OpenVMS Linker Utility Manual

December 1995

This manual describes the OpenVMS Linker utility.

Revision/Update Information: This manual supersedes the *OpenVMS Linker Utility Manual*, Version 6.2

Software Version: OpenVMS Alpha Version 7.0
OpenVMS VAX Version 7.0

Digital Equipment Corporation
Maynard, Massachusetts

December 1995

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Digital or an authorized sublicensor.

Digital conducts its business in a manner that conserves the environment and protects the safety and health of its employees, customers, and the community.

© Digital Equipment Corporation 1995. All rights reserved.

The following are trademarks of Digital Equipment Corporation: ACMS, AXP, DEC, DECdirect, DEC Fortran, DECmigrate, Digital, OpenVMS, VAX, VAX BLISS-32, VAX C, VAX FORTRAN, VAX MACRO, VMS, and the DIGITAL logo.

All other trademarks and registered trademarks are the property of their respective holders.

ZK4548

This document is available on CD–ROM.

Contents

Preface xi

Part I Linker Utility Description

1 Introduction

1.1 Overview 1–1
1.1.1 Linker Functions 1–2
1.1.2 Using the Linker 1–3
1.2 Specifying Input to the Linker 1–4
1.2.1 Object Modules as Linker Input Files 1–5
1.2.2 Shareable Images as Linker Input Files 1–6
1.2.2.1 Including a Shareable Image in a Link Operation 1–7
1.2.2.2 Installing a Shareable Image 1–8
1.2.3 Library Files as Linker Input Files 1–8
1.2.3.1 Creating a Library File 1–8
1.2.3.2 Including a Library File in a Link Operation 1–9
1.2.4 Symbol Table Files as Linker Input Files 1–10
1.2.5 Options Files as Linker Input Files 1–10
1.3 Specifying Linker Output Files 1–12
1.3.1 Creating an Executable Image 1–13
1.3.2 Creating a Shareable Image 1–13
1.3.3 Creating a System Image 1–13
1.3.4 Creating a Symbol Table File 1–14
1.3.5 Creating a Map File 1–14
1.3.6 Creating a Debug Symbol File (Alpha Images Only) 1–15
1.4 Optimizing the Performance of Alpha Images 1–15
1.4.1 Linker Default Image Optimizations (Alpha Images Only) 1–15
1.4.2 Installing Images as Resident Images (Alpha Systems Only) 1–16
1.5 Controlling a Link Operation 1–16
1.5.1 Linker Qualifiers 1–17
1.5.2 Link Options 1–19
1.6 Linking for Different Architectures 1–20

2 Understanding Symbol Resolution

2.1 Overview 2–1
2.1.1 Types of Symbols 2–1
2.1.2 Linker Symbol Resolution Processing 2–2
2.2 Input File Processing for Symbol Resolution 2–5
2.2.1 Processing Object Modules 2–6
2.2.2 Processing Shareable Images 2–10

2.2.3 Processing Library Files 2–11
2.2.3.1 Identifying Library Files Using the /LIBRARY Qualifier 2–12
2.2.3.2 Including Specific Modules from a Library Using the /INCLUDE Qualifier 2–13
2.2.3.3 Processing Default Libraries 2–13
2.2.3.4 Open Systems Library Support 2–14
2.2.4 Processing Input Files Selectively 2–15
2.3 Ensuring Correct Symbol Resolution 2–16
2.3.1 Understanding Cluster Creation 2–16
2.3.2 Controlling Cluster Creation 2–18
2.3.2.1 Using the CLUSTER= Option to Control Clustering 2–18
2.3.2.2 Using the COLLECT= Option to Control Clustering 2–18
2.4 Resolving Symbols Defined in the OpenVMS Executive 2–19
2.5 Defining Weak and Strong Global Symbols 2–20

3 Understanding Image File Creation

3.1 Overview 3–1
3.2 Creating Program Sections 3–3
3.3 Creating Image Sections 3–9
3.3.1 Processing Clusters to Create Image Sections 3–9
3.3.2 Combining Program Sections into Image Sections 3–10
3.3.3 Processing Significant Program Section Attributes 3–11
3.3.4 Allocating Memory for Image Sections 3–17
3.3.5 Image Section Attributes 3–18
3.3.6 Controlling Image Section Creation 3–21
3.3.6.1 Modifying Program Section Attributes 3–22
3.3.6.2 Manipulating Cluster Creation 3–22
3.3.6.3 Isolating a Program Section into an Image Section 3–23
3.4 Initializing an Image 3–23
3.4.1 Writing the Binary Contents of Image Sections 3–23
3.4.2 Fixing Up Addresses 3–24
3.4.3 Keeping the Size of Image Files Manageable 3–25
3.4.3.1 Controlling Demand-Zero Image Section Creation 3–25

4 Creating Shareable Images

4.1 Overview 4–1
4.2 Declaring Universal Symbols in VAX Shareable Images 4–2
4.2.1 Creating Upwardly Compatible Shareable Images (VAX Linking Only) 4–4
4.2.1.1 Creating a Transfer Vector (VAX Linking Only) 4–5
4.2.1.2 Fixing the Location of the Transfer Vector in Your Image (VAX Linking Only) 4–7
4.2.2 Creating Based Shareable Images (VAX Linking Only) 4–8
4.3 Declaring Universal Symbols in Alpha Shareable Images 4–8
4.3.1 Creating Upwardly Compatible Shareable Images (Alpha Linking Only) 4–9
4.3.2 Deleting Universal Symbols Without Disturbing Upward Compatibility (Alpha Linking Only) 4–10
4.3.3 Creating Run-Time Kits (Alpha Linking Only) 4–10
4.3.4 Specifying an Alias Name for a Universal Symbol (Alpha Linking Only) 4–11

4.3.5 Improving the Performance of Installed Shareable Images (Alpha Linking Only) 4–11

5 Interpreting an Image Map File

5.1 Overview 5–1
5.2 Components of an Image Map File 5–2
5.2.1 Object Module Synopsis 5–3
5.2.2 Module Relocatable Reference Synopsis (VAX Linking Only) 5–3
5.2.3 Image Section Synopsis Section 5–4
5.2.4 Program Section Synopsis Section 5–6
5.2.5 Symbols By Name Section 5–8
5.2.6 Symbol Cross-Reference Section 5–8
5.2.7 Symbols By Value Section 5–9
5.2.8 Image Synopsis Section 5–10
5.2.9 Link Run Statistics Section 5–11

Part II LINK Command Reference

LINK LINK–3

Qualifier Descriptions LINK–4

/ALPHA LINK–5
/BPAGE LINK–6
/BRIEF LINK–7
/CONTIGUOUS LINK–8
/CROSS_REFERENCE LINK–9
/DEBUG LINK–10
/DEMAND_ZERO (Alpha Only) LINK–11
/DSF (Alpha Only) LINK–13
/EXECUTABLE LINK–14
/FULL LINK–15
/GST (Alpha Only) LINK–16
/HEADER LINK–17
/INCLUDE LINK–18
/INFORMATIONALS LINK–19
/LIBRARY LINK–20
/MAP LINK–21
/NATIVE_ONLY (Alpha Only) LINK–22
/OPTIONS LINK–23
/P0IMAGE LINK–24
/PROTECT LINK–25
/REPLACE (Alpha Only) LINK–26
/SECTION_BINDING (Alpha Only) LINK–27
/SELECTIVE_SEARCH LINK–29
/SHAREABLE LINK–31
/SYMBOL_TABLE LINK–33
/SYSEXE (Alpha Only) LINK–35
/SYSLIB LINK–37
/SYSSHR LINK–38

/SYSTEM LINK–39
/TRACEBACK LINK–40
/USERLIBRARY LINK–41
/VAX LINK–43

Option Descriptions LINK–44

BASE= (VAX Only) LINK–45
CASE_SENSITIVE= LINK–47
CLUSTER= LINK–49
COLLECT= LINK–50
DZRO_MIN= LINK–52
GSMATCH= LINK–54
IDENTIFICATION= LINK–57
IOSEGMENT= LINK–58
ISD_MAX= LINK–59
NAME= LINK–60
PROTECT= LINK–61
PSECT_ATTR= LINK–62
RMS_RELATED_CONTEXT= LINK–63
STACK= LINK–64
SYMBOL= LINK–65
SYMBOL_TABLE= (Alpha Only) LINK–66
SYMBOL_VECTOR= (Alpha Only) LINK–67
UNIVERSAL= (VAX Only) LINK–69

A VAX Object Language

A.1 Object Language Overview A–1
A.2 Header Records A–3
A.2.1 Main Module Header Record (MHD$C_MHD) A–4
A.2.2 Language Processor Name Header Record (MHD$C_LNM) A–5
A.2.3 Source Files Header Record (MHD$C_SRC) A–5
A.2.4 Title Text Header Record (MHD$C_TTL) A–6
A.3 Global Symbol Directory Records A–6
A.3.1 Program Section Definition Subrecord (GSD$C_PSC) A–8
A.3.2 Global Symbol Specification Subrecord (GSD$C_SYM) A–10
A.3.2.1 GSD Subrecord for a Symbol Definition A–10
A.3.2.2 GSD Subrecord for a Symbol Reference A–11
A.3.3 Entry-Point-Symbol-and-Mask-Definition Subrecord (GSD$C_EPM) A–12
A.3.4 Procedure-with-Formal-Argument-Definition Subrecord (GSD$C_PRO) A–13
A.3.5 Symbol-Definition-with-Word-Psect Subrecord (GSD$C_SYMW) A–16
A.3.6 Entry-Point-Definition-with-Word-Psect Subrecord (GSD$C_EPMW) A–16
A.3.7 Procedure-Definition-with-Word-Psect Subrecord (GSD$C_PROW) . . . A–16
A.3.8 Entity-Ident-Consistency-Check Subrecord (GSD$C_IDC) A–16
A.3.9 Environment-Definition/Reference Subrecord (GSD$C_ENV) A–18

A.3.10 Module-Local Symbol Definition/Symbol Reference Subrecord (GSD$C_LSY) . . . A–19
A.3.10.1 Module-Local Symbol Definition . . . A–19
A.3.10.2 Module-Local Symbol Reference . . . A–19
A.3.11 Module-Local Entry-Point-Definition Subrecord (GSD$C_LEPM) A–19
A.3.12 Module-Local Procedure-Definition Subrecord (GSD$C_LPRO) A–20
A.3.13 Program-Section-Definition-in-Shareable-Image Subrecord (GSD$C_SPSC) . . . A–20
A.3.14 Vectored-Symbol-Definition Subrecord (GSD$C_SYMV) . . . A–20
A.3.15 Vectored-Entry-Point-Definition Subrecord (GSD$C_EPMV) . . . A–20
A.3.16 Vectored-Procedure-Definition Subrecord (GSD$C_PROV) . . . A–21
A.3.17 Symbol-Definition-with-Version-Mask Subrecord (GSD$C_SYMM) . . . A–21
A.3.18 Entry-Point-Definition-with-Version-Mask Subrecord (GSD$C_EPMM) . . . A–21
A.3.19 Procedure-Definition-with-Version-Mask Subrecord (GSD$C_PROM) . . . A–21
A.4 Text Information and Relocation Records (OBJ$C_TIR) . . . A–21
A.4.1 Stack Commands . . . A–23
A.4.2 Store Commands . . . A–25
A.4.3 Operator Commands . . . A–28
A.4.4 Control Commands . . . A–30
A.5 End-of-Module Record . . . A–31
A.6 End-of-Module-with-Word-Psect Record . . . A–32
A.7 Debugger Information Records . . . A–33
A.8 Traceback Information Records . . . A–33
A.9 Link Option Specification Records . . . A–33

B Alpha Object Language

B.1 Object Language Overview . . . B–1
B.2 Module Header Records (EOBJ$C_EMH) . . . B–5
B.2.1 Main Module Header Record (EMH$C_MHD) . . . B–6
B.2.2 Language Processor Name Header Record (EMH$C_LNM) . . . B–8
B.2.3 Source Files Header Record (EMH$C_SRC) . . . B–8
B.2.4 Title Text Header Record (EMH$C_TTL) . . . B–9
B.3 Global Symbol Directory Records (EOBJ$C_EGSD) . . . B–10
B.3.1 Program Section Definition Subrecords (EGSD$C_PSC, EGSD$C_SPSC) . . . B–11
B.3.1.1 Normal Program Section Definition Subrecord (EGSD$C_PSC) . . . B–11
B.3.1.2 Program-Section-Definition-in-Shareable-Image Subrecord (GSD$C_SPSC) . . . B–14
B.3.1.3 Standard Program Section Names and Attributes . . . B–16
B.3.2 Global Symbol Specification Subrecords (EGSD$C_SYM, EGSD$C_SYMG) . . . B–16
B.3.2.1 GSD Subrecord for a Global Symbol Definition (EGSD$C_SYM with EGSY$V_DEF Set) . . . B–17
B.3.2.2 GSD Subrecord for a Universal Symbol Definition (EGSD$C_SYMG) . . . B–20
B.3.2.3 GSD Subrecord for a Symbol Reference . . . B–23
B.3.3 Entity-Ident-Consistency-Check Subrecord (EGSD$C_IDC) . . . B–24

B.3.4 GSD Subrecords Reserved to the OpenVMS Operating System (EGSD$C_SYMV, EGSD$C_SYMM) B–26
B.3.4.1 Vectored-Symbol-Definition Subrecord (EGSD$C_SYMV) B–26
B.3.4.2 Symbol-Definition-with-Version-Mask Subrecord (EGSD$C_SYMM) B–27
B.4 Text Information and Relocation Records (EOBJ$C_ETIR) B–27
B.4.1 Stack Commands B–29
B.4.2 Store Commands B–30
B.4.3 Operator Commands B–33
B.4.4 Control Commands B–35
B.4.5 Conditional Store Commands B–36
B.4.5.1 Defining Conditional Linkage with Address-Related Commands B–36
B.4.5.2 Optimizing Instructions with Instruction-Related Commands B–38
B.4.5.2.1 Calculating JSR Hints B–42
B.5 End-of-Module Record (EOBJ$C_EEOM) B–43
B.6 Debugger Information Records (EOBJ$C_EDBG) B–46
B.7 Traceback Information Records (EOBJ$C_ETBT) B–46

Index

Examples

1–1 Hello World! Program (HELLO.C) 1–3
1–2 Sample Linker Options File 1–11
2–1 Module Containing a Symbolic Reference (my_main.c) 2–7
2–2 Module Containing a Symbol Definition (my_math.c) 2–7
3–1 Sample Program MYTEST.C 3–5
3–2 Sample Program MYADD.C 3–6
3–3 Sample Program MYSUB.C 3–6
3–4 Program Sections Generated by Example 3–1 3–7
3–5 Linking Example 3–1, Example 3–2, and Example 3–3 3–10
3–6 Image Section Information in a Map File 3–16
3–7 Program Section Information in a Map File (VAX Example) 3–16
3–8 Image Section Descriptions in an ANALYZE/IMAGE Display 3–20
3–9 Image Section Synopsis of Second Link 3–22
4–1 Shareable Image Test Module (my_main.c) 4–3
4–2 Shareable Image (my_math.c) 4–3
4–3 Transfer Vector for the Shareable Image MY_MATH.EXE 4–6

Figures

1–1 Position of the Linker in Program Development 1–2
2–1 Symbol Vector Contents 2–3
2–2 Symbol Resolution 2–4
2–3 Clusters Created for Sample Link 2–17
2–4 Linker Processing of Default Libraries and SYS$BASE_IMAGE.EXE 2–20
3–1 Communication of Image Memory Requirements 3–2

3–2 Program Sections Created for Example 3–1, Example 3–2, and Example 3–3 3–9
3–3 Combining Program Sections into Image Sections 3–11
4–1 Comparison of UNIVERSAL= Option and Transfer Vectors 4–5
4–2 Accessing Universal Symbols Specified Using the SYMBOL_VECTOR= Option 4–9
A–1 Order of Records in an Object Module A–2
A–2 GSD Record with Multiple Subrecords A–7
B–1 Order of Records in an Object Module B–3
B–2 Module Header Record with Subrecords B–6
B–3 GSD Subrecord for a Program Section Definition B–11
B–4 GSD Subrecord for a Shareable Image Program Section Definition . . . B–14
B–5 GSD Subrecord for a Global Symbol Definition (Data) B–17
B–6 GSD Subrecord for a Global Symbol Definition (Procedure) B–18
B–7 GSD Subrecords for Universal Data Definition B–20
B–8 GSD Subrecord for a Universal Procedure Definition B–21
B–9 GSD Subrecord for a Global Symbol Reference (EGSD$C_SYM with EGSY$V_DEF Clear) B–23
B–10 GSD Subrecord for an Entity Ident Consistency Check B–24
B–11 Optimization of a Standard Call B–39
B–12 Calculating a Hint to a Shareable Image B–43
B–13 End-of-Module Record B–44

Tables

1–1 Input Files Accepted by the Linker 1–5
1–2 Output Files Generated by the Linker 1–12
1–3 Linker Qualifiers 1–17
1–4 Linker Options 1–19
1–5 Logical Names for Cross-Architecture Linking 1–20
2–1 Linker Input File Processing 2–6
2–2 Linker Input File Cluster Processing 2–17
3–1 Program Section Attributes 3–4
3–2 Mapping Program Section Attributes to Image Section Attributes for Executable Images 3–13
3–3 Mapping Program Section Attributes to Image Section Attributes for Shareable Images 3–13
3–4 Significant Attributes of Program Sections in MYSUB_CLUS Cluster 3–15
3–5 Image Section Attributes 3–18
4–1 Linker Qualifiers and Options Used to Create Shareable Images 4–2
5–1 LINK Command Map File Qualifiers 5–2
5–2 Image Map Sections 5–2
5–3 Symbol Characterization Codes 5–10
A–1 Types of GSD Subrecords A–7
A–2 Alignment Field Values A–9
A–3 Stack Commands A–24
A–4 Store Commands A–26

A–5	Operator Commands	A–29
A–6	Control Commands	A–30
B–1	Object Record Types	B–1
B–2	Relationships of Structures in the Alpha Object Language	B–4
B–3	Key to Structure Prefixes	B–4
B–4	Module Header Subrecords	B–5
B–5	Types of GSD Subrecords	B–10
B–6	Alignment Field Values	B–12
B–7	Standard Program Sections	B–16
B–8	Stack Commands	B–29
B–9	Store Commands	B–30
B–10	Operator Commands	B–33
B–11	Control Commands	B–35
B–12	Summary of Store Conditional Commands for Linkage	B–36
B–13	Contents of Linkage When Symbol Is Local to the Image	B–37
B–14	Contents of Linkage When Symbol Is External to the Image	B–37
B–15	Summary of Store Conditional Commands for Instruction Replacement	B–41

Preface

Intended Audience

Programmers at all levels of experience can use this manual effectively.

Document Structure

This book has two parts and two appendixes. Part I includes five chapters that describe the linker. Part II is a reference section that describes the LINK command and its qualifiers and options. The appendixes contain the VAX Object Language specification and the Alpha Object Language specification. (The appendixes are primarily useful to compiler developers.)

In Part I, Chapter 1 introduces the OpenVMS Linker utility and how to use the LINK command and its qualifiers and parameters.

Chapter 2 describes how the linker resolves symbolic references among input files.

Chapter 3 describes how the linker creates image files.

Chapter 4 describes how to create shareable images and use them in link operations.

Chapter 5 describes how to interpret image map files.

Related Documents

For information on including the debugger in the linking operation, and on debugging in general, see the *OpenVMS Debugger Manual*.

For additional information on OpenVMS products and services, access the Digital OpenVMS World Wide Web site. Use the following URL:

`http://www.openvms.digital.com`

Reader's Comments

Digital welcomes your comments on this manual.

Print or edit the online form SYS$HELP:OPENVMSDOC_COMMENTS.TXT and send us your comments by:

Internet	**openvmsdoc@zko.mts.dec.com**
Fax	603 881-0120, Attention: OpenVMS Documentation, ZK03-4/U08
Mail	OpenVMS Documentation Group, ZKO3-4/U08 110 Spit Brook Rd. Nashua, NH 03062-2698

How To Order Additional Documentation

Use the following table to order additional documentation or information. If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Location	Call	Fax	Write
U.S.A.	DECdirect 800–DIGITAL 800–344–4825	Fax: 800–234–2298	Digital Equipment Corporation P.O. Box CS2008 Nashua, NH 03061
Puerto Rico	809–781–0505	Fax: 809–749–8300	Digital Equipment Caribbean, Inc. 3 Digital Plaza, 1st Street, Suite 200 P.O. Box 11038 Metro Office Park San Juan, Puerto Rico 00910–2138
Canada	800–267–6215	Fax: 613–592–1946	Digital Equipment of Canada, Ltd. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: DECdirect Sales
International	—	—	Local Digital subsidiary or approved distributor
Internal Orders	DTN: 264–4446 603–884–4446	Fax: 603–884–3960	U.S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063–1260

ZK–7654A–GE

Conventions

The name of the OpenVMS AXP operating system has been changed to OpenVMS Alpha. Any references to OpenVMS AXP or AXP are synonymous with OpenVMS Alpha or Alpha.

The following conventions are used to identify information specific to OpenVMS Alpha or to OpenVMS VAX:

Alpha	The Alpha icon denotes the beginning of information specific to OpenVMS Alpha.
VAX	The VAX icon denotes the beginning of information specific to OpenVMS VAX.
◆	The diamond symbol denotes the end of a section of information specific to OpenVMS Alpha or to OpenVMS VAX.

The following conventions are also used in this manual:

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.

PF1 *x* or GOLD *x*	A sequence such as PF1 *x* or GOLD *x* indicates that you must first press and release the key labeled PF1 or GOLD and then press and release another key or a pointing device button. GOLD key sequences can also have a slash (/), dash (–), or underscore (_) as a delimiter in EVE commands.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	Horizontal ellipsis points in examples indicate one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	Vertical ellipsis points indicate the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In command format descriptions, brackets indicate optional elements. You can choose one, none, or all of the options. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS file specification or in the syntax of a substring specification in an assignment statement.)
{ }	In command format descriptions, braces indicate a required choice of options; you must choose one of the options listed.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason. Boldface text is also used to show user input in Bookreader versions of the manual.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *device-name* contains up to five alphanumeric characters).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Part I

Linker Utility Description

1 Introduction

This chapter introduces the OpenVMS Linker utility (the linker), describing its primary functions and its role in software development. The chapter describes the following:

- How to invoke the linker
- How to specify input files in a link operation
- How to specify which output files the linker produces

In addition, this chapter provides an overview of how you can control a link operation by using qualifiers and options.

1.1 Overview

The primary purpose of the linker is to create images. An **image** is a file, containing binary code and data, that can be executed on an OpenVMS system. When invoked on a VAX system, the linker creates OpenVMS VAX images by default; when invoked on an Alpha system, the linker creates OpenVMS Alpha images by default.

The primary type of image the linker creates is an **executable image**. This type of image can be activated at the DCL command line by issuing the RUN command. At run time, the **image activator**, which is part of the operating system, opens the image file and reads information from the image header to set up process page tables and pass control to the location (transfer address) where image execution is to begin.

The linker can also create a **shareable image**. A shareable image is a collection of procedures that can be called by executable images or other shareable images. A shareable image is similar to an executable image; however, it cannot be executed by issuing the RUN command (shareable images do not contain a transfer address). Before a shareable image can be run, you must include it as an input file in a link operation in which an executable image is created. At run time, when the image activator processes the header in an executable image, it activates all the shareable images to which the executable image was linked.

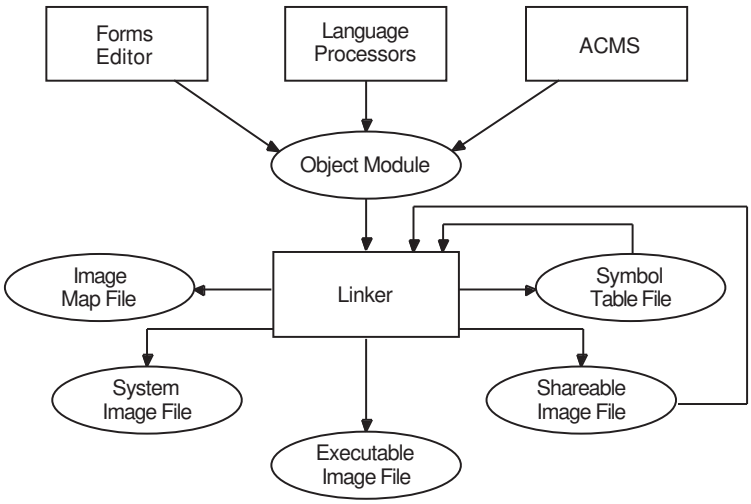
The linker can also create a **system image**, which can be run as a standalone system. System images generally do not have image headers and are not activated by the image activator.

