programs that share code are in memory at the same time.

Linking a program with procedure segments involves the same basic steps as linking a single task segment (building the control stream, executing the Link Editor, and reading the link map, as described in Section 3). This section explains the procedure/task structure and provides an example program to be linked. It also explains how to build the control stream and read the link map for the example. Execution of the Link Editor (use of the XLE command) is the same regardless of the structure used.

4.2 PROCEDURE/TASK STRUCTURE

In this type of structure, a program consists of two or three segments: one or two procedure segments and a task segment. The procedure segment(s) must contain the code to be shared. Usually, this is only executable code and/or constant data. Procedure segments must precede the task segment. This ensures the placement of shared code in the same locations for all tasks.

As previously mentioned, procedure segments have two main uses:

- To share code among several copies of the same program
- To share code among several different programs

The Link Editor allows a maximum of two procedure segments to be linked to a task segment. Usually, one procedure segment is sufficient for either of the above uses. However, two procedure segments may be useful if a program requires both uses. In this case, the first procedure segment should be the one to be shared among several different programs. Shared procedure segments must be reentrant.

The task segment must contain the entry vector for the program. The task segment may also contain variable data and code not used by other programs. Each program must contain one and only one task segment.

You must perform a separate link operation (using a different control stream) for each program that uses a shared procedure segment. In each link operation, the Link Editor produces one linked output module for each segment defined. However, you need to retain only one copy of the linked output module for a shared procedure segment.

When linking programs in this type of structure, you can use any number of object modules to form each segment, as long as the total length of all the modules does not exceed 64K bytes. As an example, this section uses five object modules to form a program with one procedure segment and a task segment. (Since linking two procedure segments is very similar to linking one procedure segment, only one procedure is used in the example.)

Figure 4-1 through Figure 4-5 contain assembly listings of the example modules (MODA, MODB, MODC, MODX, and MODY). The examples assume that each module is contained in a separate file of the same name, under a directory named VOL1.OBJ.PROC.

Since module MODA contains the entry vector for the program, it must be the first module included in the task segment. Modules MODB and MODC are also used to form the task segment. Modules MODX and MODY are used to form the procedure segment.

Each of the modules (except MODA) contains PSEG, DSEG, and CSEG directives. As explained in Section 3, the Link Editor places all the PSEGs from the included modules first, followed by the DSEGs and then the CSEGs. Therefore, even though MODX and MODY are used to form the procedure segment, the DSEGs and CSEGs from these modules are actually placed in the task segment. The placement of assembler-defined segments is discussed further in subsequent paragraphs.

```
 0001                          IDT   'MODA'
 0002                          REF   WP, START, ENDACT
 0003 0000                     PSEG
 0004 0000 0000                DATA  WP, START, ENDACT
      0002 0000
      0004 0000
 0005                          END
NO ERRORS,      NO WARNINGS
```

Figure 4-1. MODA Assembly Listing — Procedure/Task Structure

```
 0001                        IDT   'MODB'
 0002                        DEF   WP,START,ENDACT
 0003                        REF   MODC,MODX,MODY
 0004 0000                   DSEG
 0005 0000           WP      BSS   32
 0006 0020 0400      END     DATA  >0400
 0007 0000                   PSEG
 0008 0000 D802      START   MOVB  R2,@DATA
      0002 0000+
 0009 0004 06A0              BL    @MODC
      0006 0000
 0010 0008 0203              LI    R3,1
      000A 0001
 0011 000C 06A0              BL    @MODX
      000E 0000
 0012 0010 C820              MOV   @BUF1,@BUF2
      0012 0000+
      0014 0002+
 0013 0016 06A0              BL    @MODY
      0018 0000
 0014 001A 2FE0      ENDACT  XOP   @END,15
      001C 0020"
 0015 0000                   CSEG  'COM1'
 0016 0000           BUF1    BSS   2
 0017 0002           BUF2    BSS   2
 0018 0000                   CSEG  'COM2'
 0019 0000 0000      DATA    DATA  0,0,0
      0002 0000
      0004 0000
 0020                        END
NO ERRORS,     NO WARNINGS
```

Figure 4-2. MODB Assembly Listing — Procedure/Task Structure

```
 0001                        IDT   'MODC'
 0002                        DEF   MODC
 0003 0000                   PSEG
 0004 0000 C0A0      MODC    MOV   @DATA,R2
      0002 0000+
 0005 0004 A0A0              A     @CBDA,R2
      0006 0000"
 0006 0008 C802              MOV   R2,@DATA+2
      000A 0002+
 0007 000C 045B              RT
 0008 0000                   DSEG
 0009 0000 0A00      CBDA    DATA  >0A00
 0010 0000                   CSEG  'COM2'
 0011 0000 0000      DATA    DATA  0,0,0
      0002 0000
      0004 0000
 0012                        END
NO ERRORS,     NO WARNINGS
```

Figure 4-3. MODC Assembly Listing — Procedure/Task Structure

```
0000                   IDT   'MODX'
                       DEF   MODX
0000                   PSEG
0000 0202   MODX       LI    R2, BUF1
0002 0000+
0004 0283              CI    R3, 1
0006 0001
0008 1302              JEQ   NEXT
000A 0202              LI    R2, BUF2
000C 0002+
000E C4A0   NEXT       MOV   @CONST, *R2
0010 0000"
0012 045B              RT
0000                   DSEG
0000 1111   CONST      DATA  >1111
0000                   CSEG  'COM1'
0000        BUF1       BSS   2
0002        BUF2       BSS   2
                       END
NO ERRORS,      NO WARNINGS
```

Figure 4-4. MODX Assembly Listing — Procedure/Task Structure

```
                       IDT   'MODY'
                       DEF   MODY
0000                   PSEG
0000 C120   MODY       MOV   @BUF2, R4
0002 0002+
0004 A120              A     @CONST, R4
0006 0000"
0008 045B              RT
0000                   DSEG
0000 2222   CONST      DATA  >2222
0000                   CSEG  'COM1'
0000        BUF1       BSS   2
0002        BUF2       BSS   2
                       END
NO ERRORS,      NO WARNINGS
```

Figure 4-5. MODY Assembly Listing — Procedure/Task Structure

4.3 BUILDING THE CONTROL STREAM

The control stream for this type of structure contains the same basic commands as that for a single task segment, with the addition of the PROCEDURE command. The PROCEDURE command defines the beginning of a procedure segment and assigns a name to the segment. As stated earlier, the procedure segment(s) must precede the task segment. You can also use any of the other commands that can be used in single task structures. The following is the control stream for linking the example program:

```
LIBRARY      VOL1.OBJ.PROC
PROCEDURE    PROC1
INCLUDE      (MODX),(MODY)
TASK         TSK1
INCLUDE      (MODA),(MODB),(MODC)
END
```

The PSEGs from modules MODX and MODY form the procedure segment. The DSEGs from these modules are placed in the task segment, following the PSEGs from modules MODA, MODB, and MODC. The CSEGs follow the last DSEG with one exception: CSEGs defined in a module with an IDT of $BLOCK are placed in the procedure segment if that module is included in the procedure segment. The following shows the arrangement of the assembler-defined segments in the linked output, using this control stream:

Procedure segment (PROC1):

```
MODX   PSEG
MODY   PSEG
```

Task segment (TSK1):

```
MODA   PSEG
MODB   PSEG
MODC   PSEG
MODX   DSEG
MODY   DSEG
MODB   DSEG
MODC   DSEG
       CSEG COM1 (used by MODX, MODY, and MODB)
       CSEG COM2 (used by MODB and MODC)
```

This arrangement works fine when multiple copies of the same task are to share the procedure segment. However, this arrangement can cause problems when different tasks are to share this procedure segment. Notice that the DSEGs and CSEGs from the modules included in the procedure segment follow the PSEGs from the modules included in the task segment. If the PSEGs from different tasks vary in size, the DSEGs and CSEGs from the procedure segment will be placed in different locations in each task. When this happens, the references in the procedure segment vary from task to task, and the procedure segment can no longer be shared among these different tasks.

You can prevent this problem by using the ALLOCATE command. When the Link Editor encounters the ALLOCATE command, it allocates space for the DSEGs and CSEGs already included in the link, as if no more object modules were to be included. This helps ensure that the DSEGs and CSEGs from the procedure segment are placed in the same locations for every task that uses the procedure segment.

The ALLOCATE command must appear in the task segment (after the TASK command). Since the entry vector for the program must appear first in the task segment, the ALLOCATE command must also be placed after the INCLUDE command for this module (MODA in this example). In order for this to work, you must ensure that the portion of the task segment included before the ALLOCATE command (referred to as the *preallocated portion* of the task segment) is the same size for every task that shares the procedure segment. Other modules that form the task segment can be included after the ALLOCATE command; this portion of the task segment is referred to as the *postallocated portion*. However, you must ensure that the procedure segment does not reference any symbols in the postallocated modules.

Using the ALLOCATE command in this way, you can vary the size and the content of the task segment from task to task without adversely affecting the execution of the shared procedure segment. The following control stream shows the placement of the ALLOCATE command for the example program:

```
LIBRARY     VOL1.OBJ.PROC
PROCEDURE   PROC1
INCLUDE     (MODX),(MODY)
TASK        TSK1
INCLUDE     (MODA)
ALLOCATE
INCLUDE     (MODB),(MODC)
END
```

This control stream causes the Link Editor to arrange the assembler-defined segments as follows:

Procedure segment (PROC1):

```
MODX   PSEG
MODY   PSEG
```

Task segment (TSK1):

```
MODA   PSEG
MODX   DSEG
MODY   DSEG
       CSEG COM1
MODB   PSEG
MODB   DSEG
MODC   PSEG
MODC   DSEG
       CSEG COM2
```

Notice that the Link Editor no longer groups the PSEGs and DSEGs in one block for the postallocated modules. However, any new CSEGs from these modules are still placed after the last DSEG. Also note that space for CSEG COM1 is allocated before modules MODB and MODC are included. Therefore, MODB (which uses CSEG COM1) cannot add any elements to this CSEG.

As stated earlier, you must perform a separate link operation for each task that uses a shared procedure segment, but you need to keep only one copy of the linked output for this segment. After linking the procedure with the first task, you should use the DUMMY command to suppress generation of the linked output for the procedure segment. This conserves time in the linking process for each subsequent link. Use of the DUMMY command also prevents problems that may occur when the shared procedure segment is in use while it is being linked to a new task.

The following control stream shows the use of the DUMMY command in linking the shared procedure segment with a new task:

```
LIBRARY     VOL1.OBJ.PROC
PROCEDURE   PROC1
DUMMY
INCLUDE     (MODX),(MODY)
TASK        TSK2
INCLUDE     (MODA2)
ALLOCATE
INCLUDE     (MODB2),(MODC2),(MODD2)
END
```

Using this control stream, the Link Editor produces a linked output module for the task segment but does not produce one for the procedure segment. If you are using image format and are linking two procedure segments to a task, you can dummy the second procedure segment only if you dummy the first.

As in single task links, you can use the FORMAT command with the IMAGE option to install the linked output directly into a program file. If you do not use this option, you must install the segments in a program file before you can execute the program. Procedure segments must be installed either in the same program file as the task segment or in a special system file called .S$SHARED.

Once you have built the control stream, you can execute the Link Editor using the Execute Link Editor (XLE) command as described in Section 3. The Link Editor produces one linked output module for each segment defined in the control stream. When image format is not used, the Link Editor writes all the output modules to a single object file. The linked output modules for the procedure and task segments must be installed in a program file separately, using the Install Procedure Segment (IP) and the Install Task Segment (IT) commands, respectively. Refer to the *DNOS System Command Interpreter (SCI) Reference Manual* for an explanation of these commands.

In order to install the segments correctly, you must install them in the order in which they are defined in the control stream. You must also specify a logical unit number (LUNO) rather than a pathname for the object file prompt in the IP and IT commands. Using a LUNO prevents the object file from being rewound before the next output module is installed.

4.4 READING THE LINK MAP

Figure 4-6 shows the listing file produced for the example program. This file is similar to the listing file shown in Section 3 for single task segments. Page 1 contains a copy of the control stream. Page 2 lists the pathnames entered for the XLE command, the format selected, and the defined libraries. Page 3 begins the actual link map.

In this case, the link map consists of several pages. The procedure segment is defined first. The first line of this definition shows the segment type (PROCEDURE 1), the name assigned to the procedure (PROC1), and the origin and length of the procedure segment. If you use the DUMMY command, the word DUMMY also appears on this line.

The modules used to form the procedure segment are listed next. Notice that the DSEGs for these modules are listed in the procedure segment (under $DATA following each module), but the origins show locations in the task segment.

The task segment definition starts on a new page. (This does not occur if you use the NOPAGE command in the control stream.) When you use the ALLOCATE command (as in this example), the Link Editor splits the task segment definition into two parts: the preallocated portion and the postallocated portion. The preallocated portion is defined first. The postallocated portion starts a new page and is identified by the phrase (POST ALLOCATE) on the first line.

When linking a procedure segment to several different tasks, check the link maps of each task to ensure that the preallocated portion of the task segment and the DSEGs and CSEGs from the procedure modules are at the same origins in all tasks.

```
                                                  04/15/82   16:32:36              PAGE    1
 COMMAND LIST

  LIBRARY    VOL1.OBJ.PROC
  PROCEDURE  PROC1
  INCLUDE    (MODX),(MODY)
  TASK       TSK1
  INCLUDE    (MODA)
  ALLOCATE
  INCLUDE    (MODB),(MODC)
  END
```

Figure 4.6. Example Listing File — Procedure/Task Structure (Sheet 1 of 5)

```
                                          04/15/82  16:32:36         PAGE   2
LINK MAP

CONTROL FILE = VOL1.CNTRL.PROC

LINKED OUTPUT FILE = VOL1.PROCOBJ

LIST FILE = VOL1.MAP.PROC

OUTPUT FORMAT = ASCII

LIBRARIES

NO   ORGANIZATION   PATHNAME

1    RANDOM         VOL1.OBJ.PROC
```

Figure 4-6. Example Listing File — Procedure/Task Structure (Sheet 2 of 5)

```
                                          04/15/82  16:32:36         PAGE   3


PROCEDURE 1, PROC1    ORIGIN = 0000  LENGTH = 001E

MODULE    NO   ORIGIN   LENGTH    TYPE        DATE       TIME      CREATOR

MODX      1     0000     0014    INCLUDE,1   04/15/82   16:26:55   SDSMAC
$DATA     1     0026     0002
MODY      2     0014     000A    INCLUDE,1   04/15/82   16:29:04   SDSMAC
$DATA     2     0028     0002


                           D E F I N I T I O N S

 NAME     VALUE NO    NAME     VALUE NO    NAME     VALUE NO    NAME     VALUE NO

 MODX     0000   1    MODY     0014   2
```

Figure 4-6. Example Listing File — Procedure/Task Structure (Sheet 3 of 5)

```
                                             04/15/82  16:32:36           PAGE   4


PHASE 0, TSK1     ORIGIN = 0020  LENGTH = 0064

MODULE     NO   ORIGIN    LENGTH      TYPE        DATE       TIME      CREATOR

MODA       3     0020      0006     INCLUDE,1   04/15/82   16:09:25   SDSMAC


COMMON     NO   ORIGIN    LENGTH

COM1       4     002A      0004
```

Figure 4-6. Example Listing File — Procedure/Task Structure (Sheet 4 of 5)

```
                                             04/15/82  16:32:36           PAGE   5


PHASE 0, TSK1     ORIGIN = 002E  LENGTH = 0056   (POST ALLOCATE)

MODULE     NO   ORIGIN    LENGTH      TYPE        DATE       TIME      CREATOR

MODB       4     002E      001E     INCLUDE,1   04/15/82   16:20:02   SDSMAC
$DATA      4     004C      0022
MODC       5     006E      000E     INCLUDE,1   04/15/82   16:22:33   SDSMAC
$DATA      5     007C      0002


COMMON     NO   ORIGIN    LENGTH

COM2       5     007E      0006


                                D E F I N I T I O N S

 NAME      VALUE  NO     NAME      VALUE  NO     NAME      VALUE  NO     NAME      VALUE  NO

 ENDACT    0048   4      MODC      006E   5      START     002E   4      WP        004C   4


****  LINKING COMPLETED
```

Figure 4-6. Example Listing File — Procedure/Task Structure (Sheet 5 of 5)

5

Linking Program Segments

5.1 INTRODUCTION

DNOS segment management provides a great deal of flexibility in the use and handling of program segments. Although a number of segments can be in memory, a maximum of three segments can be addressed at one time. The task can issue supervisor calls (SVCs) to dynamically change the set of segments currently addressable. Thus, using program segments, you can increase the amount of code that can fit into the program's logical address space by exchanging segments as required.

