Data General

Customer Documentation

AOS/VS Link and Library File Editor (LFE) User's Manual

AOS/VS
Link and Library File Editor
(LFE)
User's Manual

093-000245-02

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000245

Printed in the United States of America
Revision 02, August 1985

NOTICE

DGC reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN DGC AND ITS CUSTOMERS. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

AOS/VS
Link and Library File Editor
(LFE)
User's Manual
093-000245-00

Revision History:
Original Release - November 1980
First Revision - April 1984
Second Revision - August 1985

Effective with:
AOS/VS Rev. 6.00
Link Rev. 5.05

A vertical bar or an asterisk in the margin of a page indicates substantive change or deletion, respectively from the previous revision, except for Appendix B and C which are entirely new.

Preface

This manual describes Link and the Library File Editor (LFE), two fundamental utilities of the Advanced Operating System/Virtual Storage (AOS/VS).

The two utilities are described in one manual because they play related roles in program development. Link consolidates object modules and library files into executable program files. LFE creates, edits, and analyzes library files.

The Link sections of this manual provide information for a broad spectrum of Link users. Some users require little more from this manual than the fundamental Link command line; others want to know more about switches, partitions, and overlays to improve program performance; others need cross-linking information for AOS, MP/AOS, RDOS, RTOS, or DG/UX™; and others may want to assemble or compile modules with their own language processors and need to know Link's internal workings for compatibility.

The LFE sections are useful for programmers who want to create or manipulate libraries.

This manual also describes CONVERT and MKABS, two additional AOS/VS utilities that manipulate RDOS and RTOS files.

To comply with the varying needs of users, this manual organizes information as follows:

Chapter 1 Introduces the Link utility, briefly describes AOS/VS memory organization, gives an overview of Link functionality, and lists memory parameters recognized by Link.

Chapter 2 Describes the Link command line and lists the optional function and argument switches in tabular form.

Chapter 3 Explains attributes, partitions, data relocation, and stack definition. These topics are prerequisites to understanding the object block formats in Chapter 4.

Chapter 4 Describes the structure of the object blocks. This information is essential for programmers writing their own compilers or assemblers.

Chapter 5 Explains how to link 16-bit programs for execution under AOS/VS. (If you developed these programs under AOS, you must relink them with the AOS/VS Link to execute them under AOS/VS.) Provides cross-linking information for AOS and MP/AOS. This chapter also contains information on overlays and resource call resolution that can help you improve program performance.

Chapter 6 Introduces the Library File Editor (LFE), presents an overview of LFE functionality, and describes the structure of the library start and library end object blocks.

Chapter 7 Explains how to operate LFE, shows the LFE command line formats, describes each LFE function-letter, and lists LFE error messages.

Appendix A Explains how to cross-link 16-bit object modules to produce program files executable under the Real-time Disk Operating System (RDOS), and the Real-Time Operating System (RTOS).

Appendix B Describes the CONVERT and MKABS utilities. You will need CONVERT if you want use Link to produce a program file that can execute under RDOS or RTOS. You will need MKABS if you are linking RDOS or RTOS files for a paper-tape system.

Appendix C Explains how to cross-link 32-bit object modules to produce program files executable under DG/UX™, Data General's UNIX™ compatible operating system.

Appendix D Lists the Link error messages.

Appendix E Lists the symbol types Link recognizes, and shows two sample map (output) listings.

Appendix F Lists the AOS/VS relocation operations (values which the language processor supplies, and which Link uses to reposition relocatable code and data).

Glossary Defines important Link and LFE terms used in this manual.

Related Manuals

Since the information presented in this manual relates to other aspects of AOS/VS or other Data General operating systems, you will find the following manuals useful for reference:

for information on the Command Line Interpreter (CLI)

- *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS)* (093-000122)

for information on AOS/VS and assembly language programming

- *AOS/VS Macroassembler Reference Manual* (093-000242)
- *AOS/VS Programmer's Manual (Volume 1 "System Concepts")* (093-000335)
- *AOS/VS Programmer's Manual (Volume 2 "System Calls")* (093-000241)
- *Principles of Operation 32-Bit ECLIPSE® Systems* (014-000704)

for information on AOS and 16-bit object modules

- *AOS Macroassembler Reference Manual* (093-000192)
- *AOS Link and Library File Editor (LFE) User's Manual* (093-000254)

for information on the RDOS and RTOS operating systems and their relocatable loader utility

- *RDOS System Reference* (093-400027)
- *RDOS/DOS Assembly Language and Program Utilities* (069-400019)
- *Real-time Operating System (RTOS) Reference Manual* (093-000135)

for information on DG/UX, Data General's UNIX compatible operating system

- */UX™ Family Programmer Reference Manual* (093-701012)
- */UX™ Family Administrator Reference Manual* (093-701008)

for information on developing programs under MP/AOS

- *MP/AOS Macroassembler, Binder, and Library Utilities* (069-400210)

Reader, Please Note:

We use these conventions for command formats in this manual:

COMMAND required *[optional]...*

Where **Means**

COMMAND You must enter the command (or its accepted abbreviation) as shown.

required You must enter some argument (such as a filename). Sometimes, we use:

$$\left\{\begin{array}{l}\text{required}_1\\ \text{required}_2\end{array}\right\}$$

which means you must enter *one* of the arguments. Do not enter the braces; they only set off the choice.

[optional] You have the option of entering this argument. Do not enter the brackets; they only set off what's optional.

... You may repeat the preceding entry or entries. The accompanying explanation will tell you exactly what you may repeat.

Additionally, we use certain symbols in special ways:

Symbol **Means**

↵ Press the NEW LINE or carriage return (CR) key on your terminal's keyboard.

< > Sets off the paraphrase of a value or symbol supplied by Link or LFE.

) The AOS/VS CLI prompt.

Finally, in examples we use

THIS TYPEFACE TO SHOW YOUR ENTRY
THIS TYPEFACE FOR SYSTEM QUERIES AND RESPONSES.

Contacting Data General

- If you have comments on this manual, please use the prepaid Remarks Form that appears after the Index. We want to know what you like and dislike about this manual.
- If you need additional manuals, please use the enclosed TIPS order form (USA only), or contact your Data General sales representative.

End of Preface

Contents

Chapter 1 - Introduction to the Link Utility

Link's Role in Program Development 1-2
AOS/VS Memory Concepts 1-3
Ring Structure 1-3
Logical and Physical Memory Structures 1-4
Relocation: An Overview 1-10
Link's Output Files 1-12
Overlay File 1-13

Chapter 2 - Link Command Line

Library Files 2-2
The System Libraries 2-2
Switches 2-3
The /CLI Argument Switch 2-4
The /REVISION Function Switch Sequence 2-4
The /STACKSIZE Switch Sequence 2-5
Output File Directive Switches 2-5

Chapter 3 - Link Terms and Concepts

Partition Attributes 3-1
Absolute, ZREL, and NREL Partition Attributes 3-1
Shared and Unshared Attributes 3-2
Normal Base and Common Base Attributes 3-2
Aligned Attribute 3-4
Code and Data Attributes 3-4
Overwrite-with-Message and Overwrite-without-Message Attributes 3-4
Partition Types 3-4
Absolute Partition 3-5
User-Defined Partitions 3-6
Partition Relocation 3-6
Data Relocation 3-8
Relocation Entries 3-8
Stack Placement 3-12

Chapter 4 - Object Block Structures

Object Block Restrictions 4-3
Block Structure: Header 4-3
Title Block 4-4
Module Revision Block 4-5
Revision Block 4-6
Partition Definition Block 4-6
Data Block 4-9
Relocation Entries 4-12
Relocation Dictionary Entries 4-13
Bit Field Relocation Entries 4-13
Alignment Block 4-13
External Symbols Block 4-14
Entry Symbols Block 4-15
Local Symbols Block 4-16
Unlabeled Common Block 4-18
Task Block 4-18
Debugger Symbols Block 4-19
Debugger Lines Block/Lines Title Block 4-20
.DL File Structure 4-20
Filler Block 4-22
End Block 4-23

Chapter 5 - Linking 16-Bit Object Modules for AOS/VS, AOS and MP/AOS

Resources 5-2
Overlays 5-2
Movable/Immovable Resources 5-3
Resource Resolution 5-3
Resource Call Optimization 5-6
Partitions 5-8
Command Line 5-8
Overlay Designators 5-8
Switches and Switch Sequences 5-9
Object Module Differences 5-10
Address Information Block 5-10
Named Common Block 5-12
Linking for MP/AOS 5-12

Chapter 6 - Introduction to the Library File Editor (LFE)

Functional Overview 6-1
Library Start/Library End Blocks 6-3
Library Start Block 6-3
Library End Block 6-4

Chapter 7 - Using the Library File Editor

A 7-3
D 7-7
I 7-8
M & N 7-11
R 7-13
T 7-14
U 7-15
X 7-16
LFE Error Messages 7-16

Appendix A - Linking 16-Bit Modules for RDOS and RTOS

System Overview A-1
Cross-Linking Differences A-1
Preliminary Steps A-3
The Command Line A-4
Switch Differences A-4
RDOS and RTOS Overlays A-5

Appendix B - Using the CONVERT and MKABS Utilities

Using CONVERT B-1
Converting .RB Files B-2
Converting Relocatable Binary Libraries B-2
Using MKABS B-3

Appendix C - Linking 32-Bit Modules for DG/UX™

Cross-Linking Differences C-1
Cross-Link Command Line C-2
Incompatible Switches C-3
Different Switch Meanings C-3
After Cross-linking C-3

Appendix D - Link Error Messages

Appendix E - Link Symbol Types and Sample Map Listings

Symbol Types E-2
Sample Map Listings E-3

Appendix F - Relocation Operations

Tables

Table

1-1 AOS/VS Memory Parameters 1-6
1-2 Memory Parameters for 16-Bit AOS/VS, AOS, and MP/AOS 1-8
1-3 Memory Parameters for RDOS and RTOS 1-9
1-4 Link Output Files 1-12

2-1 Link Function Switches and Switch Sequences 2-6
2-2 Link Argument Switches and Switch Sequences 2-11

3-1 Predefined (Default) Partitions 3-5

4-1 Object Block Types 4-2
4-2 Object Block Order 4-3

5-1 .PR Files: 16-bit/32-bit Differences 5-1
5-2 Resource-to-Resource Relocation Operations 5-3
5-3 Resource Optimization Switches 5-6
5-4 Resource Call Resolution Table 5-7
5-5 Switches Exclusive to 16-Bit Modules 5-9

7-1 LFE Function-letters 7-2

A-1 Cross-Linking: Map Partition Parameters A-2

D-1 Link Error Messages D-3

E-1 Symbol Types E-2

F-1 Relocation Operations F-1

Illustrations

Figure

1-1 Link's Role in Program Development 1-2
1-2 ZREL and NREL Memory 1-5
1-3 Relocation by the Macroassembler 1-10
1-4 Relocation by Link 1-11

3-1 Normal Base Versus Common Base Partition Relocation 3-3
3-2 Relocation Base Tables 3-7
3-3 Standard Relocation Entry 3-9
3-4 Extended 16-Bit Relocation Entry 3-9
3-5 Extended 32-Bit Relocation Entry 3-9
3-6 Standard Relocation Dictionary Entry 3-10
3-7 Extended Relocation Dictionary Entry 3-10
3-8 Bit Field Relocation Dictionary Entry 3-11

4-1 Object Block Header 4-3
4-2 Title Block 4-4
4-3 Module Revision Block 4-5
4-4 Revision Block 4-6
4-5 Partition Definition Block 4-7
4-6 Data Block 4-10
4-7 Standard and Extended 16-Bit Relocation Entries 4-11
4-8 Standard and Bit Field Relocation Dictionary Entries 4-12
4-9 Alignment Block 4-13
4-10 External Symbols Block 4-14
4-11 Entry Symbols Block 4-15
4-12 Local Symbols Block 4-17
4-13 Unlabeled Common Block 4-18
4-14 Task Block 4-18
4-15 Debugger Symbols Block 4-19
4-16 .DL File Structure 4-21
4-17 Filler Block 4-22
4-18 End Block 4-23

5-1 Optimizable Resource Calls for Argument Passed in Line (Immovable Resource) 5-4
5-2 Optimizable Resource Calls for Argument Passed On The Stack (Immovable Resource) 5-5
5-3 Non-Optimizable Resource Calls for Argument Passed In Line (Movable Resource) 5-5
5-4 Non-Optimizable Resource Calls for Argument Passed On The Stack (Movable Resource) 5-5
5-5 Unpaired Target (.PTARG) Word 5-5
5-6 Address Information Block 5-10
5-7 Named Common Block 5-12

6-1 Library Entry Resolution 6-2
6-2 Library Start Block 6-3
6-3 Library End Block 6-4

7-1 Inserting Object Modules with I Function-Letter 7-9

E-1 Mini Map Listing (/L=filename) E-3
E-2 Full Map Listing (/L=filename and other listing options) E-5

Chapter 1
Introduction to the Link Utility

The Link utility consolidates one or more object modules into an executable program file. An *object module* is an assembled or compiled source module that consists of *object blocks*, groups of binary code produced by the assembler or compiler as it interprets your source code. (Chapter 4 describes the structure of object blocks.)

To appreciate Link's role, think of program development as a process consisting of the following steps:

1. Creating the source code with one of the AOS/VS text editors
2. Compiling the source code with a high-level language compiler, or assembling it with the AOS/VS Macroassembler utility (MASM)
3. Linking the resulting object modules into a program file
4. Executing the program, and if necessary, debugging it

Notice that the term *language processor* throughout this manual refers both to high-level language compilers and to MASM.

AOS/VS is a 32-bit operating system, but it allows both 16-bit and 32-bit processes to execute. You can use the AOS/VS Link utility to build 32-bit programs that will execute under AOS/VS or DG/UX™, Data General's UNIX™ compatible operating system. You can also use AOS/VS and AOS/VS Link to develop 16-bit programs for execution under Data General's AOS/VS, AOS, MP/AOS, RDOS, and RTOS operating systems. AOS/VS also lets you execute 16-bit programs developed under AOS or MP/AOS; in most cases, you need only relink an AOS or MP/AOS program to execute it under AOS/VS. (Some programs may also require reassembling or recompiling.)

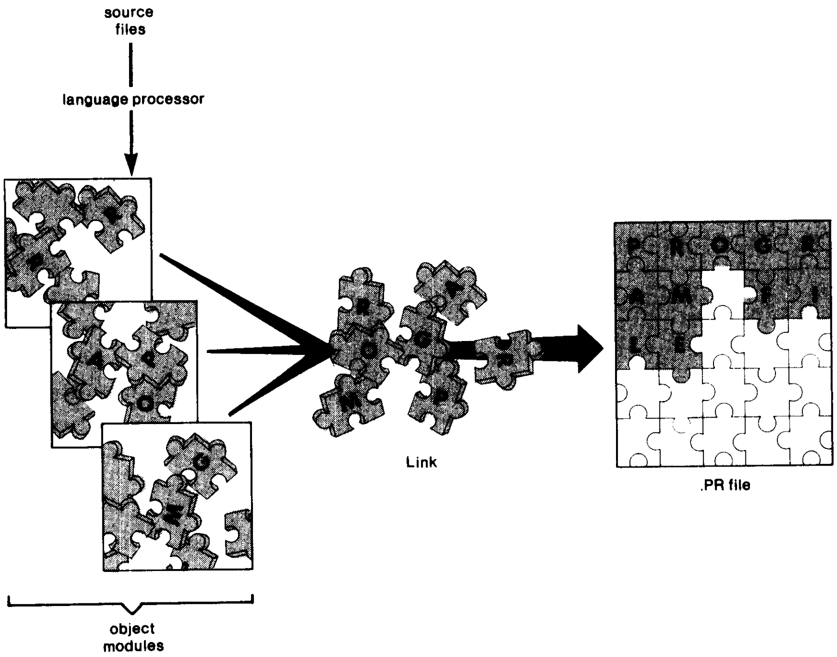
Link builds 16-bit program files differently from 32-bit program files. Chapter 5 describes these differences and explains how to link 16-bit programs. Appendix A describes differences unique to linking for RDOS and RTOS. Appendix C explains how to cross-link 32-bit programs for DG/UX.

Link's Role in Program Development

As part of the program development process, Link does the following:

- reorganizes sections of the object modules into partitions - groups of source statements with common attributes (i.e., characteristics);
- builds a symbol table for the names and values of all global symbols (symbols defined in one module and used in other modules);
- builds a memory map - an outline of each partition, its contents, and its general memory location;
- resolves and relocates (repositions) symbols, code, and data elements. (*Code* refers to executable instructions; *data* refers to constants, variables, etc.);
- builds system tables - databases that describe the program file's characteristics and its memory requirements;
- creates an executable program file, the .PR file, from the linked object modules;
- optionally creates other output files for execution and program debugging. (The creation of these files depends on the contents of the object modules and the Link command line.)

Figure 1-1 illustrates Link's place in the program development process.



SD-02187

Figure 1-1. Link's Role in Program Development

Link works in two passes. The following section outlines Link's actions during each pass and its action between passes. Subsequent chapters describe these steps in detail.

Pass	**Link's Action**
1	Scans the object modules in the command line and gauges the memory requirements of each module.
	Organizes portions of all object modules into partitions.
	Builds the symbol table for global symbols, and optionally, local symbols.
Interpass	Builds a memory map for the program.
	Assigns final relocation values to all partitions and symbols.
2	Relocates the contents of each partition, relative to each partition's relocation base (memory base).
	Resolves the contents of the data elements.
	Builds the system tables.
	Builds output files, including the .PR file.

Link supplies the operating system with a profile of the program and its memory resource requirements. AOS/VS performs the actual memory management and scheduling functions for each program at runtime.

AOS/VS Memory Concepts

AOS/VS is a demand paged, virtual operating system. The system's *demand paging* feature means that it moves only a subset of each process's logical address space into physical memory at any given time. The subset, known as the *working set*, changes in size and contents as the process demands (refers to) pages and relinquishes pages. *Logical address space* means the entire range of locations addressed in a program. Depending on the context, the term *page* refers to a 2K-byte portion of memory, or a 2K-byte portion of the logical address space.

In addition to user code, each process has its own copy of the operating system at runtime. This gives the process access to the system routines it needs.

Ring Structure

To prevent user processes from altering system routines, ECLIPSE® MV/Family processors feature a protection system based on hierarchical *rings*. There are eight rings, numbered from 0 through 7, where ring 0 is the innermost ring, and ring 7 is the outermost ring. Each ring is characterized by its function (i.e., whether it contains system code or user code) and its range of access to other rings.

Programs running in the inner rings can access programs in the outer rings without restriction. To transfer control from an outer ring to an inner ring, however, a program must have access to the *gates*, or legal entry points, for the inner ring. The *Principles of Operation, 32-Bit ECLIPSE® Systems* manual describes the assembly language instructions you use to define a gate array for a program running in an inner ring, and the instructions to address those gates from another program.

Although the physical context of the rings can change at runtime, their hierarchical relationship does not change. Rings 0 through 3 are the *system rings*; i.e., rings reserved for the operating system and related system routines. By default, user programs execute in ring 7, although rings 4, 5, and 6 are available to users as well. The Link function switch sequence /RING=n lets you direct a program to a specific ring. (See Chapter 2 for a description of the Link command line and a list of the legal switches.)

The ring structure affects relocatable addressing in a program since Link must resolve the ring field, as well as the address, of each relocatable reference. In addition, the ring structure affects the way Link builds the system tables for a program. At the end of its second pass, Link constructs the system tables in an 8K-word *preamble* to the .PR file. This preamble passes to one of the system rings at runtime. Therefore, you cannot change the values of the system tables.

The operating system supplies your program with a copy of one of the system tables - the User Status Table (UST) - but for reading purposes only. The UST contains such information as the number of tasks in the program, the program type, the size and starting address of the shared page area, and the addresses of any debugger routines. The operating system constantly updates this information as your program executes.

Logical and Physical Memory Structures

Both physical memory and the logical address space consist of two general types of pages: *shared pages* for code and data that more than one process can access, and *unshared pages* for code and data specific to each process. The shared page area occupies the highest portion of memory and grows *downward* toward numerically lower addresses. The unshared area occupies the lower end of memory and grows upward toward numerically higher addresses.

Between the shared and unshared areas there is an unused buffer area. You can use this area at runtime to augment the number of shared or unshared pages in the process's logical address space. The *AOS/VS Programmer's Manual* documents the system calls for this purpose.

Shared and unshared memory are further subdivided into the following categories:

- *lower page zero*, which by default extends from location 0 through 377_8, and consists of the following subsections:
  - an *absolute* area (for nonrelocatable references) extending by default from location 0 through 47_8
  - *ZREL*, or page zero relocatable, which by default extends from locations 50_8 through 377_8
- *NREL*, the normal relocatable area, which by default extends from location 400_8 to the highest address in memory

Note that the locations mentioned above are word addresses.

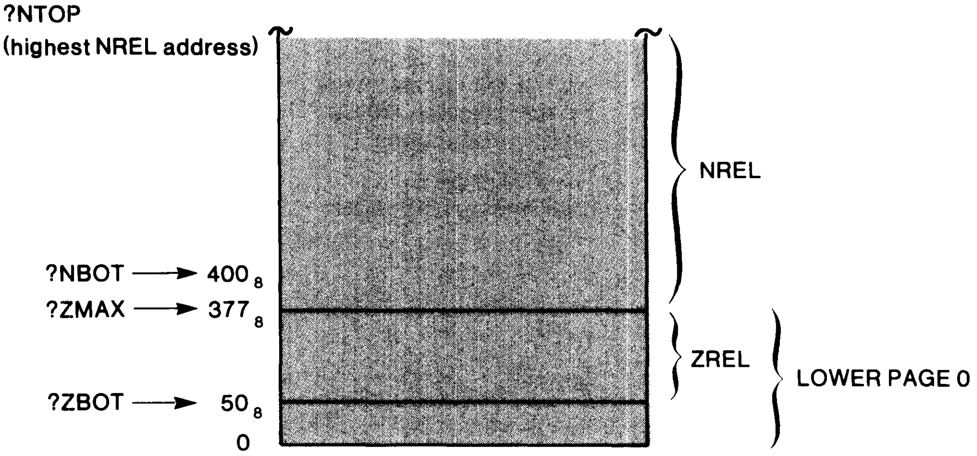
The assembly language pseudo-ops .NREL (with or without a numeric argument) and .ZREL earmark sections of your source code for the NREL and ZREL memory areas, respectively. The *AOS/VS Macroassembler Reference Manual* explains which pseudo-ops you should use for certain kinds of references. ZREL is unshared, and usually contains indirect addresses. NREL can contain both shared and unshared pages.

The mnemonics associated with the memory areas are predefined. For example, the symbol ?ZBOT, which has a default value of 50_8, always refers to the lowest available address in ZREL. The symbol ?NBOT, with a default value of 400_8, always refers to the lowest address in NREL. There are two ways to set the upper and lower boundaries of ZREL and NREL:

- by declaring the boundaries as entry symbols in your source code and defining values for them
- by declaring the ZREL and NREL boundaries using the appropriate function switch sequences in the Link command line

For information on declaring entry symbols, refer to the *AOS/VS Macroassembler Reference Manual*. Chapter 2 in this manual describes the Link function switches.

Figure 1-2 shows the relationship between ZREL and NREL and their default lower boundaries.



SD-02168

Figure 1-2. ZREL and NREL Memory

If you alter the memory boundaries, do not overlap the ZREL and NREL areas. Note that if you set ?ZBOT below its default location (50_8) or change the default value of ?NBOT, Link produces a nonexecutable program file.

If you do not alter the memory boundaries either in your source code or in the Link command line, Link assigns them the default values listed in Table 1-1. This table also briefly describes the memory parameters, and lists the function switches, if any, you can use to alter the values of the memory parameters.

Link uses the shared, unshared, ZREL, and NREL distinctions as well as other attributes to build partitions. Chapter 3 describes all of the partition attributes in greater detail.

Table 1-1. AOS/VS Memory Parameters

Symbol	Definition	Default Value	Switch or User Control
?CHAN	Number of channel control blocks to be allocated	Some language runtime libraries contain default values for this symbol (Refer to your language manual for more information.)	Generated by Link only if you use the /CHANNELS=n switch
?CLOC	Lowest address in unlabeled common area (Some compilers generate unlabeled common blocks. Chapter 4 describes this type of object block.)	-1, if there are no unlabeled common blocks	Defined by Link, no user control
?CSZE	Size of unlabeled common area	0	Declared in source code with the MASM pseudo-op .CSIZ
?LBOT	Base value of chain externals (EXTC links) See "External Symbols Block," in Chapter 4.	?UNDF	Declared in source code
?NBOT	Lowest NREL address	400_8	Declared in source code or defined with the switch /NBOT=n
?NMAX	Highest unshared NREL address that Link used plus 1	None	Defined by Link, no user control
?NTAS	Number of tasks	1	Declared in source code with the MASM pseudo-op .TSK or defined with the switch /TASKS=n
?NTOP	Highest available NREL address (i.e., the top of shared memory)	1777777_8	Declared in source code or defined with the switch /NTOP=n
?REV	Program revision	255.255.255.255	Declared in source code or defined with the switch /REVISION=n
?SBOT	Lowest address in NREL shared area	0, if there is no shared NREL	Defined by Link, but you can override with the switch /SBOT=n

(continues)

Table 1-1. AOS/VS Memory Parameters

Symbol	Definition	Default Value	Switch or User Control
?SRES	Size (in pages) of shared unlabeled common area	0	Declared in source code or defined with the switch /SRES=n
?UNDF	Value assigned to all symbols still undefined at the end of Link's first pass	-1	Declared in source code
?URTB	AOS/VS system code	A value contained in URT32.LB or URT16.LB	Defined in the system library, no user control
?USTA	Highest system table address used plus 1	None	Defined by Link, no user control
?ZBOT	Lowest ZREL address (i.e., the top of shared memory)	50_8 or 0, if you set the Link switch /UDF	Declared in source code or defined with the switch /ZBOT=n
?ZMAX	Highest address used in ZREL plus 1	None	Defined by Link, no user control
.MAIN	Starting address of the main routine	None	Declared in source code or defined with the switch /MAIN=
CFALT	Address of the commercial instruction fault handler	A value contained in URT32.LB or URT16.LB, holding the address of the default commercial instruction fault handler	Declared in source code
DEBUG	Address of SWAT® or other debugger interface	None	SWATI.OB or other debugger module
FFALT	Address of the floating-point instruction fault handler	A value contained in URT32.LB or URT16.LB holding the address of the default floating-point instruction fault handler	Declared in source code
IFALT	Address of the fixed-point instruction fault handler	A value contained in URT32.LB or URT16.LB, holding the address of the default fixed-point instruction fault handler	Declared in source code
SFALT	Address of the stack fault handler	A value contained in URT32.LB or URT16.LB, holding the address of the default stack fault handler	Declared in source code

(concluded)

Table 1-2 lists additional symbols and their memory parameters that Link might use when linking 16-bit modules to execute under AOS/VS, AOS, or MP/AOS. (See Chapter 5 for more information on linking 16-bit modules.)

Table 1-2. Memory Parameters for 16-Bit AOS/VS, AOS, and MP/AOS

Symbol	Definition	Default Value	Switch or User Control
??KCA	Calls the resource handler for ?KCALL	6013_8	Defined in URT16.LB, but you can control indirectly using resource call optimization switches (See the "Resource Resolution," section in Chapter 5.)
??RCA	Calls the resource handler for ?RCALL	6014_8	Defined in URT16.LB, but you can control indirectly using resource call optimization switches (See the "Resource Resolution," section in Chapter 5.)
??RCH	Calls the resource handler for ?RCHAIN	6015_8	Defined in URT16.LB, but you can control indirectly using resource call optimization switches (See the "Resource Resolution," section in Chapter 5.)
?AOS	One of the following: 0 for MP/AOS program, 1 for AOS or AOS/VS	None	Defined by Link, no user control
?CFLT	Address of the MP/AOS commercial instruction fault handler	A value contained in MP/AOS OSL.LB holding the address of the default MP/AOS commercial instruction fault handler	Declared in source code
?FFLT	Address of the MP/AOS floating-point instruction fault handler	A value contained in MP/AOS OSL.LB holding the address of the default MP/AOS floating-point instruction fault handler	Declared in source code

(continues)

Table 1-2. Memory Parameters for 16-Bit AOS/VS, AOS, and MP/AOS

Symbol	Definition	Default Value	Switch or User Control
?LODO	Address of the AOS overlay handler	None	Defined in AOS URT.LB
?SFLT	Address of the MP/AOS stack fault handler	A value contained in MP/AOS OSL.LB holding the address of the default MP/AOS stack fault handler	Declared in source code
STYP?	One of the following: 1 for AOS or AOS/VS, 2 for MP/AOS	None	Defined by Link, no user control
?TBOT	Lowest address in the PRSYM symbol table minus 1	0 if you do not set the /PRSYM switch	Defined By Link, no user control
?TTOP	Highest address in the PRSYM symbol table	0 if you do not set the /PRSYM switch	Defined by Link, no user control
?URHT	Address of the MP/AOS resource handler table	None	Defined by Link, no user control

(concluded)

Table 1-3 lists additional symbols and their memory parameters that Link can use when linking 16-bit modules to execute under RDOS or RTOS. (See Appendix A for more information on cross-linking RDOS and RTOS programs.)

Table 1-3. Memory Parameters for RDOS and RTOS

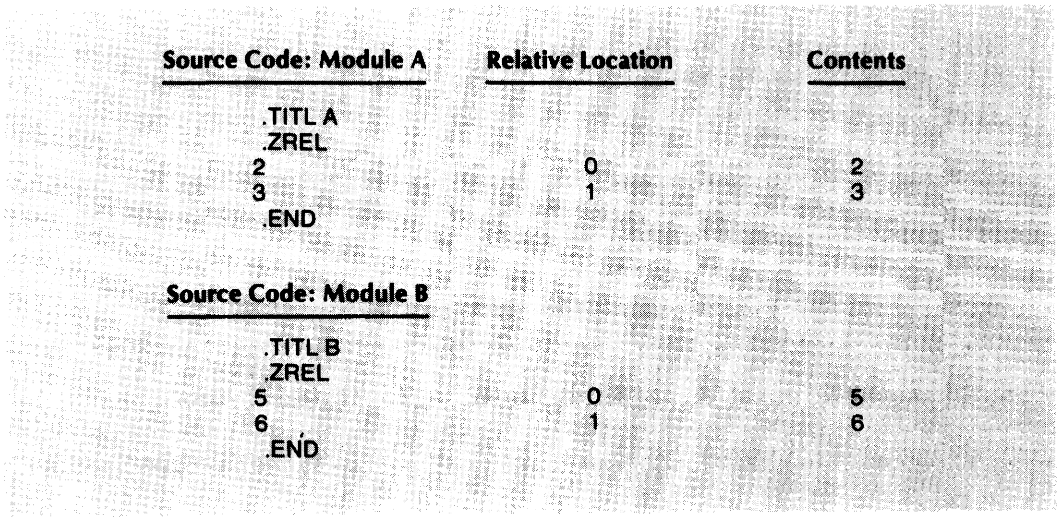
Symbol	Definition	Default Value	Switch or User Control
TMAX	Address of the RDOS multitask scheduler	None	Defined in RDOS SYS.LB
TMIN	Address of the RDOS single task scheduler	None	Defined in RDOS SYS.LB
USTAD	Address of the RDOS User Status Table	None	Defined by Link, no user control

Relocation: An Overview

Link's task of consolidating object modules into a composite .PR file requires *relocation*; that is, repositioning code and data relative to certain memory bases. This section briefly explains relocation and contrasts the relocation performed by the Macroassembler (MASM) with relocation performed by Link.

Blocks of relocatable code can reside anywhere within certain broad memory ranges at runtime. If code is relocatable, the relationship between the words is more important than their actual locations. In contrast, absolute code is locked to the absolute locations you define for it. Not all references in a source module need to be relocatable; you can declare some words for absolute locations by labeling them with the Macroassembler pseudo-ops .LOC and .GLOC. Neither MASM nor Link will relocate code or data defined for an absolute location.

Relocation occurs both at assembly or compile time and at link time. However, Link performs a more global kind of relocation than the Macroassembler, because Link handles more than one module at a time. In assigning addresses to code destined for the ZREL and NREL areas, MASM works from a temporary base address of 0 for each area. Figure 1-3 shows two hypothetical source modules, A and B, each of which places relocatable code within ZREL. If you assemble each of these modules separately, MASM relocates the code in each module as shown.



Source Code: Module A	Relative Location	Contents
.TITL A		
.ZREL		
2	0	2
3	1	3
.END		
Source Code: Module B		
.TITL B		
.ZREL		
5	0	5
6	1	6
.END		

SD-02169

Figure 1-3. Relocation by the Macroassembler

Given the same modules as those shown in Figure 1-3, and the command line X LINK A B, Link first determines the value of ?ZBOT, ZREL's base address. As we noted earlier, you can define this value from your source code or with the Link command line switch /ZBOT=n. If you do not define ZBOT, Link uses its default value, 50_8.

Once Link defines the ?ZBOT value, it uses it as the relocation base for the ZREL contributions from both module A and module B. Figure 1-4 shows how Link would relocate modules A and B if the value of ?ZBOT is 50_8, the default value.

```
Source Code:        Module A          Module B

                     .TITL A           .TITL B
                     .ZREL             .ZREL
                    2                 5
                    3                 6
                     .END              .END
```

Command Line: X LINK A B ⮐

After Relocation:

Module	Relative Location	Contents
A	50	2
	51	3
B	52	5
	53	6

SD-02170

Figure 1-4. Relocation by Link

In this example, Link uses location 50_8 as the *relocation base* for the ZREL portions of the two modules. It then relocates the ZREL elements as they appear in the source code. Link begins its relocation procedures with the first module you cited in the command line.

The preceding figures give simple illustrations of relocation. In reality, MASM and Link relocation is more complex. When the Link utility relocates code from the object modules, it considers not only the ZREL and NREL attributes, but other attributes as well. For example, Link must determine whether the code is shared or unshared, whether or not it is common to more than one module, and whether groups of elements should be aligned. A combination of these and several other attributes characterizes each partition.

Partitions first become important at assembly time, since you must be aware of the partitions and their attributes to use the Macroassembler's location counter and memory management pseudo-ops efficiently. (Chapter 3 describes in detail how Link handles partitions.)

Link's Output Files

Link's primary output is the .PR file, or program file. The .PR file is an image of the linked program as it will appear at runtime. The .PR file shows the contents of all partitions and their relative locations based on the predefined and user-defined memory boundaries.

Depending on the command line and the contents of the object modules, Link may also create one or more additional output files. Table 1-4 lists all of the possible output files for 32-bit object modules. You'll notice that you can control the creation or suppression of some of the output files with command line switches. For a complete list of Link's switches, refer to Chapter 2.

Table 1-4. Link Output Files

Output File	Description	Creation Conditions
.PR (program file)	is the core image of the linked program	always produced, unless /N function switch is present
.ST (symbol table) file	contains all global (cross-module) symbols used in the object modules, and, optionally, local symbols	always produced, unless /N or /SUPST function switch is present (To include local symbols, the modules must contain local symbols blocks, and you must modify the module names with the /LOCAL switch.)
.DS (debugger symbols) .DL (debugger lines) files	contains user-defined symbols for high-level language debugging.	produced only if one or more object modules contain debugger symbols blocks or debugger lines blocks, and you use the /DEBUG switch
error file	lists the errors encountered during the Link process.	always produced, controlled by the /E function switch
list file	lists the .PR file's symbols and memory parameters, supply statistical information on the modules and their memory use	lists primary memory parameters when you use the /L or the /L=filename switch in the Link command line. Full map produced when you use the /MAP switch. Other function switches direct Link to sort the symbols. (See Table 2-1.)

Link always produces an error file even if your input is error free. By default, Link sends the error file to the generic @OUTPUT file (e.g., your terminal in interactive mode). You can specify an alternate error file by using the /E=filename switch sequence in the command line. If you use either the /L switch or the /L=filename switch sequence, the utility also sends a copy of the errors to the list file.

Link reports errors as they occur. When it encounters an error, Link passes the error message to the error file. If you are executing Link in interactive mode, the error messages appear at your terminal. After the linking process terminates, the utility returns the message *LINK ERROR* to the calling procedure. (You can direct the error messages to the CLI pseudomacro [!STRING] by appending the /S switch to the XEQ command; i.e., XEQ/S LINK...).

If there are no errors during the linking process, the utility passes a null string to the error file, and sends the standard revision message *LINK REVISION <REV. NUMBER> ON <DATE> AT <TIME>* to the error file.

When you use the /L or /L=filename switch, Link creates a *map file* and a *mini map* and sends them to the list file. The map file lists all user-defined and system-defined symbols and the revision numbers of the object modules, if they have revision numbers. The mini map lists the memory parameters, their values, and the base values of all user-defined partitions.

The /MAP function switch directs Link to produce a *full map*, which lists all system table values and memory parameters. You can select one of several full map variations by using the map switches listed in Chapter 2.

The .DS (debugger symbols) and .DL (debugger lines) files enable you to debug your program with a high-level language debugger. Link creates a .DS or a .DL file when

- one or more object modules contain debugger symbols or debugger lines blocks, and
- you use the /DEBUG function switch in the Link command line.

Refer to Chapter 4 for descriptions of the debugger symbols and debugger lines blocks.

Overlay File

Link creates an overlay file (.OL file) when you link 16-bit object modules that contain overlays (defined in the source code or in the command line). Since overlays apply only to 16-bit modules, refer to Chapter 5 for information on overlays and the .OL file.

End of Chapter

Chapter 2
Link Command Line

The Command Line Interpreter (CLI) for the AOS and AOS/VS systems is your interface with the Link utility. The CLI command XEQ (also abbreviated X) followed by the argument LINK invokes the Link utility.

The format of the Link command line is

X*[EQ]* LINK*[fswitch...]*objectfile*[.OB]...[argswitch]*

where:

fswitch is one or more of the function switches and switch sequences listed in Table 2-1

objectfile is the name of an object module or a library file

.OB is the optional object module extension

argswitch is one or more of the argument switches and switch sequences listed in Table 2-2

Use one or more spaces, tabs, or commas to separate each object module name. Do not insert any spaces before the function and argument switches, or between them. In other words, if you use a function switch, make sure it is flush with the word LINK (e.g., X LINK/L). Similarly, if you use more than one function switch, or modify the same element with more than one argument switch, make sure there are no spaces between the switches (e.g., X LINK/L/E=errfile, and X LINK GREEN.OB/OVER/LOCAL).

The .OB extension after the object module name is optional. By default, Link searches your working directory and search list for matching object module names with the .OB extension. If it fails to find modules with this extension, it takes matching object module names without the extension.

Link takes the root filename for the program from the first module cited in the command line. For example, if the first object module is DGLPROG.OB, the .PR file will be DGLPROG.PR, the symbol table file will be DGLPROG.ST, and so on. You can specify an alternate root name by using the /O=filename switch sequence, where filename is the new root filename.

You can cite labeled common symbols and user-defined partitions in the command line, provided you modify them with the correct switches or switch sequences. Refer to Chapter 3 for details on user-defined partitions and Chapter 4 for an explanation of labeled common symbols.