The linker creates images by processing the input files you specify. The primary type of input file that can be specified in a link operation is an **object file**. Object files are produced by language processors, such as compilers or assemblers. The linker also accepts other input files such as shareable images and symbol table files, both the products of previous link operations. Section 1.2 provides more information about all the types of input files accepted by the linker. Section 1.3 provides more information about the output files created by the linker.

Introduction
1.1 Overview

Figure 1–1 illustrates the relationship of the linker to the language processor in the program development process.

Figure 1–1 Position of the Linker in Program Development



1.1.1 Linker Functions

To create an image from the input files you specify, the linker performs the following primary functions:

- **Symbol resolution**. Source modules can use symbols to represent the location of a routine entry point, the location of a data item, or a constant value. A source module may reference symbols that are defined externally to the module. When a language processor, such as a compiler or assembler, processes the source module it cannot find the value of a symbol defined externally to the module. The language processors flag these externally defined symbols as unresolved symbolic references and leaves it to the linker to find their definitions among the other input files you specify. When the linker finds the definition of a symbol, it substitutes the value of the symbol (its definition) for the reference to the symbol. Chapter 2 provides more information about symbol resolution.
- **Virtual memory allocation**. After resolving symbolic references among the input files, the linker allocates virtual memory for the image, based on the memory requirements specified in the program section definitions processed during symbol resolution. Chapter 3 provides more information about memory allocation.
- **Image initialization**. After it resolves references and specifies the memory requirements of the image, the linker initializes the image by filling it with the compiled binary data and code. The linker also inserts the actual value of resolved symbols at each instance where the symbol is referenced.

  Certain global symbols cannot be resolved at link time. For example, in shareable images, the value of a symbol that represents an address cannot be determined until run time, when the image activator loads the image in memory. The linker lists the symbols it cannot resolve in a **fix-up section** and the image activator supplies their actual address at run time by **relocating** the symbols.

When the image activator loads a shareable image in memory and relocates all the symbols in the shareable image, it must ensure that the other images that reference the symbols in the shareable image have their correct addresses. Chapter 3 provides more information about image initialization.

Alpha

- **Image optimization**. For Alpha images, the linker can perform certain optimizations to improve the performance of the image it is creating. These optimizations include replacing JSR instruction sequences with the more efficient Branch to Subroutine (BSR) instruction sequence wherever the language processors specify. For more information, see Section 1.4.♦

1.1.2 Using the Linker

You invoke the linker interactively by typing the LINK command together with the appropriate input file names at the DCL prompt. You may also invoke the linker by including the LINK command in a command procedure. (For more information about invoking the linker, see Part II.)

For example, the simple program illustrated in Example 1–1 prints the greeting ''Hello World!'' on the terminal.

Example 1–1 Hello World! Program (HELLO.C)

```
#include <studio.h>

main()
{
   printf("Hello World!\n");
}
```

To run this program, you must first compile the source file to create an object module. To compile the example, written in C, invoke the appropriate C compiler to create an object module, as in the following example:

```
$ CC HELLO
```

During compilation, the compiler translates the statements in the source file into machine instructions and groups portions of the program into program sections according to their memory use and other characteristics. In addition, the compiler lists all the global symbols defined in the module and referenced in the module in the global symbol directory (GSD). For example, in Example 1–1, the `printf` routine is referenced in the module but is not defined in it.

To create an executable image, you must then link the object file produced by the compiler, as in the following example:

```
$ LINK HELLO
```

In the link operation, you must specify as input files all the modules required to create the image. For example, you must include the C run-time library in the link of the program in Example 1–1 to resolve the reference to the `printf` routine.

Alpha

For Alpha linking, the linker processes the C run-time library shareable image [DECC$SHARE] by default because it resides in the default system shareable image library [IMAGELIB.OLB]. See Section 2.2.3.3 for more information about how the linker processes default system libraries.♦

VAX

For VAX linking, you must specify the C run-time shareable image [VAXCRTL] as an input file in the link operation explicitly, as in the following example:

```
$ LINK HELLO, SYS$INPUT/OPT
SYS$LIBRARY:VAXCRTL/SHARE
Ctrl/Z
```

(For more information about linking against the C run-time library for VAX images, see the VAX C compiler documentation.)♦

The linker processes the input files you specify in two passes. In its first pass through the input files, the linker resolves symbolic references between the modules. Because the linker processes different types of input files in different ways, the order in which you specify input files can affect symbol resolution. Chapter 2 provides more information about this topic.

After performing symbol resolution, when the linker has determined all the input modules necessary to create the image, the linker determines the memory requirements of the image based on the memory requirements of each object module. The compilers have specified the memory requirements of the object modules as program section specifications. The linker gathers together program sections with similar attributes into **image sections**. (Chapter 3 provides more information about image creation.) At run time, the image activator reads the memory requirements of the image by processing the list of image section descriptors (ISDs) the linker has stored in the image header.

If the image that results from the link operation is an executable image, it can be executed at the DCL command line. The following example illustrates how to execute the program in Example 1–1:

```
$ RUN HELLO
Hello World!
```

Note that a LINK command required to create a real application, unlike the Hello World! example, can involve specifying hundreds of input files of various types.

As with most other DCL commands, the LINK command supports numerous qualifiers with which you can control various aspects of a link operation. The linker also supports linker **options**, which you can use to further control a link operation. Linker options must be specified in an options file, which is then specified as an input file in a link operation. Section 1.2.5 describes the benefits of using options files and describes how to create them. Part II provides reference descriptions of all the qualifiers and options supported by the linker. Section 1.5 contains a summary table of these qualifiers and options.

1.2 Specifying Input to the Linker

You specify the files you want the linker to process on the LINK command line or in a linker options file. (Library files may also be specified as a user library, which the linker processes by default.) You must specify at least one input file in every link operation and, in every link operation, at least one input file must be an object module. Table 1–1 lists the different types of input files accepted by the linker, along with their default file types. (The defaults are used for both VAX and Alpha linking.) The table also describes how you can specify the file in a link operation.

Table 1–1 Input Files Accepted by the Linker

File	Default File Type	Description
Object file	.OBJ	Created by a language processor. May be specified on LINK command line or in a linker options file. This is the default input file accepted by the linker.
Shareable image	.EXE	Produced by a previous link operation. Must be specified in a linker options file; you cannot specify a shareable image as an input file on the command line. Identify the input file as a shareable image by appending the /SHAREABLE qualifier to the file specification.
Library file	.OLB	Produced by the Librarian utility. May contain object modules or shareable images. May be specified on the LINK command line, in a linker options file, or as a default user library processed by the linker. Identify the input file as a library file by appending the /LIBRARY qualifier to the library file specification. You can also include specific modules from a library in a link operation by appending the /INCLUDE qualifier to the library file specification.
Symbol table file	.STB	Produced by a previous link operation or a language processor. May be specified on the LINK command line or in an options file. Because a symbol table file is processed as an object module, it requires no identifying qualifier. Note that symbol table files produced by the linker during Alpha links cannot be specified as input files in a link operation. They are intended to be used only as an aid to debugging with the System Dump Analyzer utility (See Section 1.2.4 for more information.)
Options file	.OPT	Text file containing link option specifications or link input file specifications. May be specified only on the command line; you cannot specify an options file inside another options file. Identify the input file as an options file by appending the /OPTIONS qualifier to the end of the file specification.

Note that you must specify only native object modules and shareable images as input files when creating images. That is, you must specify only native OpenVMS Alpha object modules and shareable images as input files when creating an OpenVMS Alpha image. (Note, however, that OpenVMS VAX images can run as *translated* images on OpenVMS Alpha systems and translated images can interoperate with native OpenVMS Alpha images. See *DECmigrate for OpenVMS AXP Systems Translating Images* for more information.)

1.2.1 Object Modules as Linker Input Files

When a language processor translates a source language program, it produces an output file that contains one or more object modules. This output file, called an object file, has the default file type of .OBJ and is the primary form of linker input. At least one object file must be specified in any link operation. An object file may be specified in the command line or in an options file.

For example, in Example 1–1, the only input file specified on the LINK command line is the object module named HELLO.OBJ (the .OBJ file type does not need to be specified because it is the default):

```
$ LINK HELLO
```

An object module is comprised of a module header, global symbol directory (GSD) records, and text and information relocation commands. Language processors list the symbols defined externally to the module in the GSD. An object module is terminated by an end-of-module (EOM) record. A module can also contain debugger information and traceback information.

The linker processes the entire contents of an object file, that is, every object module in the file. It cannot selectively process object modules within an object file. The linker can process object modules selectively in an object module library (.OLB) file only.

You cannot examine an object module by using a text editor. The only way to examine an object file is by using the ANALYZE/OBJECT utility. This utility produces a report that lists the records that make up the object module. This report is primarily useful to compiler writers. (For information about using the ANALYZE command, see the *OpenVMS DCL Dictionary*.)

1.2.2 Shareable Images as Linker Input Files

A shareable image is the product of a link operation. A shareable image is not directly executable, that is, it cannot be executed by means of the DCL command RUN. To execute, a shareable image must first be included as input in a link operation that produces an executable image. (You can also activate a shareable image dynamically by using the LIB$FIND_IMAGE_SYMBOL routine. For more information, see the *OpenVMS RTL Library (LIB$) Manual*.) When that executable image is run, the shareable image is also activated by the image activator.

A shareable image file consists of an image header, one or more image sections, and a symbol table, which appears at the end of the file. This symbol table is, in fact, an object module whose records contain definitions of **universal** symbols in the shareable image. A universal symbol is to a shareable image what a global symbol is to a module, that is, it is a symbol that can be used to satisfy references in external modules.

Shareable images can provide the following benefits:

- **Reducing total link processing time**. Because the linker needs only to read the image header and to process the global symbol table in a shareable image, it takes less time for the linker to process a shareable image. The linker does not have to resolve symbolic references within the shareable image, sort program sections into image sections, or initialize the image section contents, as it does when processing object modules.
- **Avoiding relinking entire applications**. You can create a shareable image that can be modified, recompiled, and relinked without causing the images that were linked against previous versions of the shareable image to be relinked. This is called **upward compatibility**. For more information about this topic, see Chapter 4.
- **Conserving disk space**. Because many different executable images can be linked against the same shareable image, it is necessary to keep only a single copy of the shareable image on the disk. (Images that are linked with shareable images do not actually contain a copy of the shareable image.)

- **Conserving physical memory**. Because the system can map the shareable pages of an installed shareable image into the address space of many processes, each process does not need to have its own copy of these pages. Note that, to achieve this benefit, the shareable image must be installed using the Install utility, specifying the /SHARED qualifier.
- **Reduction of paging I/O**. Because a page in an installed shareable image may be mapped into the working set of several processes, it is more likely to be in physical memory, reducing paging I/O. Note that, to achieve this benefit, the shareable image must be installed using the Install utility, specifying the /SHARED qualifier.
- **Implementing memory-resident databases**. Because installed shareable images are memory resident, they simplify the implementation of applications, such as data acquisition and control systems, where response times are so critical that control variables and data readings must remain in main memory.

  For example, a shared database may be a named FORTRAN common block built into a shareable image. The shareable image may also include routines to synchronize access to such data. When applications link with the shareable image, they have easy access to the data (and routines).

  Note that, to achieve this benefit, the shareable image must be installed using the Install utility, specifying the /SHARED qualifier. If the shared database is writable, you must also specify the /WRITE qualifier.

1.2.2.1 Including a Shareable Image in a Link Operation

To include a shareable image in a link operation, you must specify the shareable image in an options file, identifying the input file as a shareable image by appending the /SHAREABLE qualifier to the file specification. You cannot specify a shareable image as an input file on the LINK command line. The following example illustrates an options file, named MY_OPTIONS_FILE.OPT, that contains an input file specification of the shareable image (the .EXE file type does not need to be specified because it is the default):

```
MY_SHARE/SHAREABLE
```

The following example illustrates the LINK command in which the options file is specified. (For more information about creating and using shareable images, see Chapter 4.) Note that the default file types for the options file and the object module do not need to be specified.

```
$ LINK MY_MAIN_PROGRAM,MY_OPTIONS_FILE/OPTIONS
```

By default, if you do not specify the device and directory in the file specification, the linker looks for shareable images in your default device and directory.

You link against shareable images in a shareable image library by specifying the library on the LINK command line or in a linker options file, identifying the file as a library by appending the /LIBRARY qualifier to the library file specification. You can include specific shareable images from the library in the link operation by appending the /INCLUDE qualifier to the library file specification, specifying which shareable images you want to include as parameters. (For more information about specifying library files in a link operation, see Section 1.2.3). By default, the linker looks for user library files in the current default directory.

Note that images that link against shareable images do not contain the shareable image but only a reference to it. When the executable image is activated, the image activator activates all the shareable images to which it has been linked. By default, each image maps its own copy of the shareable image's pages.

1.2.2.2 Installing a Shareable Image

If you install the shareable image (using the Install utility), all processes can share the same physical copy of the shareable image in memory. To take advantage of this feature, you must specify the ADD subcommand and the /SHARED qualifier on the INSTALL command line, as in the following example:

```
$ INSTALL ADD/SHARED WORK:[PROGRAMS]MY_SHARE.EXE
```

The system creates a set of global sections for those image sections in the shareable image that can be shared. The system can map these global sections into the address space of multiple processes. For those image sections that are not shareable (image sections with the copy-on-reference [CRF] attribute), each process gets a private copy. (See Chapter 3 for more information about program section and image section attributes.)

If you do not install the shareable image specifying the /SHARED qualifier, each process receives a private copy of the image. (For information about installing images, see the *OpenVMS System Manager's Manual*.)

1.2.3 Library Files as Linker Input Files

A library file is a file produced by the Librarian utility (default file type is .OLB). The linker accepts object module libraries and shareable image libraries as input files.

1.2.3.1 Creating a Library File

You create a library by specifying the /CREATE qualifier with the LIBRARY command. In the following example, the object module MY_PROG.OBJ is inserted into the library MY_LIB.OLB:

```
$ LIBRARY/CREATE/INSERT  MY_LIB MY_PROG
```

A library file contains a library header and a name table. A library name table lists all of the global symbols in all of the modules inserted in the library and associates the name of the symbol with the name of the module in which it is defined.

Object module libraries contain copies of the object module. Shareable image libraries do not contain the actual shareable image or a copy of its global symbol table (GST). Instead, shareable image libraries contain only the name of the shareable image that contains the definition. The linker looks for the shareable image in the device and directory in which the library resides. If the linker cannot find the shareable image at this location, it looks in the directory pointed to by the logical name SYS$LIBRARY for VAX links or ALPHA$LIBRARY for Alpha links.

You cannot examine a library file using a text editor. To find out which modules a library contains, invoke the Librarian utility with the /LIST qualifier. The Librarian utility lists the symbols defined in these modules if you also specify the /NAMES qualifier. In the following example, the library MYMATH_LIB.OLB contains the object module MYMATHROUTS.OBJ, which contains the definitions of the symbols myadd, mysub, mydiv, and mymul:

```
$ LIBRARIAN/LIST/NAMES MYMATH_LIB
Directory of OBJECT library WORK:[PROGS]MYMATH_LIB.OLB;1 on
3-SEP-1993 08:39:27
Creation date:   3-SEP-1993 08:39:05      Creator:  VAX-11 Librarian V04-00
Revision date:   3-SEP-1993 08:39:05      Library format:   3.0
Number of modules:      1                 Max. key length:  31
Other entries:          4                 Preallocated index blocks:     49
Recoverable deleted blocks:      0        Total index blocks used:        2
Max. Number history records:      20      Library history records:        0

Module MYMATHROUTS
MYADD                           MYDIV
MYMUL                           MYSUB
```

For more information about creating and using libraries, see the *OpenVMS Command Definition, Librarian, and Message Utilities Manual*.

1.2.3.2 Including a Library File in a Link Operation

You can specify a library file in a link operation in any of the following ways:

- **Using the /LIBRARY qualifier**. You can specify a library file on the LINK command line or in an options file, identifying the input file as a library by appending the /LIBRARY qualifier.

  When the linker processes a library file, it searches the library's name table for the definitions of symbols referenced in the other input files it has processed previously in the link operation. (Note that the order in which the linker processes a library file can affect symbol resolution. For more information, see Chapter 2.)

  When the linker finds the definition of a symbol in the library's name table, it includes the module that contains the definition in the link operation and processes it as it would any other object module or shareable image. For object module libraries, the linker extracts the object module from the library. The linker looks for the shareable image in the device and directory in which the library resides. If the linker cannot find the shareable image at this location, it looks in the directory pointed to by the logical name SYS$LIBRARY for VAX links or ALPHA$LIBRARY for Alpha links.

- **Using the /INCLUDE qualifier**. You can include specific modules from a library into a link operation by appending the /INCLUDE qualifier to the library file specification. You specify the modules you want included in the link operation as arguments to the qualifier.

  Note, however, that the linker does *not* process the name table of a library file specified using the /INCLUDE qualifier. The linker includes from the library the modules specified as arguments to the /INCLUDE qualifier into the link operation and processes them as it would any other object module or shareable image.

  If you append both the /LIBRARY qualifier and the /INCLUDE qualifier to a library file specification, the linker processes the library's name table and also includes the specified modules in the link operation.

- **Defining the library as a default user library**. You can include a library in a link operation by defining it as a default user library. To define a default user library, assign the name of the library as the value of one of the linker's LNK$LIBRARY logical names. The linker processes libraries pointed to by these logicals after processing all the other input files specified in the link operation. See Section 2.2.3.3 for more information about default library processing.

The example link of the Hello World! program in Section 1.1.2 included the C run-time library in the link operation as a default library.

1.2.4 Symbol Table Files as Linker Input Files

A symbol table file is the product of a previous link operation or a language processor. A symbol table file is an object module that contains only an object module header, a global symbol directory (GSD), and an end-of-module record.

VAX

For VAX linking, you can specify a symbol table file as input in a link operation as you would any other object module, as in the following example:

```
$ LINK MY_MAIN_PROGRAM, MY_SYMBOL_TABLE
```

The linker processes the GSD of the symbol table file during symbol resolution. If the symbol table file was the by-product of a link operation in which an executable image or system image was created, the GSD contains the names and values of every global symbol in the image. If the symbol table file is associated with a shareable image, it contains the names and values of the symbols in the image declared as universal.

Note that, for a symbol table file to be useful in link operations, the values associated with the symbols in the symbol table file must be constants. The value of symbols that represent addresses, such as a procedure entry point, can vary each time the image is activated (unless the image is based).

Note also that a symbol table file associated with a shareable image should not be specified as an input file in a link operation in place of the shareable image. The shareable image itself must be specified as input because the linker requires more information than can be found in a symbol table file, such as the memory requirements of the shareable image (contained in the image header).♦

Alpha

For Alpha linking, symbol table files created by the linker cannot be used as input files in a link operation. A symbol table in an OpenVMS Alpha shareable image does not contain the actual value of a symbol, even for symbols that represent constants. Instead, the symbol table file contains the offset of the symbol's entry in the image's symbol vector.

For example, if the symbol FOO represents the constant 6000, in a VAX image the value of FOO in the symbol table file would be 6000. In an Alpha image, the value of FOO in the symbol table file would not be 6000 but another value that represented the symbol's position in the symbol vector as an offset from the base of the symbol vector, such as 48. This entry in the symbol vector contains the value 6000.

Symbol table files created by the linker during Alpha links can be used as an aid to debugging a module with the System Dump Analyzer utility (SDA).♦

1.2.5 Options Files as Linker Input Files

An options file is a standard text file you must use to specify linker options and shareable images specified as input files. You cannot specify linker options or shareable images on the LINK command line. Linker options, similar to linker qualifiers, allow you to control various aspects of the linker operation. Part II includes reference descriptions of all the options supported by the linker.

In addition, you can use options files to perform the following tasks:

- Specifying frequently used input file specifications

- Entering LINK commands that may exceed the buffer capacity of the command language interpreter (256 characters)

When creating a linker options file, keep in mind the following restrictions:

- Separate input file specifications with a comma (,).
- Do not enter any linker qualifiers except those required to identify input files, such as the /LIBRARY or /SHAREABLE qualifier.
- Do not specify an options file within an options file.
- Enter only one option per line.
- Continue a line by entering the continuation character (the hyphen [-]) at the end of the line.
- Enter comments after an exclamation point (!).
- You may abbreviate the name of a link option to as few letters as needed to make the abbreviation unique.

Example 1–2 illustrates an options file, named PROJECT3.OPT, that contains both input file specifications and linker options.

Example 1–2 Sample Linker Options File

```
MOD1,MOD7,LIB3/LIBRARY,-
LIB4/LIBRARY/INCLUDE=(MODX,MODY,MODZ),-
MOD12/SELECTIVE_SEARCH
STACK=75
SYMBOL=JOBCODE,5
```

To use an options file in a link operation, specify the name of the options file on the command line, identifying the file as an options file by appending the linker qualifier /OPTIONS to the file specification (the .OPT file type does not need to be specified because it is the default), as in the following example:

```
$ LINK PROGA,PROGB,PROJECT3/OPTIONS
```

If you precede the link operation with the SET VERIFY command, DCL displays the contents of the options file as the file is processed.

If you want to use an options file in a command procedure or interactively on the command line, specify the input file as the logical name SYS$INPUT, appending the /OPTIONS qualifier to the logical name. DCL interprets the lines immediately following the LINK command as the contents of the options file. The following example illustrates a LINK command in a command procedure:

```
$ !  LIN command
$ LINK  MAIN,SUB1,SYS$INPUT/OPTIONS
MYPROC/SHAREABLE
SYS$LIBRARY:APPLPCKGE/SHAREABLE
STACK=75
$
```

When you specify SYS$INPUT to create an options file interactively on the command line, you must terminate the options file by entering the Ctrl/Z key sequence, as in the following example:

```
$ LINK MAIN,SUB1,SUB2,SYS$INPUT:/OPTIONS
MYPROC/SHAREABLE
SYS$LIBRARY:APPLPCKGE/SHAREABLE
STACK=75
Ctrl/Z
```

Digital recommends using command procedures to invoke the LINK command because it enables you to keep both the LINK command and all input file specifications, including any options files, together in a single file. To perform a link operation using a command procedure, you simply invoke the command procedure, as in the following example:

```
$ @LINKPROC
```

1.3 Specifying Linker Output Files

The primary output generated by the linker is an image file. In addition, the linker can generate two other output files: a symbol table file and a map file.

Alpha

For Alpha linking only, the linker can generate a debug symbol file.♦

Table 1–2 lists all the output files created by the linker.

Table 1–2 Output Files Generated by the Linker

File	Default File Type	Description
Executable image	.EXE	A program that can be executed at the command line. Specify the /EXECUTABLE qualifier to create one. This is the default output file created by the linker.
Shareable image	.EXE	A program that can be run only after being included in a link operation in which an executable image is created. Specify the /SHAREABLE qualifier to create one.
System image	.EXE	A program that is meant to be run as a standalone system. Specify the /SYSTEM qualifier to create one.
Symbol table file	.STB	An object module containing the global symbol table from an executable or system image, or the universal symbol table from a shareable image. Specify the /SYMBOL_TABLE qualifier to create one.
Map file	.MAP	A text file created by the linker that provides information about the layout of the image and statistics about the link operation. Specify the /MAP qualifier to create one.
‡Debug symbol file	.DSF	A file containing symbol information for use by the OpenVMS Alpha System-Code Debugger. Specify the /DSF qualifier to create one. See *OpenVMS Alpha Device Support: Developer's Guide* for guidelines on using the system-code debugger.

‡Alpha specific

You cannot examine an image file using a text editor. To examine an image file, check for errors in image format, and obtain other information about the image, you must use the ANALYZE/IMAGE utility. See the *OpenVMS DCL Dictionary* for information about using this utility.

The following sections describe each of the output files.

1.3.1 Creating an Executable Image

An executable image is a file that can be executed by the RUN command. An executable image is made up of an image header (which contains image identification information and the image section descriptors [ISDs] that define the memory requirements of the image), a global symbol table, and the executable machine code. An executable image may reference one or more shareable images.

To create an executable image, you can specify the /EXECUTABLE qualifier. Note, however, that the linker creates executable images by default. For example, the command used to create the executable image in Example 1–1 did not specify the /EXECUTABLE qualifier:

```
$ LINK HELLO
```

By default, the linker uses the name of the first input file specified as the name of the image file, giving the file the .EXE file type. However, you can alter this default naming convention. For more information, see the LINK command description in Part II.