This section explains the structure of tasks using program segments and provides an example program to be linked. It also explains how to build the control stream and read the link map for this type of structure.

5.2 TASK/PROGRAM SEGMENT STRUCTURE

In this type of structure, a program can consist of a number of different segments, including one procedure segment (optional), one task segment, and a number of program segments. The system maps each of these segments into one of three map positions (1, 2, or 3) of the program's logical address space. You must define the segment types in the correct order in the control stream so that they can be mapped into the right map positions. A procedure segment must precede the task segment, and program segments must follow the task segment. All linked program segments must be in the same map position.

If you use a procedure segment, the procedure segment occupies map position 1, the task segment occupies map position 2, and the program segments all occupy map position 3. You cannot use two procedure segments when also using program segments. (Refer to Section 4 for a description of procedure segments.)

If you do not use a procedure segment, the task segment occupies map position 1, and the program segments all occupy map position 2. In this case, you cannot link any program segments in map position 3. However, the task can issue SVCs to add program segments in map position 3 during execution.

As in all types of structures, the task segment must contain the entry vector for the program. The task segment may also contain the SVCs to add, delete, or exchange program segments as required. Refer to the *DNOS Supervisor Call (SVC) Reference Manual* for an explanation of these SVCs. The task segment itself cannot be exchanged with other segments.

Program segments can contain both executable code and/or data. The program segments can also vary in size. However, the total size of the procedure segment (if used), the task segment, and the largest program segment cannot exceed 64K bytes.

You can link as many program segments as needed to reduce the total size of the mapped segments. (When you use image format, the number of segments defined is limited to the number of segments that will fit in the program file.) Remember that the task can address only one linked program segment at a time. Therefore, these program segments cannot reference each other directly.

The procedure segment and task segment are loaded into memory when the task is bid. The program segments are not loaded until they are mapped in by an SVC. The call block for the SVC requires the installed ID of the segment that is to be loaded or mapped. You can obtain this ID in one of two ways. One way is to assign an ID yourself either in the link control stream (with the ID operand of the PHASE command) or when you install the segments. This way, you can assign the same ID used in the call block. In this case, you must ensure that the ID assigned is unique with respect to all other segments or overlays in the same program file.

The other way to obtain a program segment ID is to reference the segment by name in the task or procedure segment, as follows:

REF *segment name*
DATA *segment name*

The *segment name* must be the same name assigned in the SEGMENT command that defines the program segment. The *segment name* must also be unique with respect to all externally defined (DEF tag) symbols in the program and all names assigned to other segments or phases in the control stream. In this case, the Link Editor resolves the reference by assigning an available ID rather than an address to *segment name*. This ID is stored as the value of the DATA directive operand, which can be used in the call block for the SVC.

When linking programs in this type of structure, you can use any number of modules to form each segment. As an example, this section uses four object modules to form the task segment and two program segments. Figure 5-1 through Figure 5-4 contain assembly listings of the example modules (MODA, MODB, MODC, and MODD). The examples assume that each module is contained in a separate file of the same name as the module, under a directory named VOL1.OBJ.SEG.

Module MODA contains the entry vector for the program and other code that will be placed in the task segment. Modules MODB and MODC are used to form one of the program segments (SEG1). Module MODD is used to form the other program segment (SEG2).

Each of the modules contains PSEG, DSEG, and CSEG directives. For the task segment, the Link Editor reorganizes the assembler-defined segments in the same manner as it does for single task structures. (See Section 3.) The Link Editor does not reorganize the PSEGs and DSEGs within the program segments. CSEGs are still placed after the last DSEG in each segment. If the modules included in a program segment reference a CSEG not used by the task segment, this CSEG is placed in the program segment. However, if another program segment also references this CSEG, the CSEG is promoted (moved) to the task segment. Subsequent paragraphs further discuss the placement of assembler-defined segments.

```
 0001                          IDT   'MODA'
 0002                          REF   MODB, MODD
 0003 0000                     DSEG
 0004 0000           WP        BSS   32
 0005 0020 4000      CHGSEG    DATA  >4000, >00FF, >8002, 0
      0022 00FF
      0024 8002
      0026 0000
 0006 0028 0000      SEGID     DATA  0, 0, 0, 0, 0, 0, 0, 0
      002A 0000
      002C 0000
      002E 0000
      0030 0000
      0032 0000
      0034 0000
      0036 0000
 0007 0038   04      END       BYTE  >04
 0008 0000                     PSEG
 0009 0000 0000"               DATA  WP, START, ENDACT
      0002 0006'
      0004 0022'
 0010 0006 0200      START     LI    R0, 10
      0008 000A
 0011 000A C800                MOV   R0, @SEGID
      000C 0028"
 0012 000E 2FE0                XOP   @CHGSEG, 15
      0010 0020"
 0013 0012 06A0                BL    @MODB
      0014 0000
 0014 0016 0200                LI    R0, 11
      0018 000B
 0015 001A 2FE0                XOP   @CHGSEG, 15
      001C 0020"
 0016 001E 06A0                BL    @MODD
      0020 0000
 0017 0022 2FE0      ENDACT    XOP   @END, 15
      0024 0038"
 0018 0000                     CSEG  'COM1'
 0019 0000 0000      FLAG      DATA  0
 0020                          END
NO ERRORS,      NO WARNINGS
```

Figure 5-1. MODA Assembly Listing — Task/Program Segment Structure

```
 0001                         IDT   'MODB'
 0002                         DEF   MODB
 0003                         REF   MODC
 0004 0000                    PSEG
 0005 0000 C80B   MODB        MOV   R11,@SAVE11
      0002 0000"
 0006 0004 C820               MOV   @VALUE,@BUF
      0006 0002"
      0008 0000+
 0007 000A A820               A     @FLAG,@BUF
      000C 0000+
      000E 0000+
 0008 0010 06A0               BL    @MODC
      0012 0000
 0009 0014 C820               MOV   @BUF,@LUNO
      0016 0000+
      0018 0000+
 0010 001A C2E0               MOV   @SAVE11,R11
      001C 0000"
 0011 001E 045B               RT
 0012 0000                    DSEG
 0013 0000 0000   SAVE11      DATA 0
 0014 0002 9999   VALUE       DATA >9999
 0015 0000                    CSEG 'COM1'
 0016 0000        FLAG        BSS   2
 0017 0000                    CSEG 'COM2'
 0018 0000        BUF         BSS   2
 0019 0000                    CSEG 'COM3'
 0020 0000 0000   LUNO        DATA 0
 0021                         END
NO ERRORS,      NO WARNINGS
```

Figure 5-2. MODB Assembly Listing — Task/Program Segment Structure

```
 0001                         IDT   'MODC'
 0002                         DEF   MODC
 0003 0000                    PSEG
 0004 0000 A820   MODC        A     @VAL2,@BUF
      0002 0000"
      0004 0000+
 0005 0006 05E0               INCT @FLAG
      0008 0000+
 0006 000A A820               A     @LUNO,@BUF
      000C 0000+
      000E 0000+
 0007 0010 045B               RT
 0008 0000                    DSEG
 0009 0000 22B8   VAL2        DATA  8888
 0010 0000                    CSEG 'COM1'
 0011 0000        FLAG        BSS   2
 0012 0000                    CSEG 'COM2'
 0013 0000        BUF         BSS   2
 0014 0000                    CSEG 'COM3'
 0015 0000        LUNO        BSS   2
 0016                         END
NO ERRORS,      NO WARNINGS
```

Figure 5-3. MODC Assembly Listing — Task/Program Segment Structure

```
 0001                          IDT   'MODD'
 0002                          DEF   MODD
 0003 0000                     PSEG
 0004 0000 A820   MODD         A     @VAL,@FLAG
      0002 0000"
      0004 0000+
 0005 0006 0560                INV   @BUF
      0008 0000+
 0006 000A 045B                RT
 0007 0000                     DSEG
 0008 0000 10E1   VAL          DATA  4321
 0009 0000                     CSEG  'COM1'
 0010 0000        FLAG         BSS   2
 0011 0000                     CSEG  'COM2'
 0012 0000        BUF          BSS   2
 0013                          END
NO ERRORS,      NO WARNINGS
```

Figure 5-4. MODD Assembly Listing — Task/Program Segment Structure

5.3 BUILDING THE CONTROL STREAM

In this type of structure, the control stream contains the same basic commands as that for a single task, with the addition of the SEGMENT command. The SEGMENT command defines the beginning of a program segment. All program segments must follow the task segment. The following is the control stream for linking the example program:

```
LIBRARY    VOL1.OBJ.SEG
TASK       TSK2
INCLUDE    (MODA)
SEGMENT    2,SEG1
INCLUDE    (MODB),(MODC)
SEGMENT    2,SEG2
INCLUDE    (MODD)
END
```

The SEGMENT command specifies the map position the program segment will occupy (map position 2 in this example) and assigns a name to the segment. You can use as many SEGMENT commands in the control stream as necessary, but all of them must be in the same map position. Optionally, the SEGMENT command allows you to assign a load point and segment ID to the segment. The following is an example:

```
SEGMENT   2,SEG1,PROGRAM 0600,ID 1
```

The segment ID is especially useful with image format. This allows you to assign the same ID used in the call block for the SVC that loads a particular segment.

The following shows the arrangement of the assembler-defined segments in the linked output, using the above control stream:

Task segment (TSK2):

```
     MODA  PSEG
     MODA  DSEG
           CSEG COM1 (used by all modules)
           CSEG COM2 (used by modules MODB, MODC, and MODD)
```

Program segment (SEG1):

```
     MODB  PSEG
     MODB  DSEG
     MODC  PSEG
     MODC  DSEG
           CSEG COM3 (used only by MODB and MODC)
```

Program segment (SEG2):

```
     MODD  PSEG
     MODD  DSEG
```

The task segment contains the PSEG, DSEG, and CSEG COM1 defined in MODA. The task segment also contains CSEG COM2, even though it is not referenced by MODA. Since both SEG1 and SEG2 reference CSEG COM2, the Link Editor promotes (moves) this CSEG to the task segment so that both program segments have access to it.

Program segment SEG1 contains the PSEGs and DSEGs from MODB and MODC. Notice that the PSEGs and DSEGs are not grouped together but are placed in the order in which the modules are included in the link. SEG1 also contains CSEG COM3, which is placed after the last DSEG in this program segment. CSEG COM3 remains in the program segment since it is not referenced by the task segment or the other program segment.

In addition to the commands discussed in this section, you can use any of the other commands discussed in Section 3 for single tasks. If you use a procedure segment, follow the instructions given in Section 4. Program segments can also contain overlays as described in Section 6.

Using the example control stream, you can execute the Link Editor with the Execute Link Editor (XLE) command as described in Section 3. The Link Editor produces one linked output module for each segment defined in the control stream. If you select the IMAGE option in the FORMAT command, the Link Editor installs the segments directly into a program file. Otherwise, the Link Editor writes the linked output modules to a single object file. You must install the linked output modules into a program file before you can execute the program. You can install the procedure, task, and program segments using the Install Procedure (IP), Install Task (IT), and Install Program Segment (IPS) commands, respectively. Refer to the *DNOS System Command Interpreter (SCI) Reference Manual* for an explanation of these commands.

To install the segments correctly, you must install them in the same order as they are defined in the control stream. You must also specify a logical unit number (LUNO) rather than a pathname for the object file prompt in the IP, IT, and IPS commands. Using a LUNO prevents the object file from being rewound before the next output module is installed.

5.4 READING THE LINK MAP

Figure 5-5 shows the listing file produced for the example program. This file is similar to the listing file shown in Section 3 for single tasks. Page 1 contains a copy of the control stream. Page 2 lists the pathnames entered for the XLE command, the format selected, and the defined libraries. Page 3 begins the actual link map.

Since no procedure segment is used in this example, the task segment is defined first. The first line of this definition shows the segment type (PHASE 0), the name assigned to the task segment (TSK2), and the origin and length of the task segment. The module used to form the task segment is listed next.

The definition for each program segment starts on a new page unless you use the NOPAGE command in the control stream. The definitions appear in the order in which the segments are defined in the control stream. The first line of each definition shows the segment type, the name assigned, and the origin and length. Notice that the origins are the same for all the program segments, but the lengths vary.

```
                                          04/15/82  17:07:33          PAGE    1
 COMMAND LIST

  LIBRARY     VOL1.OBJ.SEG
  TASK        TSK2
  INCLUDE     (MODA)
  SEGMENT     2,SEG1
  INCLUDE     (MODB),(MODC)
  SEGMENT     2,SEG2
  INCLUDE     (MODD)
  END
```

Figure 5-5. Example Listing File — Task/Program Segment Structure (Sheet 1 of 5)

```
                                          04/15/82  17:07:33          PAGE    2
 LINK MAP

 CONTROL FILE = VOL1.CNTRL.SEG

 LINKED OUTPUT FILE = VOL1.SEGOBJ

 LIST FILE = VOL1.MAP.SEG

 OUTPUT FORMAT = ASCII

 LIBRARIES

 NO    ORGANIZATION    PATHNAME

 1     RANDOM          VOL1.OBJ.SEG
```

Figure 5-5. Example Listing File — Task/Program Segment Structure (Sheet 2 of 5)

```
                                        04/15/82  17:07:33          PAGE   3


PHASE 0, TSK2      ORIGIN = 0000  LENGTH = 0080

MODULE     NO    ORIGIN    LENGTH      TYPE         DATE        TIME       CREATOR

MODA       1      0000      0026     INCLUDE,1    04/15/82    16:46:05     SDSMAC
$DATA      1      0026      0039


COMMON     NO    ORIGIN    LENGTH

COM1       4      0060      0002
COM2       4      0062      0002
```

Figure 5-5. Example Listing File — Task/Program Segment Structure (Sheet 3 of 5)

```
                                        04/15/82  17:07:33          PAGE   4


SEGMENT SEG1       ORIGIN = 0080  LENGTH = 003A

MODULE     NO    ORIGIN    LENGTH      TYPE         DATE        TIME       CREATOR

MODB       2      0080      0020     INCLUDE,1    04/15/82    16:50:39     SDSMAC
$DATA      2      00A0      0004
MODC       3      00A4      0012     INCLUDE,1    04/15/82    16:54:15     SDSMAC
$DATA      3      00B6      0002


COMMON     NO    ORIGIN    LENGTH

COM3       3      00B8      0002


                              D E F I N I T I O N S

  NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO

  MODB      0080  2      MODC      00A4  3
```

Figure 5-5. Example Listing File — Task/Program Segment Structure (Sheet 4 of 5)

```
                                              04/15/82  17:07:33          PAGE   5


SEGMENT SEG2      ORIGIN = 0080  LENGTH = 003A

MODULE     NO   ORIGIN    LENGTH      TYPE        DATE        TIME       CREATOR

MODD       4     0080      000C     INCLUDE,1   04/15/82   17:06:51     SDSMAC
$DATA      4     008C      0002


                                   D E F I N I T I O N S

 NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO

 MODD      0080  4


****  LINKING COMPLETED
```

Figure 5-5.   Example Listing File — Task/Program Segment Structure (Sheet 5 of 5)

6 Linking Overlays

6.1 INTRODUCTION

The use of overlays allows you to increase the amount of code that can fit into the program's logical address space. An overlay consists of one portion (phase) of a task or program segment, which remains in memory only as long as it is required. When another overlay is needed, it is loaded into the same physical area and replaces an overlay currently in memory. You can define overlays at any number of levels to make maximum use of memory.