Link also lets you use a CLI macroinstruction file (macro file) to pass arguments to the Link command. A *CLI macro file* contains a series of commands or command line elements that the CLI will execute when it encounters the macro filename. For instance, When developing a program that has many object modules, you can create a CLI macro file containing a list of Link input files. Then enter the macro filename, enclosed in brackets, as an argument to the X Link command (e.g., X LINK [macrofilename]). This eliminates the need to type a long list of object module names and their switches each time you execute Link. Refer to "The /CLI Argument Switch" section later in this chapter for more information on how Link reads CLI macro files.

Library Files

You can also use the names of library files in the Link command line. A *library file* is a series of object modules preceded by a library start block and terminated by a library end block. The AOS/VS Library File Editor utility, described in Chapters 6 and 7, creates, edits, and analyzes library files.

Before linking a library module into the .PR file, Link tests to see if the module is needed, as follows:

- Does the module contain a *forced load* flag? (This flag forces Link to include the module in the .PR file.) If so, link it to the .PR file. If not, proceed to the next test.
- Does the module satisfy any outstanding (unresolved) external symbols used in the other modules in the command line? If so, link it to the .PR file; otherwise, ignore the library module.

The assembly language pseudo-op .FORC generates a forced load flag. For more information on this pseudo-op, refer to the *AOS/VS Macroassembler Reference Manual.*

The System Libraries

At the end of its first pass, Link automatically scans the appropriate *system library* to resolve any outstanding external symbols used in the object modules. The system libraries, URT32.LB for 32-bit programs and URT16.LB for 16-bit programs, contain routines written to satisfy user requirements at runtime. The system libraries usually reside in the utilities directory (:UTIL), although you can place them in your working directory or in any directory in your search list. (Unless your working directory or a search list directory contains the system library, Link will be unable to find it.)

The switch sequence /SYS=n, shown in Table 2-1, governs which system library Link will scan. URT32.LB is the default system library. Link scans this library when you omit the /SYS sequence or when you use the form /SYS=VS32. When you use the /SYS=VS16 switch sequence, Link scans URT16.LB for the proper runtime modules.

The system library for programs running under AOS is URT.LB. Link scans this library when you use the /SYS=AOS switch sequence.

To suppress the system library scan entirely, use the /NSLS function switch in the command line.

The system libraries contain the following default error handling routines: SFALT, the default stack fault handler; CFALT, the commercial instruction fault handler; FFALT, the floating-point instruction fault handler; and IFALT, the fixed-point instruction fault handler. Unless you define your own error handlers, Link includes these routines in the program file. If you define alternate error handlers, label those routines with the appropriate names (i.e., SFALT, FFALT, CFALT, IFALT), and declare them as entry symbols in the appropriate module(s).

Switches

You can modify the linking process or Link's treatment of specific object modules by using switches and switch sequences in the command line.

A *switch* is a slash character followed by one or more alphanumeric characters. A *switch sequence* is a slash character followed by one or more alphanumeric characters, an equal sign (=), and another alphanumeric character or character string. For example, the string /SUPST is a switch, and the strings /O=filename and /NBOT=n are switch sequences. In the /NBOT=n sequence, "n" represents a numeric value.

When entering a switch or switch sequence, you can abbreviate the switch name. The shortest acceptable abbreviation is the smallest number of characters, that uniquely identifies the switch. For example, you can abbreviate the /SYSTEM=VS16 switch sequence to /SYS=VS16, or even to /SY=VS16. But you cannot use /S=VS16, because there are several other switches that start with the letter S. If you enter a switch that is not unique, Link returns an error.

By default, Link interprets any numeric values in the switch sequences as decimal values. To specify an alternative radix, add the designator R*n*, where *n* represents a radix from 2 through 9, immediately after the numeric value. For example, the function switch sequence /TASKS=10R8 signals Link that your program contains a maximum of 8 (10_8) tasks.

You can enter either signed or unsigned numeric values. For example, /NBOT=256, /NBOT=+256, and ABCD/VALUE=-5 are all valid entries.

Link also lets you enter numeric values in "K" notation; that is, Link multiplies by 1024_{10} any number with a K appended to it. For example, Link interprets the switch sequence /NBOT=32K as /NBOT=32768.

Link distinguishes between two kinds of switches and switch sequences.

- *Function* switches and switch sequences modify the entire Link process. In the Link command line, place function switches and function switch sequences immediately after the word LINK; for example, X LINK/DEBUG.
- *Argument* switches and switch sequences modify individual object modules. Therefore, you place argument switches and switch sequences immediately after the object module you want them to modify. In the command line X LINK GREEN/OVER BLUE, the argument switch OVER modifies only module GREEN.

The following sections explain in detail how to use particular Link function and argument switches. Table 2-1 lists alphabetically all the function switches and switch sequences and describes their actions. Table 2-2 lists argument switches and switch sequences. (When linking a 16-bit overlaid program, you can attach certain switches to the overlay delimiters. Table 5-4 describes switches exclusive to 16-bit modules.)

The /CLI Argument Switch

As mentioned earlier, you can use a CLI macro filename as an argument to the X LINK command. The /CLI switch is useful when the size of that macro file exceeds the maximum allowable size for a command line.

When creating a CLI macro file for Link input, you must use standard CLI syntax. CLI syntax requires you to separate arguments with a separator (i.e., one or more spaces, one or more tabs, or a single comma). The CLI lets you continue a command line to the following input line by typing an ampersand (&) before pressing the NEW LINE key.

The CLI provides several special characters for use in macros. These characters let you specify arguments that modify the macro or refer to other macros or pseudo-macros. When executing a macro, the CLI substitutes the actual arguments and macro expansions; therefore, Link sees only arguments and switches on its command line.

For complete information on CLI syntax and writing macros, refer to the *Command Line Interpreter (CLI) User's Manual (AOS and AOS/VS).*

To use your CLI macro file, enter the macro filename in the Link command line as follows:

XEQ LINK *[macrofilename]*

The CLI will try to expand the Link command line, by using the contents of the macro file as arguments to XEQ LINK. If a macro file is too large to fit on a command line, the CLI will return the error message

****NOT ENOUGH MEMORY, RESTARTING CLI****

If this happens, you can use the /CLI argument switch as follows:

XEQ LINK macrofilename/CLI

This directs Link to read the file as a CLI-format file, taking arguments from it one at a time until the file is exhausted.

Note that this CLI-format file cannot contain references to other macro files, pseudomacros, or macro arguments. The file can contain only Link arguments and must adhere to CLI command syntax.

The /REVISION Function Switch Sequence

The /REV function switch sequence tells Link to assign a particular revision number (of up to four levels) to the .PR file. This switch takes the form

/REV=ww*[.xx[.yy[.zz]]]*

where:

ww is the major revision number, in the range $0\text{-}255_{10}$

The *xx*, *yy*, and *zz* characters represent optional second, third, and fourth level revision numbers in the same range ($0\text{-}255_{10}$).

You can also define a two-part or four-part revision number for a module from your source code. The title block defines a maximum two-part revision level. The module revision block defines up to a four-part revision level. Refer to Chapter 4 for complete descriptions of these object blocks.

Link obtains the .PR file's revision number as follows:

1. First, Link checks for the /REV switch sequence in the command line. If this switch sequence is present, the utility assigns the program file the specified revision number.
2. If there is no /REV switch sequence, Link scans the modules and selects the first valid revision number; i.e., any revision number *not equal to* -1. Since the title block is always the first block in a module, Link always uses the first title block with a valid revision number, even if the module also contains a valid module revision block.
3. If there is no valid revision information, Link assigns the default revision number 255.255.255.255 to the .PR file.

The /STACKSIZE Switch Sequence

The /STACK=n function switch sequence lets you define the size of the user stack for your program's initial task.

The user *stack* is a block of consecutive memory locations you set aside in your program's logical address space to hold task-specific information.

By default, Link reserves a stack of 60 words (30 double words) for a 32-bit program's initial task, and a stack of 30 words for a 16-bit program's initial task. The /STACK=n switch sequence overrides this default and sets the stack size for the initial task to n words. Refer to Chapter 3 for information on stack placement. For more information on stack concepts, refer to the *AOS/VS Programmer's Manual* and the *Principles of Operation 32-Bit ECLIPSE® Systems* manual.

Output File Directive Switches

Several function switches and switch sequences tailor Link's output listings.

The /L function switch directs Link to produce a map file and mini map, and to send them to the current list file. The /L=filename switch sequence does the same, but designates filename as the list file for this link.

Similarly, the /E=filename sequence designates filename as the error file for this link. If you omit this switch, Link sends the error file to your current @OUTPUT file (i.e., your terminal, if you entered the command line from a terminal). If you use both the /E=filename and /L=filename sequences, Link sends the error messages to both the designated error file and the list file. If you use both of these switches in the command line, use a different filename for each one; otherwise, the files will overwrite each other.

The /MAP switch gives you a full map listing; i.e., a list of all symbols, system table values, and memory parameters. You must use either /L or /L=filename with the /MAP switch. You can tailor the full map listing with one or more of the following switches: /ALPHA, which directs Link to sort the symbols alphabetically; /MODSYM, which directs the utility to group the symbols, module by module; and /NUMERIC, which directs Link to sort the symbols by their numeric values. To obtain a full map listing sorted alphabetically and listed module by module, you would specify X LINK/L/MAP/MODSYM/ALPHA. If you use the /MODMAP switch, the utility returns a module-by-module map listing.

Table 2-1 describes switches that modify the entire linking process.

Table 2-1. Link Function Switches and Switch Sequences

Switch or Switch Sequence	Description
/ALPHABETIC	Produces an alphabetically sorted list of all symbols and their values. You must use /L or /L=pathname with this switch.
/BUILDSYSTEM	Produces an AOS system (.SY) program file. You must use /SYSTEM=AOS with this switch.
/CHANNELS=n	Generates the symbol ?CHAN which some languages use as a channel directive to set the number of available I/O channels. When you use this switch in combination with /SYS=RDOS or /SYS=RTOS, Link places n in offset USTCH of the User Status Table (UST). If you use the /SYS=RDOS or /SYS=RTOS switch without /CHANNEL=n, Link puts the default value 10_8 in offset USTCH.
/COMOVR	Directs Link to use an alternative memory allocation scheme compatible with other Data General 16-bit linkers and binders. You can use this switch only when linking modules for execution under 16-bit systems.
/DEBUG	Directs Link to create a .DL file if any object module in the Link command line contains one or more debugger lines blocks or lines title blocks. This switch also directs Link to create a .DS file if any object module line contains one or more debugger symbols blocks. (See Chapter 4 for details on object blocks.) /DEBUG causes Link to emit the external symbol DEBUG before the beginning of pass one, and to place the value of that symbol in offset USTDA in the User Status Table (UST).
/E=pathname	Appends Link errors to pathname. Without this switch, Link errors go to @OUTPUT.
/ELEMENTSIZE=n	Directs Link to set the file element size of the output program and overlay file to n. The default value is 32.
/ERRORCOUNT=n	Terminates Link with a fatal error if more than n Link errors occur. The default value is 1024.
/HEXADECIMAL	Converts all numbers in Link output listings and error files from the default (octal) to hexadecimal.
/KTOP=n	Limits the logical address space for the output file to n_{10} pages (where 1 page = 2000_8 addresses). The default value of n is 512_{10} for 32-bit systems and 32_{10} for 16-bit systems.
/L	Sends Link information to @LIST. By default, this information includes the titles of all input object modules (and their revision numbers) and the values of the basic memory parameters. Link also sends error messages to this file.
/L=pathname	Same as /L except that Link information goes to pathname rather than @LIST.
/LOCAL	Directs Link to place local symbols (defined in local symbols blocks) from all object modules into the output symbol table. This switch does not affect the .PR file; however, it may simplify debugging.
/MAP	Directs Link to produce a list of all default and user-defined partitions and send it to the list file. You must use /L or /L=pathname with this switch.

(continues)

Table 2-1. Link Function Switches and Switch Sequences

Switch or Switch Sequence	Description
/MODMAP	Directs Link to produce a more detailed version of the MAP (see /MAP above). The MODMAP lists each module's contributions to all default and user-defined normal partitions. You must use /L or/L=pathname with this switch.
/MODSYM	Directs Link to produce a list of each module's entry symbols and send it to the list file. You must use /L or /L=pathname with this switch.
/MTOP=n	Sets the logical address space for the output program to n megabytes. The default value of n is 1. You can use this switch only when linking for 32-bit systems.
/N	Suppresses the creation of Link output files, but does not suppress Link output listings (i.e., information stored with /L, /L=pathname, or /E). This switch is useful when you are not ready to execute or debug a program, but you want to know how Link will allocate space in the .PR file.
/NBOT=n	Changes the lowest NREL address from 00400_8 (the default address) to n_{10}. NOTE: A program file with an NBOT other than 400 (octal) will not be executable under AOS/VS.
/NRC	Directs Link to convert all resource calls into EJSR instructions (see "Resource Resolution" in Chapter 5). You can use this switch only when linking for 16-bit systems.
/NRP	Directs Link to convert certain resource calls in overlays with EJSR instructions. You can use this switch when linking 16-bit programs for AOS or AOS/VS (i.e., /SYS=AOS or /SYS=VS16 only).
/NSLS	Suppresses Link from scanning the appropriate system library. If you do not use this switch, Link automatically scans either URT32.LB or URT16.LB.
/NTOP=n	Sets the highest logical address (highest address in NREL) in the .PR file to n. The default value is 17777777_8 for 32-bit systems and 77777_8 for 16-bit systems.
/NUMERIC	Produces a list of all symbols sorted by their numeric values. You must use /L or /L=pathname with this switch. (/ALPHA provides the same information sorted alphabetically.)
/O=filename	Forces Link to name your output program file filename.PR. Without this switch, Link uses the first argument in the command line as the root filename for output files.
/OBPRINT	Directs Link to produce an octal dump of every object block, on a module-by-module basis, and send it to the list file. This switch is useful for examining object block structure; however, it usually generates much output. You must use /L or /L=pathname with this switch.
OVERWRITE	Suppresses overwrite error messages. Without this switch, Link sends out an error message if it overwrites an address. (See Chapter 3 for more information on the overwrite-with-message attribute.)

(continued)

Table 2-1. Link Function Switches and Switch Sequences

Switch or Switch Sequence	**Description**
/PADDING=n	Directs Link to increase the size of the output program file to a multiple of 2^n; where n is an integer from 0 through 10. Link will pad the program file so that it ends on a power-of-2 memory boundary. The default value for n is 10.
/PRSYM	Directs Link to create a symbol table in the output program file. You can use this switch only when linking for 16-bit systems.
/REVISION=w*[.x[.y[.z]]]*	Sets the revision number of the output program file to w.x.y.z; where w, x, y, and z each represents an integer from 0 through 255_{10}. x, y, and z are optional.
/RING=n	Sets ring n as the ring in which the program file will execute. n must be an integer from 0 through 7. Ordinarily, the default value for n is 7. However, if you set the /UDF switch the default value for n becomes 0.
/SBOT=n	Sets the lowest address in shared NREL to n.
/SRES=n	Reserves n shared pages (1 page = 2000_8 addresses) of memory, starting at the lower boundary of shared NREL (?SBOT), for program use. Link reserves n shared pages in addition to the existing shared pages in the shared partitions. You can set n from 1 to the number of pages in the program.
/STACKSIZE=n	Directs Link to create an initial default stack of n words. If you do not use this switch, Link creates a default stack of 60 words (30 double words) for a 32-bit program or 30 words for a 16-bit program. NOTE: Because Link does not build a stack for RDOS or RTOS program files, you cannot use this switch in a command line that contains either /SYS=RDOS or /SYS=RTOS.
/STATISTICS	Directs Link to report the following Link statistics: elapsed time CPU time number of input files number of modules number of symbols modules/elapsed second modules/CPU second
/SUPST	Suppresses Link's creation of the symbol table (.ST) file. .ST files are useful during debugging, but they do not affect program execution.

(continued)

Table 2-1. Link Function Switches and Switch Sequences

Switch or Switch Sequence	Description
/SYSTEM=n	Directs Link to build a program file that will execute under the operating system or mode of AOS/VS (VS16 or VS32) specified by n. By default Link will build a 32-bit program file for AOS/VS. You can use this switch to build the following types of program files: /SYS=AOS — 16-bit programs that execute under AOS. (See Chapter 5 for more information on linking 16-bit programs for AOS.) /SYS=DGUX — 32-bit programs that execute under DG/UX. (See Appendix C for more information on linking 32-bit programs for DG/UX.) /SYS=MPAOS — 16-bit programs that execute under MP/AOS. (See Chapter 5 for more information on linking 16-bit programs for MP/AOS.) /SYS=RDOS — 16-bit programs that execute under RDOS or SOS. (See Appendix A for more information on linking 16-bit programs for RDOS.) /SYS=RTOS — 16-bit programs that execute under RTOS or DOS. (See Appendix A for more information on linking 16-bit programs for RTOS.) /SYS=VS16 — 16-bit programs that execute under AOS/VS. (See Chapter 5 for more information on linking 16-bit programs for AOS/VS.) /SYS=VS32 — 32-bit programs that execute under AOS/VS. NOTE: Since AOS/VS is the development environment for the AOS/RT32 (Real-Time 32-Bit) operating system, you can link programs to execute under AOS/RT32 with the /SYS=VS32 switch, or by default with no /SYSTEM=n switch.

(continued)

Table 2-1. Link Function Switches and Switch Sequences

Switch or Switch Sequence	Description
/TASKS=n	Informs Link that the .PR file will contain n potential tasks. (Task specification affects construction of the system tables.) If you use this switch and the object modules already contain task blocks, Link compares n and the information in the task blocks, and takes the maximum task specification.
/TEMP=pathname pointer	Sends Link's temporary files to the directory specified by the pathname pointer. Link creates and deletes several files during the course of linking. By default, Link stores these temporary files in your working directory. This switch forces Link to create these files in a different directory. NOTE: This switch has no effect on AOS/VS Link; it is implemented only for compatibility with AOS Link.
/UDF	Directs Link to build a nonexecutable user data file (UDF). A UDF file is a program file without system tables, a stack, or any routines from the system library. Link builds each UDF file from location 0 to ?NTOP. It also constructs a symbol table (.ST file) for each UDF file.
/ULAST=n	Directs Link to place the contents of partition n in the highest unshared portion of the .PR file (directly before the default stack). n must be the name of a predefined partition (UC, UD, etc.) or user-defined partition. If it is neither, Link ignores the switch and returns an error message. The default for n is UC (unshared code).
/UNUSEDSIZE=n	Sets the lowest shared NREL address to NMAX+n.
/UNX	Sets the output program file type to UNX. A UNX type file executes under MV/UX. The default file type for output program files is PRV (16- or 32-bit programs for use under AOS/VS).
/V	Directs Link to report the full pathname of all input .OB files and library files. Without this switch, Link reports only the titles of input object modules. You must use /L or /L=pathname with this switch.
/WRL	Replaces certain resource calls in overlays with EJSR or EJMP instructions. You can use this switch with /SYS=AOS, /SYS=MPAOS, or /SYS=VS16 only. (See "Resource Call Optimization" in Chapter 5.)
/XREF	Directs Link to report every reference to each user-defined symbol, on a symbol by symbol basis. You must use /L or /L=pathname with this switch.
/ZBOT=n	Sets the lowest ZREL address (?ZBOT) to n. Ordinarily, the default value for n is 50_8. However, if you set the /UDF switch the default value for n becomes 0.
ZR ZR LD LD / UD = UD UC UC SD SD SC SC	Diverts all contributions destined for the default partition on the left side of the equal sign into the default partition on the right side of the equal sign. For example, /UC=SD diverts all contributions to the predefined Unshared Code (UC) partition into the predefined Shared Data (SD) partition. You can use these switches in any combination to change the attributes of the predefined partitions from the Link command line. NOTE: Do not use the same partition on both sides of the equal sign. For instance, /ZR=ZR generates an error.

(concluded)

Table 2-2 describes switches that modify individual command line elements (e.g., input files, partitions, etc.).

Table 2-2. Link Argument Switches and Switch Sequences

Switch or Switch Sequence	Description
/ALIGNMENT=n (Append to a partition name)	Aligns the relocation base of a partition specified in the command line on a power of 2 (2^n) word boundary, where n is an integer between 0 and 12_8 inclusive. You can append this switch to any labeled common symbol or named partition in the command line. (An alignment factor of 0 is the default and means the partition is word aligned.)
/CLI (Append to a filename)	Directs Link to read filename as a CLI-format file, taking arguments from it one at a time until the file is exhausted. For more information, see "The /CLI Argument Switch" section earlier in this chapter.
/DEBUG (Append to an object module)	Directs Link to include in the .DS and .DL output files, any debugger symbols blocks data, debugger lines blocks data, and lines title blocks data contained in an object file. (See Chapter 4 for details on object blocks.)
/FORCE (Append to a library file)	Forces Link to include all modules from a library (.LB) file into the .PR file. Without the /FORCE switch, Link includes only those modules that satisfy unresolved external symbols.
/LOCAL (Append to an object module)	Directs Link to place a module's local symbols (defined in local symbols blocks) into the output symbol table. This switch does not affect the .PR file; however, it may simplify debugging.
/MAIN (Append to an object module)	Directs Link to create entry (ENT) symbol .MAIN and sets the value of this symbol to this object module's starting address. For example: `) X LINK ONE.OB TWO.OB/MAIN ↵` If Link sets TWO.OB's starting address to 16000000567_8 in the .PR file, Link also sets the value of .MAIN to 16000000567.
/MULTIPLE (Append to a library file)	Directs Link to make as many passes, linking modules from a library file, as necessary to satisfy unresolved external symbols. Without the /MULTIPLE switch, Link makes only one pass over any library file.
/OVERWRITE (Append to an object module)	Cancels the overwrite-with-message attribute for a module. (This switch is the local equivalent of the /OVER function switch; see Table 2-1.)
/SHARED (Append to a partition name)	Changes a partition's *unshared* attribute to *shared*, and places the partition in shared NREL of the output program file.
/START (Append to an object module)	Directs Link to take the starting address of this object module as the starting address of the .PR file. Without this switch, Link takes the starting address of the last object module it encounters as the starting address of the .PR file.

(continues)

Table 2-2. Link Argument Switches and Switch Sequences

Switch or Switch Sequence	Description
/VALUE=n (Append to a symbol name)	Defines a symbol as an accumulating symbol (ASYM) and assigns the value n to it. n may be a either a numeric constant or a symbol name. If n is a symbol name, Link assigns its value to the accumulating symbol you are initializing. If an object module specified in the command line defines a symbol Y as an ASYM type symbol having the value *x*, Link resets the value of Y to x+n. For example, suppose that object module TEST defined a symbol Y having value 100_8. Assume that Y has symbol type ASYM. If you issue the command `)X LINK Y/VAL=40R8 TEST.OB ↵` Link redefines Y as an accumulating symbol and assigns the value 140 ($140_8 = 100_8 + 40_8$) to it. If TEST.OB had not defined Y, then the switch would have created Y, assigned it symbol type ASYM, and set its value to 40_8. If TEST.OB defined Y with a symbol type other than ASYM, this switch would have caused a *MULTIPLY DEFINED SYMBOL ERROR*.
$/\left\{\begin{array}{l}\text{ZR}\\\text{LD}\\\text{UD}\\\text{UC}\\\text{SD}\\\text{SC}\end{array}\right\}=\left\{\begin{array}{l}\text{ZR}\\\text{LD}\\\text{UD}\\\text{UC}\\\text{SD}\\\text{SC}\end{array}\right\}$ (append to an object module)	Diverts one module's contributions destined for the default partition on the left side of the equal sign into the default partition on the right side of the equal sign. For example, /UC=SD puts the module's contributions to the predefined Unshared Code (UC) partition into the predefined Shared Data (SD) partition of the output program file. You can use these switches in any combination to change the attributes of the predefined partitions in a *single* object module. NOTE: These argument switches override any conflicting function switches. For example, `) X LINK/SC=UC A/SC=UD ↵` assigns the unshared, data attributes to all shared, code partitions in module A, despite the /SC=UC function switch sequence. When using these switches as argument switches, you can enter the same partition on both sides of the equal sign (e.g., /ZR=ZR is a valid *argument* switch).

(concluded)

End of Chapter

Chapter 3
Link Terms and Concepts

Before Link reorganizes object modules to form the .PR file, it groups the elements from each module into logical categories called partitions.

Within your source code, partitions are groups of source statements with similar attributes. After the linking process, a partition is a distinct portion of the .PR file. At runtime, each partition corresponds to a particular portion of memory.

When you assemble or compile source modules, the language processor looks at the atttributes of each source statement, and builds partitions for that module accordingly. Link's task is to consolidate the partition assignments in all of the command line object modules (and any library modules) to create partitions for the program file.

Partition Attributes

A collection of the following attributes, or characteristics, distinguishes each partition:

- absolute, ZREL, short NREL, or long NREL
- shared or unshared
- normal base or common base
- aligned
- code or data
- overwrite-with-message or overwrite-without-message

The individual attributes in the groups listed above are mutually exclusive. For instance, a partition will never have *both* the normal base and the common base attributes. Similarly, a partition cannot be part of absolute, ZREL, and long and short NREL memory at the same time.

Absolute, ZREL, and NREL Partition Attributes

Link determines the general memory location of a partition based on whether the partition has the absolute, ZREL, long NREL, or short NREL attribute.

The language processor assigns the *absolute* attribute to partitions that contain nonrelocatable code or data; that is, code or data you have assigned to specific memory addresses. Absolute source code can reside in any area of memory except the unused area (between shared and unshared memory). If you want to use a block of unused memory for absolute references, you must first declare it to be part of NREL.

If a partition contains *relocatable* code or data, it has one of the following attributes: ZREL, long NREL, or short NREL.

By default, Link places predefined and user defined ZREL partitions between addresses 50_8 and 377_8. The predefined ZREL partition, defined by the language processor and by Link, has the *unshared* attribute. (ZREL contains unshared code or data.)

Partitions with the *long NREL* attribute can reside anywhere in NREL memory, from ?NBOT to ?NTOP (the top of memory). When you refer to locations in a long NREL partition, you must use instructions with 32-bit displacement fields.

Partitions with the *short NREL* attribute must reside in the lower part of memory, from ?NBOT to location 32K-1_8. You can use instructions with 16-bit or 32-bit displacement fields to refer to locations in a short NREL partition. Refer to the *AOS/VS Macroassembler Reference Manual* and the *Principles of Operation 32-Bit ECLIPSE® Systems* manual for more information on displacement fields and the 32-bit ECLIPSE instruction set.

Shared and Unshared Attributes

Partitions with the *shared* attribute contain shared code or data; that is, code or data that more than one process can access. At runtime, a program's shared partitions reside in shared NREL memory. Shared NREL is usually *write protected* to prevent alteration at runtime.

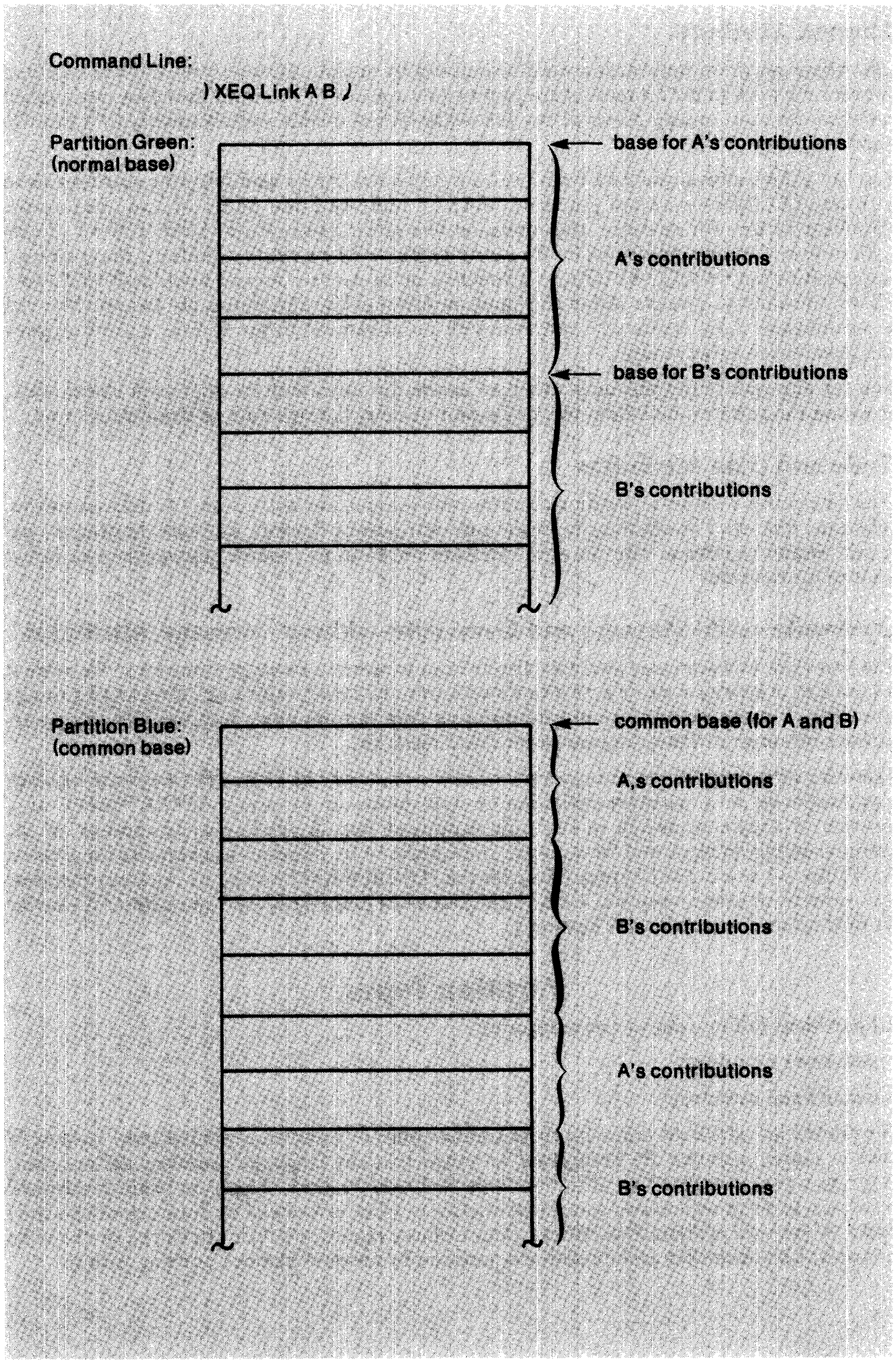
Partitions with the *unshared* attribute contain code or data reserved for the exclusive use of each process executing the program. Partitions of this kind reside in unshared memory (within ZREL or NREL) at runtime.

Normal Base and Common Base Attributes

The *normal base* and *common base* attributes enable Link to define relocation bases for each partition. Briefly, a partition's relocation base is its starting location in memory; Link displaces each word in a partition from the partition's relocation base.

If a partition has the common base attribute, Link treats its relocation base as the actual starting address for the partition. Consequently, Link assigns each partition segment a displacement value *relative to that common base*, regardless of which module the segment comes from.

If a partition has the normal base attribute, Link *normalizes* its relocation base in each module. That is, the utility assigns a *unique* relocation base to each partition segment on a module-by-module basis. Instead of interleaving, or mingling, the various segments of a normal base partition, Link concatenates them when it builds the .PR file. Figure 3-1 illustrates the difference between common base and normal base partitions.



SD-02171

Figure 3-1. Normal Base Versus Common Base Partition Relocation

Aligned Attribute

Link aligns partition elements on word boundaries by default. However, the *aligned* attribute directs Link to align the contents of the partition on a *power of two* word boundary. An aligned partition can start on any power of two boundary from a double word boundary (2^1) to a 1K word boundary (2^{10}).

You can align a named partition from the Link command line by modifying the partition name with the /ALIGN=n switch sequence, where *n* represents the power of two used as the alignment factor. For example, the command line specification PART1/ALIGN=1 directs Link to start the named partition PART1 on a double word boundary. Similarly, the command line specification PART2/ALIGN=10 directs Link to start the named partition PART2 on a 1K word boundary. You can assign an alignment factor of 0 to a partition, but this specification is meaningless. Link treats any partition with an alignment factor of 0 as a word aligned partition, which is the default.

You can also specify the alignment attribute internally, either with the alignment object block, or within the partition definition block. Chapter 4 shows the structure of these block types.

Code and Data Attributes

Link determines whether a partition contains code (executable instructions) or data (variables, constants, text, etc.) by checking it for the code attribute or the data attribute. In general, you should reserve partitions with the data attribute for data; i.e., source statements you do not want to be executed.

Overwrite-with-Message and Overwrite-without-Message Attributes

The *overwrite-with-message* attribute directs Link to return a message to the error file when it is forced to overwrite some of a partition's code or data during relocation. Overwriting occurs when two or more modules try to load different data into the same location. The *overwrite-without-message* attribute suppresses overwrite messages.

There are two alternate ways to suppress overwrite messages: by setting the overwrite message suppression bit in a partition definition or data block, or by using /OVERWRITE as a function or argument switch in the Link command line. Refer to the descriptions of the partition definition block and the data block in Chapter 4 for more on the overwrite suppression bit. Tables 2-1 and 2-2 in Chapter 2 describe the /OVERWRITE switch. (Link also suppresses the overwrite message when you use bit field relocation to partially load an address. We define bit field relocation later in this chapter.)

Partition Types

All partitions fall into one of two categories:

- predefined partitions
- user-defined partitions

The predefined partitions represent those combinations of partition attributes most frequently used in source modules. By *predefined* we mean that the language processor defines these partition types, and that the definition is carried over to Link. There are eight predefined partitions, seven of which have associated attributes. (One is reserved for future development.) Table 3-1 lists the predefined partitions, their memory ranges and their attributes. As the table indicates, Link identifies each predefined partition by an octal number, starting with 0.

Table 3-1. Predefined (Default) Partitions

Partition Number	Partition Name	Possible Memory Range	Attributes
0	AB (absolute)	0 through ?NMAX-1	absolute, common base, overwrite-with-message, data, alignment=0
1	ZR (ZREL)	?ZBOT through ?ZMAX-1	ZREL, unshared, normal base, overwrite-with-message, data, alignment=0
2	LD (relocatable data)	?NBOT through 77777	short NREL, normal base, overwrite-with-message, data, alignment=0
3	none	reserved	none
4	UC (unshared code)	?NBOT through ?NMAX-1	long NREL, unshared, normal base, code, overwrite-with- message, alignment=0
5	SD (shared data)	?SBOT through ?NTOP	long NREL, shared, normal base, data, overwrite-with-message, alignment=0
6	UD (unshared data)	?NBOT through ?NMAX-1	long NREL, unshared, normal base, data, overwrite-with- message, alignment=0
7	SC (shared code)	?SBOT through ?NTOP	long NREL, shared, normal base, code, overwrite-with-message, alignment=0

Absolute Partition

The predefined absolute partition contains words you have assigned to absolute locations. As Table 3-1 indicates, words within this partition can reside anywhere in shared or unshared memory at runtime. Therefore, the words associated with the absolute partition are not necessarily contiguous.

Link uses the base of the predefined absolute partition as the relocation base for all absolute addresses. As a result, the absolute partition is the only predefined partition with the *common base* atttribute. Note that there is a difference between *absolute addresses* and *absolute values*. You can assign a symbol an absolute value, for instance, but write your code so that the symbol itself is relocatable. When you define an absolute address, however, Link must resolve a ring field for that address. It does this by performing word relocation (a type of relocation operation) from the relocation base of the absolute partition.

User-Defined Partitions

User-defined partitions are the ZREL and NREL partitions you define in your source code, either with the assembly language pseudo-op .PART, or with a similar high-level language statement. The syntax of the .PART statement is

.PART partition-name <*attribute...*>

where the angle brackets, < >, mean that the attribute arguments are optional for every occurrence of .PART partition-name in a module except the first.

You must supply a name for each partition you define with .PART, and initially, one or more attributes for it. (You need to define the partition attributes only once per module, even if you refer to the same partition more than once.)

The attributes are those we defined in the previous section with two additional ones: *global* and *local*.

The *global* attribute allows you to define the same partition across object modules. If the partition segments in two or more modules have the same name, attributes, and the global attribute, Link regards all of them as the same partition.

Suppose, for example, you define partition NEWPART in object module A, and assign it the following attributes: long NREL, shared, common base, data, overwrite-with-message, and global. You can then define NEWPART with the same attributes in object module B. Given the command line X LINK A B, Link will use a common relocation base for both NEWPART segments. In building the .PR file, Link will append the contents of module B's NEWPART to those of module A's NEWPART. Note that each segment of a global partition must have the same attributes.

Conversely, you can assign the *local* attribute to NEWPART if you want Link to consider each NEWPART segment as a different partition. Link will then assign a different relocation base to each NEWPART segment, and will not consolidate the separate NEWPART segments in the .PR file. Consequently, two or more local partitions with the same name, in different modules, can have different attributes.

Partition Relocation

During pass one, Link scans each object module in the command line and determines the number and kind of partitions in each module. The utility then builds a temporary table of relocation bases for each module. (The language processor defines the relocation bases for each module.)

During this stage, Link assigns each partition an external number, starting at 0, on a module-by-module basis. The steps below describe the numbering scheme. (Note that the external numbers are binary values; we refer to them by their decimal equivalents for convenience.)

1. First, Link assigns external numbers 0 through 7 to the predefined partitions 0 through 7.
2. Next, Link assigns an external number to each distinct user-defined partition, starting with 8_{10}, and proceeding in ascending numeric order.
3. Finally, Link assigns an external number to each external symbol used in the module, starting with 8+n, and proceeding in ascending numeric order (where 8+n is the number assigned to the last user-defined partition).

External symbols are symbols used in one module, but defined in another module. Since the utility numbers user-defined partitions before external symbols, a module's partition definition blocks must precede its external symbol blocks. (Refer to Chapter 4 for details.)

Figure 3-2 shows temporary relocation base tables Link would construct for two sample modules. Notice that the external numbering is done on a module-by-module basis during this phase of the utility's operation. Thus, even though the same user-defined partition, NEWPART, appears in both modules, Link assigns it a different relocation base value in each module because it appears in a different order in each module.

Link Command Line: X LINK A B ↵

Module A contains the predefined partitions 0 through 7, the following user-defined partitions:

Partition Name	Attributes
NEWPART	long NREL, shared, common base, unaligned, data, overwrite-with-message, global
TJ	long NREL, unshared, normal base, unaligned, data, overwrite-with-message, local

and the external symbols:

FOO
FOO1

Module B contains the predefined partitions 0 through 7, and the following user-defined partitions:

ZZZ	long NREL, shared, common base, aligned, code, overwrite-with-message, local
NEWPART	long NREL, shared, common base, unaligned, data, overwrite-with-message, global

Temporary Relocation Base Tables:

For Module A:		For Module B:	
0	predefined partitions	0	predefined partitions
1		1	
2		2	
3		3	
4		4	
5		5	
6		6	
7		7	
8	→ NEWPART	8	→ ZZZ
9	→ TJ	9	→ NEWPART
10	→ FOO		
11	→ FOO1		

SD-02172

Figure 3-2. Relocation Base Tables

Data Relocation

Between pass one and pass two, Link builds the *memory map*, an outline showing the relocation bases assigned to the partitions in the object modules. At this point, Link also begins relocating the symbols and data words associated with each partition.