1.3.2 Creating a Shareable Image

A shareable image is similar to an executable image except that it cannot be executed from a command line. To run a shareable image, include it in a link operation in which an executable image is created. A shareable image is made up of an image header, a global symbol table, and executable machine code, just as an executable image is.

To create a shareable image, specify the /SHAREABLE qualifier in the LINK command line, as in the following example:

```
$ LINK/SHAREABLE MY_SHARE, MY_UNIVERSALS/OPT
```

Note that the preceding LINK command includes the options file MY_ UNIVERSALS.OPT. To make symbols in the shareable image available for other modules to link against, you must declare them as **universal** symbols in a linker options file. The mechanism used to declare universal symbols is different for VAX linking and Alpha linking. For complete information about creating and using shareable images with examples, see Chapter 4.

1.3.3 Creating a System Image

A system image is an image that does not run under the control of the operating system. It is intended for standalone operation only.

By default, system images do not contain an image header as executable images and shareable images do. You can create a system image with a header by specifying the /HEADER qualifier. System images do not contain global symbol tables.

To create a system image, specify the /SYSTEM qualifier in the LINK command line, as in the following example:

```
$ LINK/SYSTEM MY_SYSTEM_IMAGE
```

1.3.4 Creating a Symbol Table File

A symbol table file is an object module, produced by the linker, that contains all the global symbol definitions in the image. You can create a symbol table for any type of image: executable, shareable, or system. For executable images and system images, the symbol table contains a listing of the global symbols in the image. For shareable images, the symbol table lists the universal symbols in the image.

VAX

For VAX linking, symbol table files can be specified as input files in link operations. For more information, see Section 1.2.4.♦

Alpha

For Alpha linking, the symbol table files created by the linker cannot be used as input files in subsequent link operations. Symbol table files are intended to be used with the System Dump Analyzer utility (SDA) as an aid to debugging.♦

To create a symbol table file, specify the /SYMBOL_TABLE qualifier in the LINK command line. In the following link operation in which an executable image is created, a symbol table file is requested:

```
$ LINK/SYMBOL_TABLE MY_EXECUTABLE_IMAGE
```

By default, the linker uses the name of the first input file specified as the name of the symbol table file, giving the file the .STB file type. However, you can alter this default naming convention. For more information, see the description of the /SYMBOL_TABLE qualifier in Part II.

1.3.5 Creating a Map File

The linker can generate a diagnostic file, called an **image map**, which you can use to locate link-time errors, to study the image layout, and to keep track of global symbols. The image map provides information about the linking process, including the following types of information:

- A listing of the object modules included in the link operation
- A listing of the image sections created by the linker for the image
- A listing of all the program sections created by the linker
- A listing of all the global and universal symbols resolved by the linker for the image
- A compilation of summary statistics about the link operation

To create an image map file, specify the /MAP qualifier on the LINK command line. In batch mode, the linker creates a map file by default. When you invoke the linker interactively (at the DCL command prompt), you must request a map explicitly. By default, the linker uses the name of the first input file specified as the name of the map file, giving the file the .MAP file type. However, you can alter this default naming convention. For more information, see the LINK command description in Part II.

For example, to generate a map file in Example 1–1, you would specify the /MAP qualifier as in the following example:

```
$ LINK/MAP HELLO
```

You can determine the information contained in the image map by specifying additional qualifiers that are related to the /MAP qualifier. For example, by specifying the /BRIEF qualifier with the /MAP qualifier, you can generate a map file that contains only a subset of the total information that can be returned. For

complete information about creating a map file and the contents of a map file, see Chapter 5.

1.3.6 Creating a Debug Symbol File (Alpha Images Only)

Alpha

For Alpha linking, a debug symbol file (DSF) is a file containing debug information for use by the OpenVMS Alpha System-Code Debugger. To create a debug symbol file, specify the /DSF qualifier in the LINK command line, as in the following example:

```
$ LINK/DSF MY_PROJ.OBJ
```

By default, the linker uses the name of the first input file specified as the name of the DSF file, giving the file the .DSF file type. However, you can alter this default naming convention. For more information, see the description of the /DSF qualifier in Part II.♦

1.4 Optimizing the Performance of Alpha Images

Alpha

For Alpha linking, the linker performs certain optimizations by default to improve the performance of the images it creates. In addition, you can improve the performance of installed images by installing them as resident images. The following sections describe these optimizations.

1.4.1 Linker Default Image Optimizations (Alpha Images Only)

On Alpha systems, compilers generate a Jump to Subroutine (JSR) instruction sequence to implement procedure calls. The first instruction in this sequence, a Load Quadword (LDQ) instruction, loads the first quadword of the linkage pair, identified as an offset from the base of the linkage section (LS), into register 26. This quadword contains the code address of the procedure. The second LDQ instruction loads the second quadword of the linkage pair, which contains the address of the routine's procedure descriptor, into register 27. Once the registers have been loaded, the JSR instruction is executed with the contents of register 26 passed as an argument. The following example illustrates the JSR instruction sequence:

```
LDQ R26,x(LS)   ; x is offset into linkage section of the code address
LDQ R27,x+8(LS) ; x+8 = location of procedure descriptor address
JSR R26,R26     ;              in linkage pair
```

On Alpha systems, it is more efficient to execute a procedure call as a branch, using the BSR (Branch to Subroutine) instruction sequence, than it is to execute the call as a jump using the JSR instruction sequence. In a BSR instruction, the destination can be expressed as an offset, requiring fewer memory fetches than a JSR instruction sequence. If you replace the JSR instruction with the BSR instruction, you no longer have to load R26 with the code address.

Compilers cannot always take advantage of the efficiency of the BSR instruction because the information needed to calculate the offset is not available until link time, when the linker lays out the image sections that make up the image.

To take advantage of this performance enhancement, compilers flag each use of the JSR instruction sequence. The linker examines each use of the JSR instruction sequence and attempts to replace it with the BSR instruction sequence wherever possible. You can prevent the linker from performing code replacement by specifying the /NOREPLACE qualifier. For more information about the /REPLACE qualifier, see Part II.

When the linker replaces the JSR instruction with a BSR instruction, it also replaces the first LDQ instruction used to load R26 with a BIS instruction because it no longer needs to load R26 with the code address from the linkage pair. Independent of the JSR replacement, the linker also replaces the second LDQ instruction used to load R27 with the procedure descriptor address with a Load Address (LDA) instruction, if possible. The following example illustrates the BSR instruction sequence that replaces the JSR instruction sequence:

```
BIS R31,R31,R31 ; equivalent to a NOP
LDA R27,x(LS)   ; x is offset from linkage section to procedure descriptor
BSR R26,displ   ; branch
```

When debugging, be aware that instructions you expect to find may have been replaced as follows:

- LDQ replaced with BIS
- LDQ replaced with LDA
- JSR replaced with BSR

In addition to code replacement, the linker can also specify **hints** to improve the performance of the JSR instructions that remain in the image. A hint is used to index the instruction cache and can improve performance. Hints are generated for JSR instructions within the image and for JSR instructions to shareable images.

1.4.2 Installing Images as Resident Images (Alpha Systems Only)

On Alpha systems, another way to improve the performance of an executable image or a shareable image is to install it as a resident image. The Install utility moves certain portions of resident images into a granularity hint region (GHR) in system space where they function as a large single page with granularity hints set, which provides better performance. To create a resident image, specify the /RESIDENT qualifier on the Install utility command line, as in the following example:

```
$ INSTALL ADD/RESIDENT MY_PROG.EXE
```

To create an image that can be installed as a resident image, you must specify the /SECTION_BINDING qualifier in the link operation. When you specify the /SECTION_BINDING qualifier, the linker does not replace JSR instruction sequences with the BSR instruction sequence if it would create a dependency on image section layout. In addition, the linker also checks for data references that would create dependencies on the layout of image sections. When it creates an image that can be installed as a resident image, the linker sets a flag in the image header.

For more information, see the description of the /SECTION_BINDING qualifier in Part II.♦

1.5 Controlling a Link Operation

The linker allows you to control various aspects of the link operation by specifying qualifiers and options. The following sections summarize the qualifiers and options supported by the linker. The remaining chapters of this manual describe how to use these qualifiers and options, and Part II provides reference information about each linker qualifier and option.

1.5.1 Linker Qualifiers

As with any DCL command, the LINK command supports qualifiers that allow you to control aspects of linker processing. The qualifiers supported by the linker allow you to:

- **Identify input files**. For example, you must identify library files by appending the /LIBRARY qualifier to the file specification. Section 1.2 describes these qualifiers.
- **Specify output files**. For example, you must specify the /SHAREABLE qualifier to direct the linker to create a shareable image. Section 1.3 describes these qualifiers.
- **Control symbol resolution**. For example, if you specify the /NOSYSLIB qualifier, the linker will not process the default system object library or the default system image library. Chapter 2 contains more information about this topic.
- **Control image file creation**. For example, if you specify the /CONTIGUOUS qualifier, the linker attempts to allocate contiguous disk blocks for the image file. Chapter 3 contains more information about this topic.

Table 1–3 lists the LINK command qualifiers alphabetically.

Table 1–3 Linker Qualifiers

Qualifier	Description
/ALPHA	Directs the linker to build an OpenVMS Alpha image. Section 1.6 describes this qualifier in more detail.
/BPAGE	Specifies the page size the linker should use when creating image sections.
/BRIEF	Directs the linker to create a brief image map. Must be specified with the /MAP qualifier.
/CONTIGUOUS	Directs the linker to attempt to store the output image in contiguous disk blocks.
/CROSS_REFERENCE	Directs the linker to replace the Symbols By Name section of the image map with the Symbol Cross-Reference section. Must be specified with the /MAP qualifier.
/DEBUG	Directs the linker to generate a debugger symbol table and to give control to the OpenVMS Debugger when the image is run.
‡/DEMAND_ZERO	Controls how the linker creates demand-zero image sections in Alpha images. Not supported for VAX linking.
‡/DSF	Directs the linker to create a file called a debug symbol file (DSF) for use by the OpenVMS Alpha System-Code Debugger.
/EXECUTABLE	Directs the linker to create an executable image.
/FULL	Directs the linker to create a full image map. Used only with the /MAP qualifier.

‡Alpha specific

(continued on next page)

Table 1–3 (Cont.) Linker Qualifiers

Qualifier	Description
‡/GST	Directs the linker to include symbols that have been declared universal in the global symbol table (GST) of a shareable image. Not supported for VAX linking.
/HEADER	Directs the linker to include an image header in a system image. Used only with the /SYSTEM qualifier.
/INCLUDE	Identifies the input file to which it is appended as a library file and directs the linker to include specific modules from the library in the link operation.
/LIBRARY	Identifies the input file to which it is appended as a library file.
/MAP	Directs the linker to create an image map.
‡/NATIVE_ONLY	Directs the linker to create an Alpha image that cannot operate with a translated VAX image. Not supported for VAX linking.
/OPTIONS	Identifies an input file as a linker options file.
/P0IMAGE	Directs the linker to mark the specified executable image as one that can run only in P0 address space.
/PROTECT	Directs the linker to protect the shareable image from user-mode and supervisor-mode write access. Used with the /SHAREABLE qualifier when the linker creates a shareable image.
‡/REPLACE	Directs the linker to perform certain optimizations that improve the performance of the resultant image. Not supported for VAX linking.
‡/SECTION_BINDING	Directs the linker to check if the image contains dependencies on the layout of image sections that could interfere with a performance enhancement the Install utility can perform on images that are installed as resident images. Not supported for VAX linking.
/SELECTIVE_SEARCH	Directs the linker to include in the image's global symbol table (GST) only those global symbols that are defined in the file and referenced by previously processed files.
/SHAREABLE	Directs the linker to create a shareable image. It can also be used to identify an input file as a shareable image.
/SYMBOL_TABLE	Directs the linker to create a symbol table file.
‡/SYSEXE	Directs the linker to process the OpenVMS Alpha executive file SYS$BASE_IMAGE.EXE (located in the directory pointed to by the logical name ALPHA$LOADABLE_IMAGES) to resolve references to symbols in a link operation. Not supported for VAX images.
/SYSLIB	Directs the linker to search the default system image library and the default system object library to resolve undefined symbolic references.
/SYSSHR	Directs the linker to search the default system shareable image library to resolve undefined symbolic references.

‡Alpha specific

(continued on next page)

Table 1–3 (Cont.) Linker Qualifiers

Qualifier	Description
/SYSTEM	Directs the linker to create a system image.
/TRACEBACK	Directs the linker to include traceback information in the image.
/USERLIBRARY	Directs the linker to search default user libraries to resolve undefined symbolic references. /USERLIBRARY accepts a keyword (ALL, GROUP, PROCESS, SYSTEM, or NONE) to further specify which logical name tables to search for the definitions of default user libraries.
/VAX	Directs the linker to build an OpenVMS VAX image. Section 1.6 describes this qualifier in more detail.

1.5.2 Link Options

In addition to qualifiers, the linker supports options that allow you to control other aspects of a link operation, such as the following:

- **Specify image identification information**. Using options such as NAME=, ID=, and GSMATCH=, you can supply values for fields in the image header.
- **Declare universal symbols in shareable images**. Using the UNIVERSAL= option for VAX linking and the SYMBOL_VECTOR= option for Alpha linking, you can make symbols in shareable images accessible to external modules.
- **Group input files together**. Using the CLUSTER= option or the COLLECT= option, you can specify which input files (or program sections in those input files) the linker should group together. This can affect symbol resolution.

Note that linker options must be specified in a linker options file. (See Section 1.2.5 for information about creating linker options files and specifying them in link operations.)

Table 1–4 lists all the linker options alphabetically.

Table 1–4 Linker Options

Option	Description
†BASE=	Sets the base virtual address for the image. Not supported for Alpha linking.
CASE_SENSITIVE=	Determines whether the linker preserves the mixture of uppercase and lowercase characters used in arguments to linker options.
CLUSTER=	Directs the linker to create a cluster, assign the cluster the name specified, and insert the input files specified in the cluster.
COLLECT=	Moves the specified program sections into the specified cluster.

†VAX specific

(continued on next page)

Table 1–4 (Cont.) Linker Options

Option	Description
DZRO_MIN=	Sets the minimum number of uninitialized, contiguous pages that must be found in an image section before the linker can extract the pages from the image section and create a demand-zero image section.
GSMATCH=	Sets match control parameters for a shareable image.
IDENTIFICATION=	Sets the image ID field in the image header.
IOSEGMENT=	Specifies the size of the image I/O segment.
ISD_MAX=	Specifies the maximum number of image sections.
NAME=	Sets the image name field in the image header.
PROTECT=	Directs the linker to protect one or more clusters from user-mode or supervisor-mode write access. Can be used only with shareable images.
PSECT_ATTR=	Assigns values to program section attributes.
STACK=	Sets the initial size of the user-mode stack.
SYMBOL=	Defines a global symbol and assigns it a value.
‡SYMBOL_TABLE=	Specifies whether a symbol table file, produced in a link operation in which a shareable image is created, should contain all the global symbols as well as the universal symbols in the shareable image. By default, the linker includes only universal symbols. Not supported for VAX images.
‡SYMBOL_VECTOR=	Declares a symbol in a shareable image as universal, making it accessible to external modules. Not supported for VAX images.
†UNIVERSAL=	Declares the specified global symbol as a universal symbol, making it accessible to external modules. Not supported for Alpha images.

†VAX specific
‡Alpha specific

1.6 Linking for Different Architectures

It is possible to create OpenVMS Alpha images on an OpenVMS VAX system and to create OpenVMS VAX images on an OpenVMS Alpha system. To do this, you must mount a system disk of the target architecture and make it accessible on the system where the link is to occur. Also, you must assign several logical names to point to portions of the target architecture disk.

Table 1–5 lists the logical names and the conditions of their use.

Table 1–5 Logical Names for Cross-Architecture Linking

Logical Name	Description
ALPHA$LIBRARY	The linker uses this logical name when creating an OpenVMS Alpha image to locate the target system's shareable images and system libraries.

(continued on next page)

Table 1–5 (Cont.) Logical Names for Cross-Architecture Linking

Logical Name	Description
VAX$LIBRARY	The linker uses this logical name when creating an OpenVMS VAX image on an OpenVMS Alpha computer to locate the target system's shareable images and system libraries.
SYS$LIBRARY	The linker uses this logical name when creating an OpenVMS VAX image on an OpenVMS VAX computer to locate the target system's shareable images and system libraries.
ALPHA$LOADABLE_ IMAGES	The linker uses this logical when creating an OpenVMS Alpha image to locate the target system's base image SYS$BASE_ IMAGE.EXE when the /SYSEXE qualifier is in the link command line.

The /ALPHA and /VAX qualifiers control which architecture an image is built for:

- When you specify /ALPHA, the linker creates an OpenVMS Alpha image using the OpenVMS Alpha libraries and OpenVMS Alpha images from the target system disk that the logicals ALPHA$LIBRARY and ALPHA$LOADABLE_IMAGES point to. When you link on an OpenVMS Alpha system, these logical names initially point to the current system's libraries and images. The qualifier /ALPHA is the default on OpenVMS Alpha systems.
- When you specify /VAX on an OpenVMS Alpha system, the linker creates an OpenVMS VAX image using the OpenVMS VAX libraries and OpenVMS VAX images from the target system disk that the logical VAX$LIBRARY points to. On an OpenVMS VAX system, you create VAX images by using the OpenVMS VAX libraries and OpenVMS VAX images that the logical SYS$LIBRARY points to. The qualifier /VAX is the default on OpenVMS VAX systems.

2
Understanding Symbol Resolution

As one of its primary tasks, the linker must resolve symbolic references between modules. This chapter describes how the linker performs symbol resolution and how you can control it to ensure that the linker resolves symbolic references as you intend.

2.1 Overview

Programs are typically made up of many interdependent modules. For example, one module may define a symbol to represent a program location or data element that is referenced by many other modules. The linker is responsible for finding the correct definition of each symbol referenced in all the modules included in the link operation. This process of matching symbolic references with their definitions is called **symbol resolution**.

2.1.1 Types of Symbols

Symbols can be categorized by their scope, that is, the range of modules over which they are intended to be visible. Some symbols, called **local symbols**, are meant to be visible only within a single module. Because the definition and the references to these symbols must be confined to a single module, language processors such as compilers can resolve these references.

Other symbols, called **global symbols**, are meant to be visible to external modules. A module can reference a global symbol that is defined in another module. Because the value of the symbol is not available to the compiler processing the source file, it cannot resolve the symbolic reference. Instead, a compiler creates a global symbol directory (GSD) in an object module that lists all of the global symbol references and global symbol definitions it contains.

In shareable images, symbols that are intended to be visible to external modules are called **universal symbols**. A universal symbol in a shareable image is the equivalent of a global symbol in an object module. Note, however, that only those global symbols that have been declared as universal are listed in the global symbol table (GST) of the shareable image and are available to external modules to link against.

Language processors determine whether a symbol is local or global. For example, in VAX FORTRAN, statement numbers are local symbols and module entry points are global symbols. In other languages, you can explicitly specify whether a symbol is local or global by including or excluding particular attributes in the symbol definition. Note also that some languages allow you to specify symbols as **weak** or **strong** (see Section 2.5 for more information).

You must explicitly declare universal symbols as part of the link operation in which the shareable image is created. For more information about declaring universal symbols, see Chapter 4.

Note

In some Digital programming languages, certain types of global symbols, such as external variables in C and COMMON data in FORTRAN, are not listed in the GSD as global symbol references or definitions. Because these data types implement virtual memory that is shared, the languages implement them as program sections that are overlaid. These symbols appear as program section definitions in the GSD, not as a symbol definition or reference. (Compilers use program sections to define the memory requirements of an object module.) The linker does not include program section definitions in its symbol resolution processing. For information about how the linker processes program sections, see Chapter 3.

VAX

On VAX systems, the VAX C language extensions globalref and globaldef allow you to create external variables that appear as symbol references and definitions in the GSD. For more information, see the VAX C documentation. ♦

Alpha

On Alpha systems, the DEC C compiler supports the globalref and globaldef language extensions. In addition, DEC C supports command line qualifiers and source code pragma statements that allow you to control whether it implements external variables as program sections or as global symbol references and definitions. For more information, see the DEC C documentation. ♦

2.1.2 Linker Symbol Resolution Processing

During its first pass through the input files specified in the link operation, the linker attempts to find the definition for every symbol referenced in the input files. By default, the linker processes all the global symbols defined and referenced in the GSD of each object module and all the universal symbols defined and referenced in the GST of each shareable image. The definition of the symbol provides the value of the symbol. The linker substitutes this value for each instance where the symbol is referenced in the image.

The value of a symbol depends on what the symbol represents. A symbol can represent a routine entry point or a data location within an image. For these symbols, the value of the symbol is an address. A symbol can also represent a data constant (for example, X = 10). In this case, the value of the symbol is its actual value (in the example, the value of X is 10).

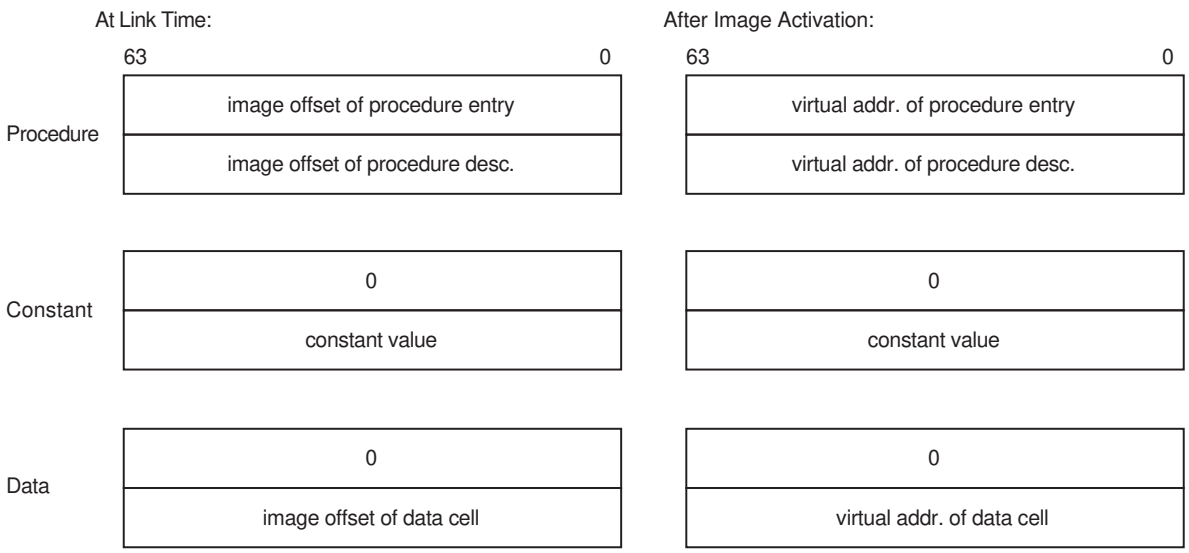
For symbols that represent addresses in object modules, the value is expressed initially as an offset into a program section. This is how language processors express addresses. Later in its processing, when the linker combines the program sections contributed by all the object modules into the image sections that define the virtual memory layout of the image, it determines the actual value of the address. (For information about how the linker determines the virtual memory layout of an image, see Chapter 3.)

For symbols that represent addresses in a shareable image, the value of the symbol at link time is architecture specific.

Alpha

For Alpha images, at link time, the value of a symbol in a shareable image (as listed in the GST of the image) is the offset of the symbol's entry in the symbol vector of the image. A symbol vector entry is a pair of quadwords that contain information about the symbol. The contents of these quadwords depend on whether the symbol represents a procedure entry point, data location, or a constant. Figure 2–1 illustrates the contents of a symbol vector entry for each of these three types of symbols. Note that, at link time, a symbol vector entry for a procedure entry point or a data location is expressed as an offset into the image. At image activation time, when the image is loaded into memory and the base address of the image is known, the image activator converts the image offset into a virtual address. Figure 2–1 shows the contents of the symbol vector at link time and at image activation time.

Figure 2–1 Symbol Vector Contents



Note that the linker does not allow programs to make procedure calls to symbols that represent data locations. ♦

VAX

For VAX images, at link time, the value of a symbol in a shareable image (as listed in the GST of the image) is the offset into the image of the routine or data location, if the symbol was declared universal using the UNIVERSAL= option. If the symbol was declared universal using a transfer vector, the value of the symbol is the offset into the image of the transfer vector entry. If the symbol represents a constant, the GST contains the actual value of the constant. ♦

The actual value of an address symbol in a shareable image is determined *at run time* by the image activator when it loads the shareable image into memory. The image activator **relocates** all the address references within a shareable image when it loads the image into memory. Once it has determined the absolute values of these addresses, the image activator **fixes up** references to these addresses in the image that linked against the shareable image. Previously, the linker created **fix-ups** that flag to the image activator where it must insert the actual addresses to complete the linkage of a symbolic reference to its definition in an image. The linker listed these fix-ups in the **fix-up section** it creates for the image. (For more information about shareable images, see Chapter 4.)