This section explains the structure of programs using overlays and provides an example program to be linked. It also explains how to build the control stream and read the link map for this type of structure.

6.2 OVERLAY STRUCTURE

You can use overlay structures in conjunction with other types of structures; however, only the segment(s) occupying the last map position in the link can have overlays. Thus, procedure segments (if used) cannot contain overlays, and if the task segment contains overlays, you cannot link any program segments. You can use overlays in any number of program segments as long as the task segment does not also contain overlays.

An overlay structure consists of one root phase defined at level 0 and a number of other phases defined at levels 1 and higher. The root phase is that portion of the task or program segment that must remain in memory while the program is executing; it cannot be replaced by another phase. The root phase must also contain code to load the overlays in that segment.

The overlays are defined as phases at level 1 and higher. You can define any number of phases at each of these levels, limited only by the number of phases that can fit in one program file. The phases can vary in size, and phases defined at the same level can overlay each other.

A series of phases, starting with the root phase and including a phase at each successively higher level, comprises an overlay path. Overlay paths are established by the order in which the phases are defined in the control stream. When establishing overlay paths, you must consider the relationship between the phases, as follows:

- Phases that interact with each other (access or transfer control to) must be in the same path. If a module is required in more than one path, you can either include it in one phase of each path or include it in a phase common to all paths that require it.

- Phases defined at the same level must be independent of each other; they cannot reference each other directly or indirectly.

When planning overlays, you also need to consider the frequency of use of each phase. Since loading an overlay requires time, code that is frequently used should be placed in a procedure segment, the root phase, or a separate program segment rather than in an overlay.

When linking overlay structures, you can use any number of object modules to form each phase. As an example, this section uses ten object modules to form a task segment with overlays. An example of a program segment with overlays using the same basic structure is also provided. (To simplify the example, the assembly listings for the object modules have been omitted in this section.) The example assumes that each object module is in a separate file of the same name as the module, under a directory named VOL1.OBJ.OVLY.

Figure 6-1 shows the structure of the example task segment. The segment consists of nine phases at four levels. Each phase is formed by one or two object modules. (Nine of the object modules are unique to one phase, and one object module is used in two phases.)

The root phase (ROOT) at level 0 is formed by module MODA. Level 1 has two phases, OVLY1A (formed by module MODB) and OVLY1B (formed by module MODF). Level 2 has four phases, OVLY2A (formed by module MODC), OVLY2B (formed by modules MODD and MODE), OVLY2C (formed by module MODG), and OVLY2D (formed by modules MODJ and MODE). Level 3 has two phases, OVLY3A (formed by module MODH) and OVLY3B (formed by module MODI). This structure has five overlay paths, as follows:

- Path 1 contains phases ROOT, OVLY1A, and OVLY2A
- Path 2 contains phases ROOT, OVLY1A, and OVLY2B
- Path 3 contains phases ROOT, OVLY1B, OVLY2C, and OVLY3A
- Path 4 contains phases ROOT, OVLY1B, OVLY2C, and OVLY3B
- Path 5 contains phases ROOT, OVLY1B, and OVLY2D

A phase that is in the path of two or more phases at the same higher level is called a *parent node*. Thus, the example structure has four parent nodes:

- Phase ROOT, which is the parent node of OVLY1A and OVLY1B
- Phase OVLY1A, which is the parent node of OVLY2A and OVLY2B
- Phase OVLY1B, which is the parent node of OVLY2C and OVLY2D
- Phase OVLY2C, which is the parent node of OVLY3A and OVLY3B

The Link Editor produces one output module for each phase defined. The root phase is installed as the root portion of the task (or program) segment. The root phase is loaded into memory when the task is bid (or, in the case of program segments, when the segment is loaded). Phases at level 1 and higher are installed as overlays. Overlays are not loaded into memory until required. The amount of memory allocated for a segment with overlays is the sum of the root phase and the longest overlay path.

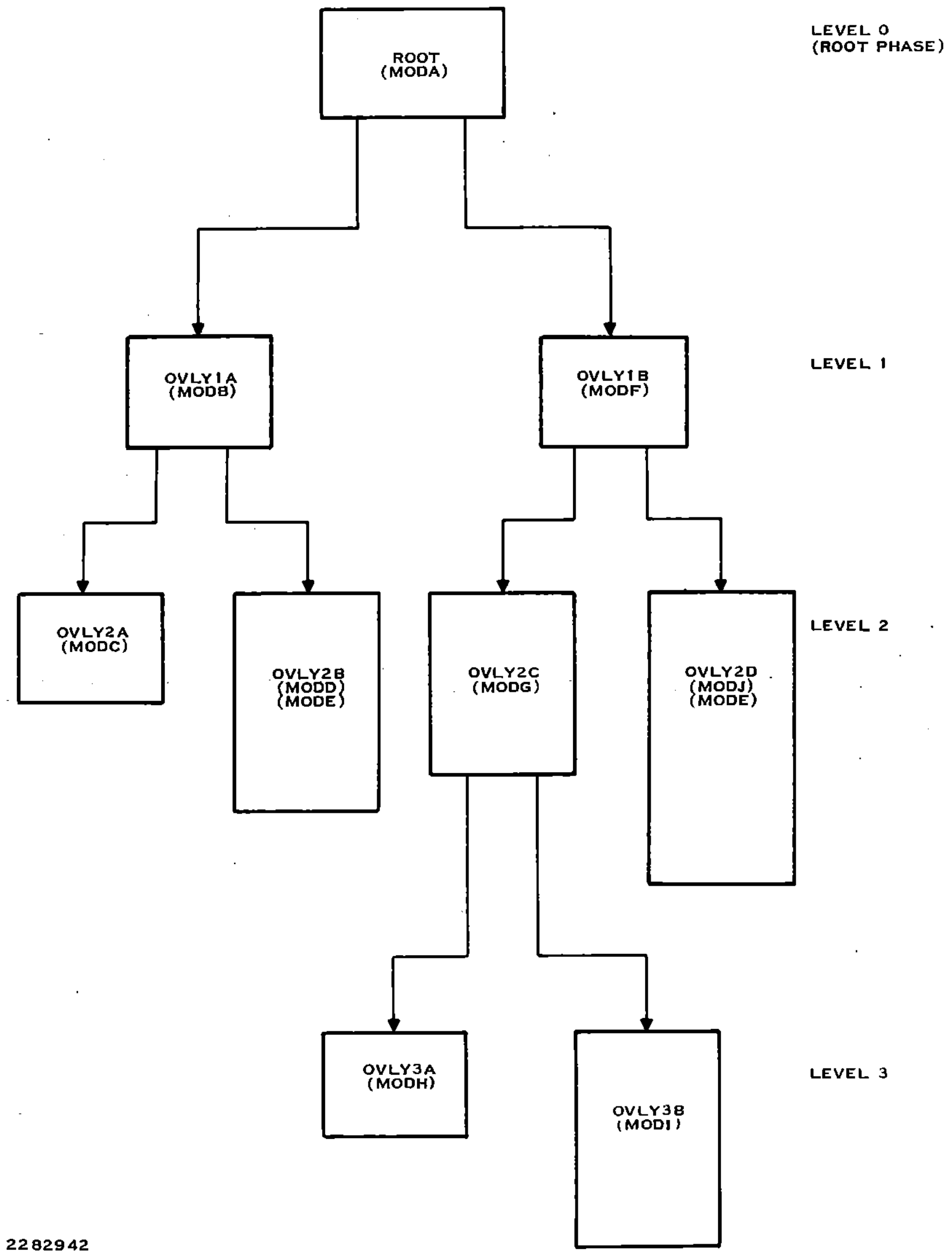


Figure 6-1. Example Overlay Structure

The linked output must contain code to load the various overlays as they are required. Overlays can be either user loaded by Load Overlay supervisor calls (SVCs) or automatically loaded by the Overlay Manager. The following paragraphs discuss user-loaded overlays and the Overlay Manager. Subsequent paragraphs also discuss the promotion of modules and the placement of assembler-defined segments in an overlay structure.

6.2.1 User-Loaded Overlays

With user-loaded overlays, the root phase of the segment containing overlays must issue Load Overlay SVCs to load the overlays in that segment. The Load Overlay SVC is described in detail in the *DNOS Supervisor Call (SVC) Reference Manual.*

When issuing these SVCs, ensure that the overlays are loaded in the proper sequence. Before a specific phase can be referenced, it must already be in memory. In addition, all preceding phases in its path must be in memory; that is, all phases between the referenced phase and the root phase must be in memory.

The call block for the Load Overlay SVC requires the installed ID of the overlay that is to be loaded. You can obtain this ID in one of two ways. One way is to assign an ID yourself either in the link control stream (with the ID operand of the PHASE command) or when you install the overlays. This way, you can assign the same ID used in the call block. In this case, you must ensure that the ID assigned is unique with respect to all other overlays in the same program file.

The other way to obtain an overlay ID is to reference the overlay by name in the root phase, as follows:

REF   *overlay name*
DATA *overlay name*

The *overlay name* must be the same name assigned in the PHASE command that defines the overlay. The *overlay name* must also be unique with respect to all externally defined (DEF tag) symbols in the program and all names assigned to segments or other phases in the control stream. In this case, the Link Editor resolves the reference by assigning an available ID to *overlay name* rather than an address. This ID is stored as the value of the DATA directive operand, which can be used in the call block for the Load Overlay SVC.

6.2.2 Overlay Manager

DNOS includes an Overlay Manager, which can be linked with the program to perform automatic overlay loading during program execution. The Overlay Manager supports the loading of overlays only in the task segment, not in a program segment.

You can use the Overlay Manager with assembly language, COBOL, or FORTRAN programs. However, you cannot use it with Pascal programs since Pascal uses different calling conventions than the Overlay Manager. Also, do not use the Overlay Manager with user-loaded overlays in the same program, since unpredictable results can occur.

The Overlay Manager loads an overlay when the program references a module or subroutine within that overlay. For the Overlay Manager to work properly, the overlays must be referenced in the proper sequence. A reference from one phase to another phase that is further from the root phase is a *downward reference*. A downward reference causes the Overlay Manager to load the required phase if it is not currently in memory. A downward reference can go down only one level from the current level. Therefore, a phase must be referenced by its parent node to be loaded. For example, referring to Figure 6-1, OVLY1B can reference OVLY2C or OVLY2D, but it cannot reference OVLY3A or OVLY3B.

A reference from one phase to another phase that is closer to the root phase is an *upward reference*. An upward reference does not cause an overlay to be loaded. If the downward references were made correctly, phases required for upward references will already be loaded.

The Overlay Manager also requires the use of specific calling conventions when referencing a module or subroutine within another phase. The reference must be made through use of the Branch and Load Workspace Pointer (BLWP) instruction. The operand of this instruction must specify one of the following:

- The symbolic name of the module or subroutine, as follows:

  ```
  BLWP   @name
  ```

- A register that contains the address of the module or subroutine, as follows:

  ```
  LI     R5,name
  BLWP   *R5
  ```

In either case, *name* is the symbol associated with a transfer vector (workspace pointer and program counter) to the module or subroutine. The symbol *name* must also be specified as an external symbol (through the use of REF and DEF directives). When resolving this symbol, the Link Editor changes the address of *name* to point to an entry in a table used by the Overlay Manager.

NOTE

You cannot use indexed addressing or direct register addressing in the BLWP instruction.

The Overlay Manager requires the use of two tables, which are built by the Link Editor. The tables are the overlay entry vector (OEV) table and the overlay phase directory (OPD) table. The following briefly describes the structure of these tables. Normally, you do not need this information to link the Overlay Manager with your program. However, it may be useful when you are debugging your program.

The OEV table is a read-only table that contains an entry for each downward reference made in the overlay structure. (The pointer to the correct entry for this table is obtained from the *name* symbol in the BLWP instruction.) Each entry in the OEV table consists of five words, as follows:

Word	Description
0	Workspace pointer (WP) for the Overlay Manager
1	Address of the new program counter (PC) value for the Branch and Link (BL) instruction in the next word (equal to current address plus 2)
2	A BL instruction to transfer control to the Overlay Manager (equivalent to BL *R1)
3	Address of the transfer vector (WP and PC) in the referenced overlay
4	Address of the entry in the OPD table that describes the overlay to be loaded for this entry

The OPD table consists of two parts: a read-only part and a read/write part. The read-only part contains a three-word entry for each phase defined in the control stream. (The pointer for each of these entries is obtained from the last word in an OEV table entry.) The format for an OPD entry is as follows:

Word	Description
0	Overlay ID
1	Address of the OPD entry for the first overlay on the same level as this overlay
2	Overlay load address

The read/write part of the OPD table consists of a bit map, which contains flags that indicate whether an overlay is currently in memory (1 equals yes, 0 equals no). The bit map is indexed by the overlay ID. The bit map always consists of 256 bits, numbered from 0 through 255. Each bit corresponds to an overlay ID. Since overlay IDs are numbered from 1 through 256, the bit corresponding to a particular overlay ID is one digit less than the ID. For example, bit 24 of the bit map corresponds to overlay ID 25. Normally, all of the bits in the bit map are not used in one program. The link map shows the starting location for the bit map.

6.2.3 Promotion of Modules

Promotion is the elevation of a module to a phase closer to the root phase in the overlay structure. The Link Editor promotes only those modules that are brought into the link as the result of a search operation (either by automatic searching or by use of the SEARCH or FIND commands).

The promotion of a module depends on which phases reference that module. When required, the Link Editor promotes the module to the highest-level phase common to the paths of all phases that reference the module.

For example, assume phases OVLY2A and OVLY2B in Figure 6-1 both reference module MODX and assume MODX is brought into the link as the result of a search operation. The Link Editor promotes MODX to phase OVLY1A (the parent node) so that both OVLY2A and OVLY2B can access it.

As another example, assume phases OVLY2B and OVLY3A both reference module MODY. In this case, the Link Editor promotes MODY to phase ROOT, since phase ROOT is the only phase in the same path as both OVLY2B and OVLY3A.

Remember that the promotion of modules does not apply to modules specifically included in the link (with INCLUDE commands). In this case, you must ensure that a module is included in a phase that is common to the paths of all phases referencing that module. Alternatively, you can include a module in several phases in different paths, as in the example. However, you must also ensure that a particular module is not included in more than one phase in the same path; otherwise, multiple definitions appear in the linked output. This can easily happen when a particular module is both specifically included in one phase and brought into the link by a search operation for other phases.

6.2.4 Placement of Assembler-Defined Segments

For the root phase in the task segment, the Link Editor reorganizes the assembler-defined segments in the same manner as it does for single task structures. (See Section 3.) The Link Editor does not reorganize the PSEGs and DSEGs of included modules within overlays. CSEGs are still placed after the last DSEG in each phase.

When necessary, the Link Editor promotes CSEGs in the same manner as it does for modules; that is, it promotes the CSEG to the highest-level phase common to the paths of all phases that reference the CSEG.