Relocation takes place in two phases. Between its first and second passes, Link relocates (repositions) the data and symbol elements within each partition. During this phase, Link also records in the symbol table the names and relocation base values of all external symbols. If a module contains local symbols (i.e., the local symbols block), you can direct Link to include those symbols in the .ST file by using the /LOCAL argument switch. Note that you cannot use local symbols as relocation bases; they are strictly for debugging purposes.

During pass two, Link resolves the value of each data word.

To relocate data and symbol elements and resolve their values, Link uses *relocation entries* generated by the language processor within data and symbols blocks.

A data or symbols block can contain the following types of relocation entries:

- *standard relocation entries*, which define the address of relocatable code or data elements
- *extended relocation entries*, which serve the same purpose as standard relocation entries, but contain a larger bit field for defining the block's relocation value, if that value is greater than 32K, and the block's relocation operation. (The *relocation operation* describes the kind of relocation Link will perform for the words in the block.)

The following relocation entries can also occur, but *only* in data blocks:

- *bit field relocation entries*, which enable Link to relocate a bit field within a word (16-bits) or a double word (32-bits)
- one or more *relocation dictionary entries*, which enable Link to look up (in its symbol table) the value of the symbols referenced by a dataword, and thereby, resolve the contents of the word (A *dataword* consists of 16 contiguous bits of code or data.)
- one or more *extended relocation dictionary entries*, which serve the same purpose as relocation dictionary entries, but provide a larger bit field for defining the relocation operation.

The data and symbols blocks in 32-bit object modules must have extended relocation formats. (The bit field format is an extended format.) However, unless you write your own assembler or compiler, these structures will be invisible to you. The assembler or compiler generates the relocation entries based on the contents of the object modules.

Chapter 4 contains detailed descriptions of the data and symbols blocks.

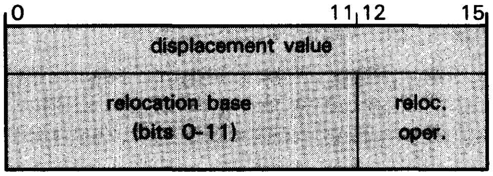
Relocation Entries

Each relocation entry contains a relocation base field and one or more relocation operation fields. The *relocation base field* contains the relocation base value of a predefined partition, user defined partition or external. The *relocation operation field* defines the type of relocation Link will perform to resolve the word's contents and address within the partition. Refer to Appendix F for a list of the legal relocation operations.

Standard and Extended Relocation Entries

Link uses standard relocation entries to determine the position of a word or symbol relative to other words in that partition. In addition to a relocation base value and relocation operation, these entries define the displacement, or position, of the word or symbol relative to the relocation base.

Figure 3-3 illustrates the standard relocation entry format.

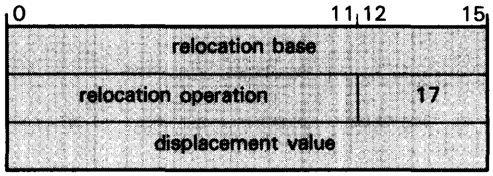


ID-02671

Figure 3-3. Standard Relocation Entry

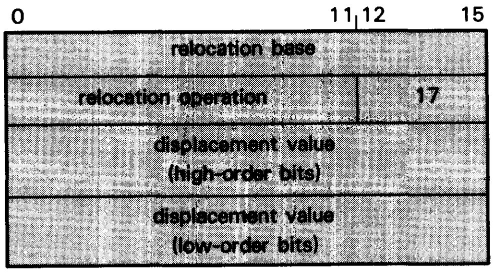
Extended relocation entries contain the same information as standard relocation entries, but contain a larger field for the actual relocation operation. Each extended relocation contains two relocation operation fields: a 4-bit field defining the format as extended (operation 17_8), and a 12-bit field for the actual relocation operation (supplied by the language processor). The extended format is required for all relocation operations greater than 15_8.

Figures 3-4 and 3-5 illustrate extended relocation entry formats for 16- and 32-bit programs, respectively.



ID-02672

Figure 3-4. Extended 16-Bit Relocation Entry


0
11
12
15
relocation base
relocation operation
17
displacement value (high-order bits)
displacement value (low-order bits)

ID-02673

Figure 3-5. Extended 32-Bit Relocation Entry

Relocation Dictionary Entries

The relocation dictionary entries in the data and symbols blocks resolve the contents of the blocks' data words. Link uses these entries during pass two.

Each relocation dictionary entry points to the relocation base value of a symbol referenced or by a word in the data block. By combining the value of the symbol with the value of the data word, Link resolves the data word. Suppose, for example, that the instruction

XWLDA 0, BETA

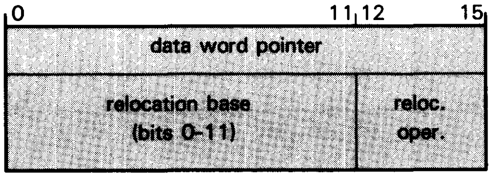
appears in a data block, and that BETA is an external symbol (i.e., BETA is defined in another object module in the command line).

After the assembly process, the instruction will appear in the data block as XWLDA 0 0. The relocation dictionary entry for the instruction will contain the following:

- a pointer to the value of XWLDA in the data block (supplied by the assembler)
- the value of BETA, as recorded in the symbol table
- a relocation operation (supplied by the assembler)

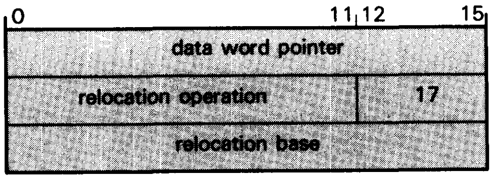
To resolve the complete XWLDA instruction, Link combines the value of XWLDA with BETA's relocation base value, according to the specified relocation operation.

Figures 3-6 and 3-7 illustrate standard and extended relocation dictionary entry formats, respectively.



ID-02674

Figure 3-6. Standard Relocation Dictionary Entry



ID-02675

Figure 3-7. Extended Relocation Dictionary Entry

Bit Field Relocation

Some high-level languages, such as PL/I, allow you to address *bit fields* as well as 16-bit words and 32-bit double words. When you address a bit field, the language processor generates a bit field relocation entry (shown in Figure 3-8).

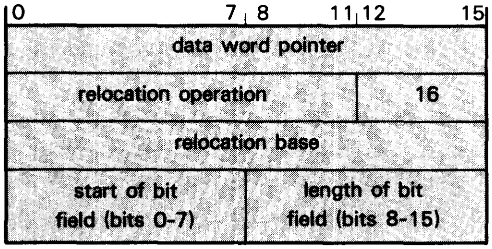
Each bit field relocation entry consists of the following elements:

- a pointer to the data element (in the data block) containing the target bit field
- a relocation operation word, where bits 12-15 define the entry as a bit field relocation entry (operation 16_8), and bits 0-11 specify the actual relocation operation
- the relocation base of the data element
- a word defining the start of the bit field and its width

The *start of bit field* specification defines the position of the first bit in the target bit field. This value must be less than 16_{10} if the entire word is 16 bits, and less than 32_{10} if the word is 32 bits. Otherwise, Link returns an error. By convention, we number bits in a field from left to right, starting with 0. Thus, the leftmost (most significant) bit in a word is bit 0, the next is bit 1, and so on.

The *width of bit field* specification defines the number of bits in the target bit field, minus 1 bit. The sum of this value and the *start of bit field* value must be less than or equal to 16_{10} or 32_{10}, depending on whether the element is a word or a double word. If it is outside the correct range, Link returns an error.

Figure 3-8 illustrates the bit field relocation dictionary entry format.



DG-02676

Figure 3-8. Bit Field Relocation Dictionary Entry

Stack Placement

The user *stack* is a block of consecutive memory locations. You usually reserve stack words for each task, to hold such information as return addresses, subroutine arguments, data, and variables.

The ECLIPSE MV/Family hardware supports two kinds of stacks: *wide stacks*, for 32-bit programs, and standard ECLIPSE stacks, for 16-bit programs. The hardware uses the following 32-bit stack registers to manage wide stacks:

- the wide stack pointer
- the wide frame pointer
- the wide stack limit
- the wide stack base

These registers occupy the lower page zero locations 20_8 through 27_8.

The *wide stack pointer* contains the address of the double word currently at the top of the stack. The operating system increments the value of the wide stack pointer as you push data onto the stack, and decrements its value as you pop data off the stack. The value of the wide frame pointer also varies dynamically. The *wide frame pointer* contains the address of the first available double word on the stack minus 2. In other words, the wide frame pointer points to the last double word used in the stack.

The *wide stack base* contains the base address, or lower boundary, of the stack. The *wide stack limit* points to the stack's upper boundary.

There are three registers for the ECLIPSE stack: the stack pointer, analogous to the wide stack pointer; the frame pointer, analogous to the wide frame pointer; and the stack limit, analogous to the wide stack limit. These 16-bit registers occupy locations 40_8 through 42_8 in lower page zero.

By default, Link allocates a stack of 60_{10} words for the initial task in a 32-bit program, and a stack of 30_{10} words for the initial task in a 16-bit program. However, you can define your own stack by loading or referring to the stack registers in your source code, or by using the /STACK=n sequence in the Link command line, where n specifies the stack size. Refer to the *Principles of Operation 32-Bit ECLIPSE® Systems* manual for documentation on the stack instructions.

Any explicit stack definition in your code overrides Link's default stack allocation and the /STACK sequence. That is, if you load the stack pointer or stack limit in your code, Link does not build a stack, even if you use the /STACK sequence in the command line.

After allocating the stack, Link checks to see if your program defines a *stack fault handler* routine to gain control in the event of a stack error. You can define a stack fault handler in your code, provided you declare it with the entry symbol SFALT. If you do not define an SFALT entry, Link searches the appropriate system library and loads in the *default stack fault handler*, a system-supplied routine.

End of Chapter

Chapter 4
Object Block Structures

Every object module consists of a series of *object blocks*, groups of binary code produced by the language processor as it interprets your source code. Assembly language pseudo-ops, such as .TITL and .REV, and certain high-level language statements generate object blocks.

This chapter shows the internal structure of the object blocks Link recognizes. If you're linking object modules produced by the AOS/VS Macroassembler or a language compiler compatible with AOS/VS, you may find this chapter useful for reference. If you're linking object modules produced by your own language processor, the information in this chapter is essential, since the structure of the object blocks you create must conform to the AOS/VS block structures. Table 4-1 lists the object blocks and their corresponding block numbers (expressed as octal values).

The block structures defined in this chapter apply primarily to 32-bit object modules. The block structures for 16-bit modules differ in some respects. For descriptions of these differences, refer to Chapter 5. For documentation on the library start and library end blocks, refer to Chapter 6.

Whenever possible, this chapter mentions which Macroassembler pseudo-ops generate the object blocks. If you're using a high-level language, refer to the appropriate language manual for comparable statements.

Table 4-1. Object Block Types

Block Type	Number
Data block	0
Title block	1
End block	2
Unlabeled common block	3
External symbols block	4
Entry symbols block	5
Local symbols block	6
Library start block	7
Address information block (AIB)*	10
(Reserved)	11
Task block	12
(Reserved)	13
Named common block*	14
(Reserved)	15
Debugger symbols block	16
Debugger lines block	17
Lines title block	20
Library end block	21
(Reserved)	22
Partition definition block	23
(Reserved)	24
(Reserved)	25
Revision block	26
Filler block	27
Module revision block	30
Alignment block	31

* Overlay designators and partition definition blocks replace AIB and named common block functionality. See Chapter 5 for descriptions of address information and named common blocks.

Object Block Restrictions

An object module need not contain all of the blocks listed in Table 4-1. However, Link imposes several restrictions on the order of the object blocks in a module, as Table 4-2 indicates.

Table 4-2. Object Block Order

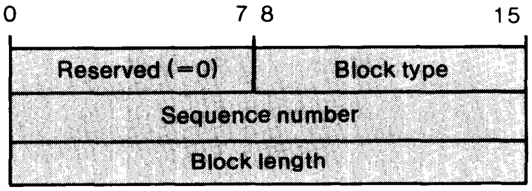
Object Block	Comments
Title block	Must be the first object block in the module; defines the title of the module and, optionally, a two-part revision level for the .PR file. (The module revision block defines a four-part revision level for the .PR file.)
Revision block	If used, must be the second object block in the module; defines which revision of the module's data block Link should use. (This block differs from the *module revision block*, which, like the title block, defines a revision level for the .PR file.)
Partition definition block(s)	Define the attributes and names of user-defined partitions. Partition definition blocks must appear before references to any symbols they define, and before data blocks and external symbols blocks.
External symbols block(s)	Describe symbols defined outside the object module. These blocks must appear before references to any symbols they describe.
Entry symbols block(s) Local symbols block(s) Data block(s)	Each entry symbols block describes symbols defined in a module; each local symbols block defines symbols used exclusively in a module. Each data block decribes a group of code or data in a module. These blocks must appear after any externals or partitions they refer to.
Other block types	(See Table 4-1)
Lines title block	Describes high-level language debugger information. (This block is a complement to the debugger lines block.)
End block	Defines the end of the object module, and its potential start address.

The object blocks in a module must be contiguous; that is, there can be no extraneous information between the end of one block and the start of the next. No object block, except the library start block, can contain more than 1024 (1K) words.

Use only one of each of the following block types per module: title block, module revision block, revision block, lines title block, and end block. Every module must begin with a title block and end with an end block; the module revision, revision, and lines title blocks are optional.

Block Structure: Header

Every object block, regardless of its type, begins with the standard three-word header shown in Figure 4-1.



SD-02174

Figure 4-1. Object Block Header

The left byte in the first header word is reserved for Link interpretation; thus, you must set this byte to 0 in most cases.

NOTE: The title block and data block are exceptions to this rule. Bit 4 in the first header word of the title block and bits 4 and 7 in the first header word of the data block are flag bits. (Title and data blocks are described later in this chapter.)

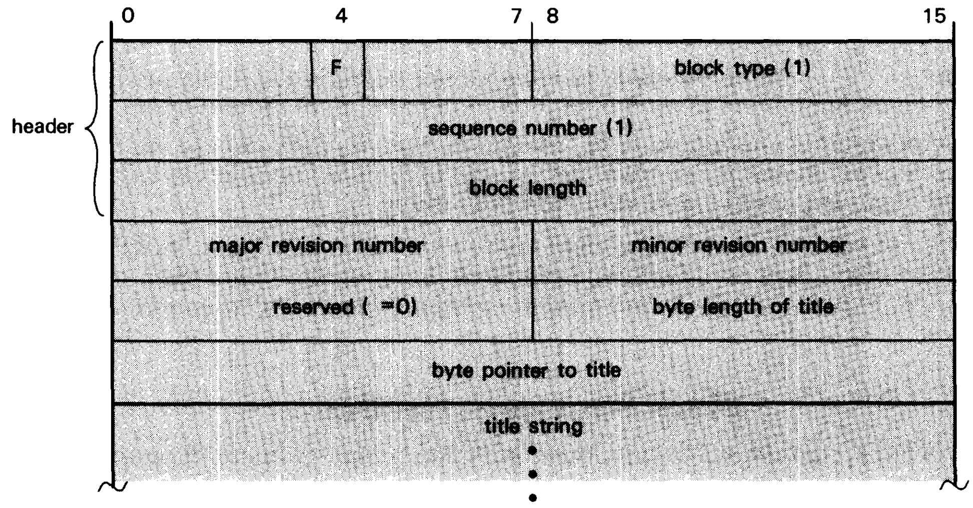
Block type in the first header word identifies the block by its octal type number. Refer to Table 4-1 for a list of the block types and their corresponding numbers.

Sequence number, the second header word, indicates the position or sequence of the block relative to the other object blocks in the module. For example, the title block always has a sequence number of 1 since it is always the first object block in the module.

Block length, the third header word, contains the total number of 16-bit words in the block, including the header words. For example, the end block contains five words including the header, and therefore, has a block length of 5.

Title Block

The title block, the first object block in every module, defines the module's title, and optionally, the revision number that Link will assign the .PR file. The Macroassembler generates a title block when it encounters the .TITL pseudo-op. Figure 4-2 shows the structure of the title block. The diagram includes (in parentheses) values for the block type and sequence number fields.



ID-02677

Figure 4-2. Title Block

As Figure 4-2 illustrates, the first three words in the title block constitute the standard object block header. Notice, however, that bit 4 in the left byte of the first header word is a flag bit, set by the Macroassembler pseudo-op .FORC. (.FORC unconditionally includes an object file from a library into your program file.) Link ignores this bit; but the Library File Editor (LFE) does not. If bit 4 is set, LFE also sets the forced load flag in the appropriate object module descriptor of the library start block (described in Chapter 6).

The *major revision number* and *minor revision number* fields supply a two-part revision number for the linked .PR file. This information is optional. As an alternative, you can define the .PR file's revision number in a module revision block, or by using the /REV switch sequence in the Link command line.

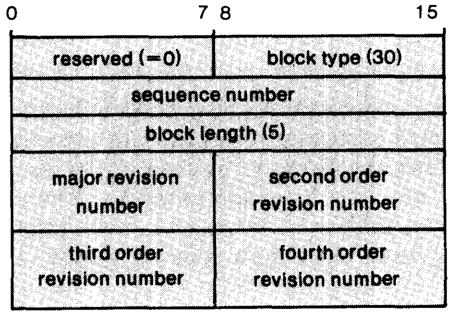
To determine the .PR file's revision number, Link first checks the command line for the /REV sequence. If it is absent, the utility uses the first valid revision number it sees; that is, the first revision number *not equal to -1*. If you do not want Link to use the revision field in the title block, set that field to -1. This forces the utility to keep searching for a valid revision number (e.g., in a module revision block).

Link places the .PR file revision number in offset USTRV of the .PR file's user status table (UST). If there is no valid revision information (i.e., the revision fields in all title and module revision blocks equal -1), Link assigns the .PR file the default revision number 255.255.255.255.

The words immediately following the revision field contain the byte length of the title and a byte pointer to the start of the title string relative to the start of the title block (i.e., 12(sub 10)). The title string, packed one character per byte in ASCII code, is the last entry in the title block. A title string can contain as many as 32 characters (bytes).

Module Revision Block

Like the title block, the module revision block defines a revision number for the .PR file. Unlike the title block, however, the module revision block can accommodate up to a four-part revision number. The Macroassembler pseudo-op .REV generates a module revision block. Figure 4-3 shows the structure of this block type.



SD-02176

Figure 4-3. Module Revision Block

The module revision block begins with the standard three-word header. The revision field, with 1 byte per revision number, follows the header. The length *(block length)* of the revision block is always 5. Even if you provide fewer than four revision numbers, the language processor pads the unused bytes with zeros. For example, a revision number of 1.0 translates to 01.00.00.00.

Revision Block

The revision block defines the earliest Link revision you need to link the modules successfully. It can also define which revisions of the data block, debugger symbols block, and debugger lines block Link will use. Link allows only one revision block per complete object block. Figure 4-4 shows the structure of the revision block.

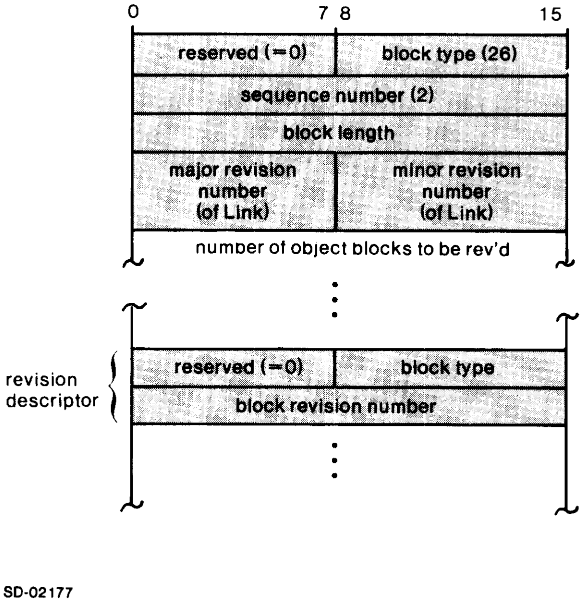


Figure 4-4. Revision Block

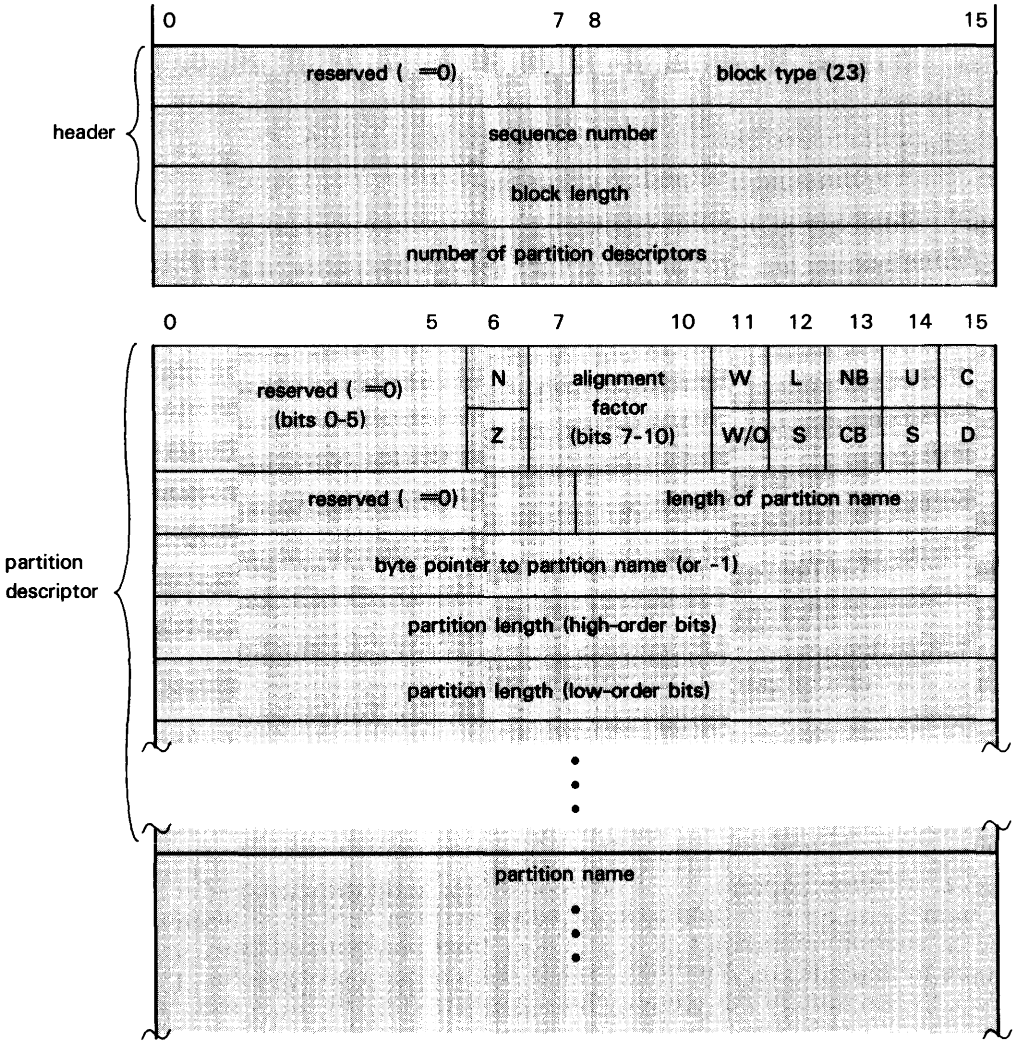
In addition to the standard block header, the revision block contains 2 bytes indicating the major and minor revision number of the Link utility. These fields specify the version of the utility you need for the linking to succeed. If the revision you specify in this block is numerically greater than the current version of Link on your system, Link returns an error and aborts.

The word following the major and minor revision fields tells the utility the number of object blocks for which you've selected particular revisions. The last entry in the block is the *revision descriptor*. This field identifies the block to be revised (by its type number), and specifies which revision Link should use. The language processor generates a revision descriptor for each block revision. Currently, Link uses the revision block information to select data blocks compatible with AOS/VS. (Other block types may be revised in future versions of AOS/VS Link.)

In effect, the revision block tells Link how to interpret the data pointers in a data block's relocation dictionary entry. If the *block revision* field contains 1, Link interprets each dictionary data pointer as a pointer to the relative position of a word in the data block. As a result, Link uses 0 to refer to the first data word in the data block, 1 to refer to the second data word, and so on - regardless of the data's final destination. If the block revision field contains 0, Link interprets the dictionary data pointer as a pointer to the relative position of the data word in its partition; i.e., the relocated address of the data word.

Partition Definition Block

The language processor generates a partition definition block for each user-defined partition in an object module. (The assembly language pseudo-op .PART generates a partition definition block.) Figure 4-5 shows the structure of this block type.



Key:

N	(bit 6) =0	if partition is in NREL
Z	(bit 6) =1	if partition is in ZREL
W	(bit 11) =0	if overwrite-with-message attribute
W/O	(bit 11) =1	if overwrite-without-message attribute
L	(bit 12) =0	if long NREL attribute
S	(bit 12) =1	if short NREL attribute
NB	(bit 13) =0	if normal base attribute
CB	(bit 13) =1	if common base attribute
U	(bit 14) =0	if unshared attribute
S	(bit 14) =1	if shared attribute
C	(bit 15) =0	if code attribute
D	(bit 15) =1	if data attribute

ID-02678

Figure 4-5. Partition Definition Block

In addition to the standard three-word header, each partition definition block contains the following elements:

- a five-word *partition descriptor* for each partition the block defines
- a word indicating the number of partition descriptors
- the name(s) of the new partition(s) (optional)

As Figure 4-5 shows, the first word in the partition descriptor contains bit fields indicating the attributes of the user-defined partition. For example, if bit 15 has a value of 1, Link gives the partition the data attribute. If the attributes defined in this block conflict with previously assigned attributes, Link returns a warning. The alignment attribute is the only exception to this rule. If you assign conflicting alignment factors to a partition, Link uses the largest one and does not return a warning.

Bits 0 through 5 in the first word of the partition descriptor are reserved for Link's use.

Bit 6 specifies whether the partition is assigned to NREL memory (bit 6=0) or ZREL memory (bit 6=1).

The *alignment* field in the partition descriptor contains the alignment factor you defined for the new partition. (The alignment factor range is 1 through 10_{10}.) The alignment factor directs Link to align the partition on a specific power of 2 (2^n) boundary. For example, an alignment factor of 10 aligns the partition on a 1K-word boundary. You can specify partition alignment in your code, or can use the /ALIGN switch sequence for this purpose.

Bit 12 in the first partition descriptor word specifies whether the new partition has the long NREL attribute (bit 12=0) or the short NREL attribute (bit 12=1). Recall that partitions with the long NREL attribute can reside anywhere in NREL memory; partitions with the short NREL attribute are restricted to the area from ?NBOT to 32K-1 words. All user-defined shared partitions must have the long NREL attribute.

Bit 13 defines whether the partition has the normal base or the common base attribute. Link relocates normal base partitions from separate relocation bases, module by module. The utility relocates the contents of common base partitions from one common base, by combining contributions to this partition from each module that defines a common base partition (see Figure 3-1). If you specify the common base attribute (i.e., bit 13 is set), the partition definition block is functionally identical to the named common block.

Bit 14 defines whether the partition is an unshared or a shared partition. Unshared partitions can reside anywhere in unshared NREL (from ?USTA to ?NMAX). Shared partitions can reside anywhere in shared NREL (from ?SBOT to ?NTOP).

Although you don't have to name user-defined partitions, each partition descriptor contains a name length field and a byte pointer to the partition name string.

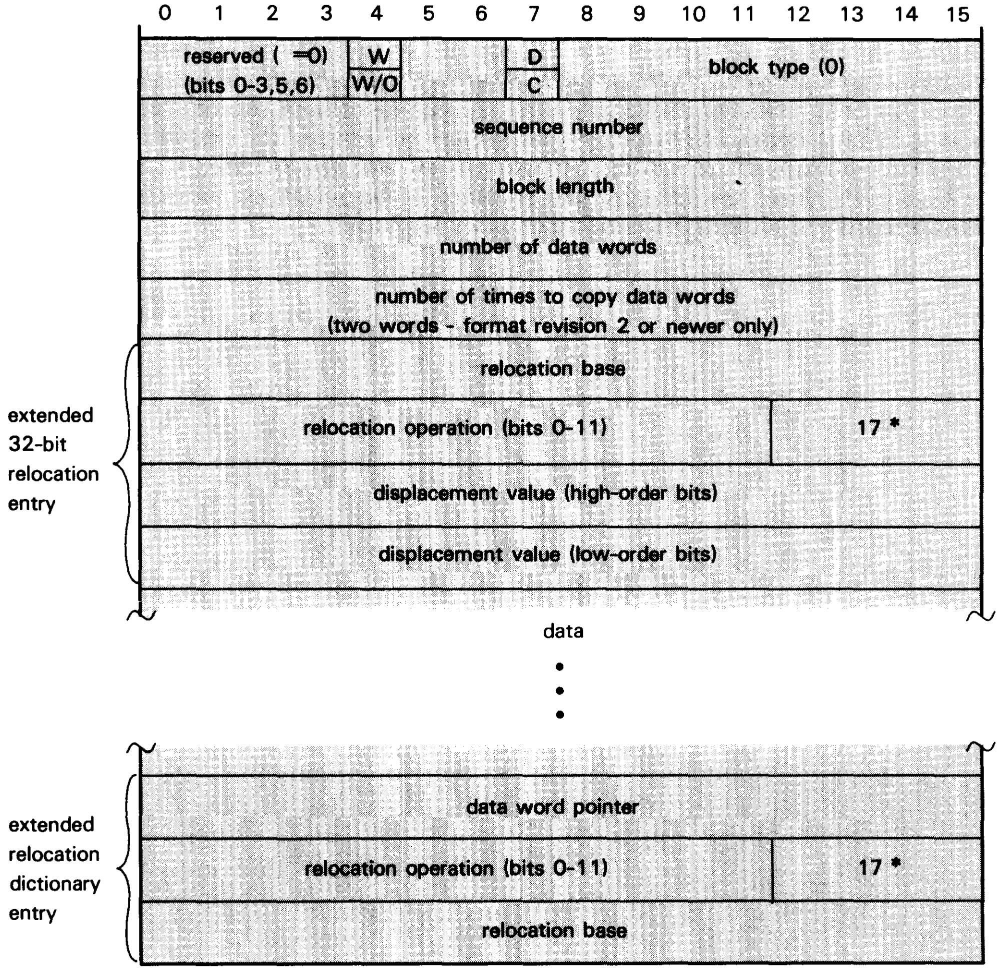
The *name length field* tells Link how many characters are in the partition's name. Link truncates partition names to 40_8 characters.

If the *byte pointer* contains -1, Link ignores the name length field, assigns the partition the *local* attribute, and gives it a unique name. The Link-defined name will appear on your map listing, but you cannot cross-reference it or use it as a relocation base since it is not always constant. See "User-Defined Partitions" in Chapter 3 for information on the local attribute. If the byte pointer contains a valid number, the partition name appears at the end of the block, packed one character per byte in ASCII code.

The *partition length* field is meaningful only if the partition has the common base attribute. If the partition length defined in this field conflicts with previous length specifications, Link uses the largest size and returns a warning.

Data Block

The language processor generates a data block for every distinct group of data or code in a module. The data block contains the actual data or code, a relocation entry defining its placement in the partition, and one or more relocation dictionary entries. Depending on your source code, a data block can contain either a standard or an extended relocation entry. The extended format is required for 32-bit modules. Similarly, a data block can contain a standard, extended, or bit field relocation dictionary entry, depending on your source code. If the data block is in a 32-bit module, the dictionary entry must have either the extended or bit field format. Figure 4-6 shows the structure of the data block with the extended 32-bit relocation formats. Figure 4-7 shows the standard and extended 16-bit relocation formats. Figure 4-8 shows the standard and bit field relocation dictionary formats.


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
reserved (=0)
(bits 0-3,5,6)
W
W/O
D
C
block type (0)
sequence number
block length
number of data words
number of times to copy data words
(two words - format revision 2 or newer only)
extended 32-bit relocation entry
relocation base
relocation operation (bits 0-11)
17 *
displacement value (high-order bits)
displacement value (low-order bits)
data
extended relocation dictionary entry
data word pointer
relocation operation (bits 0-11)
17 *
relocation base

Key:

W	(bit 4) =0	if overwrite-with-message attribute
W/O	(bit 4) =1	if overwrite-without message attribute
D	(bit 7) =0	if data attribute
C	(bit 7) =1	if code attribute
*	relocation operation 17_8 indicates that this is an extended relocation entry	

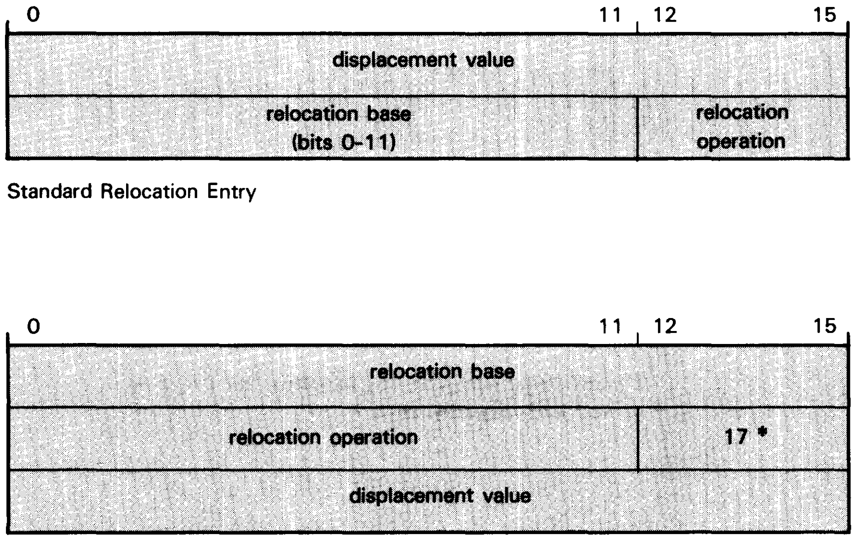
ID-02679

Figure 4-6. Data Block

As Figure 4-6 shows, the data block begins with the standard object block header. Note that bit 4 in the first header word is an overwrite flag. If this bit has a value of 0 and if overwriting occurs, Link sends an error message to the appropriate error files. If this bit has a value of 1, Link suppresses overwrite messages.

Bit 7 of the first header word is a code flag. When bit 7 has a value of 0, it means that the data block contains data (as opposed to code). When bit 7 is set (bit 7 = 1), the data block contains code. Link ignores this bit's value, but some utilities use it.

The word following the block length parameter defines the number of data words the block contains. The data words appear in order after the relocation entry.



Standard Relocation Entry

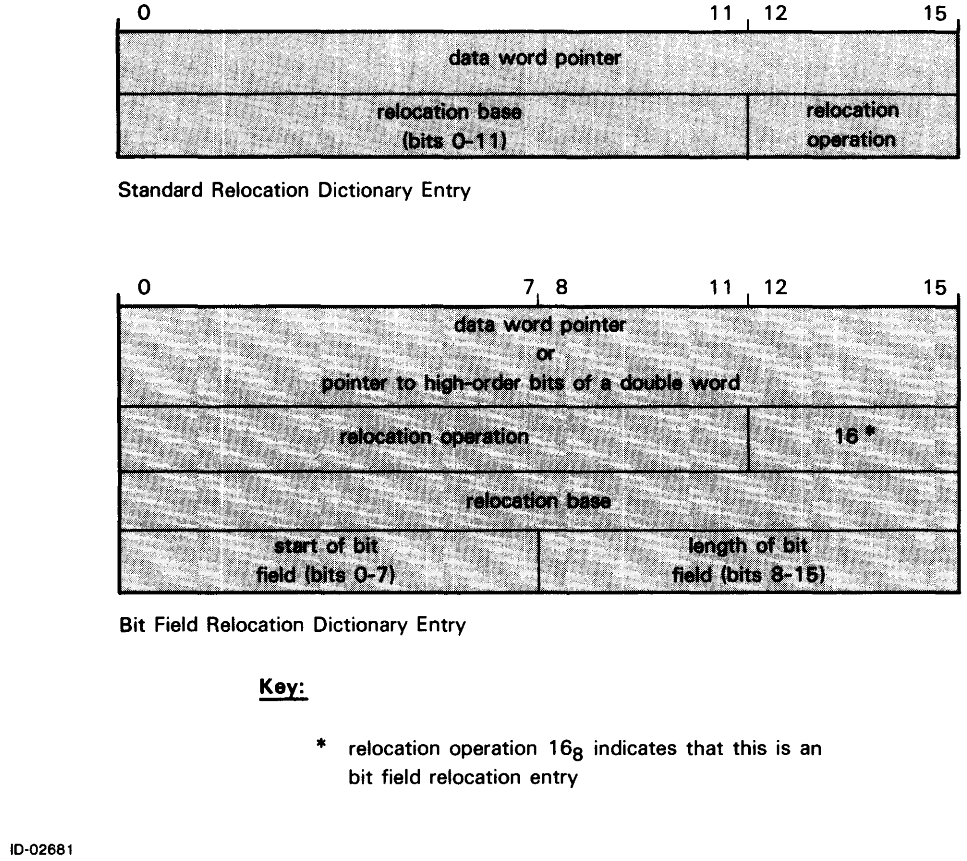
Extended 16-Bit Relocation Entry

Key:

* relocation operation 17_8 indicates that this is an extended relocation entry

ID-02680

Figure 4-7. Standard and Extended 16-Bit Relocation Entries



Standard Relocation Dictionary Entry

Bit Field Relocation Dictionary Entry

Key:

* relocation operation 16_8 indicates that this is an bit field relocation entry

Figure 4-8. Standard and Bit Field Relocation Dictionary Entries

Relocation Entries

The relocation entry tells Link where in the partition to place this group of data, relative to other groups of code and data. The *displacement value* states the starting location of this block of data relative to the partition's *relocation base.* As we noted in Chapter 3, Link determines the relocation base of each partition and external symbol between pass one and pass two.

The *relocation operation* defines the kind of relocation Link will perform to resolve the data block's actual starting address. Link determines this by combining the displacement value with the relocation base according to the relocation operation.

Only 32-bit modules can use relocation operations greater than 30_8. In addition, 32-bit modules must have the extended 32-bit and extended relocation dictionary formats. Relocation operation 17_8 in the relocation entry's second word signals that this is an extended relocation format. (See Appendix F for a complete list of relocation operations.)

Relocation Dictionary Entries

The language processor supplies at least one relocation dictionary entry for every relocatable word in the data block. Link uses these entries to look up the relocation bases of symbols or arguments referred to by the data word, and thereby, resolve the contents of the data word. Extended relocation dictionary entries are required for 32-bit modules; relocation operation 17_8, in the entry's second word, defines the entry as an extended relocation entry.

Link evaluates relocation dictionary entries in ascending order. The first word in each dictionary entry points to a data or code word in the block; i.e., the word that refers to the symbol or argument. Link combines the relocated value of the word with the relocation base of the symbol or argument according to the specified relocation operation. In this way, the utility resolves the contents of the data word.