Understanding Symbol Resolution
2.1 Overview

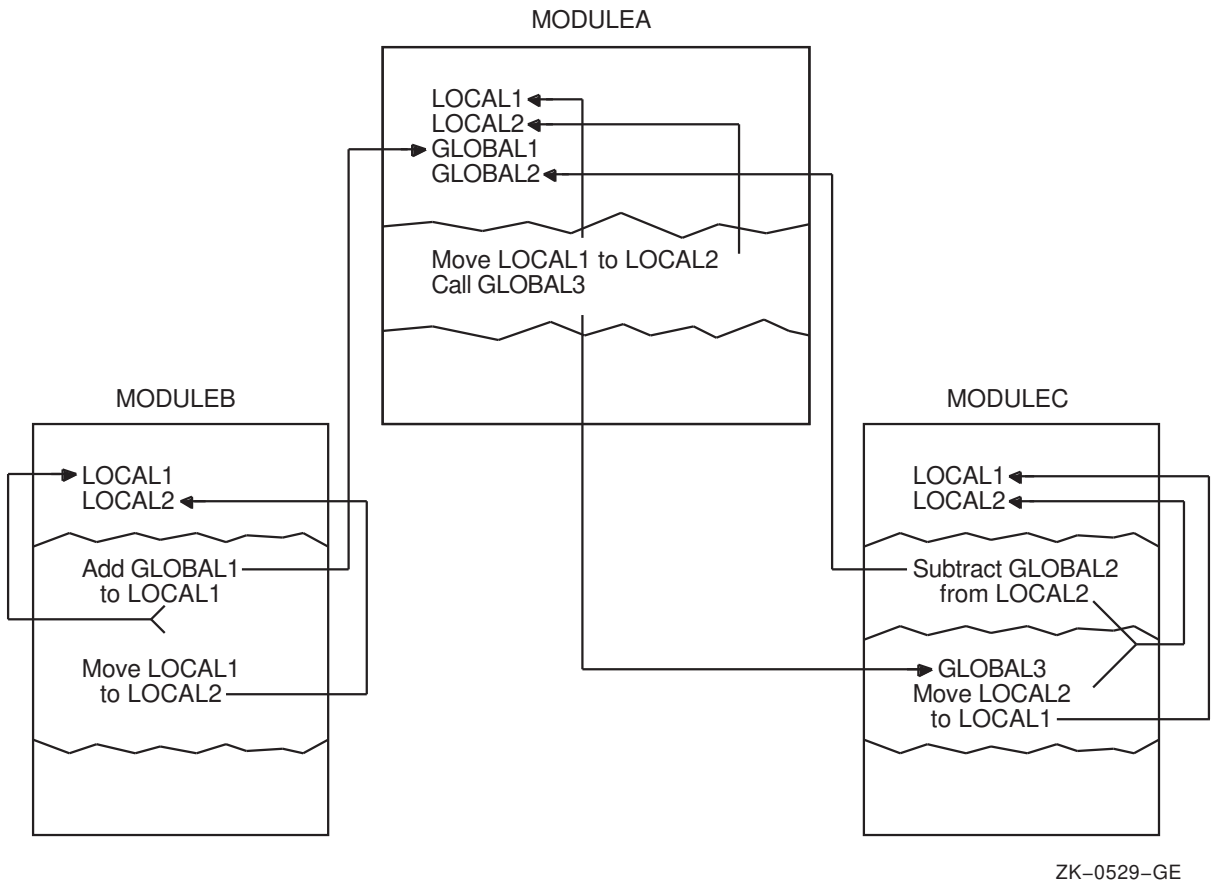
VAX

For VAX images, you can specify the address at which you want a shareable image loaded into memory by using the BASE= option. When you specify this option, the linker can calculate the absolute addresses of symbols within the shareable image because the base address of the shareable image is known. By specifying a base address, you eliminate the need for the image activator to perform fix-ups and relocations.

Note, however, that basing a shareable image can potentially destroy upward compatibility between the shareable image and other images that were linked against it. ♦

Figure 2–2 illustrates the interdependencies created by symbolic references among the modules that make up an application. In the figure, arrows point from a symbol reference to a symbol definition. (The statements do not reflect a specific programming language.)

Figure 2–2 Symbol Resolution



The linker creates an image even if it cannot find a definition for every symbol referenced in the input files it processes. The linker reports these undefined symbols as in the following example, if at least one of these unresolved references is a strong reference. (For information about strong and weak symbolic references, see Section 2.5.) The linker includes the message in the map file, if a map file was requested.

```
$ link  my_main  ! The module MY_MATH is omitted
%LINK-W-NUDFSYMS, 1 undefined symbols:
❶ %LINK-I-UDFSYM,         MYSUB
❷ %LINK-W-USEUNDEF, undefined symbol MYSUB referenced
        in psect $CODE offset %X0000001A
        in module MY_MAIN file WORK:[PROGRAMS]MY_MAIN.OBJ;1
```

❶ The linker issues an informational message for each symbol for which it cannot find a definition.

❷ The linker issues a warning message for each instance where an undefined symbol is referenced in the image.

If you run an image that contains undefined symbols and the symbols are never accessed, the program will run successfully. If you run an image that contains undefined symbols and the image accesses the symbols at run time, the image will abort, in most cases, with an access violation because the linker assigns the value zero to undefined symbols, as in the following example:

```
$ run my_main
%SYSTEM-F-ACCVIO, access violation, reason mask=00, virtual address=00000000,
PC=00001018, PSL=03C00000
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name     routine name                 line   rel PC    abs PC

MY_MAIN          main                          15   00000018  00001018
```

2.2 Input File Processing for Symbol Resolution

The linker can include object modules, shareable images, and libraries in its symbol resolution processing. For VAX images, the linker can also include a symbol table file in its symbol resolution processing. (Options files, in which linker options and input files are specified, are not included in symbol resolution.)

By default, when the linker processes an object module or shareable image, it includes all the symbol definitions from the object module or shareable image in its processing. However, if you append the /SELECTIVE_SEARCH qualifier to the object module or shareable image file specification, the linker includes in its processing only those symbols from the object module or shareable image that define symbols referenced in a previously processed input file. (For more information about selectively processing input files, see Section 2.2.4.)

Table 2–1 summarizes how the linker processes these different types of input files when performing symbol resolution. The following sections provide more detail on the linker's processing of each type of input file.

Table 2–1 Linker Input File Processing

Input File	How Processed
Object file (.OBJ)	By default, the linker processes all the symbol definitions and references listed in the GSD of the module. If you append the /SELECTIVE_SEARCH qualifier to the input file specification, the linker includes in its processing only those symbol definitions from the GSD that resolve symbolic references found in previously processed input files.
Shareable image file (.EXE)	By default, the linker processes all symbol definitions and references listed in the GST of the image. Note, however, to avoid cluttering the map file of the resultant image, the linker lists only those symbol definitions in the map file that are referenced by other modules. If you append the /SELECTIVE_SEARCH qualifier to the input file specification, the linker includes in its processing only those symbol definitions from the GST that resolve symbolic references found in previously processed input files.
†Symbol table file (.STB)	By default, the linker processes all the symbol definitions and references in the GSD of the module. If you append the /SELECTIVE_SEARCH qualifier to the input file specification, the linker includes in its processing only those symbol definitions from the module that resolve symbolic references found in previously processed input files.
Library files (.OLB)	The linker searches the name table of the library for symbols that are undefined in previously processed input files. (A library file's name table lists all the symbols available in all of the modules it contains.) If the linker finds the definition of a symbol referenced by a previously processed input file, it includes in the link operation the module in the library that contains the definition of the symbol. Once the object module or shareable image is included in the link operation, the linker processes it as any other object module or shareable image. If you append the /INCLUDE qualifier to a library file specification, the linker does *not* search the library's name table to find undefined symbolic references. Instead, the linker simply includes the specified object module or shareable image specified as a parameter to the /INCLUDE qualifier. You cannot process a library file selectively. However, if the Librarian utility's /SELECTIVE_SEARCH qualifier was specified when the object module or shareable image was inserted into the library, the linker will process the module selectively when it extracts it from the library.

†VAX specific

2.2.1 Processing Object Modules

The way the linker processes object modules to resolve symbolic references illustrates how the linker processes most other input files. (Symbol table files are object modules. The GST of a shareable image, which the linker processes in symbol resolution, is also created as an object module appended to the shareable image.)

For example, the program in Example 2–1 references the symbol mysub, which is defined in the program in Example 2–2.

Example 2–1 Module Containing a Symbolic Reference (my_main.c)

```
#include <stdio.h>

int mysub();

main()
{
   int num1, num2, result;

   num1 = 5;
   num2 = 6;
   result = 0;

   result = mysub( num1, num2 );
   printf("Result is: %d\n", result);
}
```

Example 2–2 Module Containing a Symbol Definition (my_math.c)

```
myadd(value_1,value_2)
   int value_1;
   int value_2;
{
   int result;

   result = value_1 + value_2;

   return( result)
}

mysub(value_1,value_2)
   int value_1;
   int value_2;
{
   int result;

   result = value_1 - value_2;

   return( result)
}

mymul(value_1,value_2)
   int value_1;
   int value_2;
{
   int result;

   result = value_1 * value_2;

   return( result)
}

mydiv(value_1,value_2)
   int value_1;
   int value_2;
{
   int result;

   result = value_1 / value_2;

   return( result)
}
```

The GSD created by the language processor for the object module MY_MAIN.OBJ lists the *reference* to the symbol mysub. Because object modules cannot be examined using a text editor, the following representation of the GSD is taken from the output of the ANALYZE/OBJECT utility. The example is from the

analysis of an OpenVMS Alpha object module. Differences between the format of the symbol reference between VAX object files and Alpha object files are highlighted in the list following the example.

```
4.  GLOBAL SYMBOL DIRECTORY (EOBJ$C_GSD) ❶, 344 bytes
                .
                .
                .
        9)  Global Symbol Specification (EGSD$C_SYM) ❷
                data type: DSC$K_DTYPE_Z (0)
                symbol flags:
                        (0)  EGSY$V_WEAK      0
                        (1)  EGSY$V_DEF       0
                        (2)  EGSY$V_UNI       0
                        (3)  EGSY$V_REL       0
                        (4)  EGSY$V_COMM      0
                        (5)  EGSY$V_VECEP     0 ❸
                        (6)  EGSY$V_NORM      0 ❹
                symbol: "MYSUB"
```

❶ For VAX object files, the symbol for the global symbol directory is OBJ$C_GSD.

❷ For VAX object files, the symbol for a global symbol specification is GSD$C_SYM.

❸ For VAX object files, this field is not included.

❹ For VAX object files, this field is not included. For Alpha object files, the value of this field is always zero for symbolic *references*.

The GSD created by the language processor for the object module MY_MATH.OBJ contains the *definition* of the symbol mysub, as well as the other symbols defined in the module. The definition of the symbol includes the value of the symbol.

Alpha

The following excerpt from an analysis of the OpenVMS Alpha object module (performed using the ANALYZE/OBJECT utility) shows the format of a GSD symbol definition entry. Note that, in an OpenVMS Alpha object module, a symbol definition is listed as a Global Symbol Specification.

```
4.  GLOBAL SYMBOL DIRECTORY (EOBJ$C_GSD), 46 bytes
        .
        .
        .
        9)  Global Symbol Specification (EGSD$C_SYM)
                data type: DSC$K_DTYPE_Z (0)
                symbol flags:
                        (0)  EGSY$V_WEAK      0
                        (1)  EGSY$V_DEF       1
                        (2)  EGSY$V_UNI       0
                        (3)  EGSY$V_REL       1
                        (4)  EGSY$V_COMM      0
                        (5)  EGSY$V_VECEP     0
                        (6)  EGSY$V_NORM      1 ❶
           ❷    psect: 3
           ❸    value: 64 (%X'00000040')
           ❹    code address psect: 5
           ❺    code address: 8 (%X'00000008')
                symbol: "MYSUB"
        .
        .
        .
```

❶ The value of the EGSY$V_NORM flag is 1 if the symbol represents a

procedure. The value is set to zero if the symbol represents data.

❷ The index of the program section that contains the procedure descriptor for mysub.

❸ The location of the procedure descriptor expressed as the offset from the starting address of the program section that contains the procedure descriptor.

❹ Index of program section that contains the code entry point.

❺ The location of the code entry point, expressed as the offset from the starting address of the program section that contains the entry point. ♦

VAX

The following excerpt from an analysis of the OpenVMS VAX object module (performed using the ANALYZE/OBJECT utility) shows the format of a GSD symbol definition entry. Note that, on VAX systems, a symbol definition is listed as an Entry Point Symbol and Mask Definition record.

```
4.  GLOBAL SYMBOL DIRECTORY (OBJ$C_GSD), 46 bytes
        .
        .
        .
        2)  Entry Point Symbol and Mask Definition (GSD$C_EPM)
                data type: DSC$K_DTYPE_Z (0)
                symbol flags:
                        (0)  GSY$V_WEAK       0
                        (1)  GSY$V_DEF        1
                        (2)  GSY$V_UNI        0
                        (3)  GSY$V_REL        1
                        (4)  GSY$V_COMM       0
                psect: 0
                value: 0 (%X'0000000C')
                entry mask: <>
                symbol: "MYSUB"
        .
        .
        .
```

The value of the symbol is expressed as an offset into a program section. ♦

When you link the modules shown in Example 2–1 and Example 2–2 together to create an image, you specify both object modules on the command line, as in the following example:

```
$ LINK MY_MAIN, MY_MATH
```

When the linker processes these object modules, it reads the contents of the GSDs, obtaining the value of the symbol from the symbol definition.

Alpha

Note that, for Alpha images, in the map file associated with the image, the value of the symbol mysub is the location within the image of the procedure descriptor for the routine. The procedure descriptor contains the address of the routine within the image. ♦

VAX

For VAX images, the value of the symbol mysub is represented in the map file as the location of the entry point mask. ♦

2.2.2 Processing Shareable Images

When the linker processes a shareable image specified as input in a link operation, it processes all the symbol definitions and references in the GST of the image. The GST contains all the universal symbols defined in the shareable image. Because the linker creates the GST of a shareable image in the format of an object module, the processing of shareable images for symbol resolution is similar to the processing of object modules. Note that the linker includes in the map file only those symbols that resolve references to avoid cluttering the listing with extraneous symbols.

For example, the program in Example 2–2 (in Section 2.2.1) can be implemented as a shareable image. (For information about creating a shareable image, see Chapter 4.) The shareable image can be included in the link operation as in the following example:

```
$ LINK/MAP/FULL   MY_MAIN, SYS$INPUT/OPT
MY_MATH/SHAREABLE
```

The GST created by the linker for the shareable image MY_MATH.EXE contains the *definition* of the symbol mysub, as well as the other symbols defined in the module.

Because images cannot be examined using a text editor, the following representations of the GST are taken from the output of the ANALYZE/IMAGE utility.

Alpha

For Alpha images, the universal symbol mysub in the shareable image MY_MATH.EXE appears in the GST of the image as a Universal Symbol Specification record, as illustrated in the following example:

```
SHAREABLE IMAGE - GLOBAL SYMBOL TABLE
            .
            .
            .
4.  GLOBAL SYMBOL DIRECTORY (EOBJ$C_EGSD), 200 bytes
            .
            .
            .
        3) Universal Symbol Specification (EGSD$C_SYMG)
                data type: DSC$K_DTYPE_Z (0)
                symbol flags:
                        (0)  EGSY$V_WEAK      0
                        (1)  EGSY$V_DEF       1
                        (2)  EGSY$V_UNI       1
                        (3)  EGSY$V_REL       1
                        (4)  EGSY$V_COMM      0
                        (5)  EGSY$V_VECEP     0
                        (6)  EGSY$V_NORM      1
                psect: 0
                value: 16 (%X'00000010')
                symbol vector entry (procedure)
                        %X'00000000 00010008'
                        %X'00000000 00000040'
                symbol: "MYSUB"
            .
            .
            .
```

Note that the value of the symbol, as it appears in the Universal Symbol Specification, is the location of the symbol's entry in the image's symbol vector, expressed as an offset from the base of the symbol vector. The symbol vector entry contains the address of mysub's entry point and the address of its procedure descriptor. These locations are expressed as offsets from the base of the image. The entry for a symbol in the GST of an image is a duplicate of the symbol's entry in the symbol vector. ♦

VAX

For VAX images, the universal symbol mysub in the shareable image MY_ MATH.EXE appears in the GST of the image as an Entry Point Symbol and Mask Definition record, as illustrated in the following example:

```
SHAREABLE IMAGE - GLOBAL SYMBOL TABLE
            .
            .
            .
        2)  Entry Point Symbol and Mask Definition (GSD$C_EPM)
                data type: DSC$K_DTYPE_Z (0)
                symbol flags:
                        (0)  GSY$V_WEAK       0
                        (1)  GSY$V_DEF        1
                        (2)  GSY$V_UNI        1
                        (3)  GSY$V_REL        1
                        (4)  GSY$V_COMM       0
                psect: 0
                value: 8 (%X'00000008')
                entry mask: <>
                symbol: "MYSUB"
            .
            .
            .
```

Note that the flag GSY$V_UNI is set for universal symbols to distinguish them from global symbols in object modules that use the same record format. ♦

Implicit Processing of Shareable Images

VAX

For VAX linking, when you specify a shareable image in a link operation, the linker not only processes the shareable image you specify, but also all the shareable images that the shareable image has been linked against. (A shareable image contains a global image section descriptor [GISD] for each shareable image to which it has been linked.) ♦

Alpha

For Alpha linking, the linker does not process the shareable images that the shareable image you specify has been linked against. (Shareable images on Alpha systems still contain GISDs for each shareable image that they have been linked against, however.) If your application's build procedure depends on implicit processing of shareable images, you may need to add these shareable images to your linker options file. ♦

2.2.3 Processing Library Files

Libraries specified as input files in link operations contain either object modules or shareable images. The way in which the linker processes library files depends on how you specify the library in the link operation. Section 2.2.3.1, Section 2.2.3.2, and Section 2.2.3.3 describe these differences. Note, however, that once an object module or shareable image is included from the library into the link operation, the linker processes the file as it would any other object module or shareable image.

For example, to create a library and insert the object module version of the program in Example 2–2 into the library, you could specify the following command:

```
$ LIBRARY/CREATE/INSERT MYMATH_LIB MY_MATH
```

The librarian includes the module in its module list and all of the global symbols defined in the module in its name table. To view the library's module list and name table, specify the LIBRARY command with the /LIST and /NAMES qualifiers, as in the following example:

```
$ LIBRARY/LIST/NAMES MYMATH_LIB
Directory of OBJECT library WORK:[PROGS]MYMATH_LIB.OLB;1 on
28-AUG-1994 11:11:33
Creation date:  28-AUG-1994 11:08:04      Creator:  VAX-11 Librarian V04-00
Revision date:  28-AUG-1994 11:08:04      Library format:   3.0
Number of modules:      1                 Max. key length:  31
Other entries:          5                 Preallocated index blocks:     49
Recoverable deleted blocks:      0        Total index blocks used:        2
Max. Number history records:      20      Library history records:        0

Module MY_MATH
MYADD                           MYDIV
MYMUL                           MYSUB
MY_SYMBOL
```

You can specify the library in the link operation using the following command:

```
$ LINK/MAP/FULL   MY_MATH, MYMATH_LIB/LIBRARY
```

The map file produced by the link operation verifies that the object module MY_MATH.OBJ was included in the link operation.

2.2.3.1 Identifying Library Files Using the /LIBRARY Qualifier

When the linker processes a library file identified by the /LIBRARY qualifier, the linker processes the library's name table, looking for the definitions of symbols referenced in previously processed input files.

Note that, to resolve a reference to a symbol defined in a library, the linker must process the module that references the symbol *before* processing the library file. Thus, while the ordering of object modules and shareable images is not usually important in a link operation, the ordering of library files can be important. (For more information about controlling the order in which the linker processes input files, see Section 2.3.)

Once the object module or shareable image is included from the library into the link operation, the linker processes all the symbol definitions and references in the module. If you want the linker to selectively process object modules or shareable images that are included in the link operation from a library, you must append the Librarian utility's /SELECTIVE_SEARCH qualifier to the file specification of the object module or shareable image when you insert it into the library. Appending the linker's /SELECTIVE_SEARCH qualifier to a library file specification in a link operation is illegal. For more information about processing input files selectively, see Section 2.2.4.

Processing Object Module Libraries

When the linker finds a symbol in the name table of an object module library, it extracts from the library the object module that contains the definition and includes it in the link operation. The linker then processes the GSD of the object module extracted from the library, adding an entry to the linker's list of symbol definitions for every symbol defined in the object module, and adding entries to the linker's undefined symbol list for all the symbols referenced by the module (as described in Section 2.2.1). The linker continues to process the undefined symbol list until there are no definitions in the library for any outstanding references. When the linker finishes processing the library, it has extracted all the modules that resolve references generated by modules previously extracted from the library.

Processing Shareable Image Libraries

When the linker finds a symbol in the name table of a shareable image library, it notes which shareable image contains the symbol and then looks for the shareable image to include it in the link operation. By default, the linker looks for the shareable image in the same device and directory as the library file.

VAX

For VAX linking, if the linker cannot find the shareable image in the device and directory of the library file, the linker looks for the shareable image in the directory pointed to by the logical name SYS$LIBRARY. ♦

Alpha

For Alpha linking, if the linker cannot find the shareable image in the device and directory of the library file, the linker looks for the shareable image in the directory pointed to by the logical name ALPHA$LIBRARY. ♦

Once it locates the shareable image, the linker processes the shareable image as it does any other shareable image (as described in Section 2.2.2).

2.2.3.2 Including Specific Modules from a Library Using the /INCLUDE Qualifier

If the library file is specified with the /INCLUDE qualifier, the linker does *not* process the library's name table. Instead, the linker includes in the link operation the modules from the library specified in the /INCLUDE qualifier and processes them as it would any other object module or shareable image.

If you append both the /LIBRARY qualifier and the /INCLUDE qualifier to a library file specification, the linker processes the library's name table to search for modules that contain needed definitions. When the linker finds an object module or shareable image in the library that contains a needed definition, it processes it as described in Section 2.2.3.1. In addition, the linker also includes the modules specified with the /INCLUDE qualifier in the link operation and processes them as it would any other object module or shareable image.

2.2.3.3 Processing Default Libraries

In addition to the libraries you specify using the /LIBRARY qualifier or the /INCLUDE qualifier, the linker also processes certain other libraries by default. The linker processes these default libraries in the following order:

1. **Default user library files**. You specify a default user library by associating the library with one of the linker's default logical names from the range LNK$LIBRARY, LNK$LIBRARY_1, . . . LNK$LIBRARY_999. If the /NOUSERLIBRARY qualifier is specified, the linker skips processing default user libraries. (For more information about defining a default user library, see the description of the /USERLIBRARY qualifier in Part II.)

If the default user library contains shareable images, the linker looks for the shareable image as described in Processing Shareable Image Libraries, in Section 2.2.3.1.

2. **Default system shareable image library file**. The linker processes the default system shareable image library IMAGELIB.OLB by default unless you specify the /NOSYSSHR or the /NOSYSLIB qualifier.

   Note that when the linker needs to include a shareable image from IMAGELIB.OLB in a link operation, it always looks for the shareable images in SYS$LIBRARY for VAX linking or ALPHA$LIBRARY for Alpha linking. The linker does *not* look for the shareable image in the device and directory of IMAGELIB.OLB as it does for other shareable image libraries.

3. **Default system object module library file**. The linker processes the default system object library STARLET.OLB by default unless you specify the /NOSYSLIB qualifier.

Alpha

For Alpha linking, when the linker processes STARLET.OLB by default, it also processes the shareable image (SYS$PUBLIC_VECTORS.EXE). This shareable image is needed to resolve references to system services. (For VAX linking, references to system services are resolved by linking against the file SYS$P1_VECTOR, which resides in STARLET.OLB.)

When STARLET is not processed by default (for example, when the /NOSYSLIB qualifier is specified), the linker does not process SYS$PUBLIC_VECTORS.EXE automatically, even if you explicitly specify STARLET.OLB in an options file.