6.3 BUILDING THE CONTROL STREAM

In an overlay structure, the control stream must contain commands to define the segments and PHASE commands to define the overlays within a segment. The overlays must be defined in a specific order and at the correct level to properly establish the overlay paths. This ensures that each phase is assigned the correct load point. The following is the control stream to link the example program:

```
LIBRARY    VOL1.OBJ.OVLY
TASK       ROOT
INCLUDE    (MODA)
PHASE      1,OVLY1A
INCLUDE    (MODB)
PHASE      2,OVLY2A
INCLUDE    (MODC)
PHASE      2,OVLY2B
INCLUDE    (MODD),(MODE)
PHASE      1,OVLY1B
INCLUDE    (MODF)
PHASE      2,OVLY2C
INCLUDE    (MODG)
PHASE      3,OVLY3A
INCLUDE    (MODH)
PHASE      3,OVLY3B
INCLUDE    (MODI)
PHASE      2,OVLY2D
INCLUDE    (MODJ),(MODE)
END
```

The TASK command defines the beginning of the task segment. The INCLUDE command immediately following the TASK command specifies the object module that will form the root portion of the task segment. Optionally, you can use a PHASE 0 command instead of the TASK command. (TASK and PHASE 0 commands are semantically identical.)

Each PHASE command defines the beginning of a new phase at the level specified. The INCLUDE commands following the PHASE commands specify the modules that will form each phase.

The overlay paths are established by defining one phase at each successively higher level until all the phases forming a complete path are defined. Each phase is defined only once in the control stream. Thus, once you have established one path, you can start establishing the next path by defining a new phase at the next higher level from a parent node of the previously established path. Referring to Figure 6-1, notice that the definition of phases is from top-to-bottom first and then left-to-right.

In the control stream, path 1 is established first by defining ROOT at level 0, OVLY1A at level 1, and OVLY2A at level 2. Then, path 2 is established simply by defining OVLY2B at level 2, since the other phases in this path have already been defined. However, path 3 requires several new phases to be defined. It starts with the definition of OVLY1B at level 1 and continues with OVLY2C at level 2 and OVLY3A at level 3. Then, path 4 is established by defining OVLY3B at level 3. Path 5 is established by defining OVLY2D at level 2.

When you use image format, you can also use the optional ID operand in each PHASE command to assign IDs to the phases. This is especially useful when you are loading your own overlays.

In addition to the commands already discussed, you can use any of the other commands discussed in Section 3 for single task structures. If you use the ADJUST command in an overlay structure, adjust all phases at the same level and with the same parent node in the same manner. For example, if you want to adjust OVLY2A so that it is aligned on a particular boundary, adjust OVLY2B so that it is also aligned on that boundary. Otherwise, phases that overlay each other might be assigned different load points, causing unpredictable results.

To adjust the phases properly, place the ADJUST commands so that they appear after the INCLUDE command(s) for a parent node but before the first PHASE command for the level you want adjusted. When you place an ADJUST command in this position, you do not have to adjust each phase separately; the Link Editor automatically aligns all subsequent phases at the same level and with the same parent node at the specified boundary. The following control stream shows the placement of ADJUST commands to properly align all the overlays at levels 2 and 3 on 32-byte boundaries:

```
LIBRARY     VOL1.OBJ.OVLY
TASK        ROOT
INCLUDE     (MODA)
PHASE       1,OVLY1A
INCLUDE     (MODB)
ADJUST      5
PHASE       2,OVLY2A
INCLUDE     (MODC)
PHASE       2,OVLY2B
INCLUDE     (MODD),(MODE)
PHASE       1,OVLY1B
INCLUDE     (MODF)
ADJUST      5
PHASE       2,OVLY2C
INCLUDE     (MODG)
ADJUST      5
PHASE       3,OVLY3A
INCLUDE     (MODH)
PHASE       3,OVLY3B
INCLUDE     (MODI)
PHASE       2,OVLY2D
INCLUDE     (MODJ),(MODE)
END
```

By adding some commands to the control stream, you can link the Overlay Manager with the program. You can also use the same basic overlay structure in a program segment rather than the task segment. Subsequent paragraphs discuss linking the Overlay Manager and linking program segments with overlays.

Once you have built the control stream, you can execute the Link Editor with the Execute Link Editor (XLE) command, as described in Section 3. The Link Editor produces one linked output module for each phase defined in the control stream. If you select the IMAGE option in the FORMAT command, the Link Editor installs the phases directly into a program file. Otherwise, the

Link Editor writes the linked output modules to a single object file. You must install the linked output modules into a program file before you can execute the program. You can install the root phase of the task segment using the Install Task (IT) command and the overlays using Install Overlay (IO) commands. Refer to the *DNOS System Command Interpreter (SCI) Reference Manual* for an explanation of these commands.

To install the phases correctly, you must install them in the same order as they are defined in the control stream. You must also specify a logical unit number (LUNO) rather than a pathname for the object file prompt in the IT and IO commands. Using a LUNO prevents the object file from being rewound before the next output module is installed.

6.3.1 Linking the Overlay Manager

You can link the Overlay Manager with the program by using the LOAD command in the control stream. The Overlay Manager is contained on the system subroutine library .S$SYSLIB. You must define this directory as a library in the control stream in order for the Link Editor to access it. You must also specify the IMAGE option with the FORMAT command.

The LOAD command causes the Link Editor to do the following:

- Include the Overlay Manager in the link
- Build the tables required by the Overlay Manager and resolve certain external symbols differently

The LOAD command must appear before the first overlay is defined. If you use a procedure segment, the LOAD command can optionally be placed in the procedure segment; otherwise, it must be placed in the root phase of the task segment. The following control stream links the Overlay Manager with the example program:

```
LIBRARY    .S$SYSLIB
LIBRARY    VOL1.OBJ.OVLY
FORMAT     IMAGE,REPLACE
TASK       ROOT
INCLUDE    (MODA)
LOAD
PHASE      1,OVLY1A
INCLUDE    (MODB)
PHASE      2,OVLY2A
INCLUDE    (MODC)
PHASE      2,OVLY2B
INCLUDE    (MODD),(MODE)
PHASE      1,OVLY1B
INCLUDE    (MODF)
PHASE      2,OVLY2C
INCLUDE    (MODG)
PHASE      3,OVLY3A
INCLUDE    (MODH)
PHASE      3,OVLY3B
INCLUDE    (MODI)
PHASE      2,OVLY2D
INCLUDE    (MODJ),(MODE)
END
```

6.3.2 Linking Program Segments with Overlays

An overlay structure in a program segment is similar to that in the task segment. In this case, the root phase of the overlay structure is defined by the SEGMENT command and the associated INCLUDE command(s). (You cannot use the PHASE 0 command in a program segment.) The PHASE commands (at level 1 and higher) following the SEGMENT command define the overlays in that segment.

You can use overlay structures in several different program segments within the same program. The total number of segments and overlays defined in the control stream is limited to the number that will fit in the program file.

The root phase for each segment must contain SVCs to load the overlays in that segment. (You cannot use the Overlay Manager in program segments.)

The following control stream shows the example overlay structure placed in a program segment. (Refer to Section 5 for information on program segments.) The control stream also defines a task segment and another program segment.

```
LIBRARY     VOL1.OBJ.OVLY
TASK        MAIN
INCLUDE     (MODTSK)
SEGMENT     2,SEG1
INCLUDE     (MODS)
SEGMENT     2,SEG2
INCLUDE     (MODA)
PHASE       1,OVLY1A
INCLUDE     (MODB)
PHASE       2,OVLY2A
INCLUDE     (MODC)
PHASE       2,OVLY2B
INCLUDE     (MODD),(MODE)
PHASE       1,OVLY1B
INCLUDE     (MODF)
PHASE       2,OVLY2C
INCLUDE     (MODG)
PHASE       3,OVLY3A
INCLUDE     (MODH)
PHASE       3,OVLY3B
INCLUDE     (MODI)
PHASE       2,OVLY2D
INCLUDE     (MODJ),(MODE)
END
```

6.4 READING THE LINK MAP

Figure 6-2 shows the listing file produced for the example program. The information shown is similar to that for single tasks; however, a separate definition is given for each segment and overlay defined in the control stream. Each definition starts on a new page unless the NOPAGE command is used in the control stream. The definitions appear in the order in which the segments and overlays are defined in the control stream.

The control stream used for this example contains the LOAD command to show the placement of code associated with the Overlay Manager. The LOAD command causes L$$OVM and L$$OBM to appear in the link map. L$$OVM is the Overlay Manager module. It is listed as a module at the point where the LOAD command was placed in the control stream. L$$OBM is the bit map for the Overlay Manager; it is listed as a definition (DEF tag symbol) but not as a module. The value listed for L$$OBM shows the starting location for the bit map, which always appears at the end of the root phase for the task segment. (If a procedure segment and the ALLOCATE command are used, the bit map appears at the end of the preallocated portion of the root phase.) The tables built by the Link Editor for the Overlay Manager are also placed in the root phase, but they are not listed in the link map.

```
                                           04/16/82  09:22:16          PAGE    1
 COMMAND LIST

  LIBRARY    .S$SYSLIB
  LIBRARY    VOL1.OBJ.OVLY
  FORMAT     IMAGE,REPLACE
  TASK       ROOT
  INCLUDE    (MODA)
  LOAD
  PHASE      1,OVLY1A
  INCLUDE    (MODB)
  PHASE      2,OVLY2A
  INCLUDE    (MODC)
  PHASE      2,OVLY2B
  INCLUDE    (MODD),(MODE)
  PHASE      1,OVLY1B
  INCLUDE    (MODF)
  PHASE      2,OVLY2C
  INCLUDE    (MODG)
  PHASE      3,OVLY3A
  INCLUDE    (MODH)
  PHASE      3,OVLY3B
  INCLUDE    (MODI)
  PHASE      2,OVLY2D
  INCLUDE   (MODJ),(MODE)
  END
```

Figure 6-2. Example Listing File — Overlay Structure (Sheet 1 of 11)

```
                                         04/16/82  09:22:16          PAGE   2
LINK MAP

CONTROL FILE = VOL1.CNTRL.OVLY

LINKED OUTPUT FILE = VOL1.PROGRAM

LIST FILE = VOL1.MAP.OVLY

NUMBER OF OUTPUT RECORDS = 11

OUTPUT FORMAT = IMAGE

LIBRARIES

NO    ORGANIZATION    PATHNAME

1     RANDOM          .S$SYSLIB
2     RANDOM          VOL1.OBJ.OVLY
```

Figure 6-2. Example Listing File — Overlay Structure (Sheet 2 of 11)

```
                                         04/16/82  09:22:16          PAGE   3


PHASE 0, ROOT      ORIGIN = 0000  LENGTH = 026C   (TASK ID = 2)

MODULE      NO    ORIGIN    LENGTH      TYPE        DATE        TIME        CREATOR

MODA        1      0000      0018     INCLUDE,2   04/16/82    09:11:08     SDSMAC
$DATA       1      00B2      0020
L$$OVM      2      0018      009A     INCLUDE,1   06/26/81    12:43:26     SDSMAC
$DATA       2      00D2      002D


                               D E F I N I T I O N S

 NAME     VALUE NO     NAME     VALUE NO     NAME     VALUE NO     NAME     VALUE NO

 L$$OBM   0100  2     *L$$OVM   0018  2
```

Figure 6-2. Example Listing File — Overlay Structure (Sheet 3 of 11)

```
                                   04/16/82   09:22:16           PAGE    4


PHASE  1,  OVLY1A    ORIGIN = 01C6   LENGTH = 0038    (OVERLAY  ID = 1)

MODULE      NO    ORIGIN     LENGTH        TYPE          DATE         TIME        CREATOR

MODB        3      01C6       0018      INCLUDE,2    04/16/82    09:13:05     SDSMAC
$DATA       3      01DE       0020


                           D E F I N I T I O N S

 NAME       VALUE  NO      NAME       VALUE  NO      NAME       VALUE  NO      NAME       VALUE  NO

 MODB       01C6   3
```

Figure 6-2. Example Listing File — Overlay Structure (Sheet 4 of 11)

```
                                   04/16/82   09:22:16           PAGE    5


PHASE  2,  OVLY2A    ORIGIN = 01FE   LENGTH = 0036    (OVERLAY  ID = 2)

MODULE      NO    ORIGIN     LENGTH        TYPE          DATE         TIME        CREATOR

MODC        4      01FE       0016      INCLUDE,2    04/16/82    09:19:07     SDSMAC
$DATA       4      0214       0020


                           D E F I N I T I O N S

 NAME       VALUE  NO      NAME       VALUE  NO      NAME       VALUE  NO      NAME       VALUE  NO

 MODC       01FE   4
```

Figure 6-2. Example Listing File — Overlay Structure (Sheet 5 of 11)

```
                                          04/16/82  09:22:16          PAGE   6


PHASE 2, OVLY2B   ORIGIN = 01FE  LENGTH = 006C   (OVERLAY ID = 3)

MODULE     NO   ORIGIN    LENGTH     TYPE         DATE        TIME       CREATOR

MODD       5     01FE      0016    INCLUDE,2    04/16/82    09:17:05    SDSMAC
$DATA      5     0214      0020
MODE       6     0234      0016    INCLUDE,2    04/16/82    09:20:01    SDSMAC
$DATA      6     024A      0020


                                  D E F I N I T I O N S

 NAME      VALUE  NO     NAME      VALUE  NO     NAME      VALUE  NO     NAME      VALUE  NO

 MODD      01FE   5      MODE      0234   6
```

Figure 6-2. Example Listing File — Overlay Structure (Sheet 6 of 11)

```
                                          04/16/82  09:22:16          PAGE   7


PHASE 1, OVLY1B   ORIGIN = 01C6  LENGTH = 0038   (OVERLAY ID = 4)

MODULE     NO   ORIGIN    LENGTH     TYPE         DATE        TIME       CREATOR

MODF       7     01C6      0018    INCLUDE,2    04/16/82    09:14:33    SDSMAC
$DATA      7     01DE      0020


                                  D E F I N I T I O N S

 NAME      VALUE  NO     NAME      VALUE  NO     NAME      VALUE  NO     NAME      VALUE  NO

 MODF      01C6   7
```

Figure 6-2. Example Listing File — Overlay Structure (Sheet 7 of 11)

```
                                          04/16/82  09:22:16           PAGE   8


PHASE 2, OVLY2C   ORIGIN = 01FE  LENGTH = 0038   (OVERLAY ID = 5)

MODULE     NO   ORIGIN    LENGTH      TYPE         DATE        TIME       CREATOR

MODG       8     01FE      0018    INCLUDE,2   04/16/82    09:15:48     SDSMAC
$DATA      8     0216      0020


                                   D E F I N I T I O N S

  NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO

  MODG      01FE   8
```

Figure 6-2. Example Listing File — Overlay Structure (Sheet 8 of 11)

```
                                          04/16/82  09:22:16           PAGE   9


PHASE 3, OVLY3A   ORIGIN = 0236  LENGTH = 0036   (OVERLAY ID = 6)

MODULE     NO   ORIGIN    LENGTH      TYPE         DATE        TIME       CREATOR

MODH       9     0236      0016    INCLUDE,2   04/16/82    09:20:42     SDSMAC
$DATA      9     024C      0020


                                   D E F I N I T I O N S

  NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO

  MODH      0236   9
```

Figure 6-2. Example Listing File — Overlay Structure (Sheet 9 of 11)

```
                                          04/16/82  09:22:16          PAGE  10


PHASE 3, OVLY3B   ORIGIN = 0236  LENGTH = 0036   (OVERLAY ID = 7)

MODULE     NO   ORIGIN    LENGTH      TYPE        DATE       TIME      CREATOR

MODI       10    0236      0016     INCLUDE,2   04/16/82   09:21:22    SDSMAC
$DATA      10    024C      0020



                                    D E F I N I T I O N S

 NAME     VALUE NO     NAME     VALUE NO     NAME     VALUE NO     NAME     VALUE NO

 MODI     0236  10
```

Figure 6-2. Example Listing File — Overlay Structure (Sheet 10 of 11)

```
                                          04/16/82  09:22:16          PAGE  11


PHASE 2, OVLY2D   ORIGIN = 01FE  LENGTH = 006C   (OVERLAY ID = 8)

MODULE     NO   ORIGIN    LENGTH      TYPE        DATE       TIME      CREATOR

MODJ       11    01FE      0016     INCLUDE,2   04/16/82   09:17:59    SDSMAC
$DATA      11    0214      0020
MODE       12    0234      0016     INCLUDE,2   04/16/82   09:20:01    SDSMAC
$DATA      12    024A      0020



                                    D E F I N I T I O N S

 NAME     VALUE NO     NAME     VALUE NO     NAME     VALUE NO     NAME     VALUE NO

 MODE     0234  12     MODJ     01FE  11


****  LINKING COMPLETED
```

Figure 6-2. Example Listing File — Overlay Structure (Sheet 11 of 11)

7

Linking Absolute Memory Partitions

7.1 INTRODUCTION

The Link Editor provides commands that allow you to structure programs with absolute memory partitions. These commands position assembler-defined segments on prescribed absolute boundaries so that the program can eventually be loaded into a combination of read-only memory (ROM) and random-access memory (RAM).