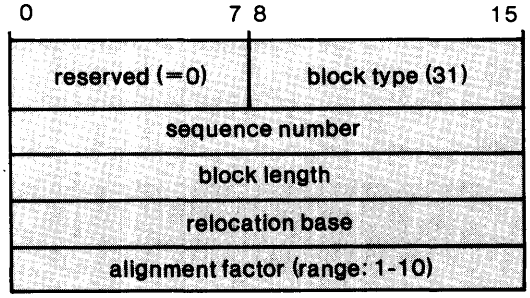
Bit Field Relocation Entries

As we noted in Chapter 3, certain compilers allow you to address bit fields, in addition to words or double words. The dictionary entry in this case is a bit field relocation entry, of the type shown in Figure 4-8. Relocation operation 16_8, in the entry's second word, defines the entry as a bit field relocation entry.

Each bit field relocation entry contains the usual relocation information, and two additional bytes indicating the start of the bit field and its width. For 16-bit modules, (start of bit field) + (width of bit field) must be less than or equal to 16_{10}; otherwise, Link generates an error. For 32-bit modules, the sum of these fields must be less than or equal to 32_{10}.

Alignment Block

The language processor generates an alignment block when you specify an alignment factor for a predefined partition or named common area in an object module. Figure 4-9 shows the structure of the alignment block.



SD-02182

Figure 4-9. Alignment Block

The *relocation base* in this block is the relocation base of the partition or named common area to be aligned. The relocation base must refer to a partition or named common area; if it refers to an external symbol, Link returns an error.

The *alignment factor* can range from 1 through 10_{10}, inclusive. Like the alignment field in the partition definition block, this word directs Link to align the relocation base of the partition or named common area on a specific power of two boundary. You can align partitions on any boundary from a double word boundary (alignment factor = 1) to a 1K-word boundary (alignment factor = 10). (An alignment factor of 0 means the partition is word aligned, the default.)

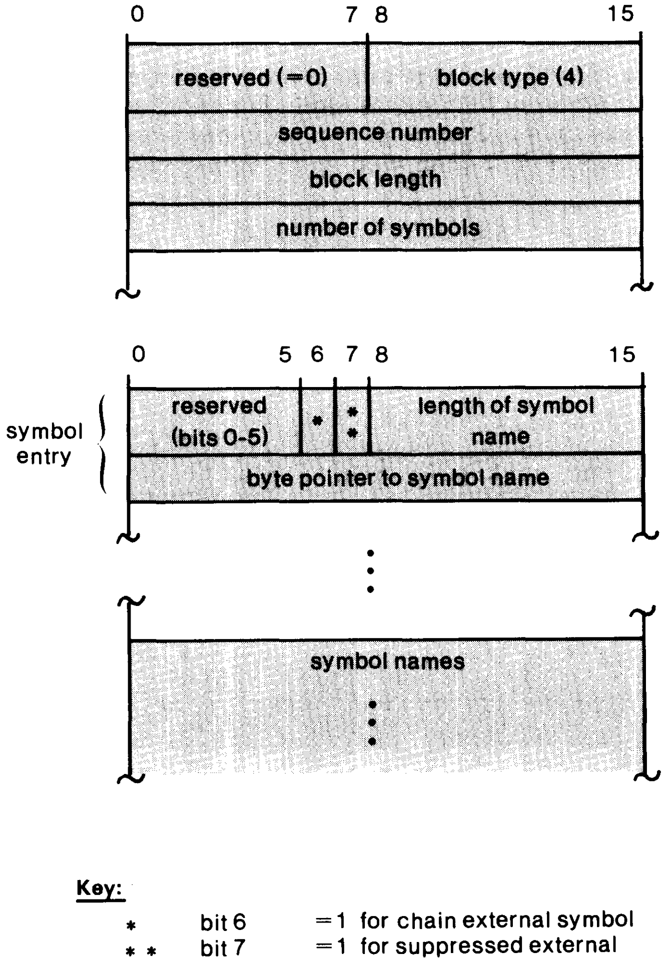
You can also use the /ALIGN switch sequence to align a predefined partition or named common area. If you use this switch against a module that contains the alignment block, Link uses the largest alignment factor specified.

Link keeps track of all memory gaps caused by partition alignment. If you use the /GAPS function switch in the command line, Link returns this information to the map file on a module-by-module basis.

External Symbols Block

The language processor generates an external symbols block when you declare one or more external symbols in an object module. *External symbols* are symbols you define (as entries) in one module, but use in other modules.

Figure 4-10 shows the structure of the external symbols block. Notice that the block contains no relocation information for the symbols. The language processor provides this information in the entry symbols blocks corresponding to the symbols.



Key:

*	bit 6	=1 for chain external symbol
* *	bit 7	=1 for suppressed external

SD-02183

Figure 4-10. External Symbols Block

The word following the block header states the number of symbols listed in the block. There is a corresponding *symbol entry* for each symbol. The first word, right byte, in the symbol entry defines the number of bytes in the symbol. The byte pointer points to the symbol as it appears in the *symbol names* portion of the block. A symbol can consist of up to 32 characters. The names appear in ASCII code, packed one character per byte.

If you declared the symbol to be a *chain external* (symbol type EXTC) in your source code, the language processor sets bit 6 in the first symbol entry word to 1. Chain externals allow you to build symbol chains across two or more object modules. Link records the name and value of each chain external in the .ST file. If the symbol already exists in the .ST file as another symbol type, Link generates an error.

During pass two, Link sets the first reference to a chain external to the value of ?LBOT, the last .PR file link. When it encounters subsequent references to a chain external, Link creates a reverse chain of addresses. There are only two possible relocation operations for chain externals: relocation operations 20 (for 16-bit relocation), and 32 (for 32-bit relocation).

If you declared the symbol to be a *suppressed external* (symbol type EXTS), the language processor sets bit 7 in the first word of the symbol entry. Link allows suppressed externals to remain undefined throughout pass one, without returning an undefined symbol error. If a suppressed external remains undefined at the end of pass one, Link assigns it the value ?UNDF, which defaults to -1.

Entry Symbols Block

Link uses the entry symbols block to resolve and relocate the entry symbols declared in the object module. *Entry symbols* are symbols defined in one object module and referred to (as externals) in other modules. The .ENT pseudo-op causes the Macroassembler to generate an entry symbols block. Figure 4-11 shows the structure of this block.

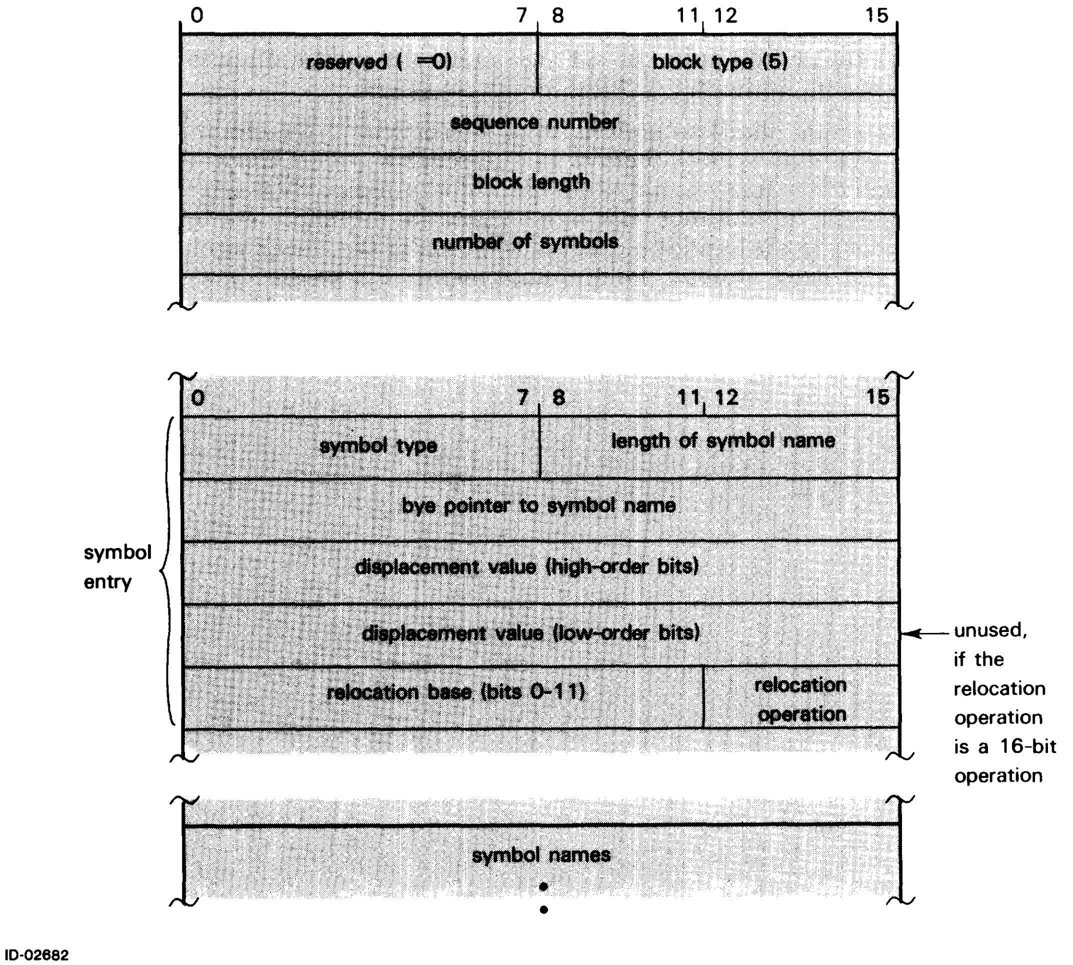

0
7
8
11
12
15
reserved (=0)
block type (5)
sequence number
block length
number of symbols
0
7
8
11
12
15
symbol entry
symbol type
length of symbol name
bye pointer to symbol name
displacement value (high-order bits)
displacement value (low-order bits)
relocation base (bits 0-11)
relocation operation
unused, if the relocation operation is a 16-bit operation
symbol names
ID-02682

Figure 4-11. Entry Symbols Block

In addition to the header, the entry symbols block consists of the following elements:

- a word indicating the number of symbols defined in the block
- one symbol entry for each symbol
- the symbol names

The first word in the symbol entry contains two fields: symbol type, in the left byte, and symbol length, in the right byte. For 32-bit modules, the only legal entry symbol type is *external entry*, symbol type 0. The .ENT pseudo-op defines a symbol as an external entry. Thus, the symbol type field must contain 0 if the block is part of a 32-bit module. Appendix E lists the symbol types available under AOS/VS.

The byte pointer in the symbol entry points to the symbol name as it appears in the *symbol names* portion of the block. Each symbol name can consist of as many as 32 characters. The names are in ASCII code, packed one character per byte.

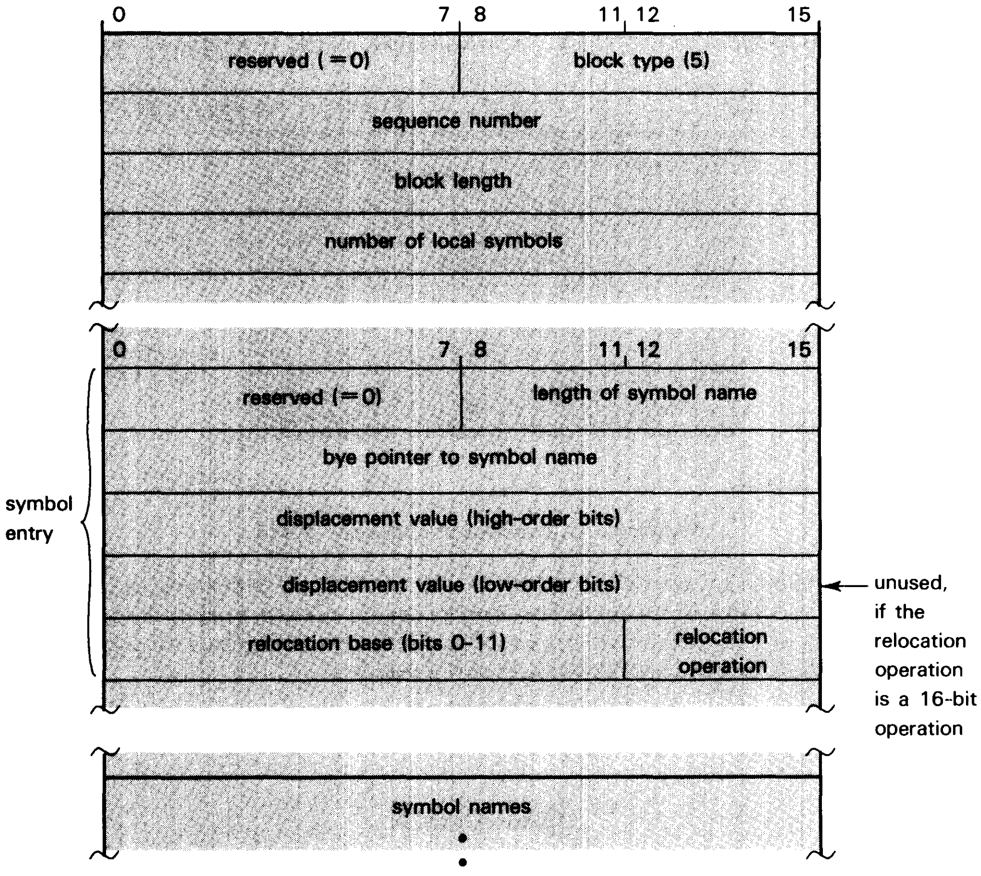
Link uses the relocation information in the symbol entry to place the symbol in the appropriate partition. The relocation base value defines the partition.

Local Symbols Block

This block type describes the local symbols in the module. Local symbols are unique to the module in which they appear; that is, they are not used in any other module. Typically, you use local symbols to label local routines for convenience and eventual debugging. The language processor regards all symbols as local by default, unless you declare them otherwise (e.g., as entry symbols, externals, etc.).

Link excludes local symbols from the .ST file, unless the module contains a local symbols block, and you modify it with the /LOCAL argument switch.

As Figure 4-12 shows, the local symbols block is identical to the entry symbols block in structure. For each local symbol, there is a corresponding *symbol entry*. The symbol entry defines the length of the local symbol name, the symbol's displacement value relative to other symbols in the block, its relocation base, and a relocation operation. The byte pointer in the symbol entry points to the location of the symbol name in the block's symbol name portion. Like external and entry symbols, local symbols names can consist of a maximum of 32 characters. The symbol names are packed one character per byte in ASCII code.

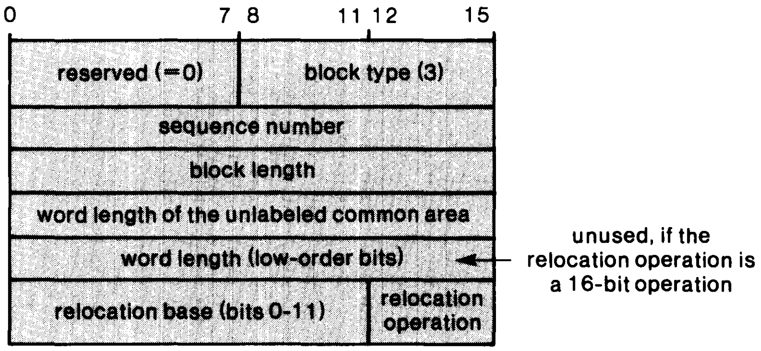

0
7
8
11
12
15
reserved (=0)
block type (5)
sequence number
block length
number of local symbols
0
7
8
11
12
15
reserved (=0)
length of symbol name
bye pointer to symbol name
displacement value (high-order bits)
displacement value (low-order bits)
relocation base (bits 0-11)
relocation operation
symbol entry
unused, if the relocation operation is a 16-bit operation
symbol names

ID-02683

Figure 4-12. Local Symbols Block

Unlabeled Common Block

The language processor generates an unlabeled common block for each unlabeled (unnamed) common area defined in your source code. The .CSZE pseudo-op causes the Macroassembler to generate an unlabeled common block. Figure 4-13 shows the structure of this block.



0
7 8
11 12
15
reserved (=0)
block type (3)
sequence number
block length
word length of the unlabeled common area
word length (low-order bits)
unused, if the relocation operation is a 16-bit operation
relocation base (bits 0-11)
relocation operation

DG-27190

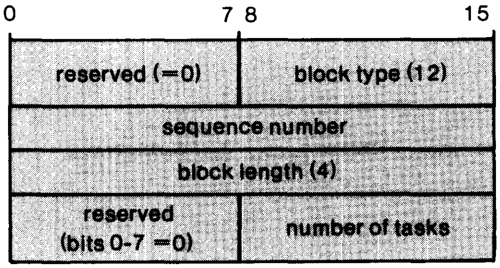
Figure 4-13. Unlabeled Common Block

The *word length* field can consist of either 16 bits or 32 bits, depending on the relocation operation. If the relocation operation is less than or equal to 30_8 or greater than 63_8, this field consists of 16 bits. If you define two or more unlabeled common areas with different lengths, Link uses the largest length specified and returns a warning.

For modules written under AOS and AOS/VS, Link associates the start of the unlabeled common area with the symbol ?CLOC, which the utility defines at the end of pass one.

Task Block

The language processor generates a task block when you use a task declaration to specify the maximum number of tasks in an object module. Link uses this information to create internal databases in the system tables for all potential tasks in the program. The Macroassembler generates a task block when it encounters the .TSK pseudo-op. Figure 4-14 shows the structure of the task block.



SD-02187

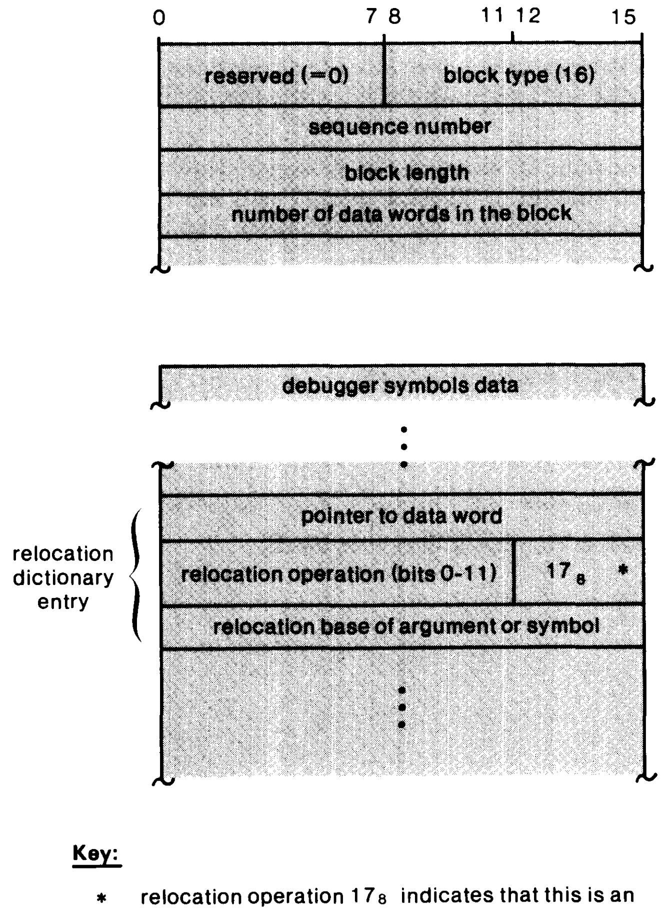
Figure 4-14. Task Block

Link always uses the maximum task specification, whether you define it internally (in the task blocks), or with the /TASKS switch sequence in the command line. For example, given three object modules with task block specifications of 2, 3, and 4, Link assumes 4 potential tasks, and constructs a comparable number of databases in the symbol tables. If the command line for the same modules specifies /TASKS=5, Link assumes a maximum of 5 tasks for the .PR file.

If you do not use the /TASKS switch sequence, and none of the modules contains a task block, Link assumes that the .PR file will consist of only one task, the initial task.

Debugger Symbols Block

The contents of both the debugger symbols block and the debugger lines block are user-defined. Typically, you use these blocks to store information for later high-level language debugging. When Link encounters one or more debugger symbols blocks and the /DEBUG switch, it takes the contents of the debugger symbols block and stores it in a .DS (debugger symbols) output file. Figure 4-15 shows the structure of the debugger symbols block.


0
7 8
11 12
15
reserved (=0)
block type (16)
sequence number
block length
number of data words in the block
debugger symbols data
pointer to data word
relocation operation (bits 0-11)
17_8 *
relocation base of argument or symbol
relocation dictionary entry

Key:

* relocation operation 17_8 indicates that this is an extended relocation entry

SD-02188

Figure 4-15. Debugger Symbols Block

There must be at least one relocation dictionary entry for every relocatable data word in the block. The dictionary entries enable Link to resolve the data in order to record it in the .DS file.

The *relocation operation* field can contain any of the legal relocation operations, except relocation operations 20_8 and 32_8.

If the relocation operation for this block is 7 or 31_8 *(link relocation)*, Link sets up a reverse chain of addresses in the .DS file, where the first word in the file is the address of the last word for which you invoked link relocation.

The .DS file shares the same root filename as the .PR file. If the .PR file is DGLPROG.PR, for example, the .DS file will be DGLPROG.DS.

Debugger Lines Block/Lines Title Block

Given one or more debugger lines blocks and the /DEBUG switch in the command line, Link creates a .DL (debugger lines) file to store the .DL data. Like the debugger symbols block, the debugger lines block contains user-defined data for eventual high-level language debugging. The two block types have similar structures; the only visible difference being the block numbers - 16 for the debugger symbols block and 17 for the debugger lines block.

Some language processors also generate a *lines title block* when an object module contains one or more debugger lines blocks. Link also uses the lines title blocks, if it is present, to create the .DL file.

The lines title block has the same structure as the debugger symbols block and the debugger lines block, but has a block number of 20. Link accepts only one lines title block per object module, even if the module contains more than one debugger lines block.

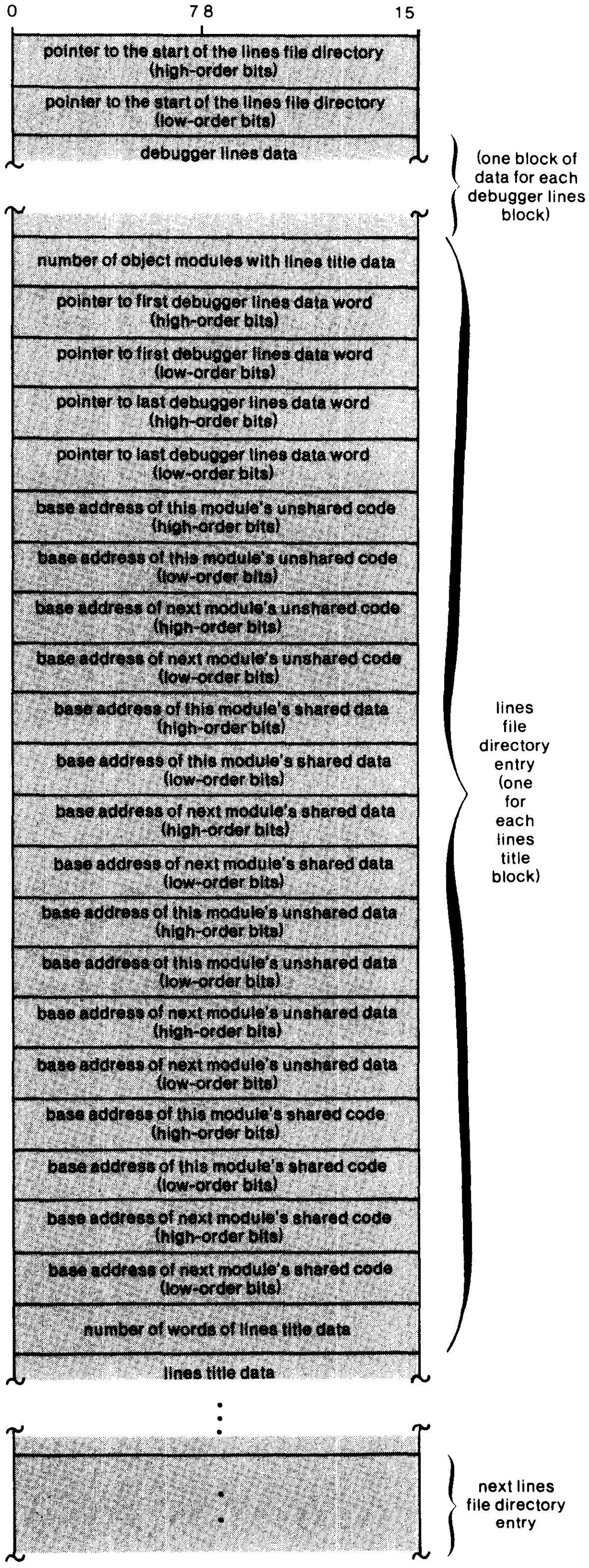
Like the .DS file, the .DL file has the same root filename as the program file.

.DL File Structure

The .DL file has the following four-part structure:

- a pointer to the start of the *lines file directory* (described below)
- debugger lines data (one block of data for each debugger lines block in the object modules)
- the *lines file directory* entries (one entry for each lines title block)
- lines title data (if any module contains a lines title block)

Figure 4-16 shows the structure of the .DL file.


0
7 8
15
pointer to the start of the lines file directory (high-order bits)
pointer to the start of the lines file directory (low-order bits)
debugger lines data
(one block of data for each debugger lines block)
number of object modules with lines title data
pointer to first debugger lines data word (high-order bits)
pointer to first debugger lines data word (low-order bits)
pointer to last debugger lines data word (high-order bits)
pointer to last debugger lines data word (low-order bits)
base address of this module's unshared code (high-order bits)
base address of this module's unshared code (low-order bits)
base address of next module's unshared code (high-order bits)
base address of next module's unshared code (low-order bits)
base address of this module's shared data (high-order bits)
base address of this module's shared data (low-order bits)
base address of next module's shared data (high-order bits)
base address of next module's shared data (low-order bits)
base address of this module's unshared data (high-order bits)
base address of this module's unshared data (low-order bits)
base address of next module's unshared data (high-order bits)
base address of next module's unshared data (low-order bits)
base address of this module's shared code (high-order bits)
base address of this module's shared code (low-order bits)
base address of next module's shared code (high-order bits)
base address of next module's shared code (low-order bits)
number of words of lines title data
lines title data
lines file directory entry (one for each lines title block)
next lines file directory entry

Note: For AOS and 16-bit AOS/VS, each base address is 16-bits.

DG-027191

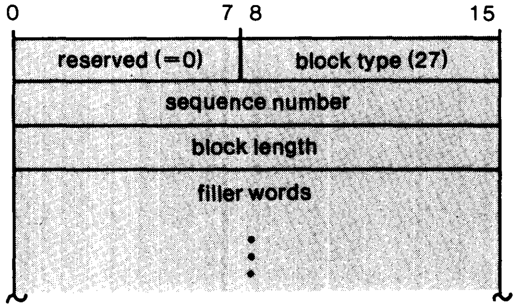
Figure 4-16. .DL File Structure

Each lines file directory entry defines the lower boundary (base) and upper boundary of the module's predefined NREL partitions; i.e., partitions 4, 5, 6, and 7. The upper boundary of the partition segment in one module determines that partition's *base address* in the next module. In Figure 4-16, for example, the base address of the next module's shared data is actually the highest address + 1 of the current module's shared data area.

Notice that the .DL file contains two kinds of pointers: pointers to the debugger lines and lines title data in the .DL file, and pointers to the memory locations of the object module's partition elements.

Filler Block

The filler block allows you to pad a short object module to a length of 1024 (1K) words. This block is useful for filling out a shorter object module in order to compare it with a longer module. Figure 4-17 shows the structure of the filler block.



SD-02190

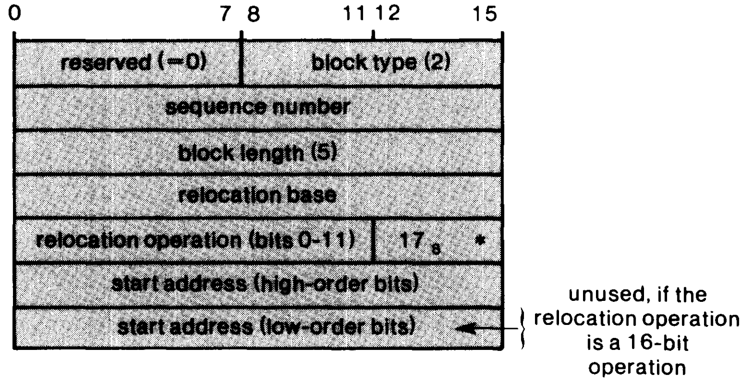
Figure 4-17. Filler Block

A filler block can contain as many as 1024 (1K) words. The three-word header is the only information Link looks at, however. The header defines the block as a filler block (type 27), and defines its length; i.e., the number of filler words plus the header.

The length specification also implicitly defines the start of the next object block, since all blocks in a module must be contiguous.

End Block

The end block defines the end of the object module and its potential starting address. The Macroassembler generates an end block when it encounters the .END pseudo-op in the module. The end block must be the last object block in the module. Figure 4-18 shows the structure of this block.


0
7 8
11 12
15
reserved (=0)
block type (2)
sequence number
block length (5)
relocation base
relocation operation (bits 0-11)
17_8 *
start address (high-order bits)
start address (low-order bits)
unused, if the relocation operation is a 16-bit operation

Key:

* relocation operation 17_8 indicates that this is an extended relocation entry

DG-027192

Figure 4-18. End Block

The *start address* field contains the starting address that the language processor calculated for the module. Link combines this value with one of the module's partition relocation bases to come up with a possible starting address for the .PR file. The relocation operation type must be 1, indicating word relocation.

The start address field can consist of either 16 bits or 32 bits. If the relocation operation is less than or equal to 30_8, or greater than 63_8, this field consists of 16 bits.

Generally, only one module in the command line contains the valid starting address for the .PR file; the language processor sets bit 0 of the other start address fields to 1. If more than one module contains a valid starting address, use the /START argument switch to specify a particular starting address. This switch overrides all other starting address specifications.

The /MAIN switch is similar to /START, but its use is language dependent. Given this switch, Link generates the symbol .MAIN, and assigns that symbol the relocation properties of the start address in the end block.

End of Chapter

Chapter 5
Linking 16-Bit Object Modules for AOS/VS, AOS and MP/AOS

Since AOS/VS supports both 32-bit and 16-bit processes, you can use the AOS/VS Link utility to create 16-bit .PR files to execute under AOS/VS, AOS, or MP/AOS. This chapter describes how Link handles 16-bit object modules. You will need this information if you're developing 16-bit programs under any of these operating systems, or if you're relinking AOS modules to execute under AOS/VS. In order to run an AOS program under AOS/VS, you *must* relink it with the AOS/VS Link.

Note that this chapter documents the *differences* between 16-bit and 32-bit linking procedures. To understand basic Link concepts, you should read the other chapters in this book as well.

This chapter is divided into five main sections. The first four sections describe resource, partition, command line, and object module differences that pertain to linking 16-bit programs. The fifth section provides additional information specific to cross-linking for MP/AOS.

Table 5-1 sketches the primary differences between 16-bit and 32-bit .PR files.

Table 5-1. .PR Files: 16-bit/32-bit Differences

Feature	16-Bit .PR file	32-Bit .PR file
Resources	can be overlays	no overlays
Partition Size	must be less than/equal to 32K words	short NREL (32K or less) long NREL (>32K)
Relocation	standard or extended relocation formats relocation operations 0 – 30_8	extended relocation formats only relocation operations 0 – 107_8
Command Line	optional overlay definition switches:	no overlay definition switches:
	/SYS=VS16 /SYS=AOS /SYS=MPAOS /SYS=RDOS * /SYS=RTOS *	/SYS=VS32 (optional)
	/MULTIPLE=n (for overlay areas)	not applicable
	/NRC (no resource passing) /NRP /WRL	not applicable
Output	.OL file, if overlays	not applicable

* (See Appendix A, "Linking 16-Bit Modules For RDOS and RTOS")

Resources

A 16-bit program file can consist of two kinds of resources: the *root*, or main portion, which is memory resident during execution, and one or more *overlays*, modules the operating system transfers in and out of memory as needed during execution.

Overlays

Overlays are useful in small memory configurations because they allow you to use the same portion of your address space (i.e., an overlay area) for more than one purpose. For example, if your address space is limited to 32K words and you've written a large program (approaching 32K or greater than 32K), you can define the main portion as the root, and then define multiple overlays to be swapped in and out of one or more overlay areas during execution. (Note that you cannot execute a 16-bit program larger than 32K; however you can build one.)

You can designate object modules as overlays in the Link command line, by delimiting the modules with *overlay designators.* (The "Command Line" section of this chapter shows how to use the overlay designators.)

Link reserves enough space in the .PR file for the overlays you've defined, but diverts the actual overlay code to an *overlay file* (.OL file). The .OL file has the same root filename as the .PR file (e.g., DGLPROG.OL corresponds to DGLPROG.PR) and resides on the disk during execution. When you call overlays from the root portion of the program, the system draws them from the overlay file.

Typically, you define two or more overlays for a single *overlay area.* You can define as many as 63 overlay areas and up to 511 overlays per area. During pass one, Link scans the modules and command line for overlays, and determines the number of overlay areas and their sizes.

A single overlay area can consist of either shared or unshared code, but not both. (MP/AOS does not have facilities for unshared overlays.) Link builds shared overlay areas in multiples of 1K words, and unshared overlay areas in multiples of 256 words (512 bytes). The utility derives the *basic size* of an overlay area by taking the size of the largest overlay destined for the area and padding it to a multiple of 1K or 256 words.

If you want several overlays to reside in one overlay area simultaneously, you can define the overlay area's *total size* to be some multiple of its basic size. The argument switch sequence for this is /MULT=n. This switch sequence increases the basic size of the overlay area by a factor of n. For example, to triple the basic size of an overlay area, specify /MULT=3. This allows you to fit three overlays of the basic size into the total overlay area simultaneously.

You can use the /MULT=n sequence only for overlay areas you define in the Link command line. (See the "Command Line" section of this chapter.) In addition, overlays destined for multiple overlay areas must be *movable resources*; that is, they can reside in any portion of the total overlay area at runtime. In contrast, an *immovable resource* is fixed to a specific memory area. (For more information see the description of "Movable/Immovable Resources" later in this chapter.)

There are two ways to acquire and release overlays:

- by issuing *primitive overlay calls* (also known as utility overlay calls). The AOS/VS system calls ?OVLOD, ?OVREL, ?OVEX, and ?OVKIL are primitive overlay calls.
- by issuing *resource calls.* The system calls ?KCALL, ?RCALL, and ?RCHAIN are resource calls. (MP/AOS does not have resource calls.)

When linking for AOS or AOS/VS, issuing the resource calls is generally the better method. If you use primitive overlay calls, you have to explicitly manage the loading and releasing of overlays. If you use resource calls, the *resource manager*, a subroutine in the system library, handles much of the loading and releasing for you. You can use resource calls to acquire and release *both* overlay and root resources. For additional documentation on the primitive overlay calls and the resource calls, refer to the *AOS/VS Programmer's Manual.*

Movable/Immovable Resources

Link classifies resources as either movable or immovable. As described earlier, a *movable resource* is a resource the system can allocate to any area within a range of memory during execution. An *immovable resource* is fixed to a specific memory area.

Movable resources must be *position-independent.* This means the locations they refer to must be outside of any moving resource, or must be offset from the program counter (pc) as the program executes.

Overlays defined for a multiple overlay area (with /MULT=n) are movable resources; that is, they can reside in any portion of the total overlay area at runtime. The root, as a whole, is an immovable resource.

Link allows only certain relocation operations between movable and immovable resources. Table 5-2 lists these relocation operations.

Table 5-2. Resource-to-Resource Relocation Operations

To:	**Immovable Resource**	**Same Movable Resource**	**Different Movable Resource**
From: **Immovable Resource**	relocation operations 0, 1, 2, 3, 4, 6, 11, 12, 14, 20	not applicable	relocation operations 10, 13
Movable Resource	relocation operations 0, 10, 13	relocation operations 10, 12, 13, 14	relocation operations 10, 13

NOTE: Relocation operation 0 indicates absolute addressing (no relocation).

Resource Resolution

This section describes how Link, together with a language processor and the system resource manager, resolves resource calls. Since this resolution is transparent to the user, we aimed this section mainly at programmers who are writing their own compilers. However, if you use overlays frequently, you should be aware of the resource call optimization switches described later in this chapter.

Resource calls take one argument: a *procedure entry symbol* (PENT). PENTs are functionally similar to standard entry symbols (ENTs); that is, Link uses both for intermodular communication. But unlike an ENT, if Link encounters a PENT defined in an overlay, Link builds a two word entry in the Resource Handler Table (RHT). This entry, called a *resource descriptor*, contains information that the resource manager uses at runtime to load and transfer control to that resource.

If Link creates one or more resource descriptors, it also loads the resource manager from the system library into the .PR file. At runtime, the resource manager uses the information in the RHT to locate PENTs with overlay values. Once the resource manager has this information, it loads the overlay and transfers control to the PENT address.

In assembly language, there are two ways to pass an argument to a resource call:

- in line — The argument immediately follows the resource call. For example, if you want to use ?RCALL to load and release the overlay containing PENT G2, issue the instruction ?RCALL G2.
- on the stack —The argument is the top word on the stack when the resource call is made. (Refer to the *AOS Macroassembler Reference Manual* for information on the pseudo-op .PTARG.)

When you pass the argument in-line, it may be possible to *optimize* the resource call by converting it to an EJSR instruction. (See "Resource Call Optimization" later in this chapter.) Depending on how your language processor translates certain resource calls (described next), Link performs this optimization automatically.

When you issue a resource call (?KCALL, ?RCALL, or ?RCHAIN or the high-level language equivalent), a language processor must translate the resource call into two 16-bit words. The first word is the *call relocation word* (or call word) and the second is the *target relocation word* (or target word). The language processor then supplies the proper relocation data for these two words.

As shown in Figures 5-1 through 5-5, the language processor must generate different values for both call words and target words depending on

- whether the argument is passed in-line or on the stack
- whether the resource is optimizable (Note that only immovable resources are optimizable.)

When you pass the resource call argument in-line, the call and target words are relocated relative to an external PENT symbol. At runtime the resource descriptor address is resolved from the target word.

When the resource call argument is passed on the stack, target relocation is unnecessary because your program pushes the resource descriptor address onto the stack immediately prior to issuing the resource call. Therefore, the language processor should set the target word to zero. (See Figures 5-2, and 5-4.)

When a resource is optimizable (immovable resource) and the argument is passed in-line, the language processor should generate values for both call and target words of 3, 4, and 5 for ?KCALL, ?RCALL, and ?RCHAIN, respectively (see Figure 5-1). These values allow Link to differentiate between paired call/target words and unpaired target words (shown in Figure 5-5).

When a resource is not optimizable (movable resource) and the argument is passed in-line, or if you expect to use the module with other Data General linkers or binders, the language processor should generate call word values of 6013, 6014, and 6015 and target word values of 0, 1, and 2 for ?KCALL, ?RCALL, and ?RCHAIN, respectively. (See Figure 5-3.)