If you specify SYS$PUBLIC_VECTORS.EXE explicitly in an options file when it is already being processed by default, the linker displays the following warning: ♦

```
%LINK-W-MULCLUOPT, cluster SYS$PUBLIC_VECTORS multiply defined
        in options file [filename]
```

2.2.3.4 Open Systems Library Support

If you are developing portable applications using the Digital Network Application Support (NAS) products, a second image library, similar to IMAGELIB, is used. The second image library contains components that conform to NAS conventions rather than to OpenVMS conventions. By default, the linker will not search this library because it may contain symbols that do not conform to the OpenVMS global symbol naming rules.

If you want the linker to include the open image library in its processing, define the logical name LNK$OPEN_LIB with any nonnull string value. If the LNK$OPEN_LIB logical is defined at link time, the linker searches OPEN_LIB in the same way it searches IMAGELIB. The open image library search is in addition to any other searches, and it is done after user libraries are searched and before other system libraries are searched, as follows:

1. User libraries, if defined with LNK$LIBRARY_*nnn*
2. OPEN_LIB, if LNK$OPEN_LIB logical is defined
3. IMAGELIB, unless /NOSYSSHR is specified
4. STARLET, unless /NOSYSLIB is specified

2.2.4 Processing Input Files Selectively

By default, the linker processes all the symbol definitions and references in an object module or a shareable image specified as input in a link operation. However, if you append the /SELECTIVE_SEARCH qualifier to an input file specification, the linker processes from the input file only those symbol definitions that resolve references in previously processed input files.

Processing input files selectively can reduce the amount of time a link operation takes and can conserve the linker's use of virtual memory. Note, however, that selective processing can also introduce dependencies on the ordering of input files in the LINK command.

Note

Processing files selectively does not affect the size of the resultant image; the entire object module is included in the image even if only a subset of the symbols it defines is referenced. (Shareable images do not contribute to the size of an image.)

For example, in the link operation in Section 2.2.2, the linker processes the shareable image MY_MATH.EXE before it processes the object module MY_MAIN.OBJ because of the way in which the linker clusters input files. (For information about how the linker clusters input files, see Section 2.3.2.1.) When it processes the shareable image, the linker includes on its list of symbol definitions all the symbols defined in the shareable image. When it processes the object module MY_MAIN.OBJ and encounters the reference to the symbol mysub, the linker has the definition to resolve the reference.

If you append the /SELECTIVE_SEARCH qualifier to the shareable image file specification and all of the other input files are specified on the command line, the link will fail. In the following example, because the linker has no symbols on its undefined symbol list when it processes the shareable image file MY_MATH.EXE, it does not include any symbol definitions from the shareable image in its processing. When it subsequently processes the object module MY_MAIN.OBJ that references the symbol mysub, the linker cannot resolve the reference to the symbol. (For information about how to correct this link operation, see Section 2.3.2.1.)

```
$ LINK MY_MAIN, SYS$INPUT/OPT
MY_MATH/SHAREABLE/SELECTIVE_SEARCH
Ctrl/Z
%LINK-W-NUDFSYMS, 1 undefined symbol:
%LINK-I-UDFSYM,         MYSUB
%LINK-W-USEUNDEF, undefined symbol MYADD referenced
        in psect $CODE offset %X00000011
        in module MY_MAIN file WORK:[PROGRAMS]MY_MAIN.OBJ;6
```

To process object modules or shareable images in a library selectively, you must specify the /SELECTIVE_SEARCH qualifier when you insert the module in the library. The following example creates the library MYMATH_LIB.OLB and inserts the file MY_MATH.OBJ into the library. (For more information about using the Librarian utility, see the *OpenVMS Command Definition, Librarian, and Message Utilities Manual*.)

```
$ LIBRARY/CREATE/INSERT MYMATH_LIB MY_MATH/SELECTIVE_SEARCH
```

2.3 Ensuring Correct Symbol Resolution

For many link operations, the order in which the input files are specified in the LINK command is not important. However, in complex link operations that specify many library files or process input files selectively, to ensure that the linker resolves all the symbolic references among the input files as you intend, you may need to be aware of the order in which the linker processes the input files. To control the order in which the linker processes input files, you must understand how the linker parses the command line.

2.3.1 Understanding Cluster Creation

As it parses the command line, the linker groups the input files you specify into **clusters** and places these clusters on a cluster list. A cluster is an internal linker construct that determines image section creation. The position of an input file in a cluster and the position of that cluster on the linker's cluster list determine the order in which the linker processes the input files you specify.

The linker always creates at least one cluster, called the **default cluster**. The linker may create additional clusters, called **named clusters**, depending on the types of input files you specify and the linker options you specify. If it creates additional clusters, the linker places them on the cluster list ahead of the default cluster, in the order in which it encounters them in the options file. The default cluster appears at the end of the cluster list. (Within the default cluster, input files appear in the same order in which they are specified on the LINK command line.)

Clusters for shareable images specified in shareable image libraries appear *after* the default cluster on the cluster list because they are created later in linker processing, when the linker knows which shareable images in the library are needed for the link operation.

The linker groups input files into clusters according to file type. Table 2–2 lists the types of input files accepted by the linker and describes how the linker processes them when creating clusters.

Table 2–2 Linker Input File Cluster Processing

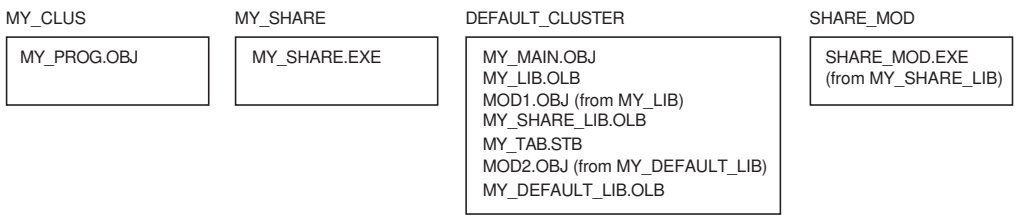
Input File	Cluster Processing
Object file (.OBJ)	Placed in the default cluster unless explicitly placed in a named cluster using the CLUSTER= option.
Shareable image file (.EXE)	Always placed in a named cluster.
†Symbol table file (.STB)	Placed in the default cluster unless explicitly placed in a named cluster using the CLUSTER= option.
Library files (.OLB)	Placed in the default cluster unless explicitly placed in a named cluster using the CLUSTER= option. If the library contains shareable images and the linker includes a shareable image from the library in the link operation, the linker creates a new cluster for the shareable image. The linker puts input files included in a link operation from a library using the /INCLUDE qualifier in the same cluster as the library. The linker puts modules extracted from any default user library that is an object library and from STARLET.OLB in the default cluster. However, because they are shareable images, the linker puts modules extracted from IMAGELIB.OLB into new clusters at the end of the cluster list (after the default cluster).
Options file (.OPT)	Not placed in a cluster.

†VAX specific

The following example illustrates how the linker puts the various types of input files in clusters. To see which clusters the linker creates for this link operation, look at the Image Section Synopsis section of the image map file. Figure 2–3 illustrates the clusters created for this link operation.

```
$ DEFINE LNK$LIBRARY  SYS$DISK:[]MY_DEFAULT_LIB.OLB
$ LINK  MY_MAIN.OBJ, MY_LIB.OLB/LIBRARY, SYS$INPUT/OPT
CLUSTER=MY_CLUS,,,MY_PROG.OBJ
MY_SHARE.EXE/SHAREABLE
MY_SHARE_LIB.OLB/LIBRARY
MY_TAB.STB
```

Figure 2–3 Clusters Created for Sample Link



ZK–5291A–GE

The linker processes input files in cluster order: processing each input file starting with the first file in the first cluster, then the second, and so on, until it has processed all files in the first cluster. Then it does the same for the second cluster, and the next, and so on, until it has processed all files in all clusters.

2.3.2 Controlling Cluster Creation

You can control in which cluster the linker places an input file by using either of the following linker options:

- CLUSTER= option
- COLLECT= option

2.3.2.1 Using the CLUSTER= Option to Control Clustering

The CLUSTER= option causes the linker to create a named cluster and to place in the cluster the object modules specified in the option. (The linker puts shareable images in their own clusters by default.)

For example, you can use the CLUSTER= option to fix the link operation illustrated in Section 2.2.4, where the link failed because a shareable image was processed selectively. To make the linker process the object module MY_MAIN.OBJ before it processes the shareable image MY_MAIN.EXE, put the object module in a named cluster. In the following example, the /EXECUTABLE qualifier is specified on the command line to specify the name of the resultant image, because MY_MAIN is not specified on the command line.

```
$ link/executable=my_main sys$input/opt
CLUSTER=mymain_clus,,,my_main
my_math/shareable/selective_search
Ctrl/Z
```

The Object Module Synopsis section of the image map file verifies that the linker processed the object module MY_MAIN before it processed the shareable image MY_MATH, as in the following map file excerpt:

```
                              +------------------------+
                              ! Object Module Synopsis !
                              +------------------------+

Module Name     Ident             Bytes      File
-----------     -----             -----      -----
MY_MAIN         V1.0                 105 MY_MAIN.OBJ;1
MY_MATH         V1.0                  12 MY_MATH.EXE;1
   .
   .
   .
```

2.3.2.2 Using the COLLECT= Option to Control Clustering

You can also create a named cluster by specifying the COLLECT= option. The COLLECT= option directs the linker to put specific program sections in a named cluster. The linker creates the cluster if it does not already exist. Note that the COLLECT= option manipulates program sections, *not* input files.

The linker sets the global (GBL) attribute of the program sections specified in a COLLECT= option to enable a global search for the definition of that program section.

2.4 Resolving Symbols Defined in the OpenVMS Executive

VAX

For VAX linking, you link against the OpenVMS executive by specifying the system symbol table (SYS$LIBRARY:SYS.STB) in the link operation. Because a symbol table file is an object module, the linker processes the symbol table file as it would any other object module. ♦

Alpha

For Alpha linking, you link against the OpenVMS executive by specifying the /SYSEXE qualifier. When this qualifier is specified, the linker selectively processes the system shareable image SYS$BASE_IMAGE.EXE located in the directory pointed to by the logical name ALPHA$LOADABLE_IMAGES. The linker does not process SYS$BASE_IMAGE.EXE by default.

Note that, because the linker is processing a shareable image, references to symbols in the OpenVMS executive are fixed up at image activation, not fully resolved at link time as they are for VAX linking. Also note that the linker looks for SYS$BASE_IMAGE.EXE in the directory pointed to by the logical name ALPHA$LOADABLE_IMAGES, *not* in the directory pointed to by the logical name SYS$LIBRARY as for VAX linking.

When the /SYSEXE qualifier is specified, the linker processes the file selectively. To disable selective processing, specify the /SYSEXE=NOSELECTIVE qualifier. For more information about using the /SYSEXE qualifier, see the description of the qualifier in Part II.

Relation to Default Library Processing

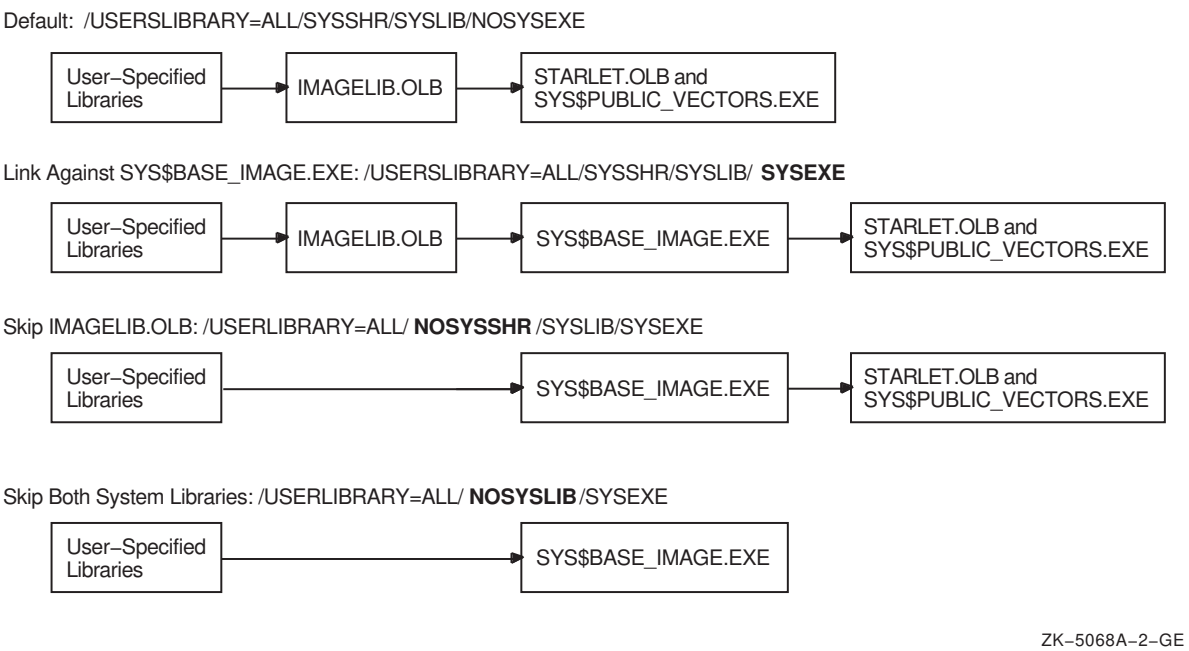
When you specify the /SYSEXE qualifier, the linker processes the SYS$BASE_IMAGE.EXE file *after* processing the system shareable image library, IMAGELIB.OLB, and *before* processing the system object library, STARLET.OLB. (Note that the linker also processes the system service shareable image, SYS$PUBLIC_VECTORS.EXE, when it processes STARLET.OLB by default.)

The /SYSSHR and /SYSLIB qualifiers, which control processing of the default system libraries, do not affect SYS$BASE_IMAGE.EXE processing. When the /NOSYSSHR qualifier is specified with the /SYSEXE qualifier, the linker does not process IMAGELIB.OLB, but still processes SYS$BASE_IMAGE.EXE and then STARLET.OLB and SYS$PUBLIC_VECTORS.EXE. When /NOSYSLIB is specified, the linker does not process IMAGELIB.OLB, STARLET.OLB, or SYS$PUBLIC_VECTORS, but still processes SYS$BASE_IMAGE.EXE.

To process SYS$BASE_IMAGE.EXE before the shareable images in IMAGELIB.OLB, specify SYS$BASE_IMAGE.EXE in a linker options file as you would any other shareable image. If you specify SYS$BASE_IMAGE.EXE in your options file, do not use the /SYSEXE qualifier.

Figure 2–4 illustrates how the /SYSEXE qualifier, in combination with the /SYSSHR and /SYSLIB qualifiers, can affect linker processing. (The default syntax illustrated in the figure is rarely specified.) ♦

Figure 2–4 Linker Processing of Default Libraries and SYS$BASE_IMAGE.EXE



ZK–5068A–2–GE

2.5 Defining Weak and Strong Global Symbols

In the dialects of MACRO, BLISS, and Pascal supported on both VAX and Alpha systems, you can define a global symbol as either strong or weak, and you can make either a strong or a weak reference to a global symbol.

In these languages, all definitions and references are strong by default. To make a weak definition or a weak reference, you must use the .WEAK assembler directive (in MACRO), the WEAK attribute (in BLISS), or the WEAK_GLOBAL or WEAK_EXTERNAL attribute (in Pascal).

The linker records each symbol definition and each symbol reference in its global symbol table, noting for each whether it is strong or weak. The linker processes weak references differently from strong references and weakly defined symbols differently from strongly defined symbols.

A strong reference can be made to a weakly defined symbol or to a strongly defined symbol.

For a strong reference, the linker checks all explicitly specified input modules and libraries and all default libraries for a definition of the symbol. In addition, if the linker cannot locate the definition needed to resolve the strong reference, it assigns the symbol a value of 0 and reports an error. By default, all references are strong.

A weak reference can be made to a weakly defined symbol or to a strongly defined symbol. In either case, the linker resolves the weak reference in the same way it does a strong reference, with the following exceptions:

- The linker will not search library modules that have been specified with the /LIBRARY qualifier or default libraries (user-defined or system) solely to resolve a weak reference. If, however, the linker resolves a strong reference to another symbol in such a module, it will also use that module to resolve any weak references.

- If the linker cannot locate the definition needed to resolve a weak reference, it assigns the symbol a value of 0, but does not report an error (as it does if the reference is strong). If, however, the linker reports any unresolved strong references, it will also report any unresolved weak references.

One purpose of making a weak reference arises from the need to write and test incomplete programs. The resolution of all symbolic references is crucial to a successful linking operation. Therefore, a problem arises when the definition of a referenced global symbol does not yet exist (as would be the case, for example, if the global symbol definition is an entry point to a module that is not yet written). The solution is to make the reference to the symbol weak, which informs the linker that the resolution of this particular global symbol is not crucial to the link operation.

By default, all global symbols in all VAX and Alpha languages have a strong definition.

A strongly defined symbol in a library module is included in the library symbol table; a weakly defined symbol in a library module is not. As a result, if the module containing the weak symbol definition is in a library but has not been specified for inclusion by means of the /INCLUDE qualifier, the linker will not be able to resolve references (strong or weak) to the symbol. If, however, the linker has selected that library module for inclusion (in order to resolve a strong reference), it will be able to resolve references (strong or weak) to the weakly defined symbol.

If the module containing the weak symbol definition is explicitly specified either as an input object file or for extraction from a library (by means of the /INCLUDE qualifier), the weak symbol definition is as available for symbol resolution as a strong symbol definition.

3
Understanding Image File Creation

This chapter describes how the linker creates an image from the input files you specify in a link operation and how you can control image file creation by using linker qualifiers and options.

3.1 Overview

After the linker has resolved all symbolic references between the input files specified in the LINK command (described in Chapter 2), the linker knows all the object modules and shareable images that are required to create the image. For example, the linker has extracted from libraries specified in the LINK command those modules that contain the definitions of symbols required to resolve symbolic references in other modules. The linker must now combine all these modules into an image.

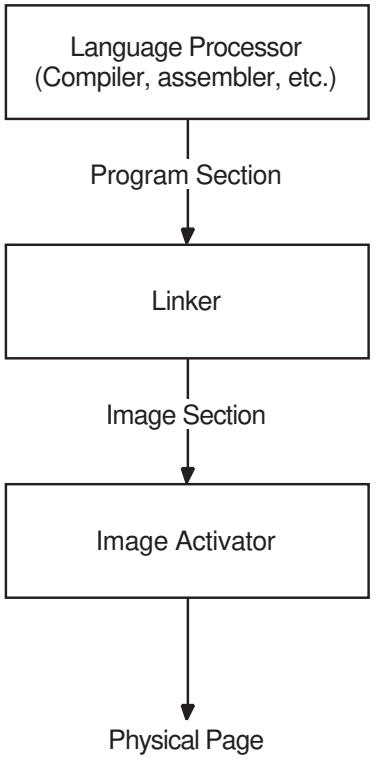
To create an image, the linker must perform the following processing:

- **Determine the memory requirements of the image**. The memory requirements of an image are the sum of the memory requirements of each object module included in the link operation. The language processors that create the object modules specify the memory requirements of an object module as **program section** definitions. A program section represents an area of memory that has a name, a length, and other characteristics, called **attributes**, which describe the intended or permitted usage of that portion of memory. Section 3.2 describes program sections.

  The linker processes the program section definitions in each object module, combining program sections with similar attributes into an **image section**. Each image section specifies the size and attributes of a portion of the virtual memory of an image. The image activator uses the image section attributes to determine the characteristics of the physical memory pages into which it loads the image, such as protection.

  Figure 3–1 illustrates how memory requirements are communicated from the language processor to the linker and from the linker to the image activator. Section 3.3 provides more information about this process.

Figure 3–1 Communication of Image Memory Requirements



ZK–5199A–GE

Note that shareable images included in link operations have already been processed by the linker. These images are separate images with their own memory requirements, as specified by their own image sections. The linker does, however, create special global image section descriptors (GISDs) for each shareable image to which an image has been linked. The image activator activates these shareable images at run time.

- **Initialize the image**. When image sections are first created, they are empty. In this step of linker processing, the linker fills the image sections with the machine code and data, as specified by the Text Information and Relocation (TIR) commands in the object module. Section 3.4 provides more information about this process.

  Alpha

  For Alpha linking, the linker also attempts to optimize the performance of an image by replacing Jump to Subroutine (JSR) instruction sequences with the more efficient Branch to Subroutine (BSR) instruction sequences. See Section 1.4 for more information. ♦

After creating image sections and filling them with binary code and data, the linker writes the image to an image file. Section 3.4.1 describes this process. To keep the size of image files manageable, the linker does not allocate space in the image file for image sections that have not been initialized with any data (that is, the linker does not write pages of zeros to the image file). Instead, the linker creates **demand-zero** image sections, which are initialized at run time by the operating system when a reference to the image section requires the operating system to move the pages into memory. Section 3.4.3 describes how the linker creates demand-zero image sections.

3.2 Creating Program Sections

Language processors create program sections and define their attributes. The number of program sections created by a language processor and the attributes of these program sections are dependent upon language semantics. For example, some programming languages implement global variables as separate program sections with a particular set of attributes. Programmers working in high-level languages typically have little direct control over the program sections created by the language processor. Medium- and low-level languages provide programmers with more control over program section creation. For more information about the program section creation features of a particular programming language, see the language processor documentation.

Alpha

In general, the linker does not create program sections. However, for Alpha linking, the linker creates a special program section for a shareable image, named $SYMVECT, which contains the symbol vector of the shareable image. ♦

Program Section Attributes

The language processors define the attributes of the program sections they create and communicate these attributes to the linker in program section definition records in the global symbol directory (GSD) in an object module. (The GSD also contains global symbol definitions and references, as described in Chapter 2.)

Program section attributes control various characteristics of the area of memory described by the program section, such as the following:

- **Access**. Using program section attributes, compilers can prohibit some types of access, such as write access. Using other program section attributes, compilers can allow access to the program section by more than one process.
- **Positioning**. By specifying certain program section attributes, compilers can specify to the linker how it should position the program section in memory.

Program section attributes are Boolean values, that is, they are either on or off. Table 3–1 lists all program section attributes with the keyword you can use to set or clear the attribute, using the PSECT_ATTR= option. (For more information about using the PSECT_ATTR= option, see Section 3.3.6.)

For example, to specify that a program section should have write access, specify the writability attribute as WRT. To turn off an attribute, specify the negative keyword. Some attributes have separate keywords that express the negative of the attribute. For example, to turn off the global attribute (GBL), you must specify the local attribute (LCL). Note that the alignment of a program section is not strictly considered an attribute of the program section. However, because you can set it using the PSECT_ATTR= option, it is included in the table.

Table 3–1 Program Section Attributes

Attribute	Keyword	Description
Alignment	–	Specifies the alignment of the program section as an integer that represents the power of 2 required to generate the desired alignment. For certain alignments, the linker supports keywords to express the alignment. The following table lists all the alignments supported by the linker with their keywords:

Power of 2	Keyword	Meaning
0	BYTE	Alignment on byte boundaries.
1	WORD	Alignment on word boundaries.
2	LONG	Alignment on longword boundaries.
3	QUAD	Alignment on quadword boundaries.
4	OCTA	Alignment on octaword boundaries.
9	–	Alignment on 512-byte boundaries.
13	–	Alignment on 8 KB boundaries.
14	–	Alignment on 16 KB boundaries.
15	–	Alignment on 32 KB boundaries.
16	–	Alignment on 64 KB boundaries.
–	PAGE	Alignment on the default target page size, which is 64 KB for Alpha linking and 512 bytes for VAX linking. You can override this default by specifying the /BPAGE qualifier.

Position Independence	PIC/NOPIC	Specifies that the program section can run anywhere in virtual address space. Applicable in shareable images only. Note that this attribute is not meaningful for Alpha images, but it is still used to sort program sections.
Overlaid /Concatenated	OVR/CON	When set to OVR, specifies that the linker may combine (overlay) this program section with other program sections with the same name and attribute settings. Program sections that are overlaid are assigned the same base address. When set to CON, the linker concatenates the program sections.
Relocatable /Absolute	REL/ABS	When set to REL, specifies that the linker can place the program section anywhere in virtual memory, according to the memory allocation strategy for the type of image being produced. When set to ABS, this attribute specifies that the program section is an absolute program section that contains no binary data or code and appears to be based at virtual address 0. Absolute program sections are used by compilers primarily to define constants.
Global/Local	GBL/LCL	When set to GBL, specifies that the linker should gather contributions to the program section *from all clusters* and place them in the same image section. When set to LCL, the linker gathers program sections into the same image section only if they are in the same cluster. The memory for a global program section is allocated in the cluster that contains the first contributing module.