Programs structured in this manner are intended for execution on development systems or stand-alone systems only; they cannot be executed under the control of the operating system.

This section explains the structure of programs that use absolute memory partitions and provides an example program to be linked. It also explains how to build the control stream and read the link map for this type of structure.

7.2 ABSOLUTE MEMORY PARTITION STRUCTURE

In this type of structure, you must include all the modules within a single task segment; you cannot define a procedure segment, a program segment, or overlays in the control stream. Also, you cannot specify image format since the output of this type of structure cannot be executed under the control of the operating system.

The object modules that form the program must contain PSEG, DSEG, and (optionally) CSEG directives to distinguish read-only code from read/write data. The PSEG directive defines program segments, which generally contain instructions and nonvariable data (read-only code). The DSEG directive defines data segments, which generally contain variable data (read/write code). The CSEG directive defines common segments, which contain variable data that can be shared by more than one module. (The Pascal, COBOL, and FORTRAN compilers automatically define these segment types.)

The Link Editor collects the PSEGs, DSEGs, and CSEGs from the modules so that each segment type can be placed in a separate area in the output. Using link control commands, you can specify the starting address for each area. Thus, the program areas (read-only code) are assigned absolute addresses corresponding to ROM locations, and the data and common areas (read/write code) are assigned absolute addresses corresponding to RAM locations. The actual placement of these areas depends on the commands used in the control stream. This is discussed in detail in the paragraph on building the control stream.

As an example, this section uses four object modules to form a program with absolute memory partitions. Figure 7-1 through Figure 7-4 contain assembly listings of the example modules (MODA, MODB, MODC, and MODD). The example assumes that each module is contained in a separate file that has the same name as the module, under a directory named VOL1.OBJ.ABS.

The example also assumes that the program will eventually be loaded into a system with the following memory scheme:

Locations >0000 through >BFFE are in ROM

Locations >C000 through >FFFE are in RAM

```
0001                         IDT   'MODA'
0002                         REF   MODB,MODC
0003 0000                    PSEG
0004 0000 C020  MDA000 MOV   @FLAG,R0
     0002 0000+
0005 0004 1604               JNE   MDA100
0006 0006 06A0               BL    @MODB
     0008 0000
0007 000A C020               MOV   @FLAG,R0
     000C 0000+
0008 000E 8800  MDA100 C     R0,@LIMIT
     0010 0000+
0009 0012 1502               JGT   MDA200
0010 0014 06A0               BL    @MODC
     0016 0000
0011 0018 05A0  MDA200 INC   @CNTR
     001A 0002+
0012 001C 10F1               JMP   MDA000
0013 0000                    DSEG
0014 0000       WP     BSS   32
0015 0000                    CSEG  'COM1'
0016 0000 0000  FLAG   DATA  0
0017 0000                    CSEG  'COM2'
0018 0000 03E7  LIMIT  DATA  999
0019 0002 0000  CNTR   DATA  0
0020                         END
NO ERRORS,      NO WARNINGS
```

Figure 7-1. MODA Assembly Listing — Absolute Memory Partitions

```
  0001                        IDT   'MODB'
  0002                        DEF   MODB
  0003 0000                   PSEG
  0004 0000 C020   MODB       MOV   @FLAG,R0
       0002 0000+
  0005 0004 8800              C     R0,@LIMIT
       0006 0000"
  0006 0008 1502              JGT   MDB100
  0007 000A 05A0              INC   @FLAG
       000C 0000+
  0008 000E 045B   MDB100     RT
  0009 0000                   DSEG
  0010 0000 0378   LIMIT      DATA  888
  0011 0000                   CSEG  'COM1'
  0012 0000 0000   FLAG       DATA  0
  0013                        END
 NO ERRORS,     NO WARNINGS
```

Figure 7-2. MODB Assembly Listing — Absolute Memory Partitions

```
  0001                        IDT   'MODC'
  0002                        DEF   MODC
  0003                        REF   MODD
  0004 0000                   PSEG
  0005 0000 02A0   MODC       STWP  R0
  0006 0002 C800              MOV   R0,@SAVEWP
       0004 0020"
  0007 0006 0200              LI    R0,MYWP
       0008 0000"
  0008 000A 02E0              LWPI  R0
       000C 0000
  0009 000E C820              MOV   @CNTR,@LOCAL
       0010 0002+
       0012 0000+
  0010 0014 A820              A     @FLAG,@LOCAL
       0016 0000+
       0018 0000+
  0011 001A 06A0              BL    @MODD
       001C 0000
  0012 001E 0620              DEC   @LIMIT
       0020 0000+
  0013 0022 C020              MOV   @SAVEWP,R0
       0024 0020"
  0014 0026 02E0              LWPI  R0
       0028 0000
  0015 002A 045B              RT
  0016 0000                   DSEG
  0017 0000        MYWP       BSS   32
  0018 0020        SAVEWP     BSS   2
  0019 0000                   CSEG  'COM1'
  0020 0000 0000   FLAG       DATA  0
  0021 0000                   CSEG  'COM2'
  0022 0000 03E7   LIMIT      DATA  999
  0023 0002 0000   CNTR       DATA  0
  0024 0000                   CSEG  'COM3'
  0025 0000        LOCAL      BSS   2
  0026                        END
 NO ERRORS,     NO WARNINGS
```

Figure 7-3. MODC Assembly Listing — Absolute Memory Partitions

```
0001                        IDT   'MODD'
0002                        DEF   MODD
0003 0000                   PSEG
0004 0000 C020   MODD       MOV   @LOCAL,R0
     0002 0000+
0005 0004 1302              JEQ   MDD100
0006 0006 05A0              INC   @CNTR
     0008 0002+
0007 000A A820   MDD100 A         @CONST,@FLAG
     000C 0000"
     000E 0000+
0008 0010 045B              RT
0009 0000                   DSEG
0010 0000 0005   CONST      DATA  5
0011 0000                   CSEG  'COM1'
0012 0000 0000   FLAG       DATA  0
0013 0000                   CSEG  'COM2'
0014 0000 03E7   LIMIT      DATA  999
0015 0002 0000   CNTR       DATA  0
0016 0000                   CSEG  'COM3'
0017 0000        LOCAL      BSS   2
0018                        END
NO ERRORS,       NO WARNINGS
```

Figure 7-4. MODD Assembly Listing — Absolute Memory Partitions

7.3 BUILDING THE CONTROL STREAM

The control stream for this type of structure contains some basic commands, plus commands to position the assembler-defined segments on prescribed boundaries. The following is the control stream for linking the example program:

```
LIBRARY     VOL1.OBJ.ABS
TASK        PROG2
PROGRAM     >2000
DATA        >C000
COMMON      >E000,COM2
INCLUDE     (MODA),(MODB),(MODC),(MODD)
END
```

The TASK command defines the beginning of the segment and assigns a name to the program.

The PROGRAM command defines the starting address for the program area. This area contains the PSEGs from all the included modules. The PSEGs are placed in the same order as the modules specified in the INCLUDE command (MODA PSEG at location >2000, followed by MODB PSEG, MODC PSEG, and MODD PSEG).

The DATA command defines the starting address for the data area, which contains all the DSEGs from the included modules. Again, the DSEGs are placed in the same order as the modules specified in the INCLUDE command.

The PROGRAM and DATA commands must appear before the INCLUDE command(s) in order for the PSEGs and DSEGs from the included modules to be placed at the specified locations. If you omit the DATA command, the DSEGs from the included modules are placed after the PSEGs in the program area. If an INCLUDE command appears before the PROGRAM and DATA commands, unpredictable results can occur.

The COMMON command defines the starting address for a common area. This area contains only specified CSEGs. You must specify the name of a CSEG in a COMMON command in order to place that CSEG in a specified common area. You can specify more than one CSEG in the COMMON command, and you can also use more than one COMMON command in the control stream. CSEGs that are not specified in any COMMON command are placed after the last DSEG encountered. Thus, in this example, only CSEG COM2 is placed at address >E000. CSEG COM1 and CSEG COM3 are placed after the DSEG for module MODD.

You can also use several PROGRAM and DATA commands to define multiple program and data areas in the linked output. The following control stream shows the use of multiple PROGRAM, DATA, and COMMON commands:

```
LIBRARY    VOL1.OBJ.ABS
TASK       PROG2
PROGRAM    >2000
DATA       >C000
INCLUDE    (MODA),(MODB)
PROGRAM    >4000
DATA       >D000
INCLUDE    (MODC),(MODD)
COMMON     >E000,COM2
COMMON     COM2,COM1,COM3
END
```

This control stream defines two program areas and two data areas. The first program area starts at address >2000 and includes only the PSEGs from MODA and MODB. The second program area starts at address >4000 and includes the PSEGs from MODC and MODD. The first data area, starting at address >C000, contains the DSEGs from MODA and MODB; the second data area, starting at address >D000, contains the DSEGs from MODC and MODD. If you omit the DATA commands, the DSEGs follow the PSEGs in the corresponding program area.

This control stream also shows the continuation of a common area. The first COMMON command defines the starting address of the common area and specifies that CSEG COM2 be placed in this area. The second COMMON command continues the same common area by using a previously specified CSEG rather than a starting address as the first operand. In this case, CSEG COM1 and CSEG COM3 are placed after CSEG COM2.

Using either of the previous control streams, the Link Editor produces one output module. In some cases, you may require multiple output modules (for example, to program multiple PROM or EPROM devices with different parts of the same program). To obtain multiple output modules, you must use the ABSOLUTE command and PHASE commands in the control stream.

The ABSOLUTE command specifies a special syntax definition for the PHASE command. When used with the ABSOLUTE command, the PHASE command specifies the beginning of a new output module, but it does not define an overlay. In this case, the PROGRAM command becomes a required operand of the PHASE commands rather than a separate command. You can specify the starting address of the data area(s) by using separate DATA commands or optional DATA operands in the PHASE commands. If you use DATA operands, you must update the starting address of the data area in each PHASE command. You can also use separate COMMON commands. The following control stream shows the use of the ABSOLUTE and PHASE commands. Figure 7-5 shows the memory configuration for this example.

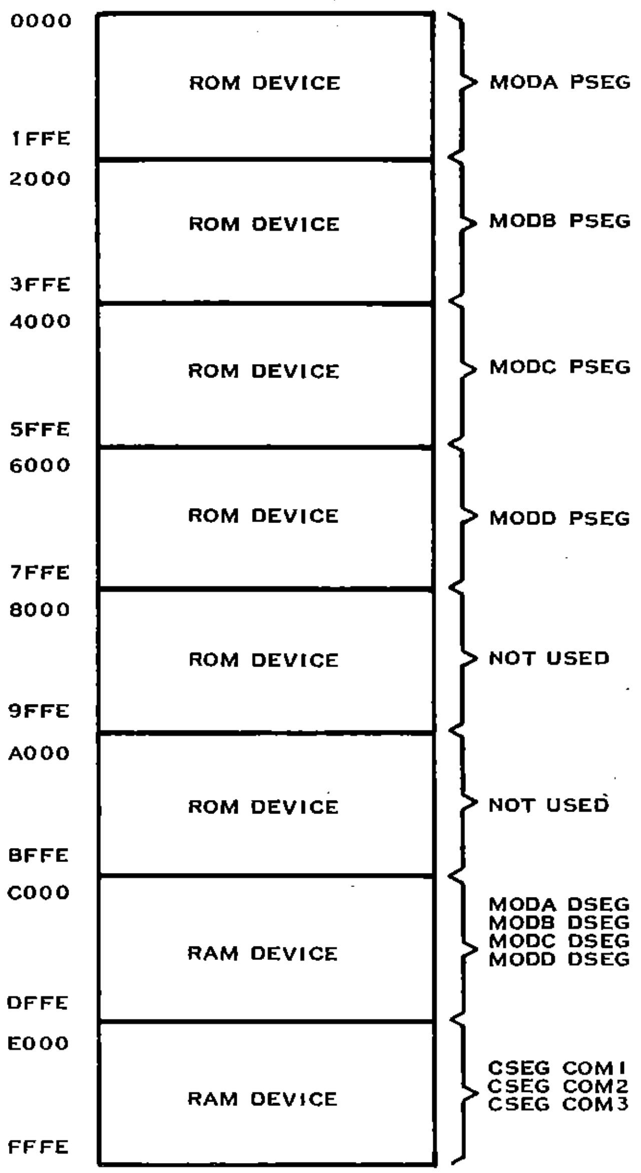
```
LIBRARY     VOL1.OBJ.ABS
ABSOLUTE
COMMON      >E000,COM1,COM2,COM3
PHASE       0,AAAA,PROGRAM >0000,DATA >C000
INCLUDE     (MODA); MODA DSEG requires >200 bytes of RAM
PHASE       1,BBBB,PROGRAM >2000,DATA >C200
INCLUDE     (MODB); MODB DSEG requires >600 bytes of RAM
PHASE       2,CCCC,PROGRAM >4000,DATA >C800
INCLUDE     (MODC); MODC DSEG requires >100 bytes of RAM
PHASE       3,DDDD,PROGRAM >6000,DATA >C900
INCLUDE     (MODD)
END
```

Although the use of DATA operands or separate DATA commands is not required, you should use one of the two to specify starting addresses for the DSEGs. Otherwise, the DSEGs start at address 0000, which may overwrite a PSEG assigned to that location. In addition, you should establish only one path in the control stream (define only one phase at each level) so that symbols are resolved correctly.

Using this control stream, the Link Editor produces one output module for each phase defined. Each output module contains the PSEG from one of the example object modules, starting at the address specified by the PROGRAM operand. All these addresses correspond to locations in ROM devices. The DATA operand in each PHASE command specifies the starting address for the DSEG for that module, which corresponds to a location in a RAM device. Notice that in each successive PHASE command, the DATA operand is updated to allow space for the DSEG from the previously included module. (The comments following the INCLUDE commands indicate the number of bytes of RAM required by that module.)

When working with absolute memory locations, make sure you specify the correct starting address for each area. You must also be aware of the amount of memory required for each area so that one area does not overflow into another area. Otherwise, data might inadvertently be destroyed.

Once you have built the control stream, you can execute the Link Editor with the Execute Link Editor (XLE) command as described in Section 3. The linked output must be written to a data file rather than to a program file.