	?KCALL	?RCALL	?RCHAIN
Call Word	3	4	5
Target Word	3	4	5

The call word is relocated from an external PENT symbol.
The target word is relocated from an external PENT symbol.

ID-02751

Figure 5-1. Optimizable Resource Calls for Argument Passed in Line (Immovable Resource)

	?KCALL	?RCALL	?RCHAIN
Call Word	0	0	0
Target Word	0	0	0

The call word is relocated from an external ENT symbols ??KCA, ??RCA, and ??RCH, respectively. The target word is not relocated.

ID-02752

Figure 5-2. Optimizable Resource Calls for Argument Passed On The Stack (Immovable Resource)

	?KCALL	?RCALL	?RCHAIN
Call Word	6013	6014	6015
Target Word	0	1	2

The call word is relocated from an external PENT symbol.
The target word is relocated from an external PENT symbol.

ID-02753

Figure 5-3. Non-Optimizable Resource Calls for Argument Passed In Line (Movable Resource)

	?KCALL	?RCALL	?RCHAIN
Call Word	6013	6014	6015
Target Word	0	1	2

Non-Optimizable Resource Calls for Argument Passed on the Stack (Movable Rescource)

ID-02754

Figure 5-4. Non-Optimizable Resource Calls for Argument Passed On The Stack (Movable Resource)

Target Word	0	1	2

The target word is relocated from an external PENT symbol.

ID-02755

Figure 5-5. Unpaired Target (.PTARG) Word

Once the call word and target word are in place, the language processor generates a relocation dictionary entry for every call relocation and target relocation word in a data block. The first word in the dictionary entry points to the ?KCALL, ?RCALL, or ?RCHAIN in the data block. The second word contains the relocation base of the data block's partition and a relocation operation. The relocation operation for call relocation is 10_8, and for target relocation, 13_8.

Resource Call Optimization

Because a resource call can be significantly slower than an equivalent EJSR instruction, Link will replace resource calls with EJSRs under certain circumstances.

- If both the calling resource and the external PENT symbol are in the root, Link will replace a ?KCALL or ?RCALL with an EJSR.
- If both the calling resource and the external PENT are in the same overlay, Link will replace a ?KCALL or ?RCALL issued for an immovable resource (call/target 4/4 or 5/5) with an EJSR. By default, all other call/target words will become resource calls.
- If you set resource call optimization switches (described next), Link will replace the appropriate resource calls with EJSRs.

Resource Call Optimization Switches

As explained in the preceding section, Link automatically converts root-to-root ?KCALLs and ?RCALLs to EJSR instructions. Link also lets you specify additional resource calls for conversion to EJSRs by using *resource call optimization switches*. However, you must observe certain restrictions when using these switches. Table 5-3 lists the resource call optimization switches and describes their actions and restrictions.

Table 5-3. Resource Optimization Switches

Switch	Replaces the Following with EJSR Instructions	Restrictions
/NRP	All resource calls from an overlay to an external PENT in the same overlay	Your program cannot use movable resources (no /MULT=n switch on overlays.) and an external PENT symbol in one overlay cannot be passed as a parameter to a module in another overlay.
/WRL	All resource calls from an overlay to an external PENT in the same overlay, plus all calls to an external PENT in the root, regardless of the location of the call	Your program must observe all /NRP restrictions, plus your program may not make resource calls from root to overlay to root to another overlay.
/NRC	All resource calls	Your program must explicitly manage overlays.

Table 5-4 shows how Link interprets resource calls ?KCALL, ?RCALL, and ?RCHAIN. Note the effects of the optimization switches.

Table 5-4. Resource Call Resolution Table

Call	Calling Resource	Target Resource	Call Word	Target Word
?KCALL	Root	Root	EJSR	A
?RCALL	Root	Root	EJSR	A
?RCHAIN	Root	Root	EJMP	A
***?KCALL	Root	Overlay	??KCA	B
***?RCALL	Root	Overlay	??RCA	B
***?RCHAIN	Root	Overlay	??RCH	B
**?KCALL	Overlay	Root	??KCA	A
**?RCALL	Overlay	Root	??RCA	A
**?RCHAIN	Overlay	Root	??RCH	A
*?KCALL	Overlay	Same Overlay	??KCA	B
*?RCALL	Overlay	Same Overlay	??RCA	B
*?RCHAIN	Overlay	Same Overlay	??RCH	B
***?KCALL	Overlay	Diff. Overlay	??KCA	B
***?RCALL	Overlay	Diff. Overlay	??RCA	B
***?RCHAIN	Overlay	Diff. Overlay	??RCH	B

Key:

??KCA calls the resource handler for ?KCALL
??RCA calls the resource handler for ?RCALL
??RCH calls the resource handler for ?RCHAIN

A is the memory address of the external PENT symbol
B is the address of a resource handler table entry

Key to Optimization Switches:

*	If /NRP, /WRL or /NRC switch is set, or the call/target words are 3/3, 4/4 or 5/5, the relocated results are the same as the root-to-root case.
**	If the /WRL or /NRC switch is set, the relocated results are the same as the root-to-root case.
***	If the /NRC switch is set, the relocated results are the same as the root-to-root case.

NOTE: Be sure to declare all external targets as .PENT (procedure entry) symbols in the modules where they're defined; otherwise, Link returns an error.

Partitions

As it does for 32-bit modules, Link groups elements from 16-bit modules into partitions. The partition attributes we defined in Chapter 3 apply to the partitions in 16-bit modules, with one exception: Link ignores the *short NREL* and *long NREL* attributes for 16-bit modules.

Only relocation operations 0 through 30_8 are valid for 16-bit object modules.

Command Line

The format of the Link command line for 16-bit object modules is

```
X[EQ] LINK[fswitch] [!*]objectfile[.OB] ...[!objectfile[.OB]...][*!][argswitch]
```

where:

fswitch is one or more function switches or function switch sequences. You must use one of the following function switches:

```
/SYS=VS16
/SYS=AOS
/SYS=MPAOS
/SYS=RDOS
/SYS=RTOS
```

objectfile is the name of an object module or a library file

.OB is the optional object module extension

!* is the leftmost overlay area designator

! delimits individual overlays within an overlay area

*! is the rightmost overlay area designator

argswitch is one or more argument switches or argument switch sequences

The CLI command XEQ (abbreviated X) invokes Link from the utilities directory.

Link searches your working directory or search list for filenames that match the object modules named in the command line. Therefore, the .OB extension after each module name is optional.

You can include the names of named common symbols and user-defined partitions in the command line, provided you modify them with the correct switches or switch sequences.

Overlay Designators

You can designate certain object modules as overlays directly in the command line, by using the overlay designators !*, !, and *!. Each exclamation point-asterisk pair delimits an overlay area; single exclamation points set off the individual overlays destined for that area.

For example, the command line

```
) X LINK A !* OV1 OV2 ! OV3 OV4 *! B ↵
```

links six object modules: A, OV1, OV2, OV3, OV4, and B. There is one overlay area with two distinct overlays: modules OV1 and OV2 constitute the first overlay, while modules OV3 and OV4 constitute the second overlay.

By using the overlay designators, you can postpone structuring your program until you link it. That is, you can decide at Link time whether you want to use certain object modules as root or overlay resources. This gives you control over your program's structure and memory usage without reassembling or recompiling the modules.

Switches and Switch Sequences

Except for /MTOP=n, /RING=n, /SYS=DGUX, and /SYS=VS32, all of the switches and switch sequences for 32-bit modules apply to 16-bit modules as well. Table 5-5 lists additional switches that apply *only* to 16-bit modules.

Table 5-5. Switches Exclusive to 16-Bit Modules

Switch	Description
/BUILDSYSTEM	When used in conjunction with the /SYSTEM=AOS switch, builds an AOS system (.SY) program file.
/COMOVR	Directs Link to use an alternate memory allocation scheme compatible with other Data General linkers and binders. (If you initially defined common areas in an overlay, Link will place them in that overlay.)
/NRC	Directs Link to optimize all resource calls by converting them to EJSR instructions. If you use this switch, your program must explicitly handle all overlay management. (See the "Resource Resolution" section in this chapter.)
/NRP	Directs Link to optimize some overlay-to-overlay resource calls. If you use this switch, your program must observe several restrictions. (See the "Resource Resolution" section in this chapter.)
/PRSYM	Includes a symbol table in the program file. This table is used primarily by the RDOS debuggers.
/SYSTEM=VS16	Builds a 16-bit program and optional overlay and symbol table file for execution under AOS/VS.
/SYSTEM=AOS	Builds a 16-bit program and optional overlay and symbol table file for execution under AOS.
/SYSTEM=MPAOS	Builds a 16-bit program and optional overlay and symbol table file for execution under MP/AOS.
/SYSTEM=RDOS	Builds a 16-bit save and optional overlay file for execution under RDOS or DOS. The optional symbol table file will be in AOS/VS 16-bit format.
/SYSTEM=RTOS	Builds a 16-bit save and optional overlay file for execution under RTOS or SOS. The optional symbol table file will be in AOS/VS 16-bit format.
!*/ALIGNMENT=n	Aligns the relocation base of an overlay area to an address which is a multiple of 2^n where n is an integer between 0 and 12_8 inclusive. Note that Link automatically sets $n=12_8$ for shared overlays. AOS shared and RDOS virtual overlay areas are always aligned to a page (1024-word) boundary.
!*/MULTIPLE=n	Increases the number of basic areas in an overlay area to n. The default is 1. You can use this switch only for AOS or AOS/VS overlay areas. (See the *AOS or AOS/VS Programmer's Reference Manual* for a description of basic areas.)
!*/VIRTUAL	Creates a virtual overlay area. *Virtual overlays* reside in extended memory; that is, within physical memory, but outside your process's logical address space. You can use this switch on RDOS overlay areas. You MUST use this switch when declaring RTOS overlay areas. (See Appendix A for more information on virtual overlays.)

You can use only the following switches and switch sequences to modify an overlay area: /ALIGN=n, /MULT=n, /VIRTUAL, and /UC=SC and all variations of that switch sequence. If you use these switches, append them to the leftmost overlay designator (!*). Note that the /VIRTUAL switch applies only to RDOS and RTOS overlays. Refer to Appendix A for further details.

You must use one of the /SYS function switch sequences in the command line. Use /SYS=AOS to produce a .PR file to run under AOS; use /SYS=MPAOS to produce a .PR file to run under MP/AOS; use /SYS=VS16 to produce a .PR file to run under AOS/VS's 16-bit mode.

Given the /SYS=VS16 sequence, Link scans the system library URT16.LB at the end of pass one, to resolve any outstanding externals used in the modules. The /SYS=AOS sequence causes Link to scan the AOS system library, URT.LB. The /SYS=MPAOS sequence causes Link to scan the MP/AOS system library, OSL.LB. The appropriate library must be in your current directory or your search list. As we explained in Chapter 2, you can suppress the library scan entirely by using the /NSLS function switch in the command line.

The /SYS=RDOS and /SYS=RTOS sequences let you cross-link modules to run under the RDOS and RTOS operating systems, respectively. For details on RDOS/RTOS cross-linking, refer to Appendix A.

Object Module Differences

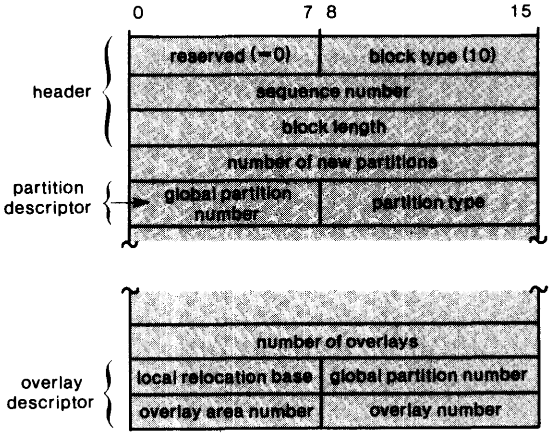
A 16-bit module can contain the same object blocks as a 32-bit module. However, there may be differences in the relocation entries generated by the language processor. For 16-bit modules, both standard and extended relocation entries and relocation dictionary entries are valid (32-bit modules must have the extended formats.) Refer to Chapter 3 for the structures of both types of relocation entries.

In addition to the object blocks described in Chapter 4, Link recognizes *address information blocks (AIBs)*, which define partitions and overlays, and *named common blocks*, which define named common areas. Although these block types are legal, partition definition blocks and overlay designators are more compatible with Link's relocation procedures.

Address Information Block

If you use AIBs, use only one per object module. You cannot apply overlay designators to a module that contains an AIB.

Once you've defined a partition with an AIB, it remains intact throughout the linking process. If you refer to that same partition in another module, however, you must redefine it with another AIB. Figure 5-6 shows the AIB structure.



SD-02193

Figure 5-6. Address Information Block

Besides the standard object block header, an AIB contains the following elements:

- a *partition descriptor* for each partition the block defines
- an *overlay descriptor* for each overlay the block defines
- a word indicating the number of partition descriptors
- a word indicating the number of overlay descriptors

Partition Descriptor

The left byte in the first partition descriptor word, *global partition number*, contains the relocation base Link will assign to the partition. Link assigns relocation bases 0-7 to the predefined partitions, and relocation bases starting at 8 to the user-defined partitions. Thus, this byte contains 8, or a value greater than 8; i.e., the next available relocation base.

The right byte of the first partition descriptor word contains the type number associated with one of the predefined partitions. Link uses this value to assign attributes to the new partition. For example, if the *partition type* in this byte is 5, the new partition will have all of the attributes of predefined partition 5.

Overlay Descriptor

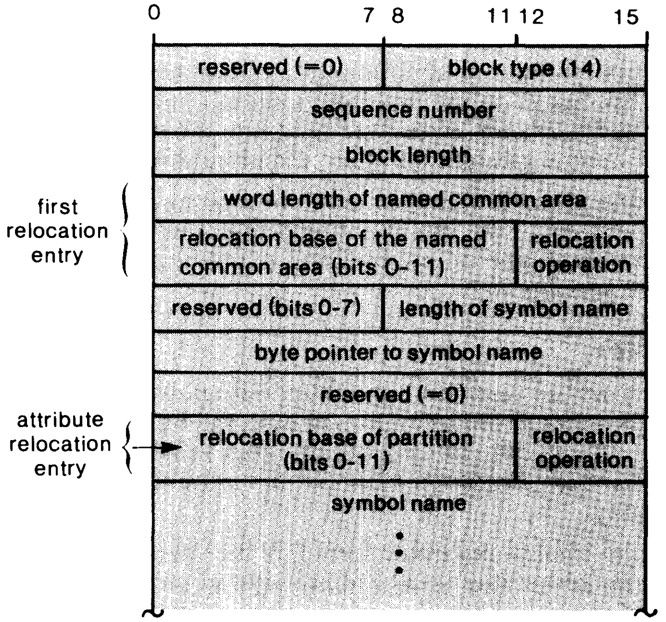
The left byte in the first overlay descriptor word, *local relocation base*, contains the relocation base Link will assign the overlay. *Global relocation base*, the right byte in this word, contains the relocation base of the corresponding partition. This value is the same as the global relocation base in the partition descriptor. Link uses this value to assign attributes and a .PR file location to the overlay.

The second word in the overlay descriptor identifies the overlay and its overlay area by number. You should number overlay areas and overlays consecutively; e.g., 00 for the first overlay in the first overlay area, 01 for the second overlay in the first overlay area, and so on.

Although both the overlay area number and overlay number occupy 1 byte each in the AIB, Link converts this to a word containing 6 bits for the overlay area number, and 9 bits for the overlay number.

Named Common Block

The named common block defines a named common area and its attributes. As Figure 5-7 indicates, each named common area shares the same attributes as one of the predefined partitions.



DG-27193

Figure 5-7. Named Common Block

Link uses the first relocation entry to resolve the contents and displacement of the named common area. The relocation base in the *attribute relocation entry* is that of the predefined partition associated with the named common area. The area inherits all of the attributes of this predefined partition.

Linking for MP/AOS

The MP/AOS execution environment is, in general, similar to the AOS and AOS/VS 16-bit execution environment. Therefore, there are not many differences between linking for AOS and linking for MP/AOS. Some of the differences that you should be aware of are:

- MP/AOS system calls differ in format and function from AOS system calls
- MP/AOS does not have facilities for unshared overlays
- MP/AOS does not have resource calls
- In MP/AOS, there is no User Status Table (UST) visible to the executing program
- The names of the default fault handlers are different for MP/AOS than for AOS/VS
- Link does not define a default stack for an MP/AOS program.

If you use a programming language that includes MP/AOS cross-development facilities, many of these differences will be invisible to you.

Before linking a program to execute under MP/AOS, you must be sure that all system-specific modules have been assembled or compiled for MP/AOS. Also be sure that you have all the appropriate libraries, including the language runtime libraries, if any, and the system library (OSL.LB) that Link scans by default. When linking an MP/AOS program, you must use the /SYSTEM=MPAOS switch sequence in the Link command line. This directs Link to build MP/AOS program, overlay and/or symbol table files from the input files.

End of Chapter

Chapter 6
Introduction to the Library File Editor (LFE)

This chapter and the next one describe the AOS/VS *Library File Editor (LFE)*, the utility you use to create, edit, and analyze library files. LFE works in parallel with Link, since you usually invoke it to create library files for Link input. (Throughout this chapter and Chapter 7, the term *library* refers to unshared libraries only. AOS/VS Link does not support shared libraries.)

A *library file* is a collection of object modules preceded by a *library start block* and terminated by a *library end block*. You can easily distinguish a library from a standard object module by its extension, .LB.

Libraries usually contain general-purpose modules applicable to more than one program. This saves you time and effort, since you need not duplicate the code for commonly used functions in your program. Instead, you can write code for a specific problem and then link in libraries of related modules. Most of the high-level languages support user runtime libraries for this reason.

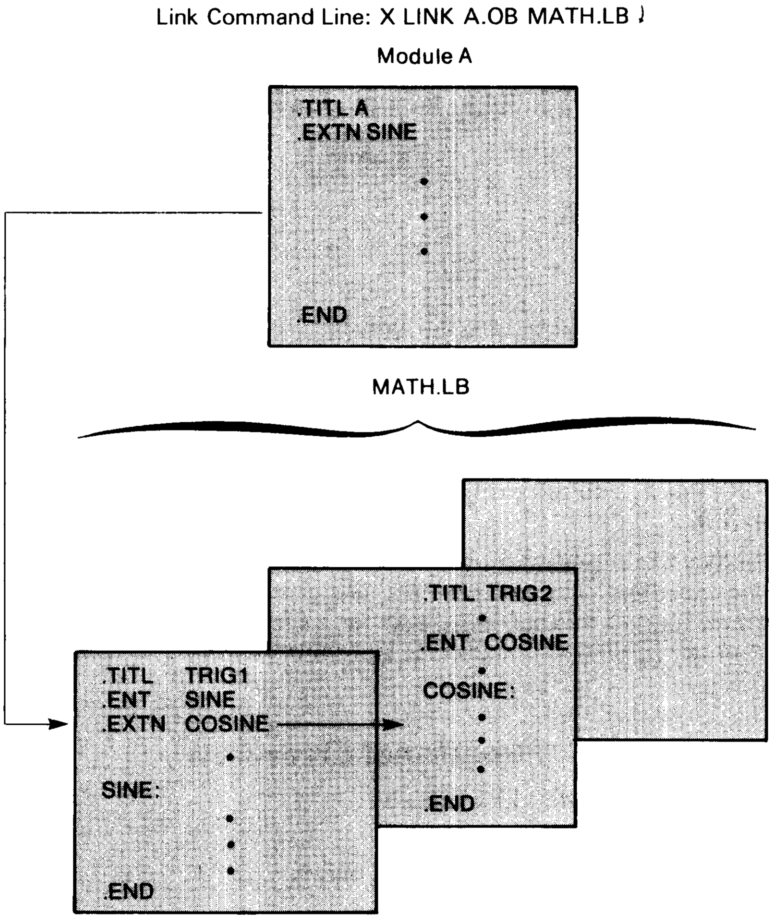
Functional Overview

In addition to building libraries, LFE can also

- analyze libraries
- delete, insert, or replace object modules in a library
- extract object modules from a library
- list the object module titles
- merge two or more libraries into a new library

LFE commands are called *function-letters*. You can use function switches to modify some of the LFE function-letters, and argument switches to modify the command line arguments. Chapter 7 describes the LFE command line in detail.

To make a library routine available to other programs, you must declare it as an entry symbol (.ENT). When Link detects an external reference to that symbol, it resolves the external by searching through the other modules in the command line for a matching entry. If the entry appears in a library module, Link includes that module in the .PR file. Figure 6-1 illustrates this concept.



DG-27194

Figure 6-1. Library Entry Resolution

Module A in Figure 6-1 contains an external reference to SINE, which appears as an entry in the module TRIG1 in the MATH.LB library. Given the command line X LINK A MATH.LB, Link sees the SINE reference in module A, recognizes it in TRIG1, and therefore, links TRIG1 into the final program file, A.PR.

Since Link scans the command line from left to right and does not rescan it, externals should appear before their entry symbol counterparts. If the command line in Figure 6-1 were X LINK MATH.LB A, for example, Link would see the SINE entry in MATH.LB first, and flag the SINE external in module A as a multiply defined symbol.

The same rule applies to the modules within libraries. For example, module TRIG1 in the sample library contains an external reference to the COSINE entry in TRIG2. Since TRIG1 appears before TRIG2, Link can easily satisfy the reference. If the order were reversed, however, Link would search unsuccessfully for the COSINE entry to satisfy TRIG1's external reference.

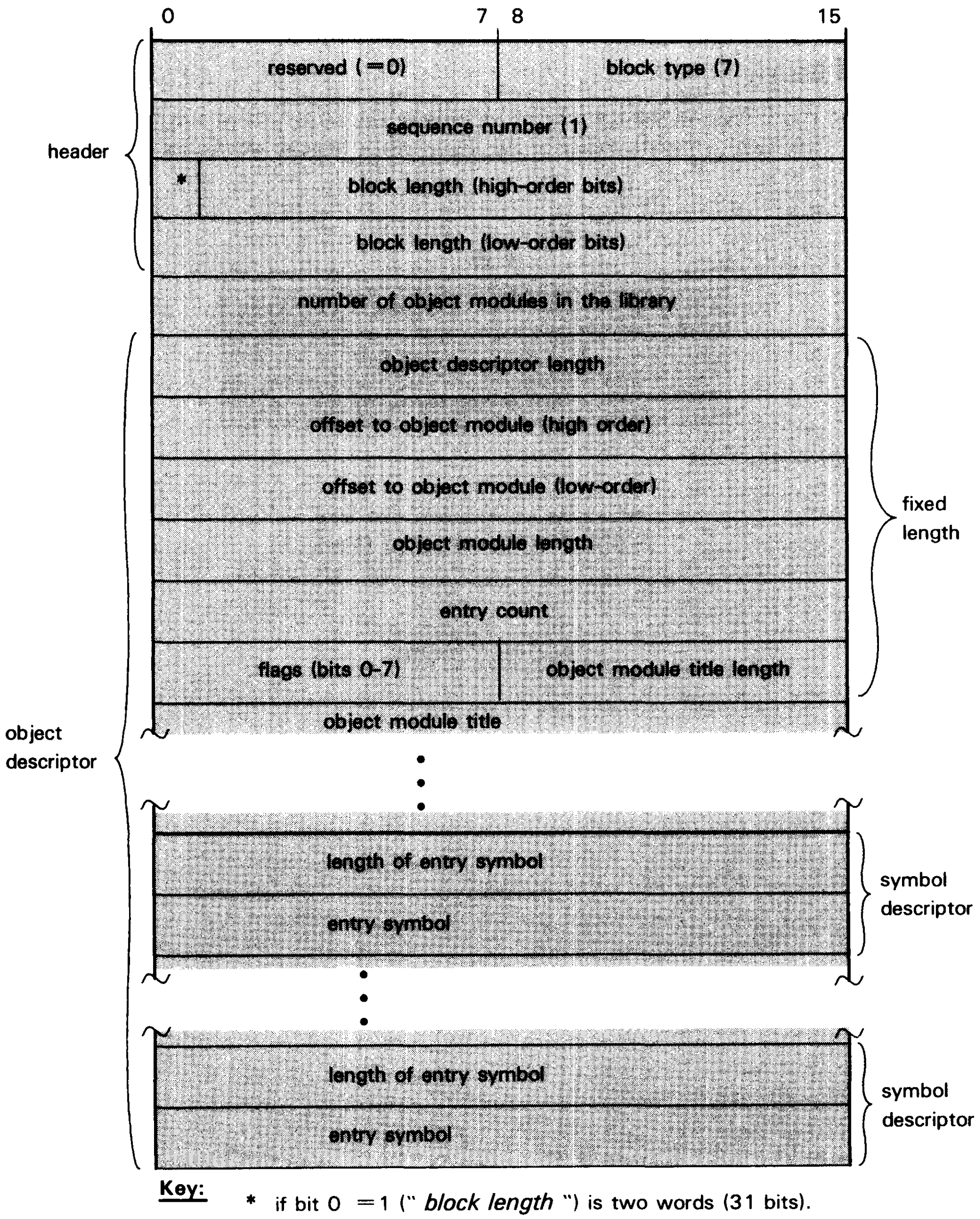
In some cases, you may want to deliberately put two or more identical entries in a library to satisfy different external conditions. (You may want Link to load one entry for certain external references, and other entries for other external references.)

Library Start/Library End Blocks

LFE precedes each library with a library start block and terminates it with a library end block. These blocks provide Link with descriptive information about each library module and its constituent entry symbols. Although you don't need to know the structures of these blocks to operate LFE, you will need this information if you plan to build unshared libraries without LFE.

Library Start Block

The library start block consists of a fixed 3 or 4-word header, followed by a series of variable-length *object descriptors*, one for each module in the library. Figure 6-2 shows the structure of the library start block.



ID-02756

Figure 6-2. Library Start Block

Unlike other blocks, library start blocks can exceed 1K (1024) words. If bit 0 in the *block length* word is 1, the block length is a double word (i.e., the block is > 32K words). Following the object block header is a word stating the number of object modules in the library.

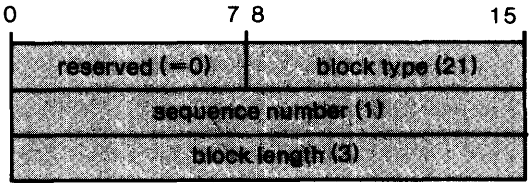
The block also contains one *object descriptor* for each object module in the library. The descriptor's fixed-length portion describes the length of the descriptor, and the length of the object module and its position relative to the other object modules in the library (i.e., its offset). The fixed-length portion also defines the number of entry symbols in the object modules. By entry symbols, we mean all ENT and PENT symbols.

The word following the entry count defines the flags in the module, including the forced load flag (forcing Link to load the module), and contains the byte length of the module's title. The title string follows this word.

LFE builds a *symbol descriptor* into the object descriptor for each entry symbol the object module contains. The first word in this descriptor states the byte length of the entry symbol.

Library End Block

The library end block follows the last object module in the library, and simply defines the end of the library. This block consists solely of the three block header words, as Figure 6-3 shows.



SD-02197

Figure 6-3. Library End Block

End of Chapter

Chapter 7
Using the Library File Editor

You invoke the Library File Editor by typing the AOS/VS CLI command XEQ (abbreviated X) followed by LFE and the appropriate function letter(s) and argument(s). The basic format of the LFE command line is

X*[EQ]* LFE*[fswitch]* function-letter argument...*[argswitch]*

where:

fswitch is one or more of the legal function switches for the LFE command.

function-letter is the LFE command.

argument is one or more arguments. (Depending on the function, an argument can be the name of a library or the name of an object module.)

argswitch is one or more of the legal argument switches for the function-letter.

To pass arguments to the X LFE command you can

- enter the arguments directly on the command line
- invoke a macro file containing the arguments
- invoke a CLI-format text file containing the arguments, and append the /CLI switch to the text filename

NOTE: The /CLI switch is useful when AOS/VS CLI is unable to expand a macro file due to insufficient memory. When using the /CLI switch, remember that your text file must adhere to CLI command syntax (i.e., separate arguments with spaces, tabs, or commas and use an ampersand (&) to continue a line.). Refer to Chapter 2 for more information on the /CLI switch.

In general, you use only one function-letter in the command line. (The exception to this is the analyze function, which accepts the combination A/M to analyze more than one library.) Table 7-1 lists the LFE function-letters. The rest of this chapter describes each function-letter individually.

Table 7-1. LFE Function-letters

Function Letter	Meaning
A	Analyze one or more library files or object files
D	Delete one or more object modules from a library
I	Insert one or more object modules into an existing library
M	Create a library by merging two or more existing library or object files
N	Create a new library from existing object files or library files
R	Create a new (output) library from an existing (input) library by replacing one or more object modules with other object modules. (This function *does not* change the input library file.)
T	List the title of each object module in the library, the starting address of each module, and the starting address of each module descriptor in the library start block
U	Update an existing library by replacing one or more object modules in the existing library with other object modules. (This function *does* change the existing library.)
X	Extract one or more object modules from a library, and place the module(s) in an object file(s). (This function *does not* change the library.)

A

Analyzes one or more library or object files.

Format Options

X*[EQ]* LFE*[/L[=listfile]][fswitch...]* A filename*[argswitch...]...*

X*[EQ]* LFE*[fswitch...]* A *[listfile/L]* filename*[argswitch...]...*

where:

/L=[listfile] or *listfile/L* — sends the analysis to *listfile*. If you do not specify a list file, LFE sends the analysis to @LIST. If you do not include /L on the command line, LFE sends the analysis to @OUTPUT.

fswitch — is one or more of the following switches:

- /F — puts the analysis of each object module on a separate page in the list file.
- /DECIMAL — sets the radix in the analysis to 10. The default output radix is 8.
- /HEXADECIMAL — sets the radix in the analysis to 16. The default output radix is 8.
- /XREF — generates an alphabetically sorted cross-referenced list of symbols and appends it to the original analysis.

A — is the function-letter.

filename — is one or more library files or object (.OB) files that you want LFE to analyze.

argswitch — is one or more of the following switches:

- /CLI — identifies the input file as a CLI-format text file.
- /LOCAL — directs LFE to include the object file's local symbols, if there are any, in the analysis. Without this switch LFE ignores local symbols.

A (continued)

Description

The A (Analyze) function-letter directs LFE to read, cross-reference and report the contents of one or more libraries or object files. The default output analysis includes:

- the title of each module
- the names and symbol types of all entry and external symbols in each module (includes local symbols in each module if /LOCAL switch is set)
- the number of words that each object module contributes to each default partition, user-defined partition, and common area
- the total number of words that all object modules contribute to all defined partitions and common areas
- the length of each module's debugger data, if any
- flags marking multiply defined, previously defined, and undefined symbols as follows:

*	Multiply defined symbol — the same entry symbol has been defined in more than one object module.
^	Previously defined symbol — this external symbol appears after its matching entry symbol.
?	Undefined symbol — LFE could not find a matching entry symbol within the object file for this external symbol.

LFE uses the following notation in its output listings:

ENTRY	entry symbol defined in a module
EXT	external symbol used in a module
COMM	named common area
ASYM	accumulating entry symbol
ENTO	overlay entry symbol
TITLE	title of a module
PENT	procedure entry symbol
EXTS	suppressible external symbol reference
LOCAL	local symbol defined in a module
EXTC	chain-link external symbol
LIMIT	limit external symbol reference
PART	normal partition
ENTS	suppressible entry symbol

Examples

1. `) X LFE/XREF/L=@LPT A BBUFFER BOPEN BWRITE BCLOSE BCBALLOC BFLUSH ↵`

2. `) X LFE/XREF A @LPT/L BBUFFER BOPEN BWRITE BCLOSE BCBALLOC BFLUSH ↵`

Each of these command lines directs LFE to analyze object modules BBUFFER, BOPEN, BWRITE, BCLOSE, BCBALLOC, and BFLUSH, and to send the results of this analysis to @LPT (the line printer). Because the /XREF switch is set, LFE includes an alphabetized, cross-referenced analysis of symbols as shown in the following sample listing.

```
  ANALYSIS BY TITLE
  -----------------

  TITLE   BBUFFER
  EXT     PAGESIZE     BCBALLOC
  EXT     PAGEBLOCKS   BCBALLOC
  EXT     BCBALLOC     BCBALLOC
  ENTRY   BBUFFER
  PART    SC           51

  TITLE   BOPEN
  EXT     PAGESIZE     BCBALLOC
  EXT     PAGEBLOCKS   BCBALLOC
? EXT     SCOPY
  EXT     BCBALLOC     BCBALLOC
  ENTRY   BOPEN
  PART    SC           62

  TITLE   BWRITE
  EXT     PAGESIZE     BCBALLOC
  EXT     PAGEBLOCKS   BCBALLOC
  EXT     BFLUSH       BFLUSH
  ENTRY   BREAD
  ENTRY   BWRITE
  PART    SC           327

  TITLE   BCLOSE
  EXT     PAGESIZE     BCBALLOC
  EXT     PAGEBLOCKS   BCBALLOC
  EXT     BFLUSH       BFLUSH
  ENTRY   BCLOSE
  PART    SC           52

  TITLE   BCBALLOC
  ENTRY   PAGESIZE     BBUFFER      BOPEN        BWRITE       BCLOSE
                       BFLUSH
  ENTRY   PAGEBLOCKS   BBUFFER      BOPEN        BWRITE       BCLOSE
                       BFLUSH
  ENTRY   BCBALLOC     BBUFFER      BOPEN
  PART    UC           1500
  PART    SC           36

  TITLE   BFLUSH
^ EXT     PAGESIZE    ^BCBALLOC
^ EXT     PAGEBLOCKS  ^BCBALLOC
  ENTRY   BFLUSH       BWRITE       BCLOSE
  PART    SC           73
```

A (continued)

```
  ANALYSIS BY SYMBOL
  ------------------

          BBUFFER     BBUFFER
          BCBALLOC    BBUFFER      BOPEN        BCBALLOC
          BCLOSE      BCLOSE
          BFLUSH      BWRITE       BCLOSE       BFLUSH
          BOPEN       BOPEN
          BOPENR      BOPEN
          BREAD       BWRITE
          BWRITE      BWRITE
          PAGEBLOCKS  BBUFFER      BOPEN        BWRITE       BCLOSE
                      BCBALLOC     BFLUSH
          PAGESIZE    BBUFFER      BOPEN        BWRITE       BCLOSE
                      BCBALLOC     BFLUSH
    TOTAL SC          665          SBUFFER      SOPENR       SWRITE
                      SCLOSE       BCBALLOC     BFLUSH
          SCOPY       BOPEN
    TOTAL UC          1500         BCBALLOC

  -------------------------

* MULTIPLY DEFINED SYMBOL
^ PREVIOUSLY DEFINED SYMBOL
? UNDEFINED SYMBOL
```

The first part of this analysis (analysis by title) lists symbol and partition data on a module-by-module basis. The first module, BBUFFER, uses external symbols PAGESIZE, PAGEBLOCKS, and BCBALLOC, all of which are defined in module BCBALLOC. The module also defines entry symbol BBUFFER and contributes 51_8 words to the shared code, normal partition. The first part of the analysis also shows that there is one undefined symbol, SCOPY, and that symbols PAGESIZE and PAGEBLOCKS are used in module BFLUSH *after* being defined in module BCBALLOC.

The second part of this analysis (analysis by symbol), lists all symbols alphabetically and shows reference and definition modules for each. This analysis also shows that the modules contribute a combined total of 665_8 words of shared code and 1500_8 words of unshared code.

D

Deletes one or more object modules from a library.

Format Options

X*[EQ]* LFE/I=library/O=newlibrary*[/DELETE]* D title *[title1...]*

X*[EQ]* LFE*[/DELETE]* D library/I newlibrary/O title *[title1...]*

X*[EQ]* LFE*[/DELETE]* D library newlibrary/O title *[title1...]*

where:

library/I
or
/I=library — is the input library, containing the modules you want to delete. If you use the third format (as demonstrated in Example 3) the /I is optional.

newlibrary/O
or
/O=newlibrary — is the output library (a reduced version of the input library).

/DELETE — is an optional switch which forces LFE to delete the output library before creating a new one. You should use this switch when you want the output library to have the same name as the input library.

title — is the title of the first object module you want to delete.

title1 — represents subsequent object modules you want to delete.

Description

The D (Delete) function-letter deletes one or more object modules from the input library (library/I) to produce a new library (newlibrary/O). The input library is unchanged.

Note that a *title* is a symbol defined within the library start block. The title does not necessarily have the same name as the .OB file that it was originally contained in.

Examples

1.) X LFE/I=MATH.LB/O=NEWMATH.LB D LOG EXP ↵
2.) X LFE D MATH.LB/I NEWMATH.LB/O LOG EXP ↵
3.) X LFE D MATH.LB NEWMATH.LB/O LOG EXP ↵

The LFE command lines in examples 1,2,and 3 are functionally equivalent. All three command lines create a new library, NEWMATH.LB, from the input library MATH.LB. NEWMATH.LB contains all of MATH.LB's object modules except LOG and EXP. MATH.LB is unchanged.

4.) X LFE/I=MATH.LB/O=MATH.LB D LOG EXP ↵
5.) X LFE/I=MATH.LB/O=MATH.LB/DELETE D LOG EXP ↵

The command line in example 4 generates an error because the input and output files have the same name. The /DELETE switch in example 5 prevents this error.

I

Inserts one or more object modules or libraries into a library.

Format Options

X*[EQ]* LFE/I=library/O=library*[/DELETE]* I title $\left\{\begin{matrix}/A\\/B\end{matrix}\right\}$ filename*[/F][/C]*...

X*[EQ]* LFE*[/DELETE]* I library/I library/O title $\left\{\begin{matrix}/A\\/B\end{matrix}\right\}$ filename*[/F][/C]*...

X*[EQ]* LFE*[/DELETE]* I library library/O title $\left\{\begin{matrix}/A\\/B\end{matrix}\right\}$ filename*[/F][/C]*...

where:

library/I
or
/I=library
is the input library. If you use the third format (as demonstrated in Example 3) the /I is optional.

library/O
or
/O=library
is the output library (an expanded version of the input library containing the inserted modules).

/DELETE is an optional switch which forces LFE to delete the output library before creating a new one. You should use this switch when you want the output library to have the same name as the input library.

title is the title of an object module in the input library. title must be followed by either /A or /B. title/A directs LFE to insert the modules immediately after title in the input library. title/B directs LFE to insert the modules immediately before title.

filename represents the object modules or libraries to be inserted.

/F sets the forced load flag in the object module; directs Link to load the object module unconditionally if the library containing the module appears in the Link command line.

/C clears the forced load flag in the object module; prevents Link from loading the module unless it satisfies an external reference.

Description

The I (Insert) function-letter inserts one or more object modules into the input library (library/I) to form a new output library (library/O). The input library remains unchanged.

Earlier versions of LFE accepted only .OB files for insertion; this revision allows you to insert any combination of .OB files and libraries.