(continued on next page)

Table 3–1 (Cont.) Program Section Attributes

Attribute	Keyword	Description
Shareability	SHR/NOSHR	Specifies that the program section can be shared between several processes. Only used to sort program sections in shareable images.
Executability	EXE/NOEXE	Specifies that the program section contains executable code. If an image transfer address is defined in a nonexecutable program section, the linker issues a diagnostic message. ‡For Alpha linking, the EXE attribute is propagated to the image section descriptor where it is used by the Install utility when it is installing the image as a resident image. (For information about resident images, see the description of the /SECTION_BINDING qualifier in Part II.)
Writability	WRT/NOWRT	Specifies that the contents of a program section can be modified at run time.
Protected Vectors	VEC/NOVEC	Specifies that the program section contains privileged change-mode vectors or message vectors. In shareable images, image sections with the VEC attribute are automatically protected.
Solitary	SOLITARY	Specifies that the linker should place this program section in its own image section. Useful for programs that map data into specific locations in their virtual memory space. Note that compilers do not set this attribute. You can set this attribute using the PSECT_ ATTR= option.
‡Unmodified	NOMOD/MOD	When set, specifies that the program section has not been initialized (NOMOD). On Alpha systems, the linker uses this attribute to create demand zero sections; see Section 3.4.3. Only compilers can set this attribute. You can clear this attribute only by specifying the MOD keyword in the PSECT_ATTR= option.
‡COM	–	Used by the DEC C compiler to implement the relaxed symbol reference/definition model for external variables. See the C documentation for more information. This attribute cannot be modified using the PSECT_ATTR= option.
Readability	RD	Reserved by Digital.
User/Library	USR/LIB	Reserved by Digital. To ensure future compatibility, this attribute should be clear.

‡Alpha specific

To illustrate program section creation, consider the program sections created by the VAX C compiler when it processes the sample programs in Example 3–1, Example 3–2, and Example 3–3.

Example 3–1 Sample Program MYTEST.C

```
extern int global_data;

int myadd();
int mysub();

main()
{
   int num1, num2, res1, res2;
   static int my_data;

   num1 = 5;
   num2 = 6;
```

(continued on next page)

Example 3–1 (Cont.) Sample Program MYTEST.C

```
   res1 = myadd( num1, num2 );
   res2 = mysub( num1, num2 );
   printf("res1 = %d, res2 =%d, globaldata=%d\n",
                 res1,res2,global_data);
}
```

Example 3–2 Sample Program MYADD.C

```
#include <stdio.h>

myadd(value_1,value_2)
 int value_1;
 int value_2;
{
 int result;
 static int add_data;

 printf("In MYADD.C\n");

 result = value_1 + value_2;
 return( result );
}
```

Example 3–3 Sample Program MYSUB.C

```
int global_data = 5;

mysub(value_1,value_2)
 int value_1;
 int value_2;
{
 int result;
 static int sub_data;

 result = value_1 - value_2;
 return( result );
}
```

To see what program sections the VAX C compiler creates for these programs, use the ANALYZE/OBJECT utility to examine the global symbol directory (GSD) in each object module. (Note that the names the language processors assign to program sections are architecture specific.)

Example 3–4 presents an excerpt from the analysis of the object module MYTEST.OBJ. Only the program section definitions are included in the excerpt. (Note that you can also determine the program sections in an object module *after* a link operation by looking at the Program Section Synopsis section of an image map file, as illustrated in Example 3–7.)

Example 3–4 Program Sections Generated by Example 3–1

```
4.  GLOBAL SYMBOL DIRECTORY (OBJ$C_GSD), 138 bytes
          .
          .
          .
      6)  Program Section Definition (GSD$C_PSC)
       ❶    alignment: 4-byte boundary        <-- psect 0
       ❷    attribute flags:
                  (0)  GPS$V_PIC        1
                  (1)  GPS$V_LIB        0
                  (2)  GPS$V_OVR        0
                  (3)  GPS$V_REL        1
                  (4)  GPS$V_GBL        0
                  (5)  GPS$V_SHR        1
                  (6)  GPS$V_EXE        1
                  (7)  GPS$V_RD         1
                  (8)  GPS$V_WRT        0
                  (9)  GPS$V_VEC        0
       ❸    allocation: 63 (%X'0000003F')
       ❹    symbol: "$CODE"
      7)  Program Section Definition (GSD$C_PSC)
          alignment: 4-byte boundary       <-- psect 1
          attribute flags:
                  (0)  GPS$V_PIC        1
                  (1)  GPS$V_LIB        0
                  (2)  GPS$V_OVR        0
                  (3)  GPS$V_REL        1
                  (4)  GPS$V_GBL        0
                  (5)  GPS$V_SHR        0
                  (6)  GPS$V_EXE        0
                  (7)  GPS$V_RD         1
                  (8)  GPS$V_WRT        1
                  (9)  GPS$V_VEC        0
          allocation: 4 (%X'00000004')
          symbol: "DATA"
      8)  Program Section Definition (GSD$C_PSC)
          alignment: 4-byte boundary       <-- psect 2
          attribute flags:
                  (0)  GPS$V_PIC        1
                  (1)  GPS$V_LIB        0
                  (2)  GPS$V_OVR        1
                  (3)  GPS$V_REL        1
                  (4)  GPS$V_GBL        1
                  (5)  GPS$V_SHR        1
                  (6)  GPS$V_EXE        0
                  (7)  GPS$V_RD         1
                  (8)  GPS$V_WRT        1
                  (9)  GPS$V_VEC        0
          allocation: 4 (%X'00000004')
          symbol: "GLOBAL_DATA"
```

(continued on next page)

Example 3–4 (Cont.) Program Sections Generated by Example 3–1

```
9)  Program Section Definition (GSD$C_PSC)
       alignment: 4-byte boundary      <-- psect 3
       attribute flags:
             (0)  GPS$V_PIC        1
             (1)  GPS$V_LIB        0
             (2)  GPS$V_OVR        0
             (3)  GPS$V_REL        1
             (4)  GPS$V_GBL        0
             (5)  GPS$V_SHR        0
             (6)  GPS$V_EXE        0
             (7)  GPS$V_RD         1
             (8)  GPS$V_WRT        1
             (9)  GPS$V_VEC        0
       allocation: 36 (%X'00000024')
       symbol: "$CHAR_STRING_CONSTANTS"
        .
        .
        .
```

The items in the following list correspond to the numbered items in Example 3–4:

❶ Alignment specifies the address boundary at which the linker places a module's contribution to the program section.

❷ Attribute flags indicate which program section attributes are set. The attributes are listed by their full symbolic name, that is, each abbreviation is preceded by the character string ''GPS$V_''. Note that, for attributes that are turned off by specifying different keywords, only the keyword that sets the attribute is listed. For example, you can see whether the program section is overlaid by checking attribute flag number 2. If the value is 1, the program section is overlaid; if the value is 0, the program section must be concatenated. Table 3–1 lists all the program section attributes. Note that the solitary attribute is not included in the GSD of an object module because that attribute is not set by language processors.

Alpha

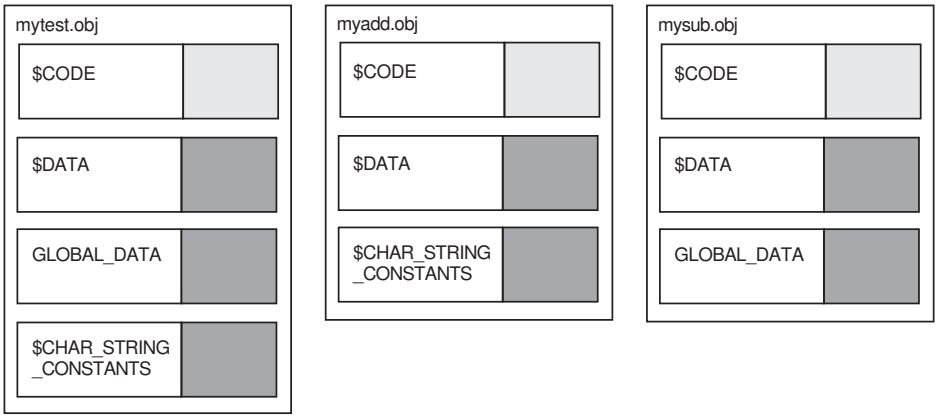
For Alpha linking, the program section display includes several additional attribute flags. The COM attribute is reserved for use by Digital. The NOMOD attribute indicates that the program section does not contain initialized data. The linker gathers program sections with this attribute into demand-zero image sections. Section 3.4.3 describes how the linker creates demand-zero image sections. ♦

❸ Allocation indicates the number of bytes required for the program section.

❹ Symbol indicates the name of the program section.

Figure 3–2 illustrates the program sections created by the VAX C compiler for the programs in Example 3–1, Example 3–2, and Example 3–3. (The shaded areas represent the settings of the program section attributes the linker considers when sorting the program sections into image sections in an executable image. See Section 3.3.3 for more information about how the linker creates image sections.)

Figure 3–2 Program Sections Created for Example 3–1, Example 3–2, and Example 3–3



3.3 Creating Image Sections

To create the image sections that define the memory requirements and page protection characteristics of an image, the linker processes the program section definitions in the object modules specified in the link operation. The number and type of image sections the linker creates depend on the number of clusters the linker creates when processing the LINK command and on the attributes of the program sections in the object modules in each cluster. Section 3.3.1 describes how the clustering of input files affects image section creation. Section 3.3.2 describes the effects of program section attributes on image section creation.

3.3.1 Processing Clusters to Create Image Sections

To create image sections, the linker processes the program section definitions in the input files you specify in the LINK command. The linker processes these input files on a cluster-by-cluster basis (as described in Section 2.3.1).

In general, only program sections in a particular cluster can contribute to a particular image section. However, the linker crosses cluster boundaries when processing program sections with the global (GBL) attribute. When the linker encounters a program section with the global attribute, it searches all the previously processed clusters for a program section with the same name and attributes and, if it finds one, places the new definition of the global program section in the same cluster as the first definition of the program section.

The linker processes input files in the order in which they appear in the clusters, making two passes through the cluster list.

On its first pass, the linker processes **based** clusters. Based clusters specify the location within memory at which the linker must position them. A based cluster can be a cluster that contains a based shareable image or a cluster, created by the CLUSTER= option, in which a base address was specified.

VAX

For VAX linking, you can also use the BASE= option to specify the base address of the default cluster. ♦

For more information about creating based clusters, see the descriptions of the CLUSTER= and BASE= options in Part II.

After processing based clusters, the linker then processes nonbased clusters. The linker ignores nonbased (position-independent) shareable image clusters because they are allocated virtual memory at run time.

For example, Example 3–5 presents a LINK command to create an image using the object modules in Section 3.2.

Example 3–5 Linking Example 3–1, Example 3–2, and Example 3–3

```
$ LINK/MAP/FULL MYTEST, MYADD, SYS$INPUT/OPT
CLUSTER=MYSUB_CLUS,,,MYSUB
SYS$LIBRARY:VAXCRTL/SHARE
Ctrl/Z
```

The CLUSTER= option in this LINK command causes the linker to create a cluster named MYSUB_CLUS, which contains the object module MYSUB.OBJ. The linker also creates a cluster for the C run-time library shareable image VAXCRTL.EXE. The linker puts the object modules MYTEST.OBJ and MYADD.OBJ in the default cluster. These clusters appear on the linker's cluster list in the following order:

1. MYSUB_CLUS
2. VAXCRTL
3. DEFAULT_CLUSTER

The linker always processes the default cluster last. (For Alpha linking, you do not need to explicitly include the C run-time library shareable image in the link operation because it resides in the default system shareable image library IMAGELIB.OLB, which the linker processes by default.)

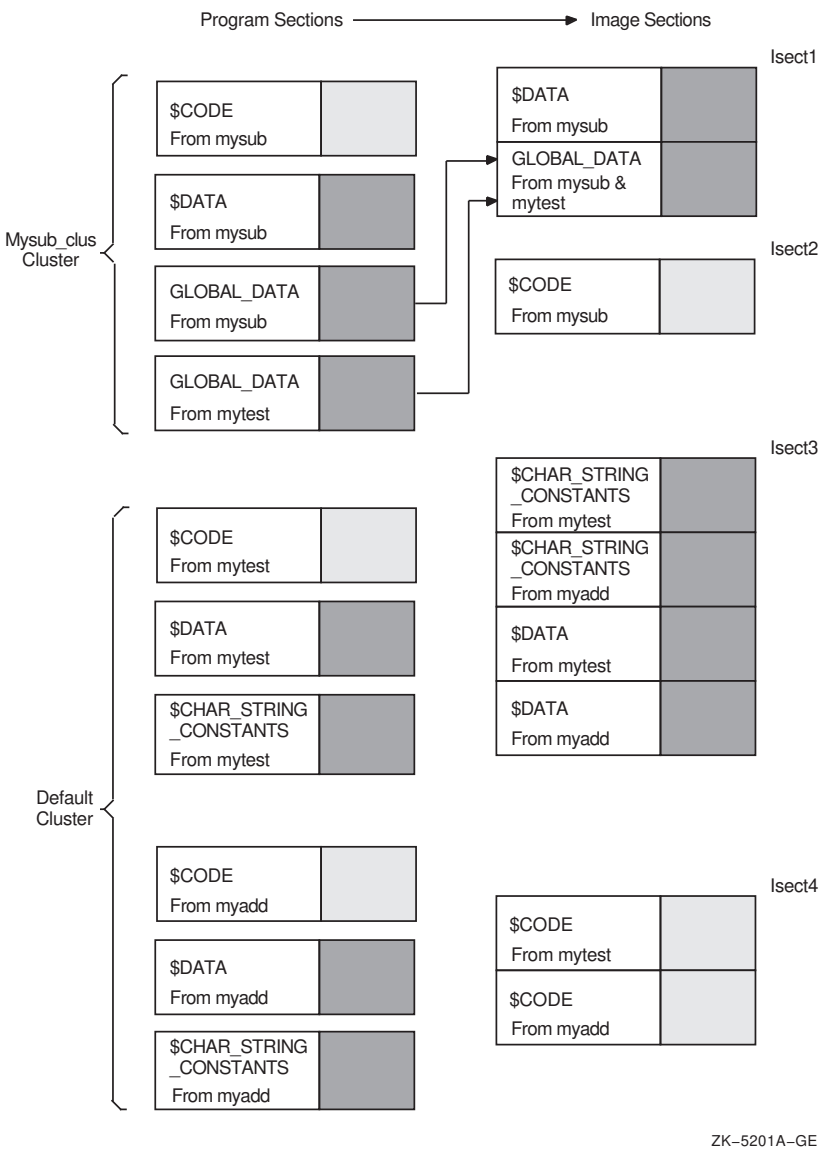
3.3.2 Combining Program Sections into Image Sections

The linker creates image sections by grouping together program sections with similar attributes. Within an image section, the linker organizes program sections alphabetically by name. If more than one object module contributes to the same program section, the linker lays out their contributions in the order it processes them.

Figure 3–3 shows how the linker groups the program sections in the object modules from the sample link into image sections, based on the setting of their significant attributes. In the figure, the settings of these significant attributes are represented by shading. (The figure considers attributes that are significant when creating executable images, not shareable images. Section 3.3.3 provides more information about which program section attributes are significant.)

Note, in the figure, that the overlaid contributions from MYSUB.OBJ and MYTEST.OBJ to the program section, GLOBAL_DATA, both appear in the MYSUB_CLUS cluster, even though the object module MYTEST.OBJ is in the default cluster. The linker puts all contributions to a global program section in the cluster in which it is first defined.

Figure 3–3 Combining Program Sections into Image Sections



3.3.3 Processing Significant Program Section Attributes

When combining program sections into image sections, the linker considers only a subset of program section attributes. The set of significant attributes varies according to the type of image being created. When creating an executable image, the linker considers all combinations of the following attributes when combining program sections into image sections:

- Writability (WRT/NOWRT)
- Executability (EXE/NOEXE)
- Protected vector (VEC/NOVEC)

Alpha

- Unmodified (NOMOD/MOD) (Alpha linking only) ♦

When creating a shareable image, the linker considers all combinations of the following attributes when combining program sections into image sections:

- Writability (WRT/NOWRT)
- Executability (EXE/NOEXE)
- Shareability (SHR/NOSHR)
- Position independence (PIC/NOPIC)
- Protected vector (VEC/NOVEC)

Alpha

- Unmodified (NOMOD/MOD) (Alpha linking only) ♦

The linker creates only one large image section for system images, so combining program sections by attributes is not applicable.

Table 3–2 and Table 3–3 list all the possible combinations of program section attributes for executable images and shareable images. Note that the order in which the combinations appear in the table (each row) is the same order in which the linker processes them. For example, the linker first processes all program sections with the NOWRT, NOEXE, and NOVEC attributes, creating an image section of program sections with these attributes. The linker then processes all program sections with the WRT, NOEXE, and NOVEC attributes, creating an image section for these program sections. The linker continues this processing until all the combinations of significant attributes have been processed and all the program sections in the cluster have been placed in an image section.

The tables include only program sections that are relocatable (with the REL attribute). Absolute program sections (with the ABS attribute), by definition, can have no allocation (they contain only constants) and cannot contribute to an image section.

Alpha

For OpenVMS Alpha images, the tables assume that the images are linked using the /DEMAND_ZERO qualifier, which is the default. (When this qualifier is specified, the linker groups program sections that do not contain any data into demand-zero image sections, allocating memory for the image section but not writing zeros to disk.) If the image is linked with the /NODEMAND_ZERO qualifier, the linker allocates space for the image section in the image file. Note that the /NODEMAND_ZERO qualifier does not affect how the linker sorts program sections; it proceeds exactly as specified by the table. However, when the image is written, the linker allocates disk space for the image section and fills the space with zeros. ♦

The tables also show how a particular combination of program section attributes determines the attributes of the image section in which it is placed. For more information about image section attributes, see Section 3.3.5.

Table 3–2 Mapping Program Section Attributes to Image Section Attributes for Executable Images

Significant Psect Attribute Settings[1]				Type of Isect	Isect Attributes Set[2]
NOWRT	NOEXE	NOVEC	‡MOD	NORMAL	–
WRT	NOEXE	NOVEC	‡MOD	"	WRT, CRF
NOWRT	EXE	NOVEC	‡MOD	"	‡EXE
WRT	EXE	NOVEC	‡MOD	"	WRT, CRF, ‡EXE
NOWRT	NOEXE	VEC	‡MOD	"	VECTOR,PROTECT
WRT	NOEXE	VEC	‡MOD	"	WRT,VECTOR, PROTECT,CRF
NOWRT	EXE	VEC	‡MOD	"	VECTOR,PROTECT, ‡EXE
WRT	EXE	VEC	‡MOD	"	WRT,VECTOR,PROTECT,‡EXE
‡NOWRT	‡NOEXE	‡NOVEC	‡NOMOD	"	DZRO
‡WRT	‡NOEXE	‡NOVEC	‡NOMOD	"	WRT,DZRO[3]

[1]For Alpha images, these attributes are prefixed with EGPS$V_. For VAX images, these attributes are prefixed with GPS$V_.

[2]For Alpha images, these attributes are prefixed with EISD$V_. For VAX images, these attributes are prefixed with ISD$V_.

[3]If the /NODEMAND_ZERO qualifier is specified, the copy-on-reference (CRF) attribute is set instead of the DZRO attribute.

‡Alpha specific

Table 3–3 Mapping Program Section Attributes to Image Section Attributes for Shareable Images

Significant Psect Attribute Settings[1]						Type of Isect	Isect Attributes Set[2]
NOWRT	NOEXE	SHR	NOPIC	NOVEC	‡MOD	SHRFXD	–
WRT	NOEXE	SHR	NOPIC	NOVEC	‡MOD	"	WRT
NOWRT	EXE	SHR	NOPIC	NOVEC	‡MOD	"	‡EXE
WRT	EXE	SHR	NOPIC	NOVEC	‡MOD	"	WRT,‡EXE
NOWRT	NOEXE	NOSHR	NOPIC	NOVEC	‡MOD	PRVFXD	–
WRT	NOEXE	NOSHR	NOPIC	NOVEC	‡MOD	"	WRT, CRF
NOWRT	EXE	NOSHR	NOPIC	NOVEC	‡MOD	"	‡EXE
WRT	EXE	NOSHR	NOPIC	NOVEC	‡MOD	"	WRT,CRF,‡EXE
NOWRT	NOEXE	SHR	PIC	NOVEC	‡MOD	SHRPIC	PIC
WRT	NOEXE	SHR	PIC	NOVEC	‡MOD	"	WRT, PIC
NOWRT	EXE	SHR	PIC	NOVEC	‡MOD	"	PIC, ‡EXE

[1]For Alpha images, these attributes are prefixed with EGPS$V_. For VAX images, these attributes are prefixed with GPS$V_.

[2]For Alpha images, these attributes are prefixed with EISD$V_. For VAX images, these attributes are prefixed with ISD$V_.

‡Alpha specific

(continued on next page)

Table 3–3 (Cont.) Mapping Program Section Attributes to Image Section Attributes for Shareable Images

Significant Psect Attribute Settings[1]						Type of Isect	Isect Attributes Set[2]
WRT	EXE	SHR	PIC	NOVEC	‡MOD	"	WRT,PIC,‡EXE
NOWRT	NOEXE	NOSHR	PIC	NOVEC	‡MOD	PRVPIC	PIC
WRT	NOEXE	NOSHR	PIC	NOVEC	‡MOD	"	WRT, CRF, PIC
NOWRT	EXE	NOSHR	PIC	NOVEC	‡MOD	"	PIC,‡EXE
WRT	EXE	NOSHR	PIC	NOVEC	‡MOD	"	WRT,CRF,PIC, ‡EXE
NOWRT	NOEXE	SHR	NOPIC	VEC	‡MOD	SHRFXD	VECTOR,PROTECT
WRT	NOEXE	SHR	NOPIC	VEC	‡MOD	"	WRT,VECTOR,PROTECT
NOWRT	EXE	SHR	NOPIC	VEC	‡MOD	"	VECTOR,PROTECT,‡EXE
WRT	EXE	SHR	NOPIC	VEC	‡MOD	"	WRT,VECTOR,PROTECT,‡EXE
NOWRT	NOEXE	NOSHR	NOPIC	VEC	‡MOD	PRVFXD	VECTOR,PROTECT
WRT	NOEXE	NOSHR	NOPIC	VEC	‡MOD	"	WRT, CRF
NOWRT	EXE	NOSHR	NOPIC	VEC	‡MOD	"	VECTOR,PROTECT,‡EXE
WRT	EXE	NOSHR	NOPIC	VEC	‡MOD	"	WRT,CRF,VECTOR,PROTECT, ‡EXE
NOWRT	NOEXE	SHR	PIC	VEC	‡MOD	SHRPIC	PIC,VECTOR,PROTECT
WRT	NOEXE	SHR	PIC	VEC	‡MOD	"	WRT,PIC,VECTOR,PROTECT
NOWRT	EXE	SHR	PIC	VEC	‡MOD	"	PIC,VECTOR,PROTECT,‡EXE
WRT	EXE	SHR	PIC	VEC	‡MOD	"	WRT,PIC,VECTOR,PROTECT, ‡EXE
NOWRT	NOEXE	NOSHR	PIC	VEC	‡MOD	PRVPIC	PIC,VECTOR,PROTECT
WRT	NOEXE	NOSHR	PIC	VEC	‡MOD	"	WRT,CRF,PIC,VECTOR, PROTECT
NOWRT	EXE	NOSHR	PIC	VEC	‡MOD	"	PIC,VECTOR,PROTECT, ‡EXE
WRT	EXE	NOSHR	PIC	VEC	‡MOD	"	WRT,CRF,PIC,VECTOR, PROTECT,‡EXE
‡NOWRT	‡NOEXE	‡SHR	‡NOPIC	‡NOVEC	‡NOMOD	SHRFXD	–
‡WRT	‡NOEXE	‡SHR	‡NOPIC	‡NOVEC	‡NOMOD	"	WRT

[1]For Alpha images, these attributes are prefixed with EGPS$V_. For VAX images, these attributes are prefixed with GPS$V_.

[2]For Alpha images, these attributes are prefixed with EISD$V_. For VAX images, these attributes are prefixed with ISD$V_.

‡Alpha specific

(continued on next page)

Table 3–3 (Cont.) Mapping Program Section Attributes to Image Section Attributes for Shareable Images

Significant Psect Attribute Settings[1]						Type of Isect	Isect Attributes Set[2]
‡NOWRT	‡NOEXE	‡NOSHR	‡NOPIC	‡NOVEC	‡NOMOD	PRVFXD	DZRO
‡WRT	‡NOEXE	‡NOSHR	‡NOPIC	‡NOVEC	‡NOMOD	"	WRT,DZRO[3]
‡NOWRT	‡NOEXE	‡NOSHR	‡PIC	‡NOVEC	‡NOMOD	PRVPIC	DZRO
‡WRT	‡NOEXE	‡NOSHR	‡PIC	‡NOVEC	‡NOMOD	"	WRT,DZRO [3], PIC
‡NOWRT	‡NOEXE	‡SHR	‡PIC	‡NOVEC	‡NOMOD	SHRPIC	PIC
‡WRT	‡NOEXE	‡SHR	‡PIC	‡NOVEC	‡NOMOD	"	WRT,PIC

[1]For Alpha images, these attributes are prefixed with EGPS$V_. For VAX images, these attributes are prefixed with GPS$V_.