2282943

Figure 7-5. Example Memory Configuration for Multiple Output Modules

7.4 READING THE LINK MAP

Figure 7-6 shows the listing file produced for the example program. The control stream used for this link contains PROGRAM, DATA, and COMMON commands. The listing file contains the same information as that for a single task structure. The PROGRAM, DATA, and COMMON commands cause the length of the segment to be listed as zero. The origins listed are absolute locations. An asterisk (*) following the value of a symbol name indicates an absolute location. Linking absolute code generated by the assembler (AORG assembler directive) also causes the length to be listed as zero and the origins to be absolute locations.

```
                                      04/16/82  09:54:17          PAGE    1
 COMMAND LIST

  LIBRARY    VOL1.OBJ.ABS
  TASK       PROG2
  PROGRAM    >2000
  DATA       >C000
  COMMON     >E000,COM2
  INCLUDE    (MODA),(MODB),(MODC),(MODD)
  END
```

Figure 7-6. Example Listing File — Absolute Memory Partitions (Sheet 1 of 3)

```
                                      04/16/82  09:54:17          PAGE    2
 LINK MAP

 CONTROL FILE = VOL1.CNTRL.ABS

 LINKED OUTPUT FILE = VOL1.ABSOBJ

 LIST FILE = VOL1.MAP.ABS

 OUTPUT FORMAT = ASCII

 LIBRARIES

 NO    ORGANIZATION    PATHNAME

 1     RANDOM          VOL1.OBJ.ABS
```

Figure 7-6. Example Listing File — Absolute Memory Partitions (Sheet 2 of 3)

```
                                              04/16/82  09:54:17           PAGE    3


PHASE 0, PROG2    ORIGIN = 0000  LENGTH = 0000

MODULE      NO   ORIGIN    LENGTH     TYPE        DATE        TIME       CREATOR

MODA        1     2000      001E    INCLUDE,1   04/16/82    09:39:00    SDSMAC
$DATA       1     C000      0020
MODB        2     201E      0010    INCLUDE,1   04/16/82    09:41:37    SDSMAC
$DATA       2     C020      0002
MODC        3     202E      002C    INCLUDE,1   04/16/82    09:50:12    SDSMAC
$DATA       3     C022      0022
MODD        4     205A      0012    INCLUDE,1   04/16/82    09:53:43    SDSMAC
$DATA       4     C044      0002


COMMON      NO   ORIGIN    LENGTH

COM2        4     E000      0004
COM1        4     C046      0002
COM3        4     C048      0002



                              D E F I N I T I O N S

 NAME     VALUE  NO     NAME     VALUE  NO     NAME     VALUE  NO     NAME     VALUE  NO

 MODB     201E*  2      MODC     202E*  3      MODD     205A*  4


****  LINKING COMPLETED
```

Figure 7-6. Example Listing File — Absolute Memory Partitions (Sheet 3 of 3)

8 Partial Linking

8.1 INTRODUCTION

Partial linking allows you to link only some of the modules required by a program at one time. The output of a partial link consists of a single module; this module is not executable and must eventually be linked with other modules to form an executable program. You can use partial links for the following purposes:

- To reduce the number of external symbols in a program. By resolving some of the external symbols in a partial link, you can reduce the amount of memory required by the Link Editor in a subsequent link.
- To link a complete set of modules that can be used by more than one program. Using partial links in this manner reduces execution time in each subsequent link. You can use the output of a partial link in either directory or sequential libraries. You must use partial links to build sequential libraries.

This section explains the structure of a partial link and provides an example. It also explains how to build the control stream and read the link map for a partial link.

8.2 PARTIAL LINK STRUCTURE

In a partial link, you must include all the modules within a single task segment; you cannot define a procedure segment, program segment, or overlay in the control stream. Also, you cannot specify image format since the output of a partial link is not executable. In a subsequent link where the entire program is being linked, you can place the output of a partial link in any segment or overlay defined.

As an example, this section uses three object modules to form an entire program. First, a partial link is used to link two of the object modules. Then, the output of the partial link is linked with the third module to form an executable program. Figure 8-1 through Figure 8-3 contain assembly listings of the example modules (MODA, MODB, and MODC). The examples assume that each module is contained in a separate file that has the same name as the module, under a directory named VOL1.OBJ.PART.

Modules MODA and MODB are used in the partial link. These modules contain PSEG, DSEG, and CSEG directives, as well as DEF and REF directives for a number of symbols. Module MODC is used in the subsequent link, along with the output of the partial link, to form the entire program.

The Link Editor resolves all the external references (REF tags) externally defined (DEF tags) by modules included in the partial link. Any references that cannot be resolved in the partial link retain the REF tags in the output. The Link Editor also provides commands that allow you to define the scope of external definitions (DEF tags) within a partial link. These symbols are defined as either global or local (not global).

The Link Editor retains the DEF tags for all global symbols in the output of the partial link. Thus, global symbols can be referenced by other modules in subsequent links. Unless you specify otherwise, the Link Editor treats all external definitions as global symbols.

Local symbols apply only to the current partial link. The Link Editor uses these symbols to resolve references to them within the partial link, but it does not retain the DEF tags for local symbols in the output. Symbols not required by modules in subsequent links should be specified as local. This reduces the symbol table size and the amount of memory required in subsequent links.

The Link Editor collects the PSEGs, DSEGs, and CSEGs from modules included in the partial link in the same manner as it does for single tasks. (Refer to Section 3.) The collection of each segment type is also tagged as a PSEG, DSEG, or CSEG(s) in the linked output. The output DSEG is the total of all input DSEGs (unless you also use the SHARE command). The Link Editor produces only one copy of a given CSEG, retaining the CSEG's original name.

```
 0001                          IDT  'MODA'
 0002                          DEF  MODS
 0003                          REF  SYM1,SYM2,SYM3,SYM4
 0004 0000                     PSEG
 0005 0000 C3E0  MDS000        MOV  @FLAG,R15
      0002 0000+
 0006 0004 06A0                BL   @SYM1
      0006 0000
 0007 0008 CBAF                MOV  @SYM2(R15),@SYM3(R14)
      000A 0000
      000C 0000
 0008 000E C820                MOV  @FLAG,@SYM4
      0010 0000+
      0012 0000
 0009 0014 0380                RTWP
 0010 0000                     DSEG
 0011 0000 0004"  MODS         DATA MDSWP,MDS000
      0002 0000'
 0012 0004        MDSWP        BSS  32
 0013 0000                     CSEG 'COM1'
 0014 0000 0000   FLAG         DATA 0
 0015                          END
NO ERRORS,      NO WARNINGS
```

Figure 8-1. MODA Assembly Listing — Partial Linking

```
 0001                          IDT  'MODB'
 0002                          DEF  SYM1, SYM2, SYM3
 0003 0000                     PSEG
 0004 0000 C3A0  SYM1          MOV  @DATA, R14
      0002 0000"
 0005 0004 05A0                INC  @DATA
      0006 0000"
 0006 0008 05A0                INC  @FLAG
      000A 0000+
 0007 000C C820                MOV  @DATA, @BUFF
      000E 0000"
      0010 0000+
 0008 0012 045B                RT
 0009      000A  SYM2          EQU  10
 0010      0004  SYM3          EQU  4
 0011 0000                     DSEG
 0012 0000 1000  DATA          DATA >1000
 0013 0000                     CSEG 'COM1'
 0014 0000 0000  FLAG          DATA 0
 0015 0000                     CSEG 'COM2'
 0016 0000       BUFF          BSS  2
 0017                          END
NO ERRORS,      NO WARNINGS
```

Figure 8-2. MODB Assembly Listing — Partial Linking

```
 0001                          IDT  'MODC'
 0002                          DEF  SYM4
 0003                          REF  MODS
 0004 0000                     PSEG
 0005 0000 0420  TOP           BLWP @MODS
      0002 0000
 0006 0004 C820                MOV  @BUFF, @SYM4
      0006 0000+
      0008 0000"
 0007 000A 10FA                JMP  TOP
 0008 0000                     DSEG
 0009 0000 0000  SYM4          DATA 0
 0010 0000                     CSEG 'COM2'
 0011 0000 0000  BUFF          DATA 0
 0012                          END
NO ERRORS,      NO WARNINGS
```

Figure 8-3. MODC Assembly Listing — Partial Linking

8.3 BUILDING THE CONTROL STREAM

The control stream for a partial link contains the same basic commands as that for a single task. In addition, the control stream must contain the PARTIAL command to specify a partial link. The PARTIAL command must appear before the TASK or PHASE 0 command. The following is a control stream for linking modules MODA and MODB in a partial link:

```
PARTIAL
LIBRARY     VOL1.OBJ.PART
TASK        PART1
INCLUDE     (MODA),(MODB)
END
```

Using this control stream, the Link Editor produces a single output module. The TASK (or PHASE 0) command assigns a name to the output module (PART1 in this example). The module name is important only if the output module is to be used in a sequential library. (Refer to Section 3 for more information on the use of libraries.)

The output module also contains a PSEG, DSEG, and CSEGs arranged in the following order:

```
MODA PSEG }
MODB PSEG } = PART1 PSEG
MODA DSEG }
MODB DSEG } = PART1 DSEG
      CSEG COM1
      CSEG COM2
```

By default, the Link Editor retains all the DEF tags in the output module. Notice from the assembly listings that MODA and MODB contain several external symbols (SYM1, SYM2, SYM3) that are not required by MODC. Therefore, you do not need to retain them in the output module. To suppress the DEF tags for these symbols in the output, you can declare the symbols as local, using the NOTGLOBAL command as follows:

```
PARTIAL
LIBRARY     VOL1.OBJ.PART
NOTGLOBAL   SYM1,SYM2,SYM3
TASK        PART1
INCLUDE     (MODA),(MODB)
END
```

Alternatively, if the modules contain more local symbols than global symbols, you can use the GLOBAL command in conjunction with the NOTGLOBAL command as follows:

```
PARTIAL
LIBRARY     VOL1.OBJ.PART
NOTGLOBAL
GLOBAL      MODS
TASK        PART1
INCLUDE     (MODA),(MODB)
END
```

Since the NOTGLOBAL command has no operands, it declares all symbols to be local. The GLOBAL command then declares only symbol MODS to be global. (Symbol MODS is the only external definition required by MODC in the subsequent link.)

Using this control stream, you can execute the Link Editor with the Execute Link Editor (XLE) command as described in Section 3. In a partial link, the linked output must be written to a data file (not a program file). This example assumes the output is written to a file named MODS under the VOL1.OBJ.PART directory, which is defined as a directory library.

The output of the partial link can then be linked with module MODC to produce an executable program. The control stream for this link is as follows:

```
FORMAT     IMAGE, REPLACE
LIBRARY    VOL1.OBJ.PART
TASK       PROG1
INCLUDE    (MODS),(MODC)
END
```

Remember that each partial link produces one output module; when a partially linked module is included in a subsequent link, all the modules included in the original partial link are also included.

8.4 READING THE LINK MAP

Figure 8-4 shows the listing file produced for the example partial link. This listing file contains the same information as that for single tasks (described in Section 3). In addition, the link map contains a listing of the global symbols. These are listed under the header GLOBAL SYMBOLS, as follows:

```
NAME   VALUE  NO   NAME   VALUE  NO   NAME   VALUE  NO
```

where:

NAME lists each external definition declared as global.

VALUE specifies the address within the link associated with the symbol.

NO indicates the number of the module in which the symbol is defined. You can map this number back to the module listings.

```
                                         04/16/82  10:17:45          PAGE    1
  COMMAND LIST

   PARTIAL
   LIBRARY    VOL1.OBJ.PART
   NOTGLOBAL
   GLOBAL     MODS
   TASK       PART1
   INCLUDE    (MODA),(MODB)
   END
```

Figure 8-4. Example Listing File — Partical Linking (Sheet 1 of 3)

```
                                         04/16/82  10:17:45          PAGE    2
LINK MAP

CONTROL FILE = VOL1.CNTRL.PART

LINKED OUTPUT FILE = VOL1.OBJ.PART.MODS

LIST FILE = VOL1.MAP.PART

OUTPUT FORMAT = ASCII

LIBRARIES

NO   ORGANIZATION   PATHNAME

1    RANDOM         VOL1.OBJ.PART
```

Figure 8-4. Example Listing File — Partical Linking (Sheet 2 of 3)

```
                                            04/16/82  10:17:45           PAGE    3

PHASE 0, PART1     ORIGIN = 0000  LENGTH = 0054

MODULE     NO   ORIGIN    LENGTH      TYPE        DATE        TIME       CREATOR

MODA       1     0000      0016    INCLUDE,1   04/16/82    10:08:00     SDSMAC
$DATA      1     0000      0024
MODB       2     0016      0014    INCLUDE,1   04/16/82    10:12:26     SDSMAC
$DATA      2     0024      0002


COMMON     NO   ORIGIN    LENGTH

COM1       2     0000      0002
COM2       2     0000      0002


                                D E F I N I T I O N S

 NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO

*MODS      0000   1     SYM1      0016   2     SYM2      000A* 2      SYM3      0004* 2

                            G L O B A L   S Y M B O L S

 NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO     NAME      VALUE NO

*MODS      0000   1

                      U N R E S O L V E D   R E F E R E N C E S

 NAME      COUNT NO     NAME      COUNT NO     NAME      COUNT NO     NAME      COUNT NO

 SYM4        1    1


****  LINKING COMPLETED
```

Figure 8-4. Example Listing File — Partical Linking (Sheet 3 of 3)

9

Error Reporting

9.1 INTRODUCTION

The Link Editor returns three different types of messages:

- Completion messages that indicate whether the Link Editor completed execution normally
- Error and/or warning messages that are written to the Link Editor listing file
- Error messages that are written to the terminal local file

Completion messages are displayed on the terminal when the Link Editor has completed execution. These messages are documented in the *DNOS Messages and Codes Reference Manual.* The Link Editor may complete execution normally and still generate errors and/or warnings.

Normally, when an error occurs, the Link Editor stops processing the control stream. If you use the ERROR command, the Link Editor does not process the line on which an error occurs but attempts to continue processing the control stream. In any case, the Link Editor produces either no linked output or incomplete linked output. You must correct the error(s) and relink before installing or executing the program.

When a warning occurs, the Link Editor continues to process the control stream and produces linked output. In most cases, you should correct the condition that caused the warning and relink before installing or executing the program. Otherwise, unpredictable results (possibly, a system crash) could occur.

The following paragraphs describe the error and warning messages that are written to the listing file and the terminal local file. When the Link Editor executes through an SCI batch stream or a command procedure, it also returns a condition code as the value of synonym $$CC. The possible values of $$CC have the following meaning:

0 —	No errors or warnings
4000 —	One or more warnings
8000 —	One or more errors
C000 —	Link Editor aborted (I/O error, end action, syntax error)

9.2 LISTING FILE MESSAGES

Error and warning messages appear in the listing file at the point at which they occur. Error messages are preceded by a string of *E characters, and warning messages are preceded by a string of *W characters. Table 9-1 explains the error messages, and Table 9-2 explains the warning messages.

Many of these messages have corresponding messages in the terminal local file. You should use the information presented in both places to determine the cause of the error and take corrective action.

If any errors or warnings occur, the second to the last line of the link map contains a total count of each type.

Table 9-1. Listing File Error Messages

CHECKSUM ERROR ENCOUNTERED ON RECORD *xx*, ACCESS NAME = *name*

Explanation:
The internal format of the input object module is faulty; *xx* is the number of the record where the error occurs and *name* is the name of the file containing the faulty input.

User Action:
Reassemble or recompile the module specified by *name*.

COMMAND EOF

Explanation:
An unexpected end-of-file marker was encountered in the control stream. A dollar sign ($) appears immediately under the error.