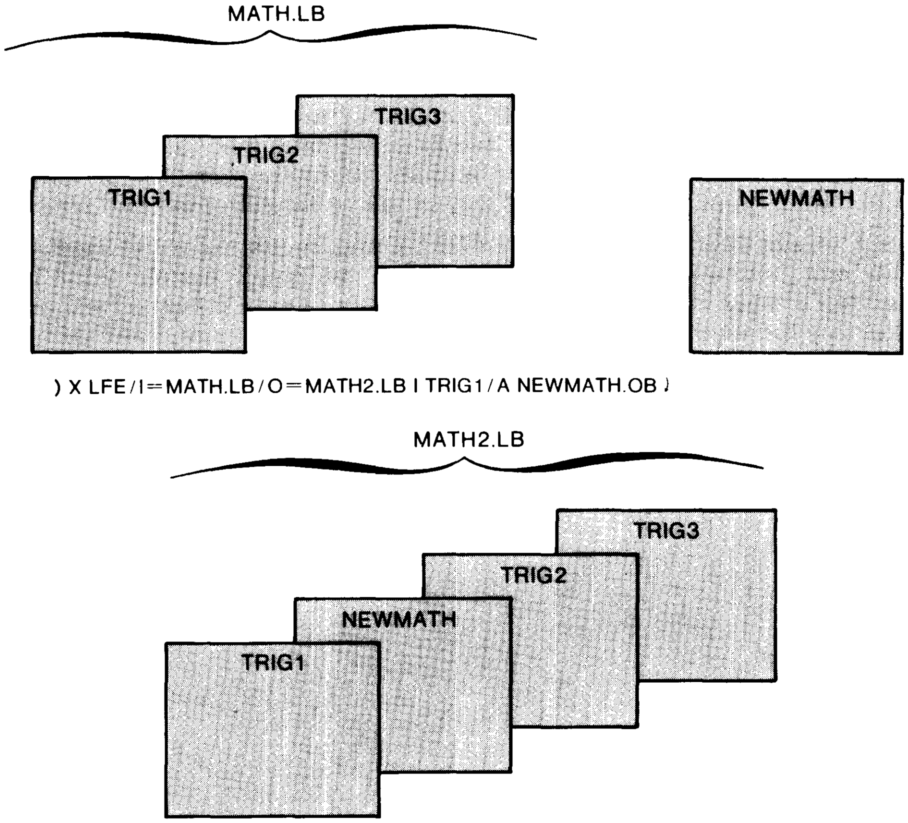
The /A and /B switches let you define where you want the new modules inserted. Title/A tells LFE to insert the modules *after* an existing module in the input library. Title/B tells LFE to insert the modules *before* an existing module in the library. (The title string specifies which module.)

The /F switch sets the forced load flags in the modules you're inserting, while /C clears the forced load flags in those modules.

Examples

1.) X LFE/I=MATH.LB/O=MATH2.LB I TRIG1/A NEWMATH.OB ↵
2.) X LFE I MATH.LB/I MATH2.LB/O TRIG1/A NEWMATH ↵
3.) X LFE I MATH.LB MATH2.LB/O TRIG1/A NEWMATH ↵

The command lines in examples 1, 2, and 3 each directs LFE to insert the object module NEWMATH into MATH.LB. The result is a new library, MATH2.LB. The new library contains all of MATH.LB's modules, plus NEWMATH. Module TRIG1 is in MATH.LB. Because we specified TRIG1/A, LFE will insert the new module after TRIG1 when creating MATH2.LB. Figure 7-1 shows the result of command lines 1, 2, or 3.



```
) X LFE/I=MATH.LB/O=MATH2.LB I TRIG1/A NEWMATH.OB ↵
```

DG-27195

Figure 7-1. Inserting Object Modules with I Function-Letter

I (continued)

```
4.  ) X LFE/I=MATH.LB/O=MATH2.LB I EXP/B CALC.LB/F ↵
5.  ) X LFE/I=MATH.LB/O=MATH2.LB I SIN/A ARCSIN.OB/C COS/A
    &) ARCCOS.OB/C TAN/A ARCTAN.OB/C ↵
6.  ) X LFE/I=MATH.LB/O=MATH2.LB I LOG/A MATRIX.LB JACOBIAN.OB ↵
```

Example 4 inserts all the object modules in CALC.LB just before the object module titled EXP. The /F switch appended to CALC.LB tells LFE to set the forced load flag of every object module in CALC.LB. In other words, whenever MATH2.LB appears on a Link command line, Link will automatically load every object module originally contained in CALC.LB.

Example 5 shows three separate insertions into MATH2.LB. First LFE inserts the object module contained in ARCSIN.OB just after the module titled SIN. Second, LFE inserts the object module contained in ARCCOS.OB immediately after COS. Finally, LFE inserts the object module contained in ARCTAN.OB after TAN. The /C switch appended to the input .OB files causes LFE to clear the forced load flags associated with these object modules. In other words, Link will load the object modules originally contained in ARCSIN.OB, ARCCOS.OB, and ARCTAN.OB only if they satisfy an unresolved external symbol.

Example 6 creates MATH2.LB from MATH.LB and inserts the object modules in MATRIX.LB and JACOBIAN.OB immediately after the object module titled LOG. For function-letter I, when two or more object files follow a title, LFE inserts them in the same order that they had on the command line. Since MATRIX.LB precedes JACOBIAN.OB on the command line, all the object modules in MATRIX.LB will precede the object module in JACOBIAN.OB in MATH2.LB.

M & N

Merges existing libraries and/or .OB files into a new library.

M Format Options

X*[EQ]* LFE*[fswitch...]* M library/O filename*[/F][/C]...*

X*[EQ]* LFE/O=library*[fswitch...]* M filename*[/F][/C]...*

N Format Options

X*[EQ]* LFE*[fswitch...]* N library/O filename*[/F][/C]...*

X*[EQ]* LFE/O=library[fswitch...] N filename*[/F][/C]...*

where:

fswitch is one or more of the following switches:

/DELETE is an optional switch which forces LFE to delete the output library before creating a new one. You should use this switch when you want the output library to have the same name as one of the input libraries.

/REV=aa.bb forces LFE to generate an object module consisting only of a title block and an end block. The object module title is always REVISION and the revision number, defined by the title block, is aa.bb. LFE then inserts the object module at the beginning of the library and sets its forced load flag.

library/O
or
/O=library is the output library.

filename is either an .OB file or a library file.

/F sets the forced load flag associated with this input file's object modules.

/C clears the forced load flag associated with this input file's object modules.

Description

The M (Merge) and N (New) function-letters merge existing libraries and/or .OB files to form a new output library. The contents of the input libraries and .OB files are unchanged.

If you specify only one input file in the command line, LFE copies the contents of that file to the designated output library (library/O). LFE does not return an error in this case.

NOTE: Function-letters M and N are functionally identical except in the way that LFE scans input files. When you use M, LFE first assumes that input files are library files; when you use N, LFE initially assumes that input files are .OB files. To illustrate, suppose that your working directory contains a library file named TRIG.LB and an .OB file named TRIG.OB. The following command line merges MATH.LB and TRIG.LB:

```
) X LFE M NEW.LB/O MATH.LB TRIG ↲
```

The following command line merges MATH.LB and TRIG.OB:

```
) X LFE N NEW.LB/O MATH.LB TRIG ↲
```

If you specify extensions in the LFE command line, M and N are functionally identical.

M & N (continued)

Examples

Using M

```
1.  ) X LFE M URT32.LB/O DGL.LB FORT.LB ↵
2.  ) X LFE/O=URT32.LB M DGL FORT.LB LANG.OB ↵
3.  ) X LFE/O=URT32.LB/DELETE/REV=2.15 M DGL FORT ↵
```

Example 1 merges two libraries, DGL.LB and FORT.LB, to form a third library, URT32.LB. Since DGL.LB is the first input library specified in the command line, its object modules will appear first.

Example 2 merges two libraries, DGL.LB and FORT.LB, and one .OB file, LANG.OB, to form library URT32.LB. In examples 1 and 2, if URT32.LB had been in your working directory prior to the merger, LFE would have sent out an error message and terminated.

Example 3 deletes URT32.LB (if present) and merges DGL.LB and FORT.LB to form a new URT32.LB. In addition, LFE inserts an object module titled REVISION, containing revision number 2.15, at the beginning of URT32.LB.

Using N

```
4.  ) X LFE N URT32.LB/O DGL.LB FORT.LB ↵
5.  ) X LFE/O=URT32.LB N DGL FORT.LB LANG.OB ↵
6.  ) X LFE/O=URT32.LB N DGL.OB/F FORT.LB ↵
```

Example 4 is functionally identical to example 1.

Example 5 creates a new library, URT32.LB, by merging FORT.LB with .OB files DGL.OB and LANG.OB. Note the difference between this and example 2.

Example 6 creates URT32.LB by merging DGL.OB with FORT.LB. The /F switch sets the forced load flag on the object module stored in DGL.OB; therefore, if URT32.LB is included in a Link command line, Link will automatically load the object module originally stored in DGL.OB.

R

Replaces a library object module with an object module or a library.

Format Options

X*[EQ]* LFE/I=library/O=library*[/DELETE]* R title filename*[/F][/C]...*

X*[EQ]* LFE R library/I library/O title filename*[/F][/C]...*

X*[EQ]* LFE R library library/O title filename*[/F][/C]...*

where:

library/I
or
/I=library — is the input library. If you use the third format (as demonstrated in example 3) the /I is optional.

library/O
or
/O=library — is the output library (a modified version of the input library).

/DELETE — is an optional switch which forces LFE to delete the output library before creating a new one. You should use this switch when you want the output library to have the same name as the input library.

title — is the title of the object module you want to replace.

filename — is one or more .OB files or libraries you want to substitute for title.

/F — sets the forced load flag associated with this input file's object modules.

/C — clears the forced load flag associated with this input file's object modules.

Description

The R (Replace) function-letter replaces object modules in the input library with new object modules to form a new library (library/O). The contents of the input library are unchanged.

On the command line, there must be a one-to-one correspondence between titles and replacement filenames. You can set (/F) or clear (/C) the forced load flags in the replacement modules.

Examples

1.
```
) X LFE/I=MATH.LB/O=NEWMATH.LB R EXP EXP2.OB/F ↵
```
2.
```
) X LFE R MATH.LB/I NEWMATH.LB/O EXP EXP2.OB/F ↵
```
3.
```
) X LFE R MATH.LB NEWMATH.LB/O EXP EXP2.OB/F ↵
```

Examples 1,2, and 3 all do the same thing. They force LFE to copy NEWMATH.LB from MATH.LB and to replace the object module titled EXP with the object module stored in EXP2.OB. The /F switch tells LFE to set the forced load flag in NEWMATH.LB associated with EXP2.OB. MATH.LB is unaffected.

4.
```
) X LFE/I=MATH.LB/O=NEWMATH.LB R SIN COSIN.LB TAN ARC.LB
&) LOG LOG3.OB ↵
```

Example 4 replaces three object modules in MATH.LB to form a new expanded library called NEWMATH.LB. First, LFE replaces the object module titled SIN with all the object modules in COSIN.LB. Next, LFE replaces the object module titled TAN with all the object modules in ARC.LB. Finally, LFE replaces the object module titled LOG with the object module stored in LOG3.OB.

T

Lists the title of each object module in a library, and certain library start block information.

Format Options

X*[EQ]* LFE*[/L[=listfile]][fswitch...]* T library...

X*[EQ]* LFE*[fswitch...]* T *[listfile/L] library...*

where:

/L[=listfile] or *listfile/L* — sends the output to *listfile*. If you do not specify a list file, LFE sends the output to @LIST. If you do not include /L on the command line, LFE sends the output to @OUTPUT.

fswitch — is one or more of the following switches:

- /F — puts the analysis of each library on a separate page in the list file.
- /DECIMAL — sets the listing radix to 10. The default output radix is 8.
- /HEX — sets the listing radix to 16. The default output radix is 8.

library — is the name of one or more library files for which you want object module titles and information.

Description

The T (Title) function-letter lists the following information:

- the title of every object module in the library
- the object descriptor address of every object module (LFE builds an object descriptor into the library start block for each object module in the library. Refer to Chapter 6 for the structure of the library start and end blocks.)
- the starting position of every object module's title block
- the size (in words) of every object module
- an asterisk in front of any object module in which the forced load flag is set

The function-letter A produces a much more detailed analysis of a library than function-letter T, but T is faster. LFE must scan an entire library to produce the full analysis A; however, LFE scans only the library start block to produce the title analysis T.

Examples

```
1.  ) X LFE/L=FAST T MATH.LB CALC.LB ↵
2.  ) X LFE T FAST/L MATH.LB CALC.LB ↵
3.  ) X LFE/HEX/F T TRIG GEOM INTEGRAL ↵
```

Examples 1 and 2 are functionally identical. Both command lines produce title analyses of libraries MATH.LB and CALC.LB.

Example 3 produces title analyses of libraries TRIG.LB, GEOM.LB, and INTEGRAL.LB. Because of the /HEX switch, the listing will be in base 16; the /F switch ensures that the title analysis of each library starts on a fresh page.

U

Updates an existing library by replacing an object module.

Format

X*[EQ]* LFE U library title filename*[/F][/C]...*

where:

library	is the library you want to update.
title	is the title of one or more object modules you want to replace.
filename	is one or more .OB files you want to substitute for title.
/F	sets the forced load flag associated with this input file's object modules.
/C	clears the forced load flag associated with this input file's object modules.

Description

The U (Update) function-letter replaces object modules in a library with other object modules. Unlike the R (Replace) function-letter, which preserves the original library, U deletes the object module from the original library and places a copy of the new module directly into the original library. When updating a large library, the U function can be significantly faster than the R function because LFE does not have to copy the entire library file.

However, because LFE puts the new module directly into the library in place of the old one, you must observe certain restrictions when using the U function.

Because LFE cannot change the size of the module descriptor in the library start block, the new module's title and entry point name(s) must not be longer than the name(s) in the module it replaces. If the new module's title does not fit in the descriptor, LFE reports an error and does not update the module. If an entry point name does not fit in the descriptor, LFE also reports an error and omits the entry point, though not the entire module, from the descriptor. Note that these restrictions apply only to the module descriptor; the module itself can be any length regardless of the length of the module it replaces. Similarly, if LFE omits an entry point name from the descriptor, the name is still present in the module itself.

When you update a library, LFE appends new modules to the existing library, but does not automatically recover space left in the library file by unused modules. To recompact a library that has been updated many times, you can use either the M (Merge) or N (New) function-letter to merge the library into itself.

Example

```
) X LFE U LIB.LB OLD NEW.OB ↵
```

This command line finds module OLD in library LIB.LB, appends NEW.OB to the library file and replaces OLD's module descriptor with a descriptor for module NEW.

X

Extracts one or more object modules from a library.

Format Options

X*[EQ]* LFE/I=library X title...

X*[EQ]* LFE X library/I title...

X*[EQ]* LFE X library title...

where:

library/I
or
/I=library — is the input library (from which you will extract one or more object modules).

title — is the title of one or more object modules to be extracted.

Description

The X (Extract) function-letter extracts copies of object modules (titles) from the input library and builds each one into a free-standing object module (.OB). The input library is left unchanged.

If your directory already contains a free-standing module by the same name, the command fails and LFE returns an error message.

Be careful not to confuse the title of an object module with the name of the .OB file in which it was originally stored.

Examples

1.) X LFE/I=MATH.LB X TRIG1 GEOM1 ↵
2.) X LFE X MATH.LB/I TRIG1 GEOM1 ↵
3.) X LFE X MATH.LB X TRIG1 GEOM1 ↵

Each command line extracts a copy of TRIG1 and GEOM1 from library MATH.LB. The extracted modules will appear in your working directory as TRIG1.OB and GEOM1.OB. If the working directory already contains a file named TRIG1.OB or GEOM1.OB, LFE will return an error.

LFE Error Messages

By default, LFE sends all error messages, as they occur, to the generic @OUTPUT file. (In most cases the @OUTPUT file is your terminal.) If you specify a list file for functions that accept the /L=listfile switch, LFE will send error messages to that list file.

This section lists the current LFE error messages and their possible causes. Some errors cause LFE to abort; others allow processing to continue. Note that the following list highlights fatal (abort) errors with a single asterisk. Angle brackets < > indicate the paraphrase of an LFE-supplied value or character string. (See the "Key" at the end of the list for an explanation of other symbols.)

BLOCK NUMBER OUT OF SEQUENCE **

The sequence word in an object block contains a value that does not conform to the block numbering order (see Chapter 4). LFE ignores the rest of the file that contains that object block.

/<switchname> = <argument> DOES NOT ACCEPT AN ARGUMENT *

You supplied an argument for a switch that does not accept arguments (e.g., /F=library).

DUPLICATE INPUT FILE SPECIFICATION *

You used a global or local /I switch more than once on the LFE command line.

DUPLICATE LIST FILE SPECIFICATION *

You used a global or local /L switch more than once on the LFE command line.

DUPLICATE OUTPUT FILE SPECIFICATION *

You used a global or local /O switch more than once on the LFE command line.

FILENAME ALREADY EXISTS, FILE <pathname> ***

You specified an output file that already exists. You must either choose a new output file or delete the existing file with the CLI command DELETE <pathname> or the LFE /DELETE switch.

FUNCTION DOES NOT ACCEPT INPUT FILE *

You specified an input file (with /I) while using function-letter A (Analyze), M (Merge), N (New), or T (Title).

FUNCTION DOES NOT ACCEPT LIST FILE *

You specified a listing file while using function-letter D, I, M, N, R, U, or X. (Only function letters A (Analyze) and T (Title) accept an /L or /L=listfile switch.)

FUNCTION DOES NOT ACCEPT OUTPUT FILE *

You specified an output file (with /O) while using function-letter A (Analyze) or T (Title).

INVALID BLOCK SIZE **

The size word in an object block contains a value less than 3 or greater than 1024_{10}. LFE ignores the rest of the file that contains that object block.

/<switchname>[=<argument>] IS A DUPLICATE OR CONFLICTING SWITCH *

Either you used the same switch more than once on one argument (e.g., /F/F) or you used mutually exclusive switches on the same argument (e.g., /F/C).

/<switchname>=<argument> IS AN ILLEGAL ARGUMENT VALUE *

Either you supplied a nonnumeric argument where a numeric argument is required (e.g., /REVISION=twelve instead of /REVISION=12) or you supplied a revision number that is not syntactically correct (/REVISION=12..34 instead of /REVISION=12.34).

<argument> IS AN UNKNOWN FUNCTION *

You did not supply a function name or function-letter as the first argument to LFE.

/<switchname>[=<argument>] IS AN UNKNOWN OR NON-UNIQUE SWITCH *

Either you used a switch that LFE does not recognize (e.g., /XYZ) or you used a switch that is not unique (e.g., /DE which might represent /DECIMAL or /DELETE).

<pathname> IS NEITHER AN OBJECT NOR A LIBRARY

You specified an input file on the command line that does not begin with either a library start block or a title block.

<pathname> IS NOT A LIBRARY *

You specified a primary input file that is not a library file while using function-letter D (Delete), I (Insert), R (Replace), T (Title), U (Update), or X (Extract). (LFE determines this by checking the block-type word of the library start block in the specified file.)

NO INPUT FILE SPECIFIED *

While using function-letters D (Delete), I (Insert), R (Replace), U (Update), or X (Extract), you did not specify the input file. If you do not specify an input file explicitly (with /I), LFE assumes that the second argument on the command line is the input file. If that second argument is not an input file name, LFE returns this error.

NO OUTPUT FILE SPECIFIED *

You did not specify an output file (with /O) while using function-letter D, I, M, N, R, U, or X. (Only function letters A (Analyze) and T (Title) do not require an /O switch.)

NOT ENOUGH ARGUMENTS *

Either you gave too few arguments (all LFE functions require at least one argument) or you did not pair the titles and objects in an R (Replace) or U (Update) command.

<title> NOT UPDATED

While using function-letter U (Update), you specified a substitute object module whose module descriptor is longer than the descriptor of the module you want to update. LFE cannot perform this update and retains the old descriptor. (For more information see the description of function-letter U in this chapter.)

OBJECT FILE NAME DOES NOT FOLLOW TITLE

While using function-letter R (Replace) or U (Update), you did not specify both an object module to be replaced or updated and an object module to substitute. (You must give these arguments in pairs.)

<symbol> OMITTED FROM DESCRIPTOR

While using function letter U (Update), you specified a substitute object module that contains an entry symbol descriptor longer than the symbol descriptor in the module you want to update. LFE omits the new symbol descriptor from the entry symbol list, but will update the rest of the module. (For more information see the description of function letter U in this chapter.)

/<switchname> REQUIRES AN ARGUMENT *

You failed to qualify a switch sequence with an argument (e.g., /O instead of /O=library).

UNKNOWN EXTERNAL NUMBER **

There is an object block that refers to an external symbol by number. If that number is either a three or a value higher than the number of externals defined in that module, LFE returns this error and continues processing.

<title> WAS NOT FOUND IN <pathname>

While using function-letter D (Delete), I (Insert), R (Replace), or U (Update), you specified an object module name that LFE did not find in the input library.

Key:

*	Fatal error.
**	Possible compiler error. (If you are certain that this error was not due to a faulty command line or some other user error, please report it to your system manager or the appropriate Data General compiler/assembler support personnel.)
***	AOS/VS system error. (Refer to the *AOS/VS Programmer's Manual* for more information.)

End of Chapter

Appendix A
Linking 16-Bit Modules for RDOS and RTOS

This appendix describes the procedures for linking 16-bit modules to run under Data General's Real-time Disk Operating System (RDOS) and Real-Time Operating System (RTOS).

The cross-linking procedures described in this section allow you to develop applications programs under the AOS/VS or AOS systems, and then execute those programs in real-time environments specially tailored to the application. Note that cross-linking will not work for many high-level language modules due to the system calls they use. Among Data General's languages, cross-linking is useful only for assembly language (provided you code your system calls for RDOS) and the DG/L™ language.

What we provide here is a summary of cross-linking from the perspective of AOS/VS Link. A full description of the RDOS and RTOS systems is outside the scope of this manual.

System Overview

RDOS is a disk-based operating system capable of supporting interactive or batch program development, and providing runtime support for real-time environments. Although RDOS has fewer features than AOS/VS or AOS, it does provide fast program execution and real-time support.

RTOS is a real-time, memory-resident subset of the RDOS system. It is compatible with RDOS and has the same general design. However, RTOS is a memory-resident rather than a disk-based system. In addition, it does not support text editors and utilities (e.g., the CLI), as RDOS does. The main advantage of RTOS is its capacity for dedicated applications and fast, limited-memory program execution. When you build RTOS programs, you build in the system modules needed for execution. Because an RTOS program contains both user code and the operating system itself, there are no system calls for RTOS.

Cross-Linking Differences

To link object modules for RDOS or RTOS, you must use the /SYS=RDOS or /SYS=RTOS function switch sequences in the Link command line. The program you produce is an RDOS or RTOS program file with the .SV extension. For example, both of the following command lines produce the program file GREEN.SV:

```
) X LINK/SYS=RDOS GREEN.OB BLUE.OB ⤶
```

```
) X LINK/SYS=RTOS GREEN.OB BLUE.OB ⤶
```

Notice that the modules specified in the command lines are object modules; Link performs the necessary conversions to create the .SV files.

In general, Link produces .SV files the same way it produces .PR files. But the linking process differs in the following respects:

- RDOS system calls are incompatible with AOS/VS and AOS system calls, and RTOS does not have system calls. Therefore, you cannot cross-link modules that use AOS/VS or AOS system calls.
- The /SYS=RDOS function switch causes Link to scan the RDOS system library, SYS.LB, which contains modules written to serve runtime needs. However, the modules in SYS.LB are RDOS .RB modules; therefore, you must convert them to object module format before you try to cross-link. (Refer to "Preliminary Steps" in this appendix for the conversion procedures and to Appendix B for a description of the CONVERT utility.)
- RTOS does not have a system library; therefore, there is no system library scan when you use the /SYS=RTOS switch sequence.
- Link does not set the stack size for an .SV program file. RDOS and RTOS rely on the stack parameters (if any) set by the language processor.
- RDOS allows a maximum of 124_{10} overlay nodes (overlay areas), numbered consecutively from 0 through 123. Each overlay node can contain a maximum 256_{10} overlays, numbered consecutively from 0-255. The left byte of an RDOS overlay descriptor word contains the overlay node number, and the right byte, the overlay number. RDOS recognizes two kinds of overlays: conventional and virtual. (See "RDOS Overlays" in this appendix for details.)
- Link cannot perform resource resolution for an .SV program file, because the resource handling methods for RDOS and RTOS differ from those for AOS/VS and AOS.
- The *shared partition* and *shared overlay* concepts are meaningless under RDOS. You can still use the predefined shared partitions in modules you're cross-linking, but Link will interpret these partitions as unshared partitions, and place them contiguously in the .SV file and the map file. Figure A-1 illustrates this.
- Link produces RDOS- or RTOS-style system tables for an .SV file.
- At execution time, the RDOS system precedes an .SV file with 16_8 words of system header information. Therefore, each address of user code in an .SV file is offset by 16_8 words.

Partition #	**Type**	**Length**	**Start**	**End**
partition 5 (shared data)	UD	00000000000		
partition 6	UD	00000000761	00000000447	00000001427
partition 7 (shared code)	UC	00000000422	00000066000	00000066421
partition 4	UC	00000007444	00000070000	00000077443

Figure A-1. Cross-Linking: Map Partition Parameters

Preliminary Steps

Link cannot create an executable program file unless it can scan the appropriate system library(s). Therefore, you must bring the system library(s) to AOS/VS from your target system.

Furthermore, since the RDOS and RTOS system libraries contain *relocatable binary* (.RB) modules rather than AOS/VS type (.OB) object modules, you must convert these libraries to object module form before doing a cross-link.

For example, to load and convert the RDOS system library (SYS.LB), perform the following steps:

Under RDOS:

1. First, use the RDOS Library File Editor to separate SYS.LB into its component .RB modules. Use the RDOS LFE T command to get the name of each .RB module, and the X command to extract each .RB from the library. Refer to the *RDOS Library File Editor (LFE) User's Manual* for details.
2. DUMP the individual .RB modules to tape.

Next, you must load the .RB tape onto your AOS/VS system and convert the .RB files to object module format. This involves the following steps:

Under AOS/VS:

3. After mounting the tape, use the CLI command X RDOS LOAD to load your .RB files. Refer to the *AOS and AOS/VS CLI User's Manual* for information on using the RDOS LOAD command. Be sure to load your files into the directory you will use for the link, or into a directory named in your search list.
4. Use the CLI command X CONVERT to convert your .RB files to object module format. Refer to Appendix B in this manual for information about the CONVERT utility.
5. Finally, use the AOS/VS Library File Editor (LFE) utility to build the converted modules into a new SYS.LB. Be sure to name the library SYS.LB.

NOTE: If you write and assemble (or compile) your RDOS or RTOS source code under RDOS, you must convert the .RB modules to .OB modules before linking.

After performing these preliminary steps, you can enter the appropriate Link command line.

The Command Line

You use the same Link command line to produce .SV files as you do to produce .PR files, except that you must use either the /SYS=RDOS or /SYS=RTOS function switch sequences. You also use the same overlay designators to define overlays (i.e., `!* ov1 ov2 ! ov3 ov4 *!`).

You *cannot* use the following function switches and switch sequences for cross-linking:

/KTOP=n
/MTOP=n
/SHARED
/SRES=n
because there are no shared partitions under RDOS or RTOS (/KTOP, /SRES, /SHARED), and both systems support memory configurations smaller than 2 megabytes (/MTOP).

/MULT=n — because you cannot increase the basic size of an RDOS overlay area.

/NRP
/WRL
because the RDOS and RTOS resource management schemes differ from those of AOS/VS.

/STACK=n — because Link does not set the stack size for an .SV program file

You *can* use the following additional switch when cross-linking for RDOS or RTOS:

/VIRTUAL — which defines a virtual overlay area for RDOS or RTOS. You can use this switch when designating overlays for RDOS. You must use this switch when designating overlays for RTOS. (See "RDOS and RTOS Overlays" in this appendix.)

If you use /VIRTUAL, place it immediately after the leftmost overlay designator. For example,

```
!*/VIRTUAL A B ! C D *!
```

defines one virtual overlay node (ABCD) with two virtual overlays, AB and CD.

Switch Differences

The following switches and switch sequences have different meanings under RDOS and RTOS: /DEBUG, /PRSYM, /ULAST=n, /SD=SC, /SC=SD, /SD=UC, /SD=UD, and /UDF.

The /DEBUG switch, which normally emits the external symbol DEBUG for high-level language debugging, emits the same external symbol under a cross-link, but for use by the RDOS Debugger utility. When you use this switch, Link resolves the debug symbol and forces a copy of the RDOS Debugger into the .SV file. If the modules contain one or more debugger symbols or debugger lines blocks, Link also produces the .DS or .DL file. RDOS does not use these files so you may want to delete them.

NOTE: If you intend to debug your RDOS or RTOS .SV file, you can include the /PRSYM switch in your cross-link command line. This switch will allow you to use on-line symbols during a debugging session.

The /ULAST=n sequence can take either a shared or an unshared partition as an argument under a cross-link, since shared partitions are meaningless under RDOS and RTOS. The equivalence statements (/SD=SC, etc.) are meaningless for the same reason. (SD and SC partitions are labeled as such on the map, but they lack the shared attribute.)

The /UDF function switch conflicts with the /SYS=RDOS and /SYS=RTOS sequences, and therefore, has no meaning under a cross-link. This switch directs Link to build a program file starting at location 0, without system tables, whereas /SYS=RDOS and /SYS=RTOS produce program files offset from location 16(sub 8), with RDOS and RTOS system tables.

RDOS and RTOS Overlays

The RDOS system supports two kinds of overlays: conventional overlays and virtual overlays.

Conventional overlays reside in the .OL file on disk, until your program calls them as it executes. When this occurs, the system reads the appropriate overlays into the overlay node Link constructed in your .SV file.

Link builds conventional RDOS overlay nodes in disk block multiples (one disk block equals 256 words). If you define an overlay node that is less than one block, Link pads it with zeros to equal one block. The same rounding occurs for overlay nodes larger than one block, but smaller than two blocks, and so on.

Virtual overlays reside in extended memory - within main memory, but outside your program's logical address space. When you call a virtual overlay, the system reads it into a virtual overlay node in your address space. Link builds virtual overlay nodes in multiples of 1K words, padding the nodes with zeros, if necessary, to equal 1K or a multiple of 1K. Link also aligns virtual overlay nodes to 1K word boundaries.

To declare virtual overlays from the Link command line, append the /VIRTUAL switch to the leftmost overlay designator. (See "Command Line" in this appendix for an example.) Normally, you must specify virtual overlays first under RDOS. However, don't worry about this if you're cross-linking, because Link automatically places virtual overlays first.

All RTOS overlays must be virtual, since RTOS is a memory-based, rather than a disk-based system. RTOS accepts a maximum of five characters in a filename, and no filename extensions. This means you must rename the Link-generated output files before you can execute a cross-linked program under RTOS.

Overlay files are contiguous under both RDOS and RTOS. Again, Link handles this conversion problem internally, so you don't have to worry about it.

End of Appendix

Appendix B
Using the CONVERT and MKABS Utilities

CONVERT and MKABS are two utilities provided as development tools with your AOS/VS system. CONVERT translates RDOS and RTOS relocatable binary (.RB) files into AOS/VS object (.OB) files. You will use CONVERT if you intend to link programs under AOS/VS for execution under RDOS or RTOS. MKABS translates RDOS and RTOS save (.SV) files into absolute binary (.AB) files. You will use MKABS if you want to execute RDOS or RTOS program files on a paper-tape system.

Using CONVERT

CONVERT accepts as input a series of .RB files and/or relocatable binary libraries and produces a series of corresponding, functionally equivalent, .OB files. You can use these .OB files as input to AOS/VS Link or LFE.

Relocatable binary files are the object files created when you write and compile (or assemble) RDOS or RTOS source code on your RDOS system. Under RDOS you can use these .RB files as input to the Relocatable Binary Loader (RLDR), which loads binaries and libraries to produce executable save files. However, if you choose to Link your relocatable binary object files under AOS/VS (with /SYS=RDOS or /SYS=RTOS), you must first convert .RBs to AOS/VS type object modules (.OBs).

As explained in Appendix A, Link scans the appropriate system library(s) when creating executable program files; therefore, you must load these libraries into your AOS/VS system before executing Link. Since RDOS and RTOS system libraries comprise .RB files, you must also convert the libraries to .OB format before cross-linking. (Note that you must transfer and convert RDOS or RTOS system libraries even if you write and assemble RDOS or RTOS source code under AOS/VS.)

Converting .RB Files

First make sure that the .RB files you want to convert are in your working directory or in a directory in your search list (see "Preliminary Steps" in Appendix A). To invoke CONVERT use the AOS/VS CLI command format

X*[EQ]* CONVERT*[/L[=listfile]]* filename*[.RB]...*

CONVERT will read filename and produce filename.OB for each filename you specify.

For example, X CONVERT MOD1 MOD2 converts MOD1.RB and MOD2.RB into MOD1.OB and MOD2.OB.

If you use */L=listfile* or */L*, CONVERT appends its output to listfile or @LIST, respectively. By default, CONVERT sends its output to @OUTPUT. Note that when you use */L* or */L=listfile*, CONVERT sends its output to a list file in CLI format; therefore, you can use listfile on the Link or LFE command line.

Converting Relocatable Binary Libraries

CONVERT can read a library of relocatable binary files, convert its constituent .RBs to .OBs, and produce as output the individual .OB files. The filenames of the individual .OB files will be the same as their respective titles.

To convert an RDOS or RTOS library into a library usable by Link, follow these steps:

1. Before executing CONVERT, load the RDOS or RTOS library from tape onto your AOS/VS system (see "Preliminary Steps" in Appendix A).
2. Execute CONVERT
3. Use the AOS/VS Library File Editor (LFE) to combine the individual .OB files into an AOS/VS format library.

The following example illustrates these steps. (This example uses the RDOS system library (SYS.LB) from the RDOS release tape.)

```
) X RDOS LOAD/V @MTxx:6 SYS.LB ↵
) X CONVERT/L=SYS_FILES SYS.LB ↵
) DELETE SYS.LB ↵
) X LFE N SYS.LB/O [SYS_FILES] ↵
```

In this example, the first command line invokes the "RDOS LOAD" program to load SYS.LB from the RDOS release tape. The second command line invokes CONVERT to change SYS.LB binary files into individual .OB files and send them to SYS_FILES. The third command deletes the old SYS.LB. The final command line invokes LFE to create a new library named SYS.LB, combining the object modules in SYS_FILES. (Note that the brackets surrounding SYS_FILES in the LFE command line direct LFE to use *the contents* of SYS_FILES as input object files.)

Using MKABS

RDOS and RTOS save (.SV) files are program files executable under RDOS or RTOS, respectively. The MKABS utility accepts .SV files as input, translates them into absolute binary (.AB) files, and outputs the .AB files to a file or a device. MKABS is useful when you need to transfer RDOS or RTOS programs to a paper-tape system. Ordinarily *absolute binary files* are written to paper-tape for use with an absolute binary loader. The *absolute binary loader* accepts .AB files on paper-tape and loads them for execution. To invoke MKABS, use the CLI command format

X*[EQ]* MKABS*[/switch]* inputfile outputfile

where

/switch is one or more of the following:

Switch	**Meaning**
/FROM=n	directs MKABS to begin reading inputfile at logical address n_8, instead of address 16_8 for RDOS files or 0 for RTOS files.
/LAST=n	directs MKABS to stop reading inputfile at logical address n_8.
/START*[=[@]n]*	places a start address in outputfile so that the system begins executing immediately after the absolute binary loader finishes. If you use /START without an argument, MKABS reads the start address from location 405_8 for RDOS or location 2 for RTOS. If you use /START=n without the @ (at-sign), the start address *is* n_8. If you use /START=@n, MKABS reads the start address from location n_8.
/ZERO	informs MKABS that inputfile is an RTOS .SV file; therefore, logical addresses are the same as file addresses. Without this switch, MKABS assumes that inputfile is an RDOS .SV file, and that logical addresses are 16_8 greater than their respective file addresses.

inputfile is an RDOS or RTOS save (.SV) file. If inputfile is an RTOS .SV file, you must use the /ZERO switch (e.g., X MKABS/ZERO RTOSFL.SV).

outputfile is the file or device to which MKABS sends the absolute binary (.AB) output.

To illustrate, assume you want to take an RTOS stand-alone program and punch it on paper-tape, using a teletypewriter. Issue the following command:

```
) X MKABS/START=@405/ZERO TEST.SV @OUTPUT ↓
```

MKABS will read TEST.SV as an RTOS save file, use the value at location 405_8 as the start address, and output the start address and TEST.AB to the teletypewriter.

End of Appendix

Appendix C
Linking 32-Bit Modules for DG/UX™

This appendix describes the procedures for linking 32-bit modules to run under DG/UX, Data General's UNIX™ compatible operating system. These cross-linking procedures allow you to develop and maintain applications programs under AOS/VS, and then execute them in a UNIX compatible environment (DG/UX).

The following sections summarize cross-linking differences from the perspective of AOS/VS Link. A full description of the DG/UX operating system is beyond the scope of this manual. For complete information on DG/UX, refer to the following manuals:

- */UX Family User Reference Manual*
- */UX Family Programmer Reference Manual*
- */UX Family Administrator Reference Manual*

Note that you can write source code for DG/UX in assembly language (MASM), as well as the AOS/VS C, FORTRAN 77, and Pascal high-level languages; however, no language completely supports cross-development at this time.

Cross-Linking Differences

To generate a DG/UX program file, you must use the /SYS=DGUX switch sequence in the Link command line. For example:

```
) X LINK/SYS=DGUX ONE.OB TWO.OB ↲
```

This command line cross-links AOS/VS 32-bit object modules ONE.OB and TWO.OB to produce DG/UX program file ONE. Notice that Link uses the root filename of the first object file in the command line as the name of the output program file, and *does not assign any extension.*

When cross-linking for DG/UX, Link generates an executable program file, an error file, and, optionally, a list file. Link does not produce a separate symbol table (.ST) file, debugger symbols (.DS) file, or debugger lines (.DL) file.

The DG/UX program file created by Link differs in content and format from other Link-generated program files. Instead of creating an .ST file, Link builds symbol table information into the DG/UX program file. Also, if one or more of your input object modules contain debugger symbols blocks or debugger lines blocks, and you use the /DEBUG switch, Link includes a user symbol table, formatted for the DG/UX debugger, in the program file.

The error file and list file (if you use /L or /L=listfile) are the same as those produced when linking for AOS/VS.

Aside from differences in Link's output, the linking process differs in the following respects:

- DG/UX system calls are incompatible with AOS/VS system calls. Therefore, you cannot cross-link modules that use AOS/VS system calls.
- When linking for DG/UX, Link does not automatically scan any system libraries. You must explicitly include the appropriate runtime libraries in the command line. (See "Cross-Link Command Line" in this appendix.)
- Link does not set the stack size for a DG/UX program file. DG/UX relies on the stack parameters set by the language processor.
- There is no User Status Table (UST) in user address space and there are no tables visible to you at run time.

Cross-Link Command Line

You use the same command line to produce DG/UX program files as you do to produce AOS/VS .PR program files, except that you must use the SYS=DGUX switch sequence. Note that some Link switches work differently (or not at all) with /SYS=DGUX. (Switch differences are explained later in this section.)