[2]For Alpha images, these attributes are prefixed with EISD$V_. For VAX images, these attributes are prefixed with ISD$V_.

[3]If the /NODEMAND_ZERO qualifier is specified, the copy-on-reference (CRF) attribute is set instead of the DZRO attribute.

‡Alpha specific

For example, Table 3–4 summarizes the settings of the significant attributes of the program sections in the module MYADD.OBJ. (Because this is an OpenVMS VAX object module, the MOD attribute is not considered.)

Table 3–4 Significant Attributes of Program Sections in MYSUB_CLUS Cluster

	Writability	Executability	Protected Vector
$CODE	NOWRT	EXE	NOVEC
DATA	WRT	NOEXE	NOVEC
$CHAR_STRING_CONSTANTS	WRT	NOEXE	NOVEC

The linker puts both the DATA and $CHAR_STRING_CONSTANTS program sections in the same image section because they both have the same settings of significant attributes. Within the image section, the linker organizes the program sections alphabetically, so the $CHAR_STRING_CONSTANTS program section appears before the DATA program section. The linker creates a separate image section for the $CODE program section.

The linker performs similar processing of the program sections in the default cluster. The Image Section Synopsis section of the map file lists the clusters the linker created and lists the image sections it created for each cluster. This section also describes the layout of the image in memory, including the base address of each image section. Example 3–6 illustrates an excerpt of the Image Section Synopsis section from the map file produced with the sample link. The listing includes clusters for contributions for the VAX C Run-Time Library. (For more information about the image section synopsis section of a map file, see Section 5.2.3.)

Example 3–6 Image Section Information in a Map File

```
                                        +------------------------+
                                        ! Image Section Synopsis !
                                        +------------------------+

   Cluster      Type Pages   Base Addr  Disk VBN PFC Protection and Paging      ...
   -------      ---- -----   ---------  -------- --- ---------------------

MYSUB_CLUS         0     1    00000200        2   0  READ WRITE   COPY ON REF
                   0     1    00000400        3   0  READ ONLY

VAXCRTL            3     4    00000000-R      0   0  READ ONLY
                   3     1    00000800-R      0   0  READ ONLY
                   4     1    00000A00-R      0   0  READ WRITE   COPY ON REF
                   3    17    00000C00-R      0   0  READ ONLY
                   3   142    00002E00-R      0   0  READ ONLY
                   4    21    00014A00-R      0   0  READ WRITE   COPY ON REF
                   4     1  P-00017400-R      0   0  READ WRITE   COPY ON REF
                   2     3    00017600-R      0   0  READ WRITE   FIXUP VECTORS

LIBRTL             3   193    00000000-R      0   0  READ ONLY
                   4     8    00018200-R      0   0  READ WRITE   DEMAND ZERO

MTHRTL             3   335    00000000-R      0   0  READ ONLY
                   2     1    00029E00-R      0   0  READ WRITE   FIXUP VECTORS

DEFAULT_CLUSTER    0     1    00000600        4   0  READ WRITE   COPY ON REF
                   0     1    00000800        5   0  READ ONLY
                   0     1    00000A00        6   0  READ WRITE   FIXUP VECTORS
                 253    20    7FFFD800        0   0  READ WRITE   DEMAND ZERO
```

To find out which program sections the linker placed in each image section, look at the Program Section Synopsis section of the map file. This section lists all the program sections in each cluster and lists the contributions (the number of bytes) to each program section from each object module. By comparing the base-address of the program sections with the base-addresses of the image sections in the Image Section Synopsis section, you can tell in which image section the program sections appear. Example 3–7 is an excerpt from the Program Section Synopsis section of the map file produced by the sample link operation. (For more information about the program synopsis section of a map file, see Section 5.2.4.)

Example 3–7 Program Section Information in a Map File (VAX Example)

```
                                      +--------------------------+
                                      ! Program Section Synopsis !
                                      +--------------------------+

Psect Name Module Name   Base     End      Length        Align      Attributes
---------- -----------   ----     ---      ------        -----      ----------

$DATA                  00000200 00000203 00000004 (  4.) LONG 2 PIC,USR,CON . . .
           MYSUB       00000200 00000203 00000004 (  4.) LONG 2

GLOBAL_DATA            00000204 00000207 00000004 (  4.) LONG 2 PIC,USR,OVR . . .
           MYSUB       00000204 00000207 00000004 (  4.) LONG 2
           MYTEST      00000204 00000207 00000004 (  4.) LONG 2

$CODE                  00000400 0000040B 0000000C ( 12.) LONG 2 PIC,USR,CON . . .
           MYSUB       00000400 0000040B 0000000C ( 12.) LONG 2

$CHAR_STRING_CONSTANTS 00000600 0000062D 0000002E ( 46.) LONG 2 PIC,USR,CON . . .
           MYTEST      00000600 00000623 00000024 ( 36.) LONG 2
           MYADD       00000624 0000062D 0000000A ( 10.) LONG 2
```

(continued on next page)

Example 3–7 (Cont.) Program Section Information in a Map File (VAX Example)

```
$DATA                   00000630 00000637 00000008 (  8.) LONG 2 PIC,USR,CON . . .
          MYTEST        00000630 00000633 00000004 (  4.) LONG 2
          MYADD         00000634 00000637 00000004 (  4.) LONG 2

$CODE                   00000800 00000858 00000059 ( 89.) LONG 2 PIC,USR,CON . . .
          MYTEST        00000800 0000083E 0000003F ( 63.) LONG 2
          MYADD         00000840 00000858 00000019 ( 25.) LONG 2
   .
   .
   .
```

3.3.4 Allocating Memory for Image Sections

When it creates an image section, the linker allocates enough memory for the image section to accommodate all the program sections it contains. Each program section definition includes its size.

The linker aligns image sections on CPU-specific page boundaries. Within an image section, the linker assigns to each program section a virtual address relative to the base address of the image section.

Concatenated Program Sections

If the program sections have the concatenated (CON) attribute set, the linker positions the program sections one after the other within an image section, inserting padding bytes between the program sections if necessary to achieve the alignment requirement of a particular contribution to a program section. The linker retains the alignment specified for each program section contribution but uses the largest alignment of a contributing module as the alignment of the whole program section.

Overlaid Program Sections

If the program sections have the overlaid (OVR) attribute set, the linker uses the same start address for the program sections so that they occupy the same virtual memory (that is, the program sections overlay each other). For overlaid program sections, the linker allocates enough space to accommodate the largest of all the program section contributions. Note that the linker does *not* generate a warning message if the contributions specify different size allocations.

Any module can initialize the contents of an overlaid program section. However, the final contents of the program section are determined by the last contributing module. Therefore, the order in which you specify the input modules is important.

Assigning Virtual Addresses

The linker keeps track of free (available) virtual addresses by maintaining a free virtual memory list. For each cluster, the linker determines the number of pages required, searches the list beginning at the lowest virtual address for a contiguous number of pages large enough to contain the cluster, allocates those addresses to the cluster, then removes those addresses from the list.

The linker allocates virtual memory to the first cluster beginning at a page size boundary for executable images in the P0 region of the user's virtual address space, unless the cluster is based, in which case it allocates virtual memory beginning at the specified address. For VAX linking, the default is 512 (200 hexadecimal). However, you can specify the page size using the /BPAGE qualifier. (For information about the /BPAGE qualifier, see Part II.)

On its first pass through the cluster list, the linker allocates virtual addresses to any based user clusters or based shareable image clusters on the cluster list, removing the allocated addresses from the free virtual memory list as it proceeds. On its second pass, it repeats this procedure for nonbased user clusters. (Remember that nonbased shareable image clusters will have memory allocated for them at run time.)

Because the linker processes clusters in the order of their appearance on the cluster list, the virtual address space of the final image will generally contain contiguous image sections of consecutive clusters on the basis of their order in the cluster list. The presence of based clusters, however, may prevent such an outcome, and for this reason they are not recommended.

After allocating memory for a cluster, the linker relocates its contents by performing the following processing:

1. **Relocating each image section**. The linker adds the starting virtual address of the cluster to the relative offset of the image section from the cluster base and places the result in the appropriate field of the image section descriptor (ISD).
2. **Relocating each program section in the image section**. The linker adds the newly calculated starting virtual address of the image section to the relative offset of the program section from the base of the image section.
3. **Relocating each global symbol in the program section**. The linker adds the newly calculated program section virtual address to the relative offset of the global symbols from the base of the program section.

3.3.5 Image Section Attributes

When it creates image sections, the linker assigns attributes to the image section based on the attributes of the program sections it contains. The image section attributes describe certain characteristics of the portion of memory they represent, for example, the protection characteristics. For example, an image section that contains program sections with the writability attribute also has the writability attribute set. Table 3–2 and Table 3–3 include the image section attributes associated with an image section that contains program sections with a particular set of attributes. Table 3–5 lists all the image section attributes. Image section attributes, like program section attributes, are Boolean values that are either on or off.

Table 3–5 Image Section Attributes

Attribute	Symbol	Function
Global	[E]ISD$M_GBL	GBL is set when the ISD came from a shareable image. On both VAX and Alpha systems, the first ISD of a shareable image is included in the base image for use by the image activator. For VAX linking, if the shareable image is based, all of its ISDs are included in the image being linked.
Copy On Reference	[E]ISD$M_CRF	CRF is set whenever the psect attributes are WRT and not SHR. CRF is also set by the linker whenever it creates fix-ups to the section (which require the image activator to write to it).

(continued on next page)

Table 3–5 (Cont.) Image Section Attributes

Attribute	Symbol	Function
Demand Zero	[E]ISD$M_ DZRO	Demand zero is set for VAX linking for executable images if: • The section was never written to with a TIR (Text and Information Relocation) command. • The section resulted from compression of empty pages from an existing section. Demand zero is set for Alpha executable and Alpha shareable images if the user has not specified /NODEMAND_ZERO and if: • The section was never written to with an ETIR command. • The program sections in the section have the NOMOD bit set. DZRO is always set for stack ISDs on both VAX images and Alpha images.
Executability	[E]ISD$M_EXE	The EXE attribute is inherited from the program section.
Write	[E]ISD$M_WRT	The WRT attribute is inherited from the program section. WRT is also set by the linker if fix-ups are made to the section. When this is done, the linker also generates a change protection fix-up so that the image activator can change the protection back to NOWRT after the fix-up is applied.
Match Control	ISD$M_ MATCHCTL	This is used only for VAX images. It is not an attribute. MATCHCTL is a 3-bit field inside the flags field. It contains the match control bits. For Alpha images, this information is contained in a completely separate field.
Last Cluster	[E]ISD$M_ LASTCLU	LASTCLU is set only for sections in executable images. LASTCLU indicates that an image section was generated off of the last cluster (which was not a shareable image cluster) in the cluster list. If FIXUPVEC is set, LASTCLU is clear.
Initial Code	[E]ISD$M_ INITALCODE	This attribute is reserved by Digital.
Based	[E]ISD$M_ BASED	BASED indicates that the section is based. This is set when BASE= is specified in the options file. This attribute may also be set if based shareable images are encountered during linking. This attribute is present but not used for Alpha linking.
Fix-Up Vectors	[E]ISD$M_ FIXUPVEC	FIXUPVEC marks the section that contains the image activator fix-ups. This section is created by the linker. The attribute cannot be set by the user.
Resident	[E]ISD$M_ RESIDENT	This attribute is reserved by Digital.
Vectored	[E]ISD$M_ VECTOR	VECTOR indicates a vectored section, either a message section or a privileged library vector.
Protected	[E]ISD$M_ PROTECT	Protect indicates that a section is protected. The linker sets the PROTECT attribute whenever VECTOR is set. PROTECT is also set if the /PROTECT qualifier is used, or if the cluster that the section is spawned from came after a PROTECT=YES option (and before a PROTECT=NO option).

The linker uses type designations instead of image section attributes to propagate the SHR and PIC program section attributes. The linker assigns the type designation [E]ISD$K_NORMAL for image sections in executable images. Image sections in shareable images can be any of the following types:

Image Section Type	Attribute Settings
Share fixed ([E]ISD$K_SHRFXD)	SHR,NOPIC
Private fixed ([E]ISD$K_PRVFXD)	NOSHR,NOPIC
Share position-independent ([E]ISD$K_SHRPIC)	SHR,PIC
Private position-independent ([E]ISD$K_PRVPIC)	NOSHR,PIC

The Image Section Synopsis section of a map file lists the attributes of each image section created in the Protection and Paging column. See Example 3–6 for an illustration. You can also get a listing of all the image sections created by the linker by using the ANALYZE/IMAGE utility. The output generated by this utility includes a list of all the image sections that make up the image, with their attributes. Example 3–8 is an excerpt from the analysis of the image file MYTEST.EXE.

Example 3–8 Image Section Descriptions in an ANALYZE/IMAGE Display

```
Image Section Descriptors (ISD)

 1)❶  image section descriptor (16 bytes)
   ❷ page count: 1
   ❸ base virtual address: %X'00000200' (P0 space)
   ❹ page fault cluster size: default
   ❺ IS flags:
    (0)  ISD$V_GBL        0
    (1)  ISD$V_CRF        1
    (2)  ISD$V_DZRO       0
    (3)  ISD$V_WRT        1
    (7)  ISD$V_LASTCLU    0
    (8)  ISD$V_INITALCODE 0
    (9)  ISD$V_BASED      0
    (10) ISD$V_FIXUPVEC   0
    (11) ISD$V_RESIDENT   0
    (17) ISD$V_VECTOR     0
    (18) ISD$V_PROTECT    0
   ❻ section type: ISD$K_NORMAL
   ❼ base VBN: 2
                        .
                        .
                        .
 9)  image section descriptor (31 bytes)
  page count: 193
  base virtual address: %X'00000000' (P0 space)
  page fault cluster size: default
  IS flags:
   (0)  ISD$V_GBL        1
   (1)  ISD$V_CRF        0
   (2)  ISD$V_DZRO       0
   (3)  ISD$V_WRT        0
   (7)  ISD$V_LASTCLU    0
   (8)  ISD$V_INITALCODE 0
   (9)  ISD$V_BASED      0
   (10) ISD$V_FIXUPVEC   0
   (11) ISD$V_RESIDENT   0
   (17) ISD$V_VECTOR     0
   (18) ISD$V_PROTECT    0
```

(continued on next page)

Example 3–8 (Cont.) Image Section Descriptions in an ANALYZE/IMAGE Display

```
section type: ISD$K_SHRPIC
base VBN: 0
❽ global section major id: %X'01', minor id: %X'00000E'
❾ match control: ISD$K_MATLEQ
❿ global section name: "LIBRTL_001"
```

The items in the following list correspond to the numbers in Example 3–8:

❶ The size of the image section descriptor.

❷ The size of the image section, expressed in pages. For Alpha images, the value is expressed in bytes.

❸ The start address assigned to the image section by the linker. Note that this address is an offset from the beginning of the image, which is assumed to start at virtual address zero. (The linker always inserts an empty page at the beginning of every executable image.) Note also that the linker does not assign a start address for image sections representing shareable images because this information cannot be determined until run time, when the shareable image is loaded into memory by the image activator.

❹ The number of pagelets that should be mapped in when the initial page fault occurs. You can set this value by using the CLUSTER= option.

❺ The settings of image section attributes. Table 3–5 lists these attributes.

❻ The type of image section, based on the combination of image section attributes.

❼ The virtual block in the image file at which the image section begins.

❽ Image sections that represent shareable images include the global section identification number, which specifies the identification number of the shareable image.

❾ Image sections that represent shareable images also include a match control field that identifies the match control algorithm the image activator should apply to the global image section identification number when it activates the shareable image this ISD describes.

❿ Image sections that represent shareable images include the global section name field, which is the name of the shareable image. The ‘‘_001"’’ is appended to the name by the linker to indicate which ISD in the image this represents.

3.3.6 Controlling Image Section Creation

You can control how the linker combines program sections into image sections in the following ways:

- By modifying the attributes of program sections
- By putting object modules into named clusters
- By using the SOLITARY attribute

3.3.6.1 Modifying Program Section Attributes

The linker combines program sections in the same cluster into the same image section if they have the same settings for the significant program section attributes. To force the linker to put the program sections into different image sections, change the attributes of one of the program sections by using the PSECT_ATTR= option.

For example, in the sample link operation, the DATA program section and the $CHAR_STRING_CONSTANTS program section are combined into the same image section. If you want the $CHAR_STRING_CONSTANTS program section to appear in a different image section, change one of the significant attributes. For example, in the following link of the sample programs, the writability attribute is set to NOWRT. (For Alpha linking, you do not need to explicitly specify the C run-time library in the link operation because it resides in the default system shareable image library [IMAGELIB.OLB], which the linker processes by default.)

```
$ LINK/MAP/FULL MYTEST,MYADD,SYS$INPUT/OPT
CLUSTER=MYSUB_CLUS,,,MYSUB
PSECT_ATTR=$CHAR_STRING_CONSTANTS,NOWRT
SYS$LIBRARY:VAXCRTL/SHARE
Ctrl/Z
```

Example 3–9 presents an excerpt from the Image Section Synopsis section of the map file produced by this link. Note that the default cluster contains one additional image section, a read-only image section beginning at virtual address 0x00000600, than the default cluster in the original link, illustrated in Section 3.3.1.

Example 3–9 Image Section Synopsis of Second Link

```
   Cluster    Type Pages Base Addr Disk VBN PFC Protection and Paging . . .
   -------    ---- ----- --------- -------- --- --------------------
                .
                .
                .
DEFAULT_CLUSTER 0    1    00000600      4    0  READ ONLY
                0    1    00000800      0    0  READ WRITE   DEMAND ZERO
                0    1    00000A00      5    0  READ ONLY
                0    1    00000C00      6    0  READ WRITE   FIXUP VECTORS
              253   20    7FFFD800      0    0  READ WRITE   DEMAND ZERO
                .
                .
                .
```

3.3.6.2 Manipulating Cluster Creation

In general, the linker creates image sections on a per-cluster basis; that is, only program sections within a particular cluster can contribute to image section creation. (The linker can collect program sections with the global attribute from all clusters into a single image section.) To ensure that a program section appears in a particular image section, put the program section in a specific cluster.

For example, in the sample link operation illustrated in Example 3–5, the linker puts all the program sections in the object module MYSUB.OBJ in the cluster named MYSUB_CLUS because the CLUSTER= option is specified. If you wanted to group all of the program sections that contain code from all the other clusters into the MYSUB_CLUS cluster, you could specify the COLLECT= option, as in the following example. (By convention, VAX language processors put the code

they generate into program sections named $CODE. Program section naming conventions are architecture specific.)

```
$ LINK/MAP/FULL MYTEST, MYADD, SYS$INPUT/OPT
CLUSTER=MYSUB_CLUS,,,MYSUB
COLLECT=MYSUB_CLUS,$CODE
SYS$LIBRARY:VAXCRTL/SHARE
Ctrl/Z
```

3.3.6.3 Isolating a Program Section into an Image Section

You can specify that the linker place a particular program section into its own image section. This can be useful for programs that map data into predefined locations within an image.

To isolate a program section into an image section, specify the SOLITARY attribute of the program section using the PSECT_ATTR= option. For example, to isolate the GLOBAL_DATA program section in the sample link into its own image section, specify the following:

```
$ LINK/MAP/FULL MYTEST,MYADD,SYS$INPUT/OPT
CLUSTER=MYSUB_CLUS,,,MYSUB
PSECT_ATTR=GLOBAL_DATA,SOLITARY
Ctrl/Z
```

Alpha

For Alpha linking, when mapping data into an existing location in the virtual memory of your program using the Create and Map Global Section ($CRMPSC) system service or the Map Global Section ($MGBLSC) system service, you must specify an address range (in the **inadr** argument) that is aligned on a CPU-specific page boundary. Because the linker aligns image sections on CPU-specific page boundaries and the program section in which the section is to be mapped is the only program section in the image section, you ensure that the start address of the location is page aligned. In addition, because Alpha systems must map at least an entire page of memory at a time, using the SOLITARY attribute allows you to ensure that no other data in the image section is inadvertently overwritten by the mapping. By default, the linker creates the next image section on the next page boundary so that no data can be overwritten. ♦

3.4 Initializing an Image

After allocating memory for the image, the linker initializes the image by writing the binary contents of the image sections by processing text information and relocation (TIR) records in the object modules. These records direct the linker in the initialization of the image section by telling it what to store in the image section buffers. In addition, the linker inserts the addresses of symbols within the image wherever they are referenced.

3.4.1 Writing the Binary Contents of Image Sections

A TIR record contains object language commands, such as stack and store commands. Stack commands direct the linker to put information on its stack, and store commands direct the linker to write the information from its stack to the buffer for that image section.

During this image section initialization, the linker keeps track of the program section being initialized and the image section to which it has been allocated. The first attempt to initialize part of an image section by storing nonzero data causes the linker to allocate a buffer in its own program region to contain the binary contents of the generated image section. This allocation is achieved by the Expand Region ($EXPREG) system service, and it requires that the linker have

available a virtually contiguous region of its own memory at least as large as the image section being initialized.

A buffer is not allocated for an image section until the linker executes a store command (with nonzero data) within that image section.

Debugger information (DBG) records and traceback information (TBT) records are processed only if the debugger was requested and traceback information was not excluded by the /NOTRACE qualifier in the LINK command. Otherwise, these records are ignored. The records contain stack and store object language commands (TIR records), but they are stored in the debugger symbol table (DST) instead of in an image section. (The linker expands its memory region to accommodate the DST, unless the /NOTRACEBACK qualifier was specified in the LINK command.)

When the linker processes end-of-module (EOM) records, it checks that its internal stack has been collapsed to its initial state. When this processing is complete, the linker has written the binary contents of all image sections to image section buffers in its own address space.

The linker writes the contents of its buffers in the following order:

1. All image sections to the image file.
2. The debugger symbol table to the image file, unless /NOTRACEBACK was specified in the LINK command.
3. The remaining sections of the map to the map file, if requested in the LINK command. (These sections include all requested sections except the Object Module Synopsis, which it already wrote, and the Link Run Statistics, which it cannot write until the linking operation finishes.)
4. The global symbol table to the image file, and also to another separate file if requested in the LINK command.
5. The image header to the image file.
6. The link statistics to the map file, if requested in the LINK command.

3.4.2 Fixing Up Addresses

Executable images and based images are loaded into memory at a known location in P0 space. The linker cannot know where in memory a shareable image will be located when it is loaded into memory at run time by the image activator. Thus, the linker cannot initialize references to symbols within the shareable image from external modules or to internal symbolic references within the shareable image itself. For shareable images, the linker creates **fix-ups** that the image activator must resolve when it activates the images at run time.

The linker uses the fix-up image section to do the following:

- It adjusts the values stored by any .ADDRESS directives that are encountered during the creation of the nonbased shareable image. This action, together with subsequent adjustment of these values by the image activator, preserves the position independence of the shareable image.

VAX

- For VAX linking, it processes all general-address-mode code references to targets in position-independent shareable images. In this way, it creates the linkage between these code references and their targets, whose locations are not known until run time. ♦

3.4.3 Keeping the Size of Image Files Manageable

Because neither language processors nor the linker initialize data areas in a program with zeros, leaving this task to the operating system instead, some image sections may contain uninitialized pages. To keep the size of the image file as small as possible, the linker does not write pages of zeros to disk for these uninitialized pages. Instead, the linker searches image sections that contain initialized data for groups of contiguous, uninitialized pages and creates demand-zero image sections out of these pages (called **demand-zero compression**). Demand-zero image sections reduce the size of the image file and enhance the performance of the program. At run time, when a reference is made that initializes the section, the operating system initializes the allocated page of physical memory with zeros (hence the name ''demand-zero'').

Note that the linker creates demand-zero image sections only for executable images.

Alpha

The Alpha compilers identify to the linker program sections that have not been initialized by setting the NOMOD attribute of the program section. The linker groups these uninitialized program sections into a demand-zero image section.

If two modules contribute to the same program section and one contribution has the NOMOD attribute set and the other does not, the linker performs a logical AND of the NOMOD bits so that the two contributions end up in the same (non-demand-zero) image section.

The linker can create demand-zero image sections for both executable and shareable images. Program sections with the SHR and the NOMOD attributes set are not sorted into demand-zero image sections in shareable images.♦

3.4.3.1 Controlling Demand-Zero Image Section Creation

When performing demand-zero compression, by default the linker searches the pages of existing image sections looking for the default minimum of contiguous, uninitialized pages. You can specify a different minimum by using the DZRO_MIN= option. For more information about the effect of this option on image size and performance, see the description of the DZRO_MIN= option in Part II.

You can control demand-zero compression by specifying the maximum number of image sections that the linker can create using the ISD_MAX= option.

4

Creating Shareable Images

This chapter describes how to create shareable images and how to declare universal symbols in shareable images.

4.1 Overview

To create a shareable image, specify the /SHAREABLE qualifier on the LINK command line. You can specify as input files in the link operation any of the types of input files accepted by the linker, as described in Chapter 1.

Note, however, to enable other modules to reference symbols in the shareable image, you must declare them as universal symbols. High- and mid-level languages do not provide semantics to declare universal symbols. You must declare universal symbols at link time using linker options. The linker lists all universal symbols in the global symbol table (GST) of the shareable image. The linker processes the GST of a shareable image specified as an input file in a link operation during symbol resolution. (For more information about symbol resolution, see Chapter 2.)

VAX

For VAX linking, you declare universal symbols by listing the symbols in a UNIVERSAL= option statement in a linker options file. You can create shareable images that can be modified, recompiled, and relinked without causing the images that were linked against previous versions of the shareable image to be relinked. To provide this upward compatibility, you must create a **transfer vector** that contains an entry for each universal symbol in the image. For more information about these topics, see Section 4.2. ♦

Alpha

For Alpha linking, you declare universal symbols by listing the symbols in a SYMBOL_VECTOR= option statement in a linker options file. You do not need to create a transfer vector to create an upwardly compatible shareable image. The symbol vector can provide upward compatibility. For more information about this topic, see Section 4.3 ♦

The linker supports qualifiers and options that control various aspects of shareable image creation. Table 4–1 lists these qualifiers and options. (For more information about linker qualifiers and options, see Part II.)

Table 4–1 Linker Qualifiers and Options Used to Create Shareable Images

Qualifier	Description
‡/GST	For Alpha images, directs the linker to include universal symbols in the global symbol table (GST) of the shareable image, which is the default. When you specify the /NOGST qualifier, the linker creates an empty GST for the image. See Section 4.3.3 for more information about using this qualifier to create run-time kits. Not supported for VAX images.
/PROTECT	Directs the linker to protect the shareable image from write access by user or supervisor mode.
/SHAREABLE	Directs the linker to create a shareable image, when specified in the link command line. When appended to a file specification in a linker options file, this qualifier identifies the input file as a shareable image.

Option	Description
GSMATCH=	Sets the major and minor identification numbers in the header of the shareable image and specifies the algorithm the linker uses when comparing identification numbers.
PROTECT=	When specified with the YES keyword in a linker options file, this option directs the linker to protect the clusters created by subsequent options specified in the options file. You turn off protection by specifying the PROTECT=NO option in the options file.
‡SYMBOL_TABLE=	For Alpha linking, when specified with the GLOBALS keyword, this option directs the linker to include in a symbol table file all the global symbols defined in the shareable image, in addition to the universal symbols. By default, the linker includes only universal symbols in a symbol table file associated with a shareable image (SYMBOL_TABLE=UNIVERSALS). Not supported for VAX linking.
‡SYMBOL_VECTOR=	For Alpha linking, specifies symbols in the shareable image that you want declared as universal. Not supported for VAX linking.
†UNIVERSAL=	For VAX linking, specifies symbols in the shareable image that you want declared as universal. Not supported for Alpha linking.

†VAX specific
‡Alpha specific

4.2 Declaring Universal Symbols in VAX Shareable Images

VAX

For VAX linking, you declare universal symbols by specifying the UNIVERSAL= option in an options file. List the symbol or symbols you want to be universal as an argument to the option. The symbols listed in a UNIVERSAL= option can represent procedures, relocatable data, or constants. For each symbol declared as universal, the linker creates an entry in the global symbol table (GST) of the image. At link time, when the linker performs symbol resolution, it processes the symbols listed in the GSTs of the shareable images included in the link operation.

To illustrate how to declare universal symbols, consider the programs in Example 4–1 and Example 4–2.

Example 4–1 Shareable Image Test Module (my_main.c)

```
#include <stdio.h>

extern int my_data;

globalref int my_symbol;

int mysub();

main()
{
   int num1, num2, result;

   num1 = 5;
   num2 = 6;

   result = mysub( num1, num2 );
   printf("Result= %d\n", result);
   printf("Data implemented as overlaid psect= %d\n", my_data);
   printf("Global reference data is= %d\n", my_symbol);
}
```

Example 4–2 Shareable Image (my_math.c)

```
int my_data = 5;

globaldef int my_symbol = 10;

myadd(value_1, value_2)
 int value_1;
 int value_2;
{
  int result;

  result = value_1 + value_2;
  return( result );
}
mysub(value_1,value_2)
 int value_1;
 int value_2;
{
 int result;

 result = value_1 - value_2;
 return( result );
}
mydiv( value_1, value_2 )
  int value_1;
  int value_2;
{
  int result;

  result = value_1 / value_2;
  return( result );
}
mymul( value_1, value_2 )
  int value_1;
  int value_2;
{
  int result;

  result = value_1 * value_2;
  return( result );
}
```

To implement Example 4–2 as a shareable image, you must declare the universal symbols in the image by using the following LINK command:

```
$ LINK/SHAREABLE MY_MATH, SYS$INPUT/OPT
PSECT_ATTR=my_data,NOSHR
UNIVERSAL=myadd
UNIVERSAL=mysub
UNIVERSAL=mymul
UNIVERSAL=mydiv
UNIVERSAL=my_symbol
Ctrl/Z
```

Note that the symbol `my_data` in Example 4–2 does not have to be declared universal because of the way in which VAX C implements it. Several Digital programming languages, including VAX C and VAX FORTRAN, implement certain external variables as program sections with the overlaid (OVR), global (GBL), and relocatable (REL) attributes. When the linker processes these object modules, it **overlays** the program sections so that the various object modules that reference the variable access the same virtual memory. Symbols implemented in this way are declared universal (appear in the GST of the image) by default.

In the sample link operation, the SHR attribute of the program section that implements the data symbol `my_data` is reset to NOSHR. If you do not reset the shareable attribute for program sections that are writable, you must install the shareable image to run the program. (The shareable attribute [SHR] determines whether multiple processes have shared access to the memory.)

The following example illustrates how to link the object module MY_MAIN.OBJ with the shareable image MY_MATH.EXE. Note that the LINK command sets the shareability attribute of the program section my_data to NOSHR, as in the link operation in which the shareable was created.

```
$ LINK  MY_MAIN, SYS$INPUT/OPT
MY_MATH/SHAREABLE
PSECT_ATTR=my_data,NOSHR
Ctrl/Z
```

4.2.1 Creating Upwardly Compatible Shareable Images (VAX Linking Only)

For VAX linking, you can create a shareable image that can be modified, recompiled, and relinked without causing the images that were linked against previous versions of the image to be relinked. To provide this upward compatibility, you must ensure that the values of relocatable universal symbols within the image remain constant with each relinking.

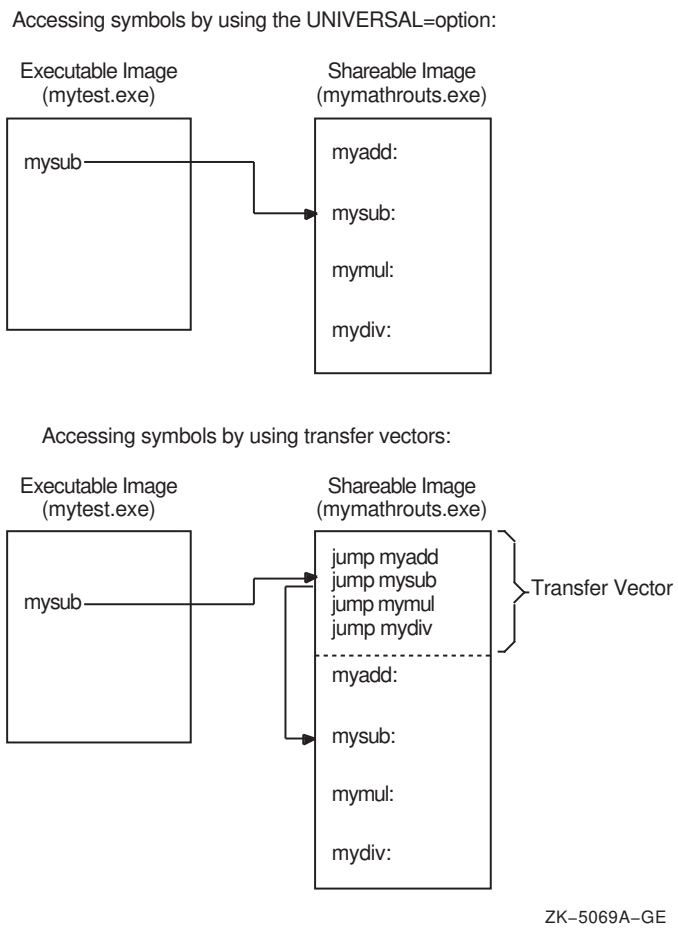
Universal Symbols that Represent Procedures

To fix the locations of universal symbols that represent procedures in a shareable image, create a **transfer vector** for the shareable image. In a transfer vector, you create small routines in VAX MACRO that define an entry point in the image and then transfer control to another location in memory. You declare the entry points defined in the transfer vector as the universal symbols and have each routine transfer control to the actual location of the procedures within the shareable image. As long as you ensure that the location of the transfer vector remains the same with each relinking, images that linked with previous versions of the shareable image will access the procedures at the locations they expect.

Figure 4–1 illustrates the flow of control at run time between a main image and a shareable image in which the actual routines are declared as universal symbols (as shown in Section 4.2) and between a main image and a shareable image in

which the transfer vector entry points are declared as universal symbols (as shown in Section 4.2.1.1).

Figure 4–1 Comparison of UNIVERSAL= Option and Transfer Vectors



Universal Symbols that Represent Data

To provide upwardly compatible symbols that represent data locations, you must also fix these locations within memory. You can accomplish this by allocating the data symbols at the end of the transfer vector file. In this way, when you fix the location of the transfer vector within an image, the data locations also remain the same. See Section 4.2.1.1 for more information.

4.2.1.1 Creating a Transfer Vector (VAX Linking Only)

You create a transfer vector using VAX MACRO. Specify the .TRANSFER directive because it declares the symbol that you specify as its argument as a universal symbol by default. Digital recommends the following coding conventions for creating a transfer vector:

```
❶ .transfer    FOO       ;Begin transfer vector to FOO
❷ .mask        FOO       ;Store register save mask
❸ jmp          L^FOO+2   ;Jump to routine
```

❶ The .TRANSFER directive causes the symbol, named FOO in the example, to be added to the shareable image's global symbol table. (You do not need to also specify the symbol in a UNIVERSAL= statement in a linker options file.)

❷ The .MASK directive causes the assembler to allocate 2 bytes of memory, find the register save mask accompanying the entry point (FOO in the example), and store the register save mask of the procedure. (According to the OpenVMS calling standard, procedure calls using the CALLS or CALLG instructions include a word, called the register save mask, whose bits represent which registers must be preserved by the routine.)

❸ The JMP instruction transfers control to the address specified as its argument. In the example, this address is two bytes past the routine entry point FOO (the first two bytes of the routine are the register save mask).

Digital recommends that you use a jump instruction (for example, JMP L^) in the transfer vector. Transfering control with a BSBW or JSB instruction results in saving the address of the next instruction from the transfer vector on the stack. In addition, the displacement used by the BSBW instruction must be expressible in 16 bits, which may not be sufficient to reach the target routine. Also, to avoid making the image position dependent, do not use an absolute mode instruction.

Note that the preceding convention assumes that the routine is called using the procedure call format, the default for most high-level language compilers. If a routine is called as a subroutine, using the JSB instruction, you do not need to include the .MASK directive. When creating a transfer vector for a subroutine call, Digital recommends adding bytes of padding to the transfer vectors. This padding makes a subroutine transfer vector the same size as a transfer vector for a procedure call. If you need to replace a subroutine transfer vector with a procedure call transfer vector, you can make the replacement without disturbing the addresses of all the succeeding transfer vectors.

The following example illustrates a subroutine transfer vector that uses the .BLKB directive to allocate the padding:

```
.TRANSFER    FOO      ;Begin transfer vector to FOO
JMP          L^FOO    ;Jump to routine
.BLKB        2        ;Pad vector to 8 bytes
```

To ensure upward compatibility, follow these guidelines when creating a transfer vector:

- Preserve the order and placement of entries in a transfer vector. Once you establish the order in which entries appear in a transfer vector, do not change it. Images that were linked against the shareable image depend on the location of the symbol in the transfer vector.

  You can reserve space within a transfer vector for future growth by specifying dummy transfer vector entries at various positions in a transfer vector.

- Add new entries to the end of a transfer vector. When including universal data in a transfer vector file, use padding to leave adequate room for future growth between the end of the transfer vector and the beginning of the list of universal data declarations.

Example 4–3 illustrates a transfer vector for the program in Example 4–2.

Example 4–3 Transfer Vector for the Shareable Image MY_MATH.EXE

(continued on next page)

Example 4–3 (Cont.) Transfer Vector for the Shareable Image MY_MATH.EXE

```
.transfer myadd
.mask  myadd
jmp  l^myadd+2
.transfer mysub
.mask  mysub
jmp  l^mysub+2
.transfer mymul
.mask  mymul
jmp  l^mymul+2
.transfer mydiv
.mask  mydiv
jmp  l^mydiv+2
.end
```

Assemble the transfer vector file to create an object module that can be included in a link operation:

```
$ MACRO MY_MATH_TRANS_VEC.MAR
```

4.2.1.2 Fixing the Location of the Transfer Vector in Your Image (VAX Linking Only)

For VAX linking, you include a transfer vector in a link operation as you would any other object module. However, to ensure upward compatibility, you must make sure that the transfer vector always appears in the same location in the image. The best way to accomplish this is to make the transfer vector always appear at the beginning of the image by forcing the linker to process it first. If you put the transfer vector file in a named cluster, using the CLUSTER= option, and specify it as the first option in an options file that can generate a cluster, the transfer vector will appear at the beginning of the file. (For more information about controlling cluster creation, see Section 2.3.)

The following example illustrates how to include the transfer vector in the link operation, using the CLUSTER= option, so that the linker processes it first:

```
$ LINK/SHAREABLE   MY_MATH, SYS$INPUT/OPT
❶ GSMATCH=lequal,1,1000
❷ CLUSTER=trans_vec_clus,,,MY_MATH_TRANS_VEC.OBJ
Ctrl/Z
```

❶ To enable images that linked against a shareable image to run with various versions of the shareable image, you must specify the identification numbers of the image. By default, the linker assigns a unique identification number to each version of a shareable image. At run time, if the ID of the shareable image as it is listed in the executable image does not match the ID of the shareable image the image activator finds to activate, the activation will abort. For information about using the GSMATCH= option to specify ID numbers, see the description of the GSMATCH= option in Part II.

❷ This CLUSTER= option causes the linker to create the named cluster TRANS_VEC_CLUS and to put the transfer vector file in this cluster.

4.2.2 Creating Based Shareable Images (VAX Linking Only)

For VAX linking, you can create a **based** shareable image by specifying the BASE= option in a linker options file. In a based image, you specify the starting address at which you want the linker to begin allocating memory for the image. For more information about the BASE= option, see Part II.

Digital does not recommend using based shareable images.

Based shareable Alpha images are not supported. ♦

4.3 Declaring Universal Symbols in Alpha Shareable Images

Alpha

For Alpha linking, you declare universal symbols by listing them in a SYMBOL_VECTOR= option. For each symbol listed in the SYMBOL_VECTOR= option, the linker creates an entry in the shareable image's symbol vector and creates an entry for the symbol in the shareable image's global symbol table (GST). When the shareable image is included in a subsequent link operation, the linker processes the symbols listed in its GST.

To implement Example 4–2 as an Alpha shareable image, you must declare the universal symbols in the image by using the following LINK command:

```
$ LINK/SHAREABLE  MY_MATH, SYS$INPUT/OPT
GSMATCH=lequal,1,1000
SYMBOL_VECTOR=(myadd=PROCEDURE,-
               mysub=PROCEDURE,-
               mymul=PROCEDURE,-
               mydiv=PROCEDURE,-
               my_symbol=DATA,-
               my_data=PSECT)
Ctrl/Z
```

You must identify the type of symbol vector entry you want to create by specifying a keyword. The linker allows you to create symbol vector entries for procedures, data (relocatable or constant), and for global data implemented as an overlaid program section.

A symbol vector entry is a pair of quadwords that contains information about the symbol. The contents of these quadwords depends on what the symbol represents. If the symbol represents a procedure, the symbol vector entry contains the address of the procedure entry point and the address of the procedure descriptor. If the symbol represents a data location, the symbol vector entry contains the address of the data location. If the symbol represents a data constant, the symbol vector entry contains the actual value of the constant.

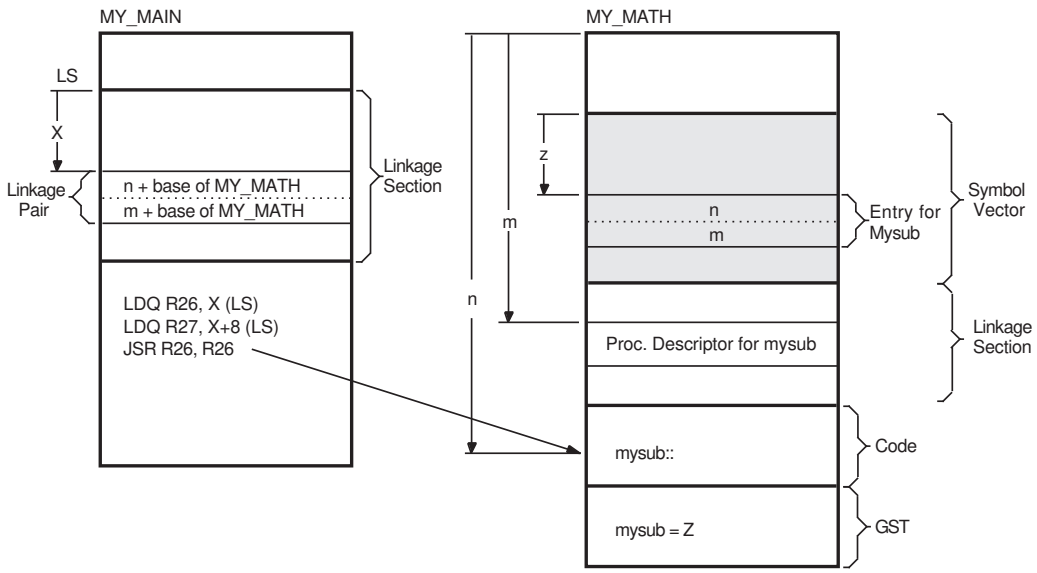
When you create the shareable image (by linking it specifying the /SHARE qualifier), the value of a universal symbol listed in the GST is the offset of its entry into the symbol vector (expressed as the offset *z* in Figure 4–2).

When you include this shareable image in a subsequent link operation, the linker puts this value in the linkage pair in the linkage section of the executable image that references the symbol. (A linkage pair is a data structure defined by the OpenVMS calling standard.)

At run time, when the image activator loads the shareable image into memory, it calculates the actual locations of the routines and relocatable data within the image and stores these values in the symbol vector. The image activator then fixes up the references to these symbols in the executable image that references symbols in the shareable image, moving the values from the symbol vector in the shareable image into the linkage section in the executable image. When

the executable image makes the call to the procedure, shown as the Jump-to-Subroutine (JSR) instruction sequence in Figure 4–2, control is transferred directly to the location of the procedure within the shareable image.

Figure 4–2 Accessing Universal Symbols Specified Using the SYMBOL_ VECTOR= Option



Note that, unlike VAX linking, global symbols implemented as overlaid program sections are not universal by default. Instead, you control which of these symbols is a universal symbol by including it in the SYMBOL_VECTOR= option, specifying the PSECT keyword. The example declares the program section `my_data` as a universal symbol. (Note that you must specify the qualifier /EXTERN_MODEL=COMMON on the compile command line to make the DEC C for OpenVMS Alpha compiler implement the symbol as an overlaid program section. For more information, see the DEC C for OpenVMS Alpha documentation.)

The name of a symbol implemented as an overlaid program section can duplicate the name of a symbol representing a procedure or data location. If the program section specified in a SYMBOL_VECTOR= option does not exist, the linker issues a warning, places zeros in the symbol vector entry, and does not create an entry for the program section in the image's GST.

4.3.1 Creating Upwardly Compatible Shareable Images (Alpha Linking Only)

The SYMBOL_VECTOR= option allows you to create upwardly compatible shareable images without requiring you to create transfer vectors as for VAX images.

However, as with transfer vectors, to ensure upward compatibility when using a SYMBOL_VECTOR= option, you must preserve the order and placement of the entries in the symbol vector with each relinking. Do not delete existing entries. Add new entries only at the end of the list. If you use multiple SYMBOL_

VECTOR= option statements in a single options file to declare the universal symbols, you must also maintain the order of the SYMBOL_VECTOR= option statements in the options file. If you specify SYMBOL_VECTOR= options in separate options files, make sure the linker always processes the options files in the same order. (The linker creates only one symbol vector for an image.)

Note, however, that there is no need to anchor the symbol vector at a particular location in memory, as you would anchor a transfer vector for a VAX link. The value at link time of a universal symbol in an Alpha shareable image is its location in the symbol vector, expressed as an offset from the base of the symbol vector, and the location of the symbol vector is stored in the image header. (For VAX linking, the value of a universal symbol at link time is the location of the symbol in the image, expressed as an offset from the base of the image.) Thus, the relative position of the symbol vector within the image does not affect upward compatibility.

4.3.2 Deleting Universal Symbols Without Disturbing Upward Compatibility (Alpha Linking Only)

To delete a universal symbol without disturbing the upward compatibility of an image, use the PRIVATE_PROCEDURE or PRIVATE_DATA keywords. In the following example, the symbol `mysub` is deleted using the PRIVATE_PROCEDURE keyword:

```
$ LINK/SHAREABLE   MY_MATH, SYS$INPUT/OPT
GSMATCH=lequal,1,1000
SYMBOL_VECTOR=(myadd=PROCEDURE,-
               mysub=PRIVATE_PROCEDURE,-
               mymul=PROCEDURE,-
               mydiv=PROCEDURE,-
               my_symbol=DATA,-
               my_data=PSECT)
Ctrl/z
```

When you specify the PRIVATE_PROCEDURE or PRIVATE_DATA keyword in the SYMBOL_VECTOR= option, the linker creates symbol vector entries for the symbols *but does not create an entry for the symbol in the GST of the image*. The symbol still exists in the symbol vector and none of the other symbol vector entries have been disturbed. Images that were linked with previous versions of the shareable image that reference the symbol will still work, but the symbol will not be available for new images to link against.

Using the PRIVATE_PROCEDURE keyword, you can replace an entry for an obsolete procedure with a private entry for a procedure that returns a message that explains the status of the procedure.

4.3.3 Creating Run-Time Kits (Alpha Linking Only)

If you use shareable images in your application, you may want to ship a run-time kit with versions of these shareable images that cannot be used in link operations.

To do this, you must first link your application, declaring the universal symbols in the shareable images using the SYMBOL_VECTOR= option so that references to these symbols can be resolved. After the application is linked, you must then relink the shareable images so that they have fully populated symbol vectors but empty global symbol tables (GSTs). The fully populated symbol vectors allow your application to continue to use the shareable images at run time. The empty GSTs prevent other images from linking against your application.

To create this type of shareable image for a run-time kit (without having to disturb the SYMBOL_VECTOR= option statements in your application's options files), relink the shareable image after development is completed, specifying the /NOGST qualifier on the LINK command line. When you specify the /NOGST qualifier, the linker builds a complete symbol vector, containing the symbols you declared universal in the SYMBOL_VECTOR= option, but does not create entries for the symbols that you declared universal in the GST of the shareable image. For more information about the /GST qualifier, see Part II.

4.3.4 Specifying an Alias Name for a Universal Symbol (Alpha Linking Only)

For Alpha linking, a universal symbol can have a name, called a **universal alias**, different from the name contributed by the object module in which it is defined. You specify the universal alias name when you declare the global symbol as a universal symbol using