User Action:
Make sure there is an END command in the control file.

COMMAND SEQUENCE

Explanation:
Either the command is not in the proper sequence in the control stream or two incompatible commands appear in the control stream. A dollar sign ($) appears immediately under the error.

User Action:
Correct any semantically invalid constructions in the control stream.

DUPLICATE NAME

Explanation:
The same name has been used needlessly or ambiguously two or more times. The rest of the command record is ignored. A dollar sign ($) appears immediately under the error.

User Action:
Remove from the control stream all duplicate references to commons or external symbols.

Table 9-1. Listing File Error Messages (Continued)

ILLEGAL BACK CHAIN ACCESS NAME = *name* SYMBOL = *external symbol*

Explanation:
The internal format of the input object module is faulty; *name* is the name of the file containing the faulty input and *external symbol* is the symbol involved in the error.

User Action:
Reassemble or recompile the module specified by *name*.

ILLEGAL COMMON REFERENCE ENCOUNTERED ON RECORD *xx* ACCESS NAME = *name*

Explanation:
The internal format of the object module is faulty; *xx* is the number of the record where the error occurs and *name* is the name of the file containing the faulty input.

User Action:
Reassemble or recompile the module specified by *name*.

ILLEGAL DUMMY COMMAND, *name*

Explanation:
A DUMMY command appears after a procedure that was not dummied; *name* refers to the segment or phase in which the illegal DUMMY command appears.

User Action:
Either dummy the preceding procedure(s) or remove the DUMMY command from the control stream.

ILLEGAL LIBRARY NAME ACCESS NAME = *name*

Explanation:
The file or directory specified by *name* does not exist or cannot be opened.

User Action:
Use the I/O error code shown in the terminal local file to refer to SVC >00 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

ILLEGAL OVERLAY SEGMENT

Explanation:
Appears only when linking COBOL programs; the internal format of an input object module is bad.

User Action:
Recompile the modules containing COBOL segments.

ILLEGAL TAG ENCOUNTERED ON RECORD *xx*, ACCESS NAME = *name*

Explanation:
The internal format of the input object module is faulty; *xx* is the number of the record where the error occurs and *name* is the name of the file containing the faulty input.

User Action:
Verify that the files specified by *name* is an object file. If so, reassemble or recompile the module.

Table 9-1. Listing File Error Messages (Continued)

INTERNAL LINKER BUG, *xx*

Explanation:
A bug (*xx*) in the Link Editor has caused processing to terminate.

User Action:
Communicate the problem to your customer representative.

NO FIRST INPUT RECORD, ACCESS NAME = *name*

Explanation:
The file specified by *name* exists, but an end-of-file marker was encountered on the first read.

User Action:
Reassemble or recompile the module specified by *name*.

NO TASK COMMAND

Explanation:
The control stream does not contain a TASK or PHASE 0 command.

User Action:
Make sure there is either a TASK or PHASE 0 command in the control stream.

PREMATURE END OF FILE ENCOUNTERED, ACCESS NAME = *name*

Explanation:
The last record of the input object module is not a colon (:) record.

User Action:
Reassemble or recompile the module specified by *name*.

SIZE

Explanation:
Either a number exceeds the allowable range or a LIBRARY or INCLUDE command operand has too many characters. A dollar sign ($) appears immediately under the error.

User Action:
Correct the erroneous number or pathname in the control stream.

SYNTAX

Explanation:
A rule of syntax for Link Editor commands has been violated. A dollar sign ($) appears immediately under the error.

User Action:
Correct the syntax errors in the control stream. (Refer to Section 2 for correct syntax).

Table 9-1. Listing File Error Messages (Continued)

UNABLE TO ASSIGN OVERLAY ID

Explanation:
IMAGE format only. Indicates that no overlay IDs are available in the program file.

User Action:
Delete some overlays or use a different program file.

UNABLE TO BACKSPACE INPUT FILE ACCESS NAME = *name*

Explanation:
A backspace operation on a sequential library failed.

User Action:
Use the I/O error code shown in the terminal local file to refer to SVC >00 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

UNABLE TO CLOSE OUTPUT FILE ACCESS NAME = *name*

Explanation:
An error occurred while trying to clean up and close the output file.

User Action:
Use the error code shown in the terminal local file to refer to SVC >00, SVC >25, and SVC >27 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

UNABLE TO FIND PROCEDURE *name*

Explanation:
IMAGE format only. A procedure that was dummied does not exist on either the output program file or the S$SHARED program file.

User Action:
Make sure you have specified the correct program file or do not dummy the procedure.

UNABLE TO INSERT OVERLAY

Explanation:
IMAGE format only. An install operation failed. The error may have been caused either by a delete-protected segment or overlay or by failure to specify the REPLACE option in the control stream.

User Action:
Unprotect the segment or specify REPLACE. If these methods do not work, use the error code shown in the terminal local file to refer to SVC >00, SVC >25, SVC >26, and SVC >27 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

Table 9-1. Listing File Error Messages (Continued)

UNABLE TO INSERT PROCEDURE

Explanation:
IMAGE format only. An install operation failed. The error may have been caused either by a delete-protected segment or overlay or by failure to specify the REPLACE option in the control stream.

User Action:
Unprotect the segment or specify REPLACE. If these methods do not work, use the error code shown in the terminal local file to refer to SVC >00, SVC >25, SVC >26, and SVC >27 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

UNABLE TO INSERT TASK

Explanation:
IMAGE format only. An install operation failed. The error may have been caused either by a delete-protected segment or overlay or by failure to specify the REPLACE option.

User Action:
Unprotect the segment or specify REPLACE. If these methods do not work, use the I/O error code shown in the terminal local file to referto SVC >00 and SVC >26 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

UNABLE TO OPEN CONTROL FILE ACCESS NAME = *name*

Explanation:
The file specified by *name* does not exist or cannot be opened.

User Action:
Use the I/O error code shown in the terminal local file to refer to SVC >00 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

UNABLE TO OPEN INCLUDE FILE ACCESS NAME = *name*

Explanation:
The file specified by *name* does not exist or cannot be opened.

User Action:
Use the I/O error code shown in the terminal local file to refer to SVC >00 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

UNABLE TO OPEN OUTPUT FILE ACCESS NAME = *name*

Explanation:
The file specified by *name* does not exist or cannot be opened.

User Action:
Use the I/O error code shown in the terminal local file to refer to SVC >00 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

Table 9-1. Listing File Error Messages (Continued)

UNABLE TO READ CONTROL RECORD ACCESS NAME = *name*

Explanation:
A read operation on the control file failed.

User Action:
Use the I/O error code shown in the terminal local file to refer to SVC >00 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

UNABLE TO READ OVERFLOW RECORD

Explanation:
A read operation on a Link Editor temporary file failed.

User Action:
Use the I/O error code shown in the terminal local file to refer to SVC >00 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

UNABLE TO READ WORK RECORD

Explanation:
A read operation on a Link Editor temporary file failed.

User Action:
Use the I/O error code shown in the terminal local file to refer to SVC >00 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

UNABLE TO WRITE OUTPUT RECORD ACCESS NAME = *name*

Explanation:
A write operation to the output file failed.

User Action:
Use the I/O error code shown in the terminal local file to refer to SVC >00 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

UNABLE TO WRITE OVERFLOW RECORD

Explanation:
A write operation to a Link Editor temporary file failed.

User Action:
Use the I/O error code shown in the terminal local file to refer to SVC >00 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

Table 9-1. Listing File Error Messages (Continued)

UNABLE TO WRITE PROGRAM FILE RECORD ACCESS NAME = *name*

Explanation:
A write operation to the output file failed.

User Action:
Use the I/O error code shown in the terminal local file to refer to SVC >00 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

UNABLE TO WRITE WORK RECORD

Explanation:
A write operation to a Link Editor temporary file failed.

User Action:
Use the I/O error code shown in the terminal local file to refer to SVC >00 errors in the SVC section of the *DNOS Messages and Codes Reference Manual.*

Table 9-2. Listing File Warning Messages

ADDRESS SPACE OVERFLOW

Explanation:
The program counter (PC) value in the linked output exceeded >FFFF (64K).

User Action:
Reduce the size of the program by using overlays or program segments.

DUPLICATE IDT NAME

Explanation:
The IDT for the module matches a module already included in the link. This warning can occur when a module is included more than once within the same segment or phase in the control stream or when more than one module has the same IDT.

The Link Editor retains all original IDTs in the symbol table for the linked output (I tags). A duplicate IDT prevents full symbolic debugging. Refer to Appendix A for an explanation of the I tag.

User Action:
Determine the cause of the duplicate IDT. If the duplicate IDT is caused by a module being included more than once, correct the control stream. If the duplicate IDT is caused by different modules, change the IDT name for a module to a unique symbol and reassemble or recompile the module.

MULTIPLE SYMBOL DEFINITION

Explanation:
The listed symbol is defined more than once. It is assigned the value of the first occurrence.

User Action:
Determine the cause of the multiple symbol definition. In some cases, this may be desirable.

SHARE SPACE

Explanation:
When two or more modules share a data area in the linked output (see the SHARE command), the first module included must have the largest DSEG. This message warns that the DSEG from a subsequently included module is larger.

User Action:
Either include the module with the largest DSEG first or adjust the size of the DSEG in the subsequent module and reassemble or recompile the module.

9.3 TERMINAL LOCAL FILE MESSAGES

The terminal local file contains error messages only. Table 9-3 explains these messages. Some of the messages include the phrase CODE = *xxxx* ; Table 9-4 explains the value of *xxxx*.

Table 9-3. Terminal Local File Error Messages

BAD OBJECT FORMAT

Explanation:
An input module was in a format which the Link Editor could not understand.

User Action:
See listing file for more information.

CHECKSUM ERROR

Explanation:
Checksum did not verify on an input record.

User Action:
See listing file for more information.

CONTROL FILE I/O ERROR, CODE = *xxxx*

Explanation:
An I/O operation in the control file failed.

User Action:
See listing file for more information; refer to Table 9-4 for an explanation of the error code.

ILLEGAL TAG

Explanation:
The object code contains an illegal tag or is missing a zero tag.

User Action:
See listing file for more information.

INPUT FILE I/O ERROR, CODE = *xxxx*

Explanation:
Unable to open or access input file.

User Action:
See listing file for more information; refer to Table 9-4 for an explanation of the error code.

Table 9-3. Terminal Local File Error Messages (Continued)

INSUFFICIENT MEMORY REQUESTED

Explanation:
The link requested is too large to successfully complete.

User Action:
Reduce the number of external symbols or modules using partial links, or reduce the number of phases defined in the control stream.

INVALID LIBRARY NAME

Explanation:
A library specified could not be opened.

User Action:
See listing file for more information; refer to Table 9-4 for an explanation of the error code.

LINK EDITOR BUG

Explanation:
Error occurred within the Link Editor.

User Action:
See listing file and notify your customer representative.

LINK EDITOR TASK ERROR *xxxx*, WP = *nnnn*, PC = *nnnn*, ST = *nnnn*

Explanation:
Task error *xxxx* has occurred. WP indicates the value of the workspace pointer; PC indicates the value of the program counter; ST indicates the value of the status register.

User Action:
See the System Log Messages section of the *DNOS Messages and Codes Reference Manual* for an explanation of the task error code. If the condition is not correctable, contact your customer representative.

LIST FILE I/O ERROR, CODE = *xxxx*

Explanation:
Unable to open or write the list file.

User Action:
See Table 9-4 for an explanation of the error code.

MISSING OR MISPLACED COMMANDS

Explanation:
The link control file contains semantic errors.

User Action:
See listing file for more information.

Table 9-3. Terminal Local File Error Messages (Continued)

NO FIRST INPUT RECORD

Explanation:
The first read operation of an input file failed.

User Action:
See listing file for more information.

OUTPUT FILE I/O ERROR, CODE = *xxxx*

Explanation:
An I/O operation on the output file failed.

User Action:
See listing file for more information; refer to Table 9-4 for an explanation of the error code.

OVERFLOW FILE I/O ERROR, CODE = *xxxx*

Explanation:
An I/O operation on a Link Editor temporary file failed.

User Action:
See listing file for more information; refer to Table 9-4 for an explanation of the error code.

OVERLAY IMAGE ERROR, CODE = *xxxx*

Explanation:
An install operation failed.

User Action:
See listing file for more information; refer to Table 9-4 for an explanation of the error code.

PREMATURE END OF FILE

Explanation:
The last record of an input file was not a colon record.

User Action:
See listing file for more information.

PROCEDURE IMAGE ERROR, CODE = *xxxx*

Explanation:
An install procedure operation failed.

User Action:
See listing file for more information; refer to Table 9-4 for an explanation of the error code.

Table 9-3. Terminal Local File Error Messages (Continued)

SYNTAX ERROR

Explanation:
The link control file contains a syntax error.

User Action:
See listing file for more information.

TASK IMAGE ERROR, CODE = *xxxx*

Explanation:
An install operation failed.

User Action:
See listing file for more information; refer to Table 9-4 for an explanation of the error code.

UNABLE TO GET MEMORY

Explanation:
A Get Memory operation failed.

User Action:
Try the link again when more memory is available.

UNABLE TO LOAD OVERLAY

Explanation:
One of the overlays of the Link Editor could not be loaded.

User Action:
Verify that the .S$LANG program file has not been changed. If you find the file has not been changed, notify your customer representative about your situation.

WORK FILE I/O ERROR, CODE = *xxxx*

Explanation:
An I/O operation on a Link Editor temporary file failed.

User Action:
See listing file for more information; refer to Table 9-4 for an explanation of the error code.

Table 9-4. Error Codes

CODE = xxxx	Meaning
80xx or 00xx	DNOS, I/O Internal error code. Refer to the SVC section of the *DNOS Messages and Codes Reference Manual* for errors for SVC >00.
81FF	An attempt was made to suppress (dummy) the linked output of the task segment when image format is used.
8100	File is already open (Internal Link Editor error). Call your customer representative.
8102	File is not open (Internal Link Editor error). Call your customer representative.
nmxx	DNOS Internal error code. Refer to the SVC section of the *DNOS Messages and Codes Reference Manual* for errors for SVC >nm.

Appendix A

Object Code Format

A.1 INTRODUCTION

This appendix explains the format of the object code produced by either the assembler, a compiler, or the Link Editor. This information is not required for understanding Link Editor operation. However, it is helpful if you want to read object code to check certain errors or to determine if an object file is in valid format.

Figure A-1 shows an example object file containing two object modules. Object modules consist of two or more records. The records contain a number of tag characters; each tag character is followed by one to three fields. The first character of a record is the first tag character. The next tag character follows the end of the field or fields associated with the preceding tag. The end of each record is marked by an F tag, which terminates the object code portion of the record.

An identifier appears to the right of each record. The identifier consists of the first four characters of the module name and, in standard object format, four digits that specify the record number within the module.

Each object module is terminated by a colon (:) record. (The colon appears in the first column of the record.) The colon record indicates the date and time the module was created and the utility that created the module (LINKER for the Link Editor and SDSMAC for the assembler).

Table A-1 lists the tag characters and the field values associated with each tag. Unless otherwise noted in the table, the size of each field is either four characters (ASCII object format) or four binary digits (compressed object format). Variations in this size are noted by an integer value in parentheses following the field value description. The tags are listed in groups, where the group heading indicates the general function of the tags in that group. The following paragraphs explain the tags by functional group. The order in which these tags appear will vary from module to module.

```
0001EPROC1    A0000B0202C002AB0283B0001B1302B0202C002CBC4A0C00267F274F    PROC0001
B045BBC120C002CBA120C0028B045BI0000MODX     I0014MODY     7F3B4F          PROC0002
:                  04/15/82  16:32:36    LINKER        1.1.0              PROC0003
00064TSK1     A0020C004CC002EC0048A0026B1111B2222I0020MODA     A002E7F209FTSK10001
BD802C007EB06A0C006EB0203B0001B06A0C0000BC820C002AC002CB06A0C00147F1BFF TSK10002
B2FE0C006CA006CB0400BC0A0C007EBA0A0C007CBC802C0080B045BB0A00A007E7F161F TSK10003
B0000B0000B0000I002EMODB     I006EMODC     7F73AF                        TSK10004
:                  04/15/82  16:32:36    LINKER        1.1.0              TSK10005
```

Figure A-1. Example Object File

Table A-1. Object Code Tags

Tag	Field 1	Field 2	Field 3
Module Definition Tags:			
0	PSEG length	Module name (8)	—
M	DSEG length	$DATA	0000
M	Blank common length	$BLANK	Common #
M	CSEG length	Common name (6)	Common #
M	CBSEG length	$CBSEG	CBSEG #
Entry Point Definition Tags:			
1	Absolute address	—	—
2	P/R address	—	—
Load Address Tags:			
9	Absolute address	—	—
A	P/R address	—	—
S	D/R address	—	—
P	C/R address	Common or CBSEG #	—
Data Tags:			
B	Absolute value	—	—
C	P/R address	—	—
T	D/R address	—	—
N	C/R address	Common or CBSEG #	—
External Definition (DEF) Tags:			
6	Absolute value	Symbol (6)	—
5	P/R address	Symbol (6)	—
W	D/R or C/R address	Symbol (6)	Common #
External Reference (REF) Tags:			
3	P/R chain address	Symbol (6)	—
4	Absolute chain address	Symbol (6)	—
X	D/R or C/R chain address	Symbol (6)	Common #
E	Symbol index number	Absolute offset	—
Secondary External Reference Tags:			
V	P/R chain address	Symbol (6)	—
Y	Absolute chain address	Symbol (6)	—
Z	D/R or C/R chain address	Symbol (6)	Common #
U	0000	Symbol (6)	—

Table A-1. Object Code Tags (Continued)

Tag	Field 1	Field 2	Field 3
Symbol Definition Tags:			
G	P/R address	Symbol (6)	—
H	Absolute value	Symbol (6)	—
J	D/R or C/R address	Symbol (6)	Common #
Checksum and End-of Record Tags:			
7	Value	—	—
8	Any value	—	—
F	—	—	—
Load Bias Tag:			
D	Absolute address	—	—
Repeat Count Tag (FORTRAN Only):			
R	Value	Repeat count	—
Program ID (IDT) Tag (Link Editor Only):			
I	P/R address	Module name (8)	—
CBSEG Reference Tag (COBOL Only):			
Q	Record offset	CBSEG #	—

Notes:

P/R indicates program relocatable (within a PSEG).

D/R indicates data relocatable (within a DSEG).

C/R indicates common relocatable (within a CSEG).

CBSEG indicates COBOL segment.

A.2 MODULE DEFINITION TAGS

The module definition tags include the 0 tag and M tags. The 0 tag defines the program-relocatable code (or PSEG portion of the module). This tag consists of an ASCII zero in standard object code format and a binary one in compressed object code format.

The 0 tag is followed by two fields; field 1 contains the number of bytes required for the PSEG area for that module and field 2 contains the module name. For a Link Editor module, the module name is obtained from the name assigned in a PROCEDURE, TASK, PHASE, or SEGMENT command. For an assembler module, the module name is obtained from the IDT directive. When no IDT directive is used, the field contains blanks.

An M tag is used only when the module contains DSEGs, CSEGs, or CBSEGs (COBOL segments). The M tag is followed by three fields. Field 1 contains the number of bytes required for the DSEG, CSEG or CBSEG area for that module. Field 2 contains a six-character identifier, which is $DATA for DSEGs, $BLANK for blank (unnamed) CSEGs, and a defined name for named CSEGs. Field 3 consists of a four-character hexadecimal number defining a unique common number to be used by other tags that reference or initialize data in that particular CSEG. For DSEGs, this common number is always zero. For CSEGs (including blank CSEGs), the common numbers are assigned in increasing order, beginning at one and ending with the number of different CSEGs. Modules produced by the assembler can contain a maximum of 127 different CSEGs.

A.3 ENTRY POINT DEFINITION TAGS

The 1 and 2 tags define the entry points for a program. The 1 tag is used when the entry address is absolute and the 2 tag is used when the entry address is program relocatable. Field 1 of the tag contains the entry address in hexadecimal. One of these tags may appear in the object file. The field value is used by a ROM loader to determine the entry point at which execution starts when loading is complete.

A.4 LOAD ADDRESS TAGS

The 9, A, S, and P tags are used with load addresses for data that follows. A load address is required for a data word that is to be placed in memory at some address other than the next address. A 9 tag is used when the load address is absolute; an A tag is used when the load address is program relocatable; an S tag is used when the load address is data relocatable; and a P tag is used when the load address is common relocatable.

Field 1 of these tags contains the address at which the following data word is to be loaded. Field 2 of a P tag contains a common number.

A.5 DATA TAGS

The B, C, T, and N tags are used with data words. The B tag is used when the data is absolute, that is, an instruction word or a word that contains text characters or absolute constants. The C tag is used for a word that contains a program-relocatable address. The T tag is used for a word that contains a data-relocatable address. The N tag is used for a word that contains a common-relocatable address.

Field 1 of these tags contains the data word. This data word is placed in the memory location specified in the preceding load address field or in the memory location that follows the preceding data word. Field 2 of an N tag contains a common number.

A.6 EXTERNAL DEFINITION (DEF) TAGS

The 5, 6, and W tags are used for external definitions (DEF symbols). The 5 tag is used when the address of an external definition is program relocatable. The 6 tag is used when the address is absolute. The W tag is used when the address is data or common relocatable.

The field values for these tags provide linking information for these external definitions. Field 1 contains the address and field 2 contains the external symbol. Field 3 of a W tag contains a common number.

A.7 EXTERNAL REFERENCE (REF) TAGS

The 3, 4, and X tags are used for external references (REF symbols). The 3 tag is used when the last appearance of the symbol is in program-relocatable code. The 4 tag is used when the last appearance is in absolute code. The X tag is used when the last appearance is in data- or common-relocatable code.

The field values for these tags provide linking information for external references. Resolution of a reference is done through a back chain operation. Each location in the chain points to the preceding appearance of the symbol. Field 1 contains the location of the last appearance of the symbol. Field 2 contains the symbol, and field 3 of an X tag contains a common number.

When field 1 of a 4 tag contains zero, it signifies that there is no back chain for the referenced symbol. Otherwise, the value corresponding to the referenced symbol is placed in the location specified. The specified location's previous value is used as a pointer to the next location in the chain until an absolute zero is encountered. This marks the end of the chain.

The E tag is also used for external references. An E tag is used when a nonzero quantity is to be added to a reference. Field 1 identifies the reference by occurrence in the object code (0, 1, 2, and so forth). Field 2 contains the value to be added to the reference after the reference is resolved. The value in field 1 is an index into references identified by 3, 4, V, X, Y or Z tags in the object code. The list is maintained by order of occurrence; that is, the first entry in the list is the symbol located in field 2 of the first 3, 4, V, X, Y, or Z tag. The index to that reference in the E tag would be 0000.

A.8 SECONDARY EXTERNAL REFERENCE TAGS

The V, Y, and Z tags are used for secondary external references. The V tag is used when the last appearance of the symbol is in program-relocatable code. The Y tag is used when the last appearance of the symbol is in absolute code. The V tag is used when the last appearance of the symbol is in data or common relocatable code.

Field 1 of these tags contains the location of the last appearance of the symbol and field 2 contains the symbol. Field 3 of a Z tag contains a common number.

The U tag is generated by the assembler LOAD directive, which is used with secondary references. The symbol specified is treated as an external reference. Field 1 contains zeros. Field 2 contains the symbol for which the loader will search for a definition.

A.9 SYMBOL DEFINITION TAGS

The G, H, and J tags are used when the symbol tables are included in the output modules. The G tag is used when the location or value of the symbol is program relocatable. The H tag is used when the location or value is absolute. The J tag is used when the location or value is data or common relocatable.

Field 1 of these tags contains the location or value of the symbol. Field 2 contains the symbol to which the location is assigned. Field 3 of a J tag contains a common number.

A.10 CHECKSUM AND END-OF-RECORD TAGS

The 7 tag precedes the checksum, which is an error detection word. The checksum is formed as the record is being written. It is the two's complement of the sum of the eight-bit ASCII values of the characters of the record, from the first tag of the record through the checksum tag.

The 8 tag is used to ignore the checksum, which is contained in field 1.

The F tag indicates the end of the record. It may be followed by blanks.

A.11 LOAD BIAS TAG

The D tag is used to specify a load bias. Field 1 contains the absolute address that will be used by the loader to relocate the symbols when loaded. The Link Editor does not accept modules containing a D tag.

A.12 REPEAT COUNT TAG (FORTRAN ONLY)

The R tag is used to initialize a number of data words in a FORTRAN module with the same value. Field 1 contains the value to be placed in the data words. Field 2 contains the number of data words that are to contain the value.

A.13 PROGRAM ID (IDT) TAG (LINK EDITOR ONLY)

The I tag specifies the name originally assigned to a module (prior to linking). The Link Editor preserves the original name so it can be used during debugging. Field 1 contains the starting address of the module in the link. Field 2 contains the original module name (assigned by an IDT directive or program ID statement).

A.14 CBSEG REFERENCE TAG (COBOL ONLY)

The Q tag is used to access referenced CBSEGs (COBOL segments). Field 1 contains a record offset. Field 2 contains the CBSEG number.

Appendix B

High-Level Language Information

B.1 INTRODUCTION

This appendix explains how the Pascal, COBOL, and FORTRAN compilers generate some of the object tags used by the Link Editor. The tags covered are as follows:

- IDT tags
- DEF and REF tags
- PSEG, DSEG, and CSEG tags.

This information is provided so you can relate the material and examples covered in this manual to a specific high-level language. You should also refer to the appropriate programmer's guide when linking programs written in these languages.

B.2 PASCAL COMPILER

The Pascal compiler generates one object module for each procedure or function declared in the program. The IDT tag is obtained from the entry point name in the declaration.

The compiler automatically generates a DEF tag from the entry point name in each declaration. Pascal modules cannot have multiple entry points as in assembly language programs. Procedure or functions that are referenced but not included in the program must be declared as EXTERNAL. The compiler also automatically generates a REF tag for each procedure or function referenced.

The executable statements in a Pascal program form the PSEGs. The COMMON declarations are tagged as CSEGs. Pascal programs do not contain DSEGs since data allocation is done dynamically.

B.3 COBOL COMPILER

The COBOL compiler generates one or more object modules for a program, depending on the structure. The IDT tag is obtained from the PROGRAM-ID paragraph.

The compiler generates a DEF tag for all defined modules and a REF tag for all CALL statements.

The interpreted statements and constant data comprise the PSEG in a COBOL program. The DSEG is formed by the Data Division and other modifiable data in the program. COBOL programs do not contain CSEGs.

B.4 FORTRAN COMPILER

The FORTRAN compiler generates one object module for each program unit defined. The IDT tag for the module is obtained from the program name or program unit name. If you do not supply a program name, the default $MAIN is used.

The FORTRAN compiler automatically generates a DEF tag for each program unit. REF tags are generated for subprogram calls.

The executable statements in a FORTRAN program form the PSEGs. Variable and array declarations are tagged as DSEGs. COMMON statements are tagged as CSEGs.

Appendix C

Command Syntax

This appendix provides a quick reference for the link control commands. The following is an alphabetical list of the commands, showing their syntax definitions. A brief description of syntax rules is given, following the command list.

Command List:

ABSOLUTE
ADJUST [*n*]
ALLGLOBAL
ALLOCATE
AUTO
COMMON *base*,*name*[, *name* . . .,*name*]
DATA *base*
DUMMY
END
ERROR
FIND [*name*. . .,*name*]
FORMAT {ASCII/COMPRESSED/IMAGE[,REPLACE][,*priority*]}
GLOBAL [*symbol*. . .,*symbol*]
INCLUDE [*name*. . .,*name*]   (see note 1)
LIBRARY *name*[,*name*. . .,*name*]
LOAD
MAP {REFS/NO'*string*'[,NO'*string*'. . .,NO'*string*']}
NOAUTO
NOERROR
NOLOAD
NOMAP
NOPAGE
NOSYMT
NOTGLOBAL [*symbol*. . .,*symbol*]
PAGE
PARTIAL
PHASE *level*,*name*[,PROGRAM *base*][,ID *n*]   (see note 2)
PROCEDURE *name*
PROGRAM *base*
SEARCH [*name*. . .,*name*]
SEGMENT *map*,*name*[,PROGRAM *base*][,ID *n*]
SHARE *name*,*name*[,*name*. . .,*name*]
SYMT
TASK [*name*][,PROGRAM *base*]

Notes:

1. If you do not specify a *name* operand, the object module(s) to be included must immediately follow the INCLUDE command in the control stream.

2. If you use the PHASE command with the ABSOLUTE command, the PROGRAM *base* operand is required and can be followed by an optional DATA *base* operand.

Syntax Rules:

1. Commands are entered by typing in the command name, followed by at least one blank and then any operands required.

2. Command names entered can be either the full name or only the first four characters of the name. This also applies to the PROGRAM operand in the PHASE, SEGMENT, and TASK command.

3. Multiple operands must be separated by commas.

4. Each command must be on a separate line (record).

5. Comments must be preceded by a semicolon (;). Comments can be on a separate line or they can follow a command and its operand(s).

6. Synonyms and/or logical names can be used to specify pathnames.

7. The notations used in the syntax definitions are as follows:

   - Items in uppercase must be entered exactly as shown except for the command names and PROGRAM operand, which you can enter in the four-character abbreviated form.

   - Items in lowercase italics indicate a type of operand. Replace this with a specific operand of the appropriate type.

   - Items in square brackets ([ ]) indicate optional operands; items not enclosed in square brackets are required.

   - Items in braces ({}) indicate a choice of enclosed operands. The choices are separated by slashes (/). You can enter only one of the choices.

   - An ellipsis (. . .) indicates that you can repeat the preceding operand as many times as necessary. You must separate the operands with commas.

Alphabetical Index

Introduction

HOW TO USE INDEX

The Index, table of contents, list of illustrations, and list of tables are used in conjunction to obtain the location of the desired subject. Once the subject or topic has been located in the Index, use the appropriate paragraph number, figure number, or table number to obtain the corresponding page number from the table of contents, list of illustrations, or list of tables.

INDEX ENTRIES

The following Index lists key words and concepts from the subject material of the manual together with the area(s) in the manual that supply major coverage of the listed concept. The numbers along the right side of the listing reference the following manual areas:

- Sections — Reference to Sections of the manual appear as "Sections x" with the symbol x representing any numeric quantity.
- Appendixes — Reference to Appendixes of the manual appear as "Appendix y" with the symbol y representing any capital letter.
- Paragraphs — Reference to paragraphs of the manual appear as a series of alphanumeric or numeric characters punctuated with decimal points. Only the first character of the string may be a letter; all subsequent characters are numbers. The first character refers to the section or appendix of the manual in which the paragraph may be found.
- Tables — References to tables in the manual are represented by the capital letter T followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the table). The second character is followed by a dash (-) and a number.

  Tx-yy

- Figures — References to figures in the manual are represented by the capital letter F followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the figure). The second character is followed by a dash (-) and a number.

  Fx-yy

- Other entries in the Index — References to other entries in the index preceded by the word "See" followed by the referenced entry.