Before executing the Link command line, make sure that Link has access to all the required program elements; including your input object files (.OB) and libraries (.LB), as well as the appropriate system libraries.

When cross-linking for DG/UX, Link does not automatically scan any system library; therefore, you must specify the appropriate libraries on the command line. Although it is possible to transfer libraries (or archives) from your DG/UX system to AOS/VS, the libraries that Link needs in order to build a DG/UX program file are available under AOS/VS. These are: LIBECS.A, LIBA.A, LIB3.A, LIBC.A, LIBPAS.A, and LIBF77.A. LIBECS.A and LIBA.A, are general purpose runtime libraries essential for cross-linking; LIB3.A is a runtime library that may be necessary, depending on the requirements of your object files; LIBC.A, LIBPAS.A, and LIBF77.A are language runtime libraries required by Link if your source code is written in C, Pascal, or FORTRAN 77, respectively.

NOTE: When cross-linking for DG/UX, always append the /MULTIPLE switch to any library filenames you specify on the command line. This switch directs Link to make as many passes, linking modules from a library, as necessary to satisfy unresolved external symbols. Without /MULT, Link makes only one pass over each library file.

For example, assume that ONEC.OB and TWOC.OB are object modules written in the C programming language. To perform the cross-link, issue the following command line:

```
X LINK/SYS=DGUX ONEC TWOC LIBECS.A/MULT LIBA.A/MULT LIBC.A/MULT ⤶
```

Incompatible Switches

There are several switches that you should not use on the cross-link command line. Some because they do not work at all for DG/UX (e.g., /BUILDSYSTEM can only be used with /SYS=AOS); others because they have no useful application in the DG/UX environment.

The following is a list of these switches:

/BUILDSYSTEM	Use with /SYS=AOS.
/COMOVR	Use for 16-bit systems.
/ELEMENTSIZE=n	DG/UX does not have variable file element sizes.
/MAIN	Not required by any high-level language currently available (C, F77, Pascal).
/NRC	DG/UX does not support resource calls.
/NRP	DG/UX does not support resource calls.
/NSLS	Link does not automatically scan system libraries for DG/UX.
/PRSYM	Use with /SYS=RDOS or /SYS=RTOS.
/STACKSIZE=n	Link does not create an initial default stack for DG/UX.
/START	Not required by any high-level language currently available (C, F77, Pascal).
/TASKS=n	DG/UX does not support multitasking.
/UDF	Not applicable to DG/UX.
/UNX	Use for MV/UX only.
/WRL	DG/UX does not support resource calls.

Different Switch Meanings

The following switches have different meanings when cross-linking for DG/UX:

/DEBUG	Link does not create separate .DS or .DL files for DG/UX. If you use this switch, and your object modules contain one or more debugger symbols blocks or debugger lines blocks, Link builds debugger information into the DG/UX program file.
/LOCAL	Link does not create a separate .ST file for DG/UX. If you use this switch, Link adds local symbols information to the DG/UX program file.
/SUPST	Link does not create a separate .ST file for DG/UX. If you use this switch, Link does not build the ST portion of the DG/UX program file.

After Cross-linking

After Link creates an executable program file, you can use the following procedure to bring that file to your target DG/UX system:

1. While running MV/UX, the UNIX emulator, under AOS/VS, use the tape archive utility (MV/UX TAR) to dump your DG/UX program file to tape.
2. Transport the tape to your DG/UX system.
3. Under DG/UX, load the program file from tape using the tape archive utility (DG/UX TAR).

End of Appendix

Appendix D
Link Error Messages

Link reports errors as they occur, and sends them to one or possibly both of the following files:

- the error file the default error file is @OUTPUT, but you can override the default by using the /E=filename switch. If you use /E=filename, Link opens filename even if there are no errors.
- the list file if the Link command line contains either the /L or /L=filename switch, Link also sends errors to @LIST or filename respectively.

Also, if Link encounters any errors, it sends the character string LINK ERROR to the calling process as soon as the linking process terminates. (Refer to the ?RETURN system call in the *AOS/VS Programmer's Manual*. The CLI will display *LINK ERROR*. You can also direct the character string to the CLI pseudo macro [!STRING] by appending the /S switch to the XEQ command (i.e., XEQ/S LINK).

There are two classes of linking errors: fatal and non-fatal. Upon detecting a fatal error, Link terminates immediately, signals *LINK ERROR*, and sends the appropriate error message to the list file or error file (assuming that the error was not due to Link's inability to open one of these files). Any of the following conditions can cause a fatal error:

- an unexpected error return from some system calls
- user input errors (e.g., no arguments on the command line)
- Link internal consistency errors (e.g., symbol table overflow)

Non-fatal errors do not force Link to terminate. When the utility encounters a non-fatal error, it writes the appropriate error message to the error file or list file and continues executing the command line. Then, when the linking process terminates normally, Link signals *LINK ERROR* to the calling process. The following conditions can cause non-fatal errors:

- expected error returns from some system calls (e.g., object module does not exist, illegal pathname)
- user input errors (e.g., invalid object block, relocation errors)

Link precedes each error message with as much information as possible about the location of the error. Link also indicates whether or not the error is fatal and whether there might be a problem with the language processor that generated the object file. The first line of the message will always report one of the following: "ERROR:", "FATAL ERROR:", or "POSSIBLE COMPILER ERROR:". The line may also contain any of the following:

IN MODULE <title>
OBJECT BLOCK NUMBER <number> (for "POSSIBLE COMPILER ERROR:" only)
AT LOCATION <address>
(AREA <overlay area>, OVERLAY <overlay number>)

You will not see angle brackets (<>) in Link error messages; we use them here only to indicate the paraphrase of a value Link supplies.

NOTE: The following two messages are *not* error messages, although they appear in error files:

> *LINK REVISION <rev. number> ON <date> AT <time>*
> *OPTIONS: <Link switches used>*
>
> and
>
> *<program name> FILE CREATED*

Link includes this information at the beginning of the error file and the end of the list file.

Table D-1 lists the error messages reported by revision 5.05 of AOS/VS Link. Since some messages may change with each new release of Link, we advise you to check the current release notice for any enhancements or changes. As you read the table, notice that the messages *appear in this typeface* and that angle brackets (<>) set off the paraphrase of values supplied by Link. The table highlights different types of errors as follows:

*	FATAL ERROR
**	POSSIBLE COMPILER ERROR — (If you are certain that this error was not due to a faulty command line or some other user error, please report it to your system manager or the appropriate Data General compiler/assembler support personnel.)
***	AOS/VS SYSTEM ERROR — (Refer to the *AOS/VS Programmer's Manual* for more information.)

Table D-1. Link Error Messages

A 32-BIT RELOCATION OPERATION CANNOT BE USED IN A 16-BIT CONTEXT

You have included a 32-bit object file while linking for a 16-bit target operating system.

[SHARED/UNSHARED][DATA/CODE][COMM/PART]ATTRIBUTE CONFLICT

Two object modules each defined a partition with the same name, but having one or more conflicting attributes. Link cannot resolve a partition that is both shared and unshared, or both data and code, or both common base and normal base.

ATTEMPT TO LOAD DATA OUTSIDE FILE LIMITS **

A data block in one of your modules is trying to load datawords into a location that Link has not allocated to those datawords.

ATTEMPT TO LOAD OUTSIDE COMMON PARTITION <symbol name> **

A data block in one of your object modules is trying to store datawords at an address in a common base partition; however, that address is outside the defined boundaries of the partition.

CALL OR TARGET REFERENCE TO <symbol name> WHICH IS NOT A PENT

Link cannot resolve a target word and call word because the target argument is in an overlay, but is not a PENT. (For more information on call and target words, refer to the 'Resource Resolution' section in Chapter 5.)

CHAIN REFERENCE TO <symbol name> WHICH IS NOT AN EXTC

Link cannot reference <symbol name> because it is used in a chain-linking relocation operation, but is not described in the external symbols block as an EXTC type symbol.

COMMAND REQUIRES ARGUMENTS *

You did not pass any arguments on the Link command line. (Sometimes this error results from a missing & in a CLI macro.)

[<argument>]/<switchname>=<argument> DOES NOT ACCEPT AN ARGUMENT

You supplied an argument for a switch that does not accept arguments (e.g., /HEX=16)

END OF FILE, FILE <filename> ***

An object file is either missing an end block, or is zero-length.

<symbol name> FAULT ADDRESS IS OUT OF RANGE

The starting address of either the Stack Fault Handler, the Floating-Point Fault Handler, or the Commercial Fault Handler is not in the first 32K of memory. Link initializes reserved locations with these starting addresses.

FILE DOES NOT EXIST, FILE <filename> ***

<filename> is neither in your working directory nor in one of the directories in your search list. This message tells you that Link did not find either <filename> or <filename>.OB.

INVALID BLOCK SIZE <number> **

The size word in an object block contains a value less than 3 or greater than 1024_{10}. Link ignores the rest of the file that contains that object block.

INVALID BLOCK TYPE <number> **

An object block contains a block type (the right byte of the first word of every object block) that is not recognized by this revision of Link (see Table 4-1). Link ignores the rest of the file that contains that object block.

INVALID DICTIONARY OFFSET **

The relocation dictionary entry in a data block, debugger symbols block, debugger lines block, or lines title block contains an offset which does not point to a word in that block.

(continues)

Table D-1. Link Error Messages

INVALID OVERLAY SYNTAX

You used incorrect syntax when designating an overlay area in the command line. You must begin the overlay area designator with '!*', delimit each overlay with '!', and close the overlay area with '*!'. Use spaces, tabs, or commas to separate !*, !, and *! from other arguments.

INVALID RELOCATION OPERATION <number> **

You have specified a relocation operation not supported by your revision of Link.

[<argument>]/<switchname>[=<argument>] IS A DUPLICATE OR CONFLICTING SWITCH

You used the same switch more than once on one argument (e.g., /N/N); or you used the same switch on more than one argument, resulting in conflicting arguments (e.g., A/START B/START); or you used mutually exclusive switches (e.g., LINK/KTOP=5/MTOP=3).

[<argument>]/<switchname>=<argument> IS AN ILLEGAL ARGUMENT VALUE

You supplied a non numeric argument where a numeric argument is required (e.g., /REVISION=twelve instead of /REVISION=12); or you supplied a numeric constant that is not correctly formed (e.g., KTOP=999R8); or you supplied a revision number that is not syntactically correct (e.g., /REVISION=12..34); or you supplied an argument whose value falls outside the acceptable range (e.g., /ALIGNMENT=17 where legal values are 0 through 12_8).

<number>,<number> IS AN INVALID BIT STRING SPECIFIER **

A data block contains a bit field relocation dictionary entry that is specifying a bit field outside the dataword it is pointing to. (This is a possible compiler error.)

[<argument>]/<switchname>[=<argument>] IS AN UNKNOWN OR NON-UNIQUE SWITCH

You used a switch that Link does not recognize (e.g., /XYZ), or you used a switch that is not unique (e.g., /ST which might represent /STACK or /STATISTICS).

<filename> IS NOT AN OBJECT FILE

The header (first three words in an object block) in <filename> does not conform to standard object block header structure (see Chapter 4).

<symbol name> IS ONE OF <number> CIRCULARLY DEFINED EXTERNALS

Link cannot resolve one or more user symbols because you have defined them in terms of one another. For instance, if you define entry symbol A (.ENT A) in one module as A=B, and entry symbol B (.ENT B) in another module as B=A, then use external (.EXT) symbol A or B anywhere in you program, Link will not be able to resolve either A's or B's value. Link also returns this error when you define an entry symbol in terms of itself.

BLOCK NUMBER <number> IS OUT OF SEQUENCE **

The sequence word in an object block contains a value that does not conform to the block numbering order (see Chapter 4). Link ignores the rest of the file that contains that object block.

LINK REVISION <xx>.<yy> REQUIRED TO LINK THIS MODULE

A revision block in one of the input object modules contains a revision number newer than the revision of Link you are using. You need a newer revision of Link to process this module. Link ignores the rest of the module that contains this revision block.

(continued)

Table D-1. Link Error Messages

NESTED COMMAND FILES

You used <argument>/CLI (which designates a command file) inside a command file.

NO START ADDRESS HAS BEEN SPECIFIED

None of the object modules have a starting address in their end block. Unless you are linking a non-executable file with the /UDF switch, at least one module must contain a valid starting address. (The /START switch cannot correct this error because it must be used with a module that already has a possible starting address.)

NOT ENOUGH CONTIGUOUS BLOCKS * ***

A Link-generated file cannot fit on a device because the file's element size is greater than the number of free contiguous blocks on the device. This error is most frequently seen with RDOS and RTOS overlay files, which are built entirely contiguously, and therefore require a great deal of contiguous disk space.

OVERWRITE, PREVIOUS <number> PRESENT <number>

Two object modules are trying to initialize the same word to different values. Link retains the newer value and reports this error. (You can suppress this message by using any of the /OVERWRITE switches.)

REFERENCE TO LENGTH OF <symbol name> WHICH IS NOT A PARTITION

Link cannot refer to <symbol name> with the relocation operation given. If an object block uses one of the offset relocation operations, the base of the operation must be either a partition or common area.

REFERENCE TO UNDEFINED SYMBOL <symbol name>

Link cannot resolve an external symbol. Make sure that you included all the necessary object files on the Link command line, and that the appropriate system library is available to Link.

[<argument>]/<switchname>[=<argument>] REFERENCES AN UNDEFINED SYMBOL

The global switch /ULAST and the local switches /ALIGNMENT=n and /SHARED operate on partitions or common areas. Link returns this error if you append any of these switches to an argument that is not a partition or common area. Link also returns this error if it cannot resolve the symbol because the partition is undefined.

[<argument>]/<switchname> REQUIRES AN ARGUMENT

You failed to qualify a switch sequence with an argument (e.g., /E instead of /E=pathname)

RESOURCE HANDLER TABLE OVERFLOWS FIRST 16 PAGES

The resource-call (?RCALL) handler expects the Resource Handler Table (RHT) to be entirely within the first 16K of address space. Link returns this error when the RHT goes beyond address 37777_8. To reduce the size of the RHT, define fewer PENTs in overlays.

SHORT PARTITIONS OVERFLOW FIRST 32 PAGES

Link cannot fit all *short* NREL partitions into the first 32K of address space. You must remove the short attribute from some partitions.

<symbol name> SOENTO SYMBOL OVERFLOW

System overlay entry symbols (SOENTOs) are not properly placed in system overlays. (SOENTOs are created only when you set the /BUILDSYSTEM switch.)

START ADDRESS MAY NOT BE IN AN OVERLAY

The starting address of the program must not be in an overlay because the system will not load the overlay before transferring control to that address.

(continued)

Table D-1. Link Error Messages

/START OR /MAIN SWITCH HAS BEEN SET ON A MODULE WITHOUT A START ADDRESS

To execute the /START or /MAIN switch, Link uses the relocatable start address (relocation entry) found in the end block of the object module. Link returns this error if there is no start address in the end block (i.e., relocation entry value is negative).

SYMBOL <symbol name> IS MULTIPLY DEFINED

Two object modules are trying to define an entry symbol with the same name.

SYMBOL <symbol name> IS UNDEFINED

Link could not find a matching entry symbol for this external symbol. The undefined external symbol was emitted by either your object modules or by Link itself. If Link emitted the undefined external, it generally means that Link is not reading the appropriate system library. Make sure that the system library for your target operating system is in your working directory or in a directory on your search list.

<symbol name> SYMBOL TYPE INVALID **

An entry block contains a symbol type that this revision of Link does not recognize.

SYMBOLIC DATA LOCATION <symbol name> IS UNDEFINED

Link is unable to resolve a data block's relocation entry because it corresponds to an external symbol that is either not defined, or is defined in a subsequent module. (Ordinarily, the relocation entry in a data block defines an external number corresponding to a partition; however, external symbols are allowed. For more information on initializing data fields relative to external symbols refer to the .GLOC pseudo-op in the *AOS/VS Macroassembler Reference Manual.*)

SYSTEM TABLES OVERFLOW FIRST PAGE

AOS and mapped versions of RDOS expect the User Status Table (UST), the Task Control Block (TCB), and the Overlay Directory to be entirely within the first page (1024_{10} words) of a program file. Your program contains too many TCBs, or too many overlay areas, or too many AOS basic areas.

TOO MANY OVERLAY AREAS

The root contains more overlay areas than the target operating system allows. AOS, AOS/VS16, and MP/AOS permit 63_{10} overlay areas. RDOS and RTOS permit 126_{10} overlay areas.

TOO MANY OVERLAYS IN AREA <number>

There are more overlays in overlay area <number> than the target operating system allows. AOS, AOS/VS16, and MP/AOS permit 511_{10} overlays per overlay area. RDOS and RTOS permit 127_{10} overlays per overlay area.

UNKNOWN EXTERNAL NUMBER <number> **

There is an object block that refers to an external symbol by number. Link returns this error, if that number is either a 3 or a value higher than the number of externals defined in that module.

UNSHARED OVERFLOWS INTO SHARED

Your program's shared page area occupies the highest portion of memory and grows *downward* toward numerically lower addresses. The unshared area occupies the lower end of memory and grows *upward*. Link returns this error when the shared and unshared areas overlap.

ZERO-LENGTH SYMBOL NAME **

A title block, entry block, external symbols block, partition definition block, accumulating symbol block, or named common block contains a name-length field that has a value of zero.

ZREL OVERFLOWS INTO NREL

The total length of all ZREL partitions is greater than NBOT minus ZBOT.

(concluded)

End of Appendix

Appendix E
Link Symbol Types and Sample Map Listings

This appendix lists the symbol types recognized by Link and shows two sample map listings: a mini map, generated with the /L=filename switch; and a more comprehensive listing, generated with /L=filename plus other switches invoking additional Link listing options.

When producing a map listing file (list file), Link shows the symbol type and value of each symbol in the .PR file. Depending on the symbol type, Link may also supply additional information; for example, when listing a 16-bit symbol defined in an overlay, Link shows the symbol's overlay area and overlay number.

When listing the names of partitions or common areas, Link shows the length, lowest address and highest address of each partition or common area. Link also includes a two-letter mnemonic representing the attributes of each partition or common area. Link pairs the following letters to create these mnemonics:

Letter	Definition
A	absolute
Z	relocatable zrel
L	relocatable short
U	relocatable long unshared
S	relocatable long shared
C	code
D	data

For example, UC indicates relocatable long unshared code. (Refer to Table 3-1 for attributes and memory ranges of predefined partitions.)

Symbol Types

Table E-1 lists symbol types and type numbers that Link recognizes.

Table E-1. Symbol Types

Symbol Type	Type Number	Meaning
ENTRY	0	entry definition
EXT	1	external reference
COMM	2	common or external static area
ASYM	3	accumulating-value symbol
ENTO	4	overlay entry symbol
TITLE	5	title symbol (used with LOCALs)
PENT	6	procedure entry definition
EXTS	7	suppressible external reference
LOCAL	10	local symbol definition
EXTC	11	chain-link external
LIMIT	12	(not supported)
BSLPENT	13	(not supported)
SLPENT	14	(not supported)
PART	15	normal partition
SOENTO	16	system overlay entry (AOS only)
ENTS	17	suppressible entry definition

Sample Map Listings

Figures E-1 and E-2 are map listings produced by Link while creating program file CLIBT.PR. Figure E-1 shows a minimum listing generated by using the /L=filename switch. Figure E-2 shows a more comprehensive listing generated with /L=filename plus the /MAP, /MODMAP, /MODSYM, /NUMERIC, /V and /XREF switches.

Both listings begin with a two-line header: the first line shows the Link revision number, the date, and the time; the second line shows the function switches used to invoke linking options. Note that both listings include the /RING=3 switch, indicating that Link is building a part of the operating system, and the /NSLS switch, indicating that Link will not scan the system library. Both listings also show the /L=<filename> switch sequence, but specify different list files.

Figure E-1, the mini map, lists the names of the four input object modules: CLIBT, UWART, CRESOLVE, and XYZZY3. Next, it lists the major program parameters:

ZBOT and ZMAX are the lower and upper bounds of ZREL;

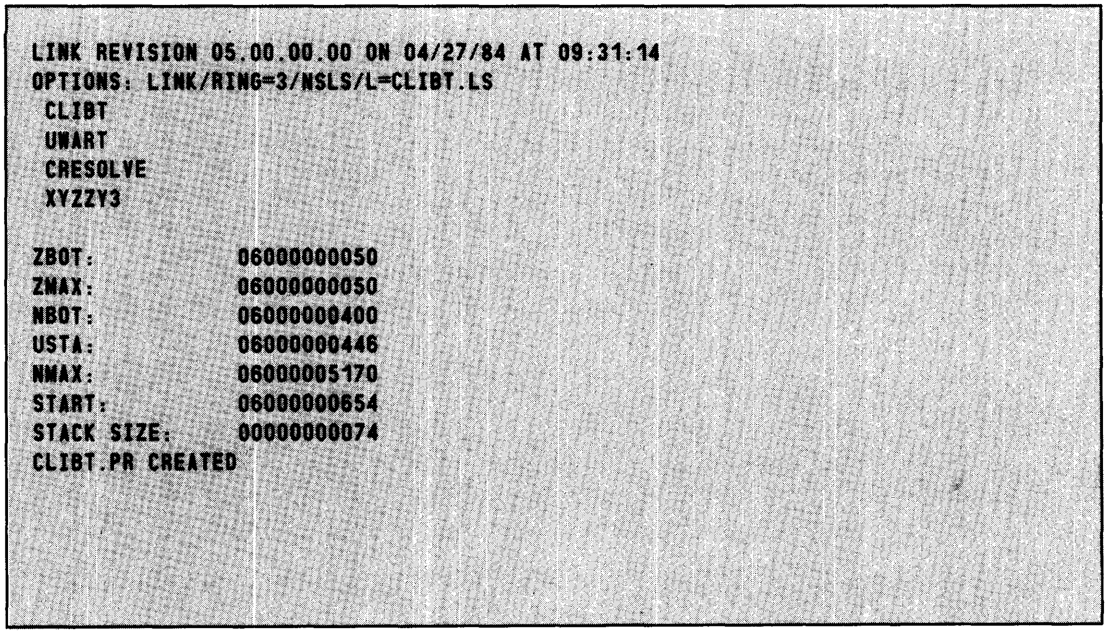
NBOT and NMAX are the lower and upper bounds of unshared NREL;

USTA is the upper boundary for system tables (which start at NBOT) and the start of user code and data;

START is the address where program execution begins.

Finally, the mini map displays the STACK SIZE of the initial default stack, and the name of the output program file, CLIBT.PR.

(This program has no shared NREL. If it did, Link would show the lower and upper bounds of shared NREL as SBOT and NTOP, respectively.)



```
LINK REVISION 05.00.00.00 ON 04/27/84 AT 09:31:14
OPTIONS: LINK/RING=3/NSLS/L=CLIBT.LS
 CLIBT
 UWART
 CRESOLVE
 XYZZY3

ZBOT:          06000000050
ZMAX:          06000000050
NBOT:          06000000400
USTA:          06000000446
NMAX:          06000005170
START:         06000000654
STACK SIZE:    00000000074
CLIBT.PR CREATED
```

Figure E-1. Mini Map Listing (/L=filename)

Figure E-2, the full map listing, contains all the information in the mini map, plus additional information that Link produced as a result of the following switches:

/V — adds a pathname preceding each object module file name (e.g., :SYSGEN:CLIBT.LB is the pathname to CLIBT).

/MODSYM — adds a list of entry symbols defined by each module (e.g., DEBUG and CLIBT are entry symbols defined in CLIBT).

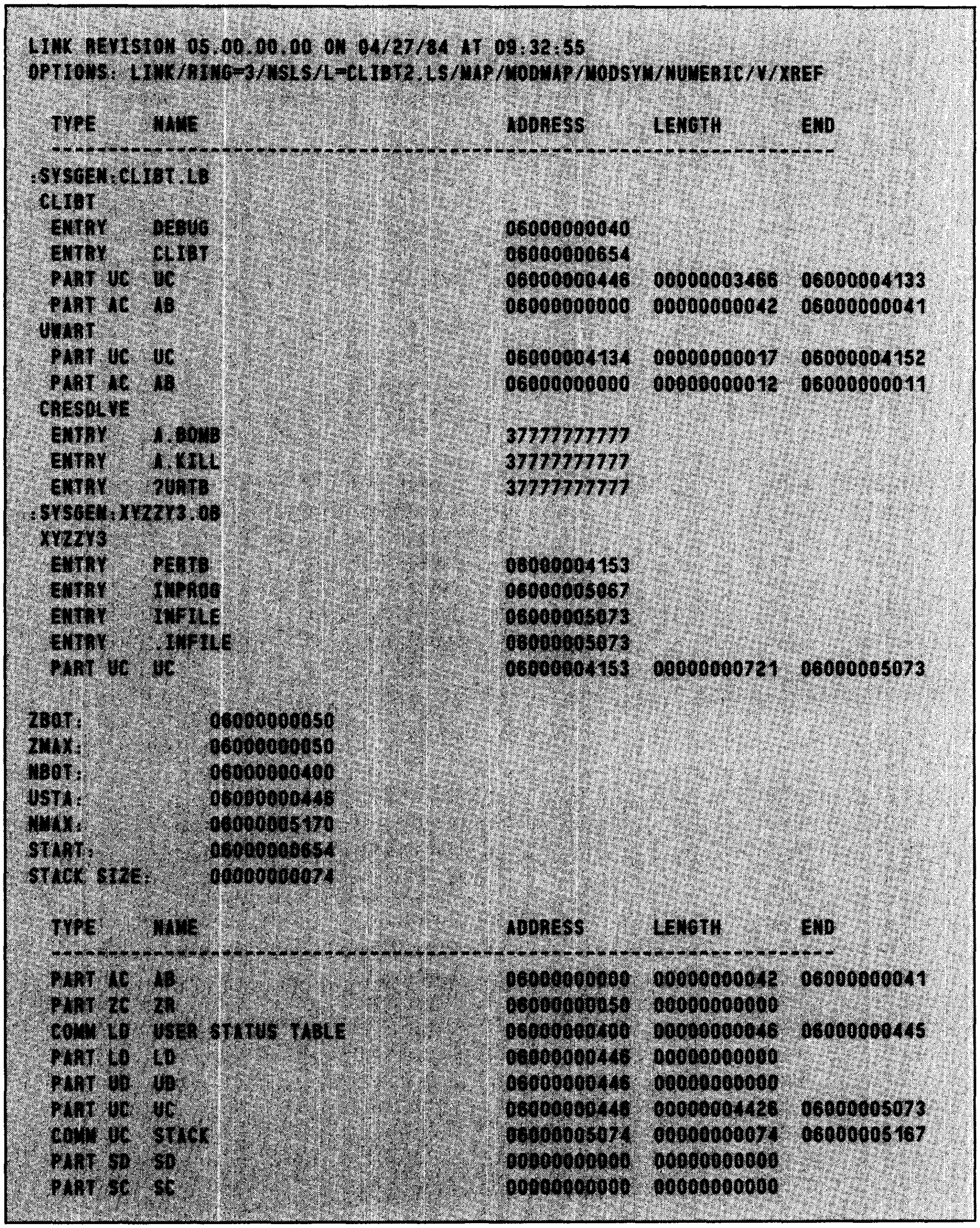
/MODMAP — adds a list of all normal partitions to which each module contributes (e.g., CLIBT contributes unshared code (UC) to the unshared code partition (UC) and absolute code (AC) to the absolute partition (AB)).

/MAP — adds a detailed map of memory use (following the list of major memory parameters).

/XREF — adds a cross-referenced, alphabetically sorted, symbol listing showing the type, name, and value of each symbol. Following each referenced symbol, this listing shows the name of the object module making the reference and the location of the reference (e.g., entry symbol PERTB is used twice by object module CLIBT; first at location 06000001045, and again at location 06000001257.).

/NUMERIC — adds a numerically sorted symbol listing (You can use /ALPHA to generate an alphabetically sorted symbol listing without the cross-referencing information provided by /XREF.)

NOTE: Figure E-2 is a full map listing for a small program; a listing for a large program might be very long, particularly if you use the /XREF switch.



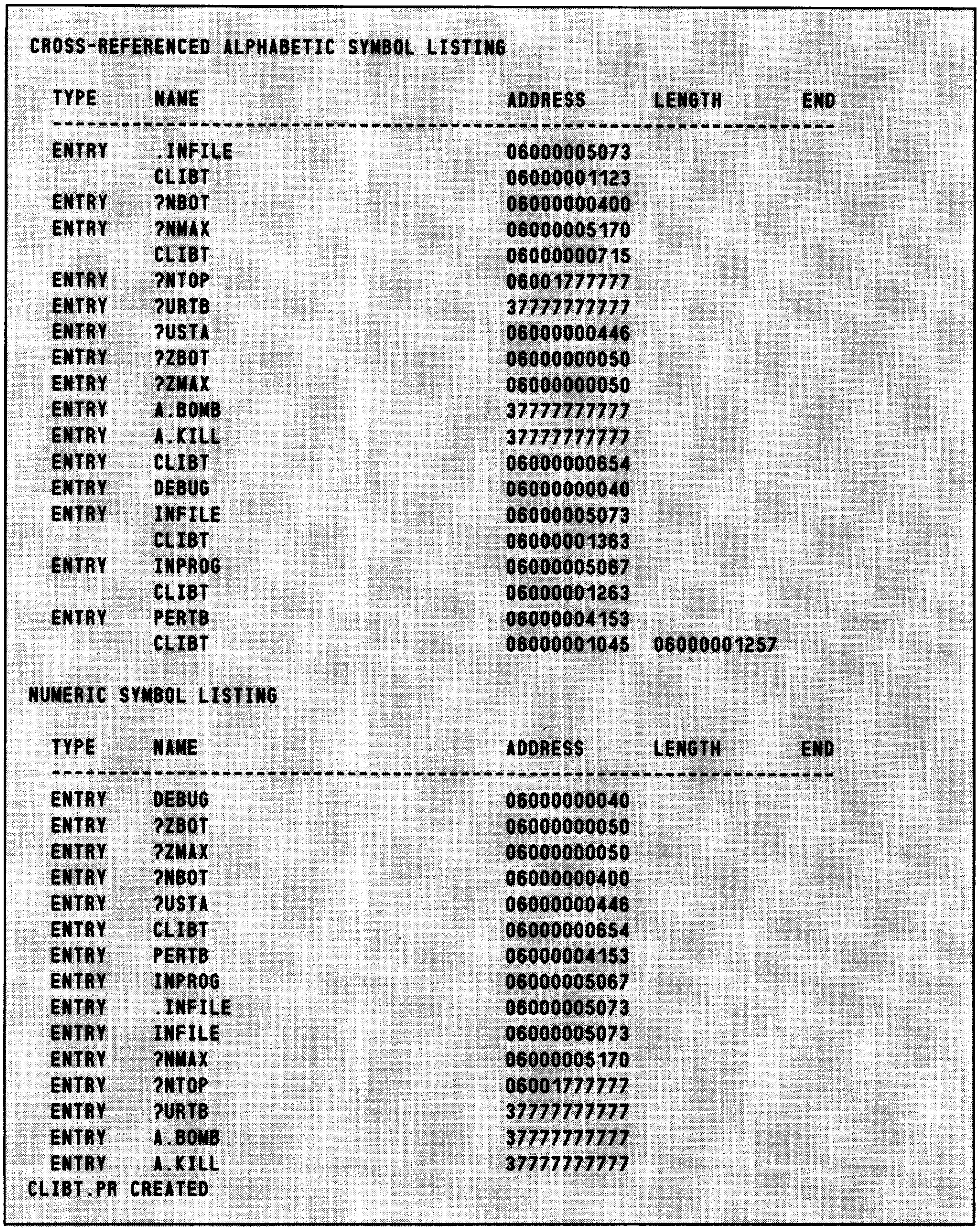
```
LINK REVISION 05.00.00.00 ON 04/27/84 AT 09:32:55
OPTIONS: LINK/RING=3/NSLS/L=CLIBT2.LS/MAP/MODMAP/MODSYM/NUMERIC/V/XREF

  TYPE     NAME                            ADDRESS      LENGTH       END
  ---------------------------------------------------------------------
:SYSGEN:CLIBT.LB
 CLIBT
  ENTRY    DEBUG                           06000000040
  ENTRY    CLIBT                           06000000654
  PART UC  UC                              06000000446  00000003466  06000004133
  PART AC  AB                              06000000000  00000000042  06000000041
 UWART
  PART UC  UC                              06000004134  00000000017  06000004152
  PART AC  AB                              06000000000  00000000012  06000000011
 CRESOLVE
  ENTRY    A.BOMB                          37777777777
  ENTRY    A.KILL                          37777777777
  ENTRY    ?URTB                           37777777777
:SYSGEN:XYZZY3.OB
 XYZZY3
  ENTRY    PERTB                           06000004153
  ENTRY    INPROG                          06000005067
  ENTRY    INFILE                          06000005073
  ENTRY    .INFILE                         06000005073
  PART UC  UC                              06000004153  00000000721  06000005073

ZBOT:           06000000050
ZMAX:           06000000050
NBOT:           06000000400
USTA:           06000000446
NMAX:           06000005170
START:          06000000654
STACK SIZE:     00000000074

  TYPE     NAME                            ADDRESS      LENGTH       END
  ---------------------------------------------------------------------
  PART AC  AB                              06000000000  00000000042  06000000041
  PART ZC  ZR                              06000000050  00000000000
  COMM LD  USER STATUS TABLE               06000000400  00000000046  06000000445
  PART LD  LD                              06000000446  00000000000
  PART UD  UD                              06000000446  00000000000
  PART UC  UC                              06000000446  00000004426  06000005073
  COMM UC  STACK                           06000005074  00000000074  06000005167
  PART SD  SD                              00000000000  00000000000
  PART SC  SC                              00000000000  00000000000
```

Figure E-2. Full Map Listing (/L=filename and other listing options) (continues)



```
CROSS-REFERENCED ALPHABETIC SYMBOL LISTING

  TYPE     NAME                             ADDRESS      LENGTH       END
  -----------------------------------------------------------------------
  ENTRY    .INFILE                          06000005073
           CLIBT                            06000001123
  ENTRY    ?NBOT                            06000000400
  ENTRY    ?NMAX                            06000005170
           CLIBT                            06000000715
  ENTRY    ?NTOP                            06001777777
  ENTRY    ?URTB                            37777777777
  ENTRY    ?USTA                            06000000446
  ENTRY    ?ZBOT                            06000000050
  ENTRY    ?ZMAX                            06000000050
  ENTRY    A.BOMB                           37777777777
  ENTRY    A.KILL                           37777777777
  ENTRY    CLIBT                            06000000654
  ENTRY    DEBUG                            06000000040
  ENTRY    INFILE                           06000005073
           CLIBT                            06000001363
  ENTRY    INPROG                           06000005067
           CLIBT                            06000001263
  ENTRY    PERTB                            06000004153
           CLIBT                            06000001045  06000001257

NUMERIC SYMBOL LISTING

  TYPE     NAME                             ADDRESS      LENGTH       END
  -----------------------------------------------------------------------
  ENTRY    DEBUG                            06000000040
  ENTRY    ?ZBOT                            06000000050
  ENTRY    ?ZMAX                            06000000050
  ENTRY    ?NBOT                            06000000400
  ENTRY    ?USTA                            06000000446
  ENTRY    CLIBT                            06000000654
  ENTRY    PERTB                            06000004153
  ENTRY    INPROG                           06000005067
  ENTRY    .INFILE                          06000005073
  ENTRY    INFILE                           06000005073
  ENTRY    ?NMAX                            06000005170
  ENTRY    ?NTOP                            06001777777
  ENTRY    ?URTB                            37777777777
  ENTRY    A.BOMB                           37777777777
  ENTRY    A.KILL                           37777777777
CLIBT.PR CREATED
```

Figure E-2. Full Map Listing (/L=filename and other listing options) (concluded)

End of Appendix

Appendix F
Relocation Operations

The language processor generates a relocation operation for every relocatable reference in your source code. The type of relocation operation depends on the addressing mode and the element being addressed (i.e., whether it is a 16-bit word, a 32-bit double word, a byte, or a bit).

Your ECLIPSE MV/Family processor supports three addressing modes: absolute mode, program counter (pc) relative mode, and accumulator (ac) relative mode. To resolve all possible address and ring field combinations, AOS/VS provides 58_{10} relocation operations, of which 55_{10} are currently defined. (Three are reserved.)

Table F-1 lists the relocation operations and classifies them according to function. This information is primarily for reference. Unless you're writing your own language processor, you won't need to know the relocation operations and their functions. For further information on the addressing modes and logical-to-physical address translations, consult the *Principles of Operation 32-Bit ECLIPSE® Systems* manual.

Table F-1. Relocation Operations

Relocation Operation	Description	Comments
Operations for 16-bit References		
0	absolute relocation	For absolute addresses; logical address = physical address
1	word relocation (base+data)	Data is sign extended, result must be >= -200000_8, and < 200000_8
2	byte relocation (2*base+data)	Result must be >= -200000_8, and < 200000_8.
3	16-bit displacement, where bits 1-15 of data = 0 (base + sign extended data)	Result must be >= -200000_8, and and < 200000_8
(3)	16-bit displacement, where bits 6-7 of data = 0 (low-order 8 bits base + low-order 8 bits data = intermediate result.) Intermediate result + high-order 8 bits data = result.	Intermediate result must be >= 0, and < 400_8.
(3)	16-bit displacement, where bits 6-7 of data are non-zero (low-order 8 bits base + low-order 8 bits base = intermediate result.) Intermediate result + high-order 8 bits data = result.	Data is sign extended. Intermediate result must be >= -200_8 and < 200_8.
4	subtraction (base-data = intermediate result.) Intermediate result (low-order 15 bits) + bit 0 of data = result.	Data is sign extended. Intermediate result must be >= -20000_8, and < 200000_8.

(continues)

Table F-1. Relocation Operations

Relocation Operation	Description	Comments
Operations for 16-bit References		
5	overlay entry (.ENTO)	Data replaced by overlay area, overlay numbers.
6	multiplication (data*base)	
7	link relocation (16-bit)	Creates a reverse chain of data addresses in 16-bit .DS file
10	call relocation	Method used to resolve resource calls in 16-bit modules. If base is in an overlay, it must be declared as a PENT.
11	GREF (global reference) relocation (base + sign extended data = intermediate result.) Intermediate result (low-order 15 bits) + bit 0 of data = result.	Intermediate result must be >= -200000$_8$, and < 200000$_8$.
12	pc relative 15 ((base-pc)+ sign extended data = intermediate result.) Intermediate result (low-order 15 bits) + bit 0 of data = result.	Intermediate result must be >= -200000$_8$, and < 200000$_8$.
13	target relocation	Method used to resolve target in resource calls (16-bit modules). If base is in an overlay, it must be declared as a PENT.
14	pc relative 16 ((base-pc)+ sign extended data)	Result must be >= -200000$_8$, and < 200000$_8$.
15	reserved	
17	extended relocation designator (not an actual relocation op.)	(See "Relocation Entries", this chapter.)
20	.PR file link (16-bit) (data word replaced by address of previous reference)	Creates a reverse chain of locations using .EXTC symbols.
21	offset (16-bit) (Length of partition + data)	Relocation base must be a partition or common area
22	subtraction, type 2 result = data - base	
23	bit (16-bit) result = 4 * base + data	Result must be >= -100000 and < 100000$_8$

(continued)

Table F-1. Relocation Operations

Relocation Operation	Description	Comments
Operations for 32-bit References Exclusively		
31	link relocation (32-bit)	Creates a reverse chain of data addresses in 32-bit .DS file.
32	.PR file link (32-bit)	Creates a reverse chain of addresses in the .PR file; used to resolve .EXTC external symbols.
33	32-bit absolute relocation	Analogous to relocation operation 0; used to identify a 32-bit symbol entry.
34	bit (32-bit) result = 4 * base + data	
35	offset (32-bit) (Length of partition + data)	Relocation base must be a partition or common area
32-bit base,	31-bit data:	
40	word relocation (32-bit base + 31-bit data = intermediate result.) Intermediate result masked to 31 bits and "ORd" with @ sign (bit 0) of data word.	Intermediate result must be >= -2000000000_8 and < 2000000000_8.
41	byte relocation ((2* 32-bit base) + 32-bit data = relocated address)	Result must be >= -4000000000_8, and < 4000000000_8.
42	word pc relative (32-bit base + (masked)31-bit data = intermediate result.) Intermediate result, minus 28-bit pc, is masked to 31 bits and "ORd" with @ sign (bit 0) of data word.	Intermediate result must be >= -2000000000_8, and < 2000000000_8
43	byte pc relative ((2* 32-bit base) + 32-bit data = intermediate result.) Intermediate result - (masked) 28-bit pc = relocated address.	Intermediate result must be >= -4000000000_8, and < 4000000000_8.

Table F-1. Relocation Operations

Relocation Operation	Description	Comments
	Operations for 32-bit Address Relocation	
32-bit base,	28-bit data:	
44	word relocation (32-bit base + (masked) 28-bit data = intermediate result.) Intermediate result is "ORd" with ring field (bits 1-3) and @ sign (bit 0) of data word.	Intermediate result must be >=0, and < 2000000000_8.
45	byte relocation (32-bit base + (2* 28-bit data) = intermediate result.) Intermediate result is "ORd" with 2* ring field (bits 1-3) of data word.	Intermediate result must be >=0, and < 4000000000_8.
46	pc relative (32-bit base + (masked) 28-bit data = intermediate result.) Intermediate result, minus (masked) 28-bit pc is masked to 31 bits and "ORd" with @ sign (bit 0) of data word.	Intermediate result must be >= 0, and < 2000000000_8. (Ring field of data must be the same as the global ring field)
47	byte pc relative (32-bit base + (2* 28-bit data) = intermediate result.) Intermediate result - (2* 28-bit pc) = relocated address.	Intermediate result must be >= 0, and < 4000000000_8. (Ring field of data must be the same as the global ring field.)
28-bit base,	31-bit data:	
50	word relocation ((masked) 28-bit base + 31-bit data = intermediate result.) Intermediate result is "ORd" with @ sign (bit 0) of data, and ring field (bits 1-3) of base.	Intermediate result must be >=0, and < 2000000000_8.
51	byte relocation ((2* (masked) 28-bit base)+ 32-bit data = intermediate result.) Intermediate result is "ORd" with 2* ring field (bits 1-3) of base.	Intermediate result must be >= 0, and < 4000000000_8.
52	pc relative ((28-bit base) + 31-bit data = intermediate result.) Intermediate result, minus (masked) 28-bit pc is masked to 31 bits and "ORd" with @ sign (bit 0) of data word.	Intermediate result must be >=0, and < 2000000000_8. (Ring field of the base must be the same as global ring field.)
53	((2* (masked) 28-bit base) + 32-bit data = intermediate result.) Intermediate result - (2* (masked) 28-bit pc = relocated address.	Intermediate result must be >= 0, and < 4000000000_8. (Ring field of the base must be the same as global ring field.)

Table F-1. Relocation Operations

Relocation Operation	Description	Comments
	Operations for 32-bit Address Relocation	
32-bit base,	15-bit data (unsigned):	
100	word relocation (28-bit base + 15-bit (masked) data = intermediate result.) Intermediate result is masked to 15 bits and "ORd" with @ sign (bits 1-3) of data word.	Intermediate result must be >=0, and < 100000_8.
101	byte relocation ((2* 28-bit base) + 16-bit data = intermediate result.) Low-order bits of intermediate result = relocated address.	Intermediate result must be >=0, and < 200000_8.
102	pc relative ((28-bit base - (masked) 28-bit pc) + masked 15-bit data = intermediate result.) Intermediate result is masked to 15 bits and "ORd" with @ sign (bits 1-3) of data word.	Intermediate result must be >= 0, and < 100000_8. (Ring field of the base must be the same as the global ring field.)
103	byte pc relative ((2* 28-bit base)- (2* (masked) 28-bit pc) + 16-bit data = intermediate result.) Low-order bits of intermediate result = relocated address.	Intermediate result must be >=0, and < 200000_8. (The ring field of the base must be the same as the global ring field.)
32-bit base,	15-bit data (signed):	
104	word relocation (28-bit base + 15-bit data (data masked and sign extended) = intermediate result.) Intermediate result is masked to 15 bits, and "ORd" with @ sign (bits 1-3) of data word.	Intermediate result must be >= -40000_8 and < 40000_8.
105	byte relocation ((2* 28-bit base) + 16-bit data (data is sign extended) = intermediate result.) Low-order bits of intermediate result = relocated address.	Intermediate result must be >= -100000_8, and < 100000_8.
106	pc relative ((2* 28-bit base) - (2* (masked) 28-bit pc) + 15-bit data (data is masked and sign extended) = intermediate result.) Intermediate result is masked to 15 bits, and "ORd" with @ sign (bits 1-3) of data word.	Intermediate result must be >= -40000_8 and < 40000_8. (Ring field of the base must be the same as the global ring field.)
107	byte pc relative ((2* 28-bit base)-(2* (masked) 28-bit pc) + 16-bit data (data is sign extended) = intermediate result.) Low-order bits of intermediate result = relocated address.	Intermediate result must be >= -10000_8 and < 10000_8.

Table F-1. Relocation Operations

Relocation Operation	Description	Comments
	Operations for Data Generation	
32-bit base,	32-bit data:	
60	(32-bit base + 32-bit data)	
61	(32-bit base - 32-bit data)	
62	(32-bit base * 32-bit data)	
63	(32-bit data - 32-bit base)	
32-bit base,	16-bit data (data is sign extended):	
64	(32-bit base) + 16-bit data)	
65	(32-bit base - 16-bit data)	
66	(32-bit base * 16-bit data)	
67	(16-bit data - 32-bit base)	
32-bit base,	16-bit data (data is unsigned):	
70	(32-bit base + 16-bit data)	Data is 0 extended, result is masked to 16 bits. (Bits 0-15 of result must = 0)
71	(32-bit base) - 16-bit data)	Data is 0 extended, result is masked to 16 bits. (Bits 0-15 of result must = 0.)
72	(32-bit base * 16-bit data)	Data is 0 extended, result is masked to 16 bits. (Bits 0-15 of result must = 0.)
73	(16-bit data - 32-bit base)	Data is 0 extended, result is masked to 16 bits. (Bits 0-15 of result must = 0.)
32-bit base,	16-bit data (no checking of base value):	
74	(32-bit base + 16-bit data)	Data is 0 extended. Result is masked to 16 bits.
75	(32-bit base - 16-bit data)	Data is 0 extended. Result it masked to 16 bits.
76	(32-bit base * 16-bit data)	Data is 0 extended. Result is masked to 16 bits.
77	(16-bit data - 32-bit base)	Data is 0 extended. Result is masked to 16 bits.

End of Appendix

Glossary

Term	Meaning
absolute code absolute data	code or data assigned to specific (i.e., absolute) locations.
alignment factor	a value within the range $1\text{-}10_{10}$ describing a partition's alignment along some memory boundary (e.g., double word alignment, 1K-word alignment).
argument switch	a switch used in the Link command line to modify the utility's handling of a specific object module or partition.(See Table 2-2.)
attributes	the set of characteristics defining a partition. The attributes are: • absolute, ZREL, long NREL, or short NREL • shared or unshared • normal base or common base • alignment • code or data • overwrite-with-message or overwrite-without-message • local or global (user-defined partitions only)
bit field	a portion (series of bits) of a word or double word.
bit field relocation entry	a relocation entry generated by the language processor to describe the relocation of a bit field.
block types	octal values that identify object blocks and implicitly define their functions.
byte pointer	a 32-bit (or 16-bit) value that points to the start of a byte string.
command line	a set of directives that invokes Link, LFE, CONVERT, or MKABS and names the files and/or program elements the utility will process.
CONVERT	the utility that converts RDOS or RTOS relocatable binary files to AOS/VS object file format.
cross-linking	using AOS/VS Link to create a program file for execution under the AOS, DG/UX, MP/AOS, RDOS, or RTOS operating systems.
dataword	16 contiguous bits; code or data.
demand paging	the AOS/VS technique of moving pages in and out of a process's working set as that process uses and ceases to use them.

Term	Meaning
displacement	in object blocks — a dataword's position relative to some other point in the object block or object module. In memory reference instructions — a value that tells the CPU which logical address to access.
.DL (debugger lines) file	an output file Link creates to store information for eventual high-level language debugging; built when the object module(s) contains one or more debugger lines blocks and the command line includes the /DEBUG function switch. When cross-linking for DG/UX, Link includes debugger information in the program file and does not create a .DL file.
.DS (debugger symbols) file	an output file Link creates to store information for eventual high-level language debugging; built when the object module(s) contains one or more debugger symbols blocks and the command line includes the /DEBUG function switch. When cross-linking for DG/UX, Link includes debugger information in the program file and does not create a .DS file.
entry symbol	a symbol defined in an entry symbols block. Entry symbols can be accessed by any object module.
error file	an output file Link creates to record object module or command file errors and Link-generated messages (i.e., the .PR file creation message); produced unconditionally, even if there are no errors.
extended relocation entry	a portion of an object block generated by the language processor; allows Link to define a block's relocation value, if that value is greater than 32K; contains larger relocation base/relocation operation fields than a standard relocation entry.
external symbol	a symbol defined in an external symbols block; a symbol declared in, and defined by, one object module, but used in one or more other modules.
forced load flag	a bit in an object module that, when set, directs Link to load that module into the program file unconditionally.
function-letters	Library File Editor (LFE) commands.
function switch	a command line switch that modifies the entire linking process. (See Table 2-1.)
immovable resource	resources that occupy the same memory area throughout the program's execution.
language processor	an assembler utility (e.g., MASM), or a high-level language compiler.
library file	a series of object modules that begin with a library start block and end with a library end block; produced by the Library File Editor utility (LFE).
Library File Editor (LFE)	the utility that creates, edits, and analyzes library files.
lines file directory	a section of the .DL (debugger lines) file that describes the memory destination of the data in one or more lines title blocks.
Link	the name of the AOS/VS relocatable linker.
logical address space	the entire range of locations that a process can access.

Term	Meaning
lower page zero	the memory area extending from location 0 through location 377$_8$; it is subdivided into two sections: 0 through 47$_8$, for system references, and ZREL (page zero relocatable), locations 50$_8$ through 377$_8$.
map files	(mini map and full map) listing files Link produces when the command line includes the /L or /L=filename switch. Map files record the program file's partitions, memory parameters, and other statistical information.
memory map	an image of the program file produced internally between Link's first and second pass; outlines each partition, its contents and general location.
MKABS	the utility that converts RDOS or RTOS .SV files to absolute binary (.AB) files for use on a paper-tape system.
movable resource	a position-independent resource that the operating system can assign to any location within a general memory range.
NREL	*normal relocatable* memory that extends by default from location 400$_8$ to the extent of memory; subdivided into two sections: *short*NREL (400$_8$ through 32K), and *long* NREL (400$_8$ to the highest address in memory).
.OB file	a file, containing one object module, produced by a language processor.
object blocks	blocks of binary code produced by the language processor as it interprets your source code.
object block header	the first three words in every object block; states the block's type, sequence number (relative to the other object blocks in a module), and length.
object module	an assembled or compiled source module that consists of object blocks; identified by the .OB filename extension.
.OL (overlay) file	an output file Link builds to retain a program's overlays (16-bit programs only).
optimization	the process by which Link replaces certain resource calls with EJSR instructions to create a more efficient program file.
overlay	a shared or unshared routine the system swaps in and out of memory as necessary during execution; a type of resource (16-bit programs only).
overlay area	a section of the .PR file, the .OL file, or memory reserved for shared or unshared overlays (16-bit programs only).
overlay designators	the character series !*, !, and *!; used in the Link command line to designate certain object modules as overlays (16-bit programs only).
page	a 2K-byte portion of a process's logical address space; a 2K-byte portion of memory.
partition	a group of source statements with common attributes; a section of the .PR file or of memory having common attributes.
position-independent routines	routines whose references are limited to immovable or pc relative resources.

Term	Meaning
.PR file (program file)	an executable program file Link creates from one or more object modules and/or library files; identified by the .PR filename extension.
.RB file	a file containing one relocatable object module, produced by an RDOS language processor.
relocatable binary module	an RDOS or RTOS object module, containing a series of RLDR-compatible object blocks and usually having the file name extension .RB.
relocatable code relocatable data	datawords (code or data) that Link and the operating system can relocate (reposition) to any location within a broad memory range; code or data in which the relationship between the data elements is more important than their actual locations.
relocation	the process by which Link repositions code and/or data relative to relocation bases in memory.
relocation base	a value generated by the language processor and/or Link; indicates a partition's base address in memory.
relocation dictionary entry	an element within a data block generated by the language processor and used by Link to resolve the value of a dataword.
relocation entries	relocation information within data blocks, partition and overlay definition blocks, and various symbols blocks; generated by the language processor and used by Link to relocate partitions in memory.
relocation operation	an element within a relocation entry or relocation dictionary entry; generated by the language processor to describe the kind of relocation Link will perform for a dataword or object block.
resource	the root or overlay portion of the program file (16-bit programs only).
resource (procedure) calls	the system calls ?RCALL, ?KCALL, and ?RCHAIN; used to reference external root or overlay resources.
resource handler routines	system-defined routines (in the system library) that load, manage, and release resources at runtime.
rings	seven hierarchical memory contexts (numbered 0 through 7); the basis of the AOS/VS protection system.
root	a type of resource; that part of a program which is memory resident during execution (16-bit programs only).
shared pages	memory pages accessible to more than one process.
.ST file	a nonexecutable file generated by Link and used for debugging. The file contains the names and values of all global symbols (and, optionally, local symbols) in the program. When cross-linking for DG/UX, Link includes this information in the program file and does not create a .ST file.
stack	a block of consecutive memory locations generally reserved for storing variables, return addresses, subroutine arguments, and task-specific information.
.SV (save) file	a program file executable under RDOS; or, a program file executable under RTOS.

Term	Meaning
switch	an optional element of the Link or LFE command line; a slash followed by an alphanumeric character or character string (e.g., /ALPHA). In the Link command line, function switches modify the entire linking process; argument switches modify the utility's handling of a particular module or partition.
switch sequence	an optional element of the Link command line; a slash following by an alphanumeric character or character string, an equal sign, and another character or character string (e.g., /UC=SC).
symbol table	an output file produced by Link; contains the names and values of all global symbols (and optionally, local symbols) in the program.
system library	a file of system routines written to satisfy user runtime requirements.
system tables	internal databases built by Link to describe the .PR file's characteristics and memory requirements.
unshared pages	memory pages reserved for the exclusive use of a single process.
virtual overlays	overlays that reside in extended memory, outside a program's logical address space. (RDOS and RTOS systems only.)
wide frame pointer	one of four hardware registers used to manage the wide pointer stack; points to the last double word currently in use on the stack.
wide stack	the AOS/VS 32-bit user stack. (See stack.)
wide stack base	one of four hardware registers used to manage the wide stack; contains the stack's base address (lower boundary).
wide stack limit	one of four hardware registers used to manage the wide stack; contains the stack's upper boundary.
wide stack pointer	one of four hardware registers used to manage the wide stack; contains the double word currently at the top of the stack.
ZREL	*page zero relocatable* memory; by default, extends from 50_8 through 377_8.

End of Chapter

Index

Within this index, "f" or "ff" after a page number means "and the following page" or "pages". In addition, primary page references for each topic are listed first. Commands, calls, and acronyms are in uppercase letters (e.g., CREATE); all others are lowercase.

A

AB (absolute binary file) B-1f
absolute
  address 3-5
  binary file B-1f
  code 1-11, Glossary-1
  data Glossary-1
  memory area 1-5
  partition 3-1, 3-5
  value 3-5
accumulating symbol 2-12
address information block (AIB) 5-10
address, logical space 1-3
aligned partition, *see* partition attributes
alignment block 4-13
alignment factor Glossary-1
!*/ALIGNMENT=n (switch) 5-9
/ALIGNMENT=n (switch) 2-11, 3-3, 4-17
/ALPHABETIC (switch) 2-6
AOS/RT32, linking for 2-9
AOS/VS memory concepts 1-3ff
argument passing to resource calls 5-5
argument switch
  definition 2-3, Glossary-1
  table 2-6ff
ASYM (symbol type) *see* /VALUE=n
attributes
  definition Glossary-1
  see also partition attributes

B

base
  relocation 1-12, 3-7
  see also partition relocation
binary file
  absolute B-1f
  relocatable A-3
bit field Glossary-1
bit field relocation entries, *see* relocation entries
block
  address information 5-10
  data, *see* data block
  debugger lines 4-20
  debugger symbols 4-19
  end, *see* end block
  entry symbols 4-15, 4-3
  external symbols 4-14, 4-3
  filler 4-22
  library end, *see* library end block
  library start, *see* library start block
  lines title 4-21, 4-3
  local symbols 4-16f
  module revision 4-6
  named common 5-12
  object 1-1
  task 4-18
  types Glossary-1
  unlabeled common 4-6, 4-18
/BUILDSYSTEM (switch) 2-6, 5-9
byte pointer Glossary-1

C

call relocation word 5-4ff
CFALT (fault handler) 2-3
CFALT (symbol) 1-7
?CFLT (symbol) 1-8
chain external symbol 4-18
?CHAN (symbol) 1-6; *see also* /CHANNELS=n
/CHANNELS=n (switch) 2-6
CLI macroinstruction file 2-2, 2-4; *see also* /CLI switch
/CLI (switch) 2-4, 2-11
?CLOC (symbol) 1-6, 4-18
code
  absolute 1-11
  definition 1-2
  partition 3-3
  relocatable 1-11
command line 2-1ff, Glossary-1
common area 4-18
common base partition, *see* partition attributes
/COMOVR (switch) 2-6, 5-9
concepts
  AOS/VS memory 1-3ff
  MV/Family ring structure 1-3
CONVERT B-1f, Glossary-1

cross-linking
  definition Glossary-1
  for AOS 5-1ff
  for AOS/VS 16-bit 5-1ff
  for DG/UX C-1ff
  for MP/AOS 5-1ff 5-19
  for RDOS and RTOS A-1ff
CSZE (pseudo-op) 4-18
?CSZE (symbol) 1-6

D

data
  block 4-9f, 4-3
  definition 1-2f
  partition 3-3
  relocation 3-9ff
  dataword Glossary-1
  see also data block
DEBUG (symbol) 1-7; *see also* /DEBUG
/DEBUG (switch) 2-6, 2-11, 4-23
debugger
  lines block 4-21
  lines file 1-13f
  symbols block 4-19
  symbols file 1-13f
default partitions 3-4f
demand paging 1-3, Glossary-1
descriptor
  object 6-4
  overlay 5-11
  partition 5-11
  symbol 6-4
designators, overlay 5-8
DG/UX, cross-linking for C-1ff
displacement Glossary-2
DL (debugger lines file) 1-13; *see also* /DEBUG
DL (file) 4-20f
DS (debugger symbols file) 1-13; *see also* /DEBUG
DS (file) 4-19f

E

/E=pathname (switch) 2-6f
EJSR instruction 5-6f
/ELEMENTSIZE=n (switch) 2-6
end block 4-3, 4-23
ENT (pseudo-op) 5-5; *see also* entry symbols block
entering numeric values 2-3
entries, relocation 3-9ff
entry symbol (ENT) 5-3, Glossary-2
entry symbols block 4-15, 4-3
error
  file 1-13f, Glossary-2
  handling routines 2-3
  message 1-14
  messages
    LFE 7-23ff
    LINK D-1ff
  reporting 1-14
/ERRORCOUNT=n (switch) 2-6
extended relocation entries, *see* relocation entries
external numbers 3-7; *see also* partition
external symbol 3-7, Glossary-2
external symbols block 4-17, 4-3
EXTS (symbol type), *see* suppressed external symbol

F

fault handling routines 2-3
FFALT (fault handler) 2-3
FFALT (symbol) 1-7
?FFLT (symbol) 1-8
file
  .AB B-1f
  .DL 1-13; *see also* /DEBUG
  .DS 1-13
  .LB 2-11; *see also* /FORCE
  .OL 1-14, 5-3
  .PR 1-13
  .RB A-3, B-1
  .ST 1-13; *see also* /SUPST
  .SV S-2
  .SY 2-6; *see also* /BUILDSYSTEM
files
  error 1-13f
  library 2-2
  list 1-13f
  macro 2-2, 2-4
  map 1-14
  output 1-13f
  see also /CLI switch
filler block 4-30
flag, forced load 2-2
FORC (pseudo-op) 2-2, 4-5
/FORCE (switch) 2-11
forced load flag 2-2, Glossary-2
function-letters
  definition 7-1f, 6-1 Glossary-2
  table 7-2
function switch
  definition 2-3, Glossary-2
  table 2-6ff

G

gates 1-3
global partition, *see* partition
GLOC (pseudo-op) 1-11

H

/HEXADECIMAL (switch) 2-6

I

IFALT (fault handler) 2-3
IFALT (symbol) 1-7
immovable resource 5-2, Glossary-2

K

K notation 2-3
KCA?? (symbol) 1-9
KCALL? (resource call) 5-2ff
/KTOP=n (switch) 2-6

L

/L (switch) 2-6f 1-14
/L=filename 1-14
/L=pathname (switch) 2-6
language processor 1-1, Glossary-2
LB (library file) 2-11; *see also* /FORCE
?LBOT (symbol) 1-6
/LD=default partition (switch) 2-12
LFE, *see* Library File Editor
library
  files 2-2, 6-1 Glossary-2
  start block 6-3f
  system 2-2
library end block 6-4, 6-1
Library File Editor (LFE)
  command line format 7-1
  error messages 7-16ff
  function-letter definition 7-1ff
  function-letters
    A (Analyze) 7-3
    D (Delete) 7-7
    I (Insert) 7-8
    M (Merge) 7-11
    N (New) 7-11
    R (Replace) 7-13
    T (Title) 7-14
    U (Update) 7-15
    X (Extract) 7-16
  functional overview 6-1ff
  output listing notation 7-4
lines file directory Glossary-2
lines title block 4-21, 4-3
Link
  actions between passes 1-2
  actions during pass 1 1-2
  actions during pass 2 1-2
  command line 2-1
  output files 1-13f
  role in program development 1-2
  uses 1-1
list file 1-13f
LOC (pseudo-op) 1-11
local partition, *see* partition
local symbols block 4-16, 4-3
/LOCAL (switch) 2-6, 2-11, 3-9
LODO? (symbol) 1-9
logical address space 1-3, Glossary-2
logical memory structure 1-5ff
lower page zero 1-5, Glossary-3

M

macro file 2-1, 2-4; *see also* /CLI switch
MAIN (symbol) 1-7; *see also* /MAIN
/MAIN (switch) 2-11
map file 1-13, Glossary-2
/MAP (switch) 2-6
MASM (macroassembler) 1-1
  source code relocation 1-11
memory
  absolute 1-5
  concepts, AOS/VS 1-3ff
  map, definition 3-8, Glossary-3
  NREL 1-5
  parameter symbols 1-7ff
  parameters 1-6ff (table)
  shared and unshared 1-5
  structures, logical and physical 1-5ff
  ZREL 1-5
mini map 1-13
MKABS B-1ff
/MODMAP (switch) 2-7
/MODSYM (switch) 2-7
module revision block 4-6
module, object 1-1
movable resource 5-2
MP/AOS, cross-linking differences 5-12
/MTOP=N (switch) 2-9
/MULT=n (switch) 5-2
!*/MULTIPLE=n (switch) 5-10
/MULTIPLE (switch) 2-11

N

/N (switch) 2-7
named common block 5-12, 5-10
/NBOT=N (switch) 2-7
?NBOT 1-5
NBOT? (symbol) 1-6
? NMAX(symbol) 1-6
normal base partition, *see* partition attributes
normal relocatable memory area (NREL) 1-5
/NRC (switch) 2-7, 5-9
NREL 1-5
NREL partition 5-11
/NRP (switch) 2-7, 5-9
/NSLS (switch) 2-7
?NTAS (symbol) 1-6
/NTOP=N (switch) 2-7
?NTOP 1-6
NTOP? (symbol) 1-7
numbers
  external 3-7
  see also partition
numeric values, entering 2-3
/NUMERIC (switch) 2-7

O

/O=filename (switch) 2-7
OB extension 2-1
object block
  header 4-4
  order 4-3
  sequence number (octal) 4-2
  structures 4-1ff
  types 4-2
object blocks 4-1ff, 1-1, Glossary-3
object descriptor 6-3
object module 1-1, Glossary-3
/OBPRINT (switch) 2-7
OL (overlay file) 1-14
optimization Glossary-3
optimization switches 5-6
optimizing resource calls 5-6ff
OSL.LB 5-10
output files 1-13f
output files, filenames 2-1
overlay
  area 5-2, Glossary-3
  area number 5-2
  definition 5-2f, Glossary-3
  descriptor 5-11
  designators 5-8, 5-2, Glossary-3
  file 1-14
  number 5-11
  primitive overlay calls 5-4
  resource calls 5-4
overlays
  conventional A-5
  RDOS A-5
  RTOS A-5
  virtual A-5
overwrite-without-message partition, *see* partition attributes
/OVERWRITE (switch) 2-7, 2-11, 3-3

P

/PADDING=n (switch) 2-8
page zero relocatable memory area (ZREL) 1-5
page
  definition 1-3, Glossary-3
  shared 1-5
  unshared 1-5
paging, demand 1-3
parameters, memory 1-6ff (table)
PART (pseudo-op) 3-5; *see also* partition
partition
  default 3-4f
  definition 3-1, Glossary-3
  for 16-bit modules 5-8
  global user-defined 3-6
  local user-defined 3-6
  NREL 5-8
  numbers 3-4
  relocation 3-7
  relocation base 3-2
  types, predefined 3-4f
  types, user-defined 3-6
partition attributes
  definition 3-1ff
  absolute 3-1, 3-5
  alignment 3-3, 4-16
  code and data 3-3
  definition 3-1ff
  normal base and common base 3-2
  NREL (long and short) 3-1
  overwrite-with-message 3-4
  overwrite-without-message 3-4
  shared and unshared 3-2
  ZREL 3-1
partition definition block 4-6f, 4-3
partition descriptor 5-11
passing arguments to resource calls 5-2
PENT (symbol type), *see* procedure entry symbol
physical memory structure 1-5ff
position independent routines Glossary-3
PR (program file) 1-13
predefined partition, *see* partition
procedure entry symbol (PENT) 5-3
processor, language 1-1

program development cycle 1-1
program file 1-13
/PRSYM (switch 5-9, 2-8

R

radix 2-3
RB (relocatable binary file) A-3, B-1
RCA?? (symbol) 1-9
RCALL? (resource call) 5-6ff
RCH?? (symbol) 1-9
RCHAIN? (resource call) 5-6ff
RDOS
  cross-linking for A-1ff
  overlay areas A-2
  overlays A-6
  save file (.SV) A-2
  system calls A-2
  system library A-2
relocatable binary file A-3, B-1
relocatable code 1-11
relocatable data Glossary-4
relocation
  base 1-11, 3-7
  base, of a partition 3-2
  call word, *see* call relocation word
  data 3-9ff
  definition 1-10, Glossary-4
  dictionary entry 4-13
  entries
    bit field 3-11, 3-8f, 4-13
    definition Glossary-4
    dictionary 3-11, 3-8f, 4-12
    extended 3-11, 3-8f, 4-12
    extended dictionary 3-11, 3-8f, 4-13
    operation 4-12, 5-5
    operations F-1ff
    standard 3-10, 3-9, 4-12
    see also data block
  partition 3-7
  performed by Link 1-12ff
  performed by MASM 1-11
resource
  definition 5-2ff, Glossary-4
  descriptor 5-4
  handler table (RHT) 5-3
  immovable 5-2
  manager 5-2f
  movable 5-42
  resolution 5-3ff
resource call
  definition 5-2ff, Glossary-4
  optimization 5-4, 5-6ff
  optimization, switches 5-6
  see also overlay
?REV (symbol) 1-6
revision block 4-6, 4-3
revision number 4-5f
revision number, default 2-5
/REVISION (switch) 2-5, 4-5
/REVISION=w.x.y.z (switch) 2-8
RHT (resource handler table) 5-7
ring structure 1-3
/RING=n (switch) 2-8
rings 1-3, Glossary-4
Rn (radix designator) 2-3
root 5-2, Glossary-4
root filename 2-1
RTOS
  cross-linking for A-1ff
  overlays A-6

S

/SBOT=n (switch) 2-8
?SBOT (symbol) 1-6
/SC=default partition (switch) 2-12
/SD=default partition (switch) 2-12
SFALT (fault handler) 2-3, 3-12
SFALT (symbol) 1-7, 3-12
?SFLT (symbol) 1-9
shared overlay area 5-3
shared pages 1-4, Glossary-4
shared partition, *see* partition attributes
/SHARED (switch) 2-11
/SRES=n (switch) 2-8
?SRES (symbol) 1-7
ST (symbol table file) 1-13; *see also* /SUPST
stack
  definition 3-5, Glossary-4
  default 2-5, 3-12
  fault handler 3-12
  registers 3-12
/STACKSIZE (switch) 2-5, 3-12
/STACKSIZE=n (switch) 2-8, 3-12
standard relocation entries, *see* relocation entries
/START (switch) 2-11
/STATISTICS (switch) 2-8
STYP? (symbol) 1-9
/SUPST (switch) 2-8
SV (RDOS save file) A-2

switch
  argument 2-3, 2-11 (table)
  definition 2-3ff, Glossary-5
  function 2-3
  output file directives 2-6
  sequence 2-3, 2-6 (table)
switches
  for 16-bit modules 5-9ff
  resource call optimization 5-6
SY (AOS system program file), *see* /BUILDSYSTEM
symbol
  accumulating 2-12
  chain external 4-15
  external 3-6
  procedure entry 5-3
  standard entry 5-3
  suppressed external 4-15
symbol table file 1-13, Glossary-4
symbols, memory parameter 1-7ff
SYS.LB (RDOS system library) A-2
system libraries 2-2, Glossary-5
system rings 1-3, Glossary-5
system tables Glossary-5
/SYSTEM=AOS (switch) 5-9
/SYSTEM=MPAOS (switch) 5-9
/SYSTEM=n (switch) 2-9
/SYSTEM=n (switch)
  /SYS=AOS 2-9
  /SYS=DGUX 2-9
  /SYS=MPAOS 2-9
  /SYS=RDOS 2-9, A-1
  /SYS=RTOS 2-9, A-1
  /SYS=VS16 2-9
  /SYS=VS32 2-9
/SYSTEM=RDOS (switch) 5-9
/SYSTEM=RTOS (switch) 5-9
/SYSTEM=VS16 (switch) 5-9

T

target relocation word 5-4ff
task block 4-23
/TASKS=n (switch) 2-10, 4-23
?TBOT (symbol) 1-10
/TEMP=pathname pointer (switch) 2-10
TITL (pseudo-op) 4-5
title block 4-5, 4-3
TMAX (symbol) 1-9
TMIN (symbol) 1-9
?TTOP (symbol) 1-9

U

/UC=default partition (switch) 2-12
/UD=default partition (switch) 2-12
/UDF (switch) 2-10
/ULAST=n (switch) 2-10
?UNDF (symbol) 1-7, 4-15
unlabeled common block 4-18
unshared overlay area 5-2
unshared pages 1-4, Glossary-5
unshared partition, *see* partition attributes
/UNUSEDSIZE=n (switch) 2-10
/UNX (switch) 2-10
?URHT (symbol) 1-9
URT.LB (system library) 2-2, 5-10
URT16.LB 5-10
URT16.LB (system library) 2-2
URT32.LB (system library) 2-2
?URTB (symbol) 1-7
User Status Table (UST) 1-3
user-defined partition, *see* partition
UST (User Status Table) 1-3; *see also* /CHANNELS=n
?USTA (symbol) 1-7
USTAD (symbol) 1-9

V

/V (switch) 2-10
/VALUE=n (switch) 2-12
virtual overlays A-6, Glossary-5
!*/VIRTUAL (switch) 5-9

W

wide frame pointer Glossary-4
wide stack
  base Glossary-4
  limit Glossary-4
  pointer Glossary-4
working set 1-3
/WRL (switch) 2-10

X

/XREF (switch) 2-10

Z

/ZBOT=n (switch) 2-10
ZBOT ? 1-5
?ZBOT (symbol) 1-7
ZMAX ? 1-5
?ZMAX (symbol) 1-7
/ZR=default partition (switch) 2-10
ZREL 1-5
ZREL partition, *see* partition attributes

Data General

TP________

TIPS ORDER FORM
Technical Information & Publications Service

BILL TO:
COMPANY NAME________________________
ADDRESS ________________________
CITY ________________________
STATE____________ ZIP ____________
ATTN: ________________________

SHIP TO: (if different)
COMPANY NAME________________________
ADDRESS ________________________
CITY ________________________
STATE____________ ZIP ____________
ATTN: ________________________

QTY	MODEL #	DESCRIPTION	UNIT PRICE	LINE DISC	TOTAL PRICE

(Additional items can be included on second order form)

[Minimum order is $50.00]

Tax Exempt #__________
or Sales Tax (if applicable)

TOTAL	
Sales Tax	
Shipping	
TOTAL	

METHOD OF PAYMENT

□ Check or money order enclosed
For orders less than $100.00

□ Charge my □ Visa □ MasterCard
Acc't No.__________ Expiration Date__________

□ Purchase Order Number:______________

SHIP VIA

□ DGC will select best way (U.P.S or Postal)

□ Other:
□ U.P.S. Blue Label
□ Air Freight
□ Other ______________________
____

NOTE: ORDERS LESS THAN $100, INCLUDE $5.00 FOR SHIPPING AND HANDLING.

Person to contact about this order ______________ Phone __________ Extension ______

Mail Orders to:

Data General Corporation
Attn: Educational Services/TIPS F019
4400 Computer Drive
Westboro, MA 01580
Tel. (617) 366-8911 ext. 4032

____
Buyer's Authorized Signature Date
(agrees to terms & conditions on reverse side)

____
Title

____
DGC Sales Representative (If Known) Badge #

DISCOUNTS APPLY TO
MAIL ORDERS ONLY

educational services

CUT ALONG DOTTED LINE

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form shown on the reverse hereof which is accepted by DGC.

1. **PRICES**
   Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. **PAYMENT**
   Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice.

3. **SHIPMENT**
   Shipment will be made F.O.B. Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment.

4. **TERM**
   Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. **CUSTOMER CERTIFICATION**
   Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

6. **DATA AND PROPRIETARY RIGHTS**
   Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

7. **DISCLAIMER OF WARRANTY**
   DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS SUPPLIED HEREUNDER.

8. **LIMITATIONS OF LIABILITY**
   IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. **GENERAL**
   A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer.

DISCOUNT SCHEDULES

DISCOUNTS APPLY TO MAIL ORDERS ONLY.

LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20%
15 or more manuals of the same part number - 30%

DISCOUNTS APPLY TO PRICES SHOWN IN THE CURRENT TIPS CATALOG ONLY.

DataGeneral

TIPS ORDERING PROCEDURE:

Technical literature may be ordered through the Customer Education Service's Technical Information and Publications Service (TIPS).

1. Turn to the TIPS Order Form.
2. Fill in the requested information. If you need more space to list the items you are ordering, use an additional form. Transfer the subtotal from any additional sheet to the space marked "subtotal" on the form.
3. Do not forget to include your MAIL ORDER ONLY discount. (See discount schedules on the back of the TIPS Order Form.)
4. Total your order. (MINIMUM ORDER/CHARGE after discounts of $50.00.)

   If your order totals less than 100.00, enclose a certified check or money order for the total (include sales tax, or your tax exempt number, if applicable) plus $5.00 for shipping and handling.
5. Please indicate on the Order Form if you have any special shipping requirements. Unless specified, orders are normally shipped U.P.S.
6. Read carefully the terms and conditions of the TIPS program on the reverse side of the Order Form.
7. Sign on the line provided on the form and enclose with payment. Mail to:

   TIPS
   Educational Services - M.S. F019
   Data General Corporation
   4400 Computer Drive
   Westboro, MA 01580
8. We'll take care of the rest!

educational services

134-755-01

moisten & seal

CUSTOMER DOCUMENTATION COMMENT FORM

Your Name ____________________ Your Title ____________

Company ______________________________________

Street ____

City ____________________ State ________ Zip ______

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title ____________________ Manual No. ____________

Who are you?
□EDP/MIS Manager
□Senior Systems Analyst
□Engineer
□Analyst/Programmer
□Operator
□End User
□Other ____________
____

How do you use this manual? *(List in order: 1 = Primary Use)*

___ Introduction to the product
___ Reference
___ Tutorial Text
___ Operating Guide
___ Other
____

fold

		Yes	No
About the manual:	Is it easy to read?	□	□
	Is it easy to understand?	□	□
	Are the topics logically organized?	□	□
	Is the technical information accurate?	□	□
	Can you easily find what you want?	□	□
	Does it tell you everything you need to know?	□	□
	Do the illustrations help you?	□	□

If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA 01772

Postage will be paid by addressee

DataGeneral

Customer Documentation
MS E-219
4400 Computer Drive
Westboro, MA 01581-9973

134-755-01

moisten & seal

CUSTOMER DOCUMENTATION COMMENT FORM

Your Name ____________________ Your Title ____________________

Company ____________________

Street ____________________

City ____________________ State ____________ Zip ______

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title ____________________ Manual No. ____________________

Who are you?
□EDP/MIS Manager
□Senior Systems Analyst
□Engineer
□Analyst/Programmer
□Operator
□End User
□Other ____________
____

How do you use this manual? *(List in order: 1 = Primary Use)*

___ Introduction to the product
___ Reference
___ Tutorial Text
___ Operating Guide
___ Other
____

fold

		Yes	No
About the manual:	Is it easy to read?	□	□
	Is it easy to understand?	□	□
	Are the topics logically organized?	□	□
	Is the technical information accurate?	□	□
	Can you easily find what you want?	□	□
	Does it tell you everything you need to know?	□	□
	Do the illustrations help you?	□	□

If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA 01772

Postage will be paid by addressee

DataGeneral

Customer Documentation
MS E-219
4400 Computer Drive
Westboro, MA 01581-9973

Data General Corporation, Westboro, MA 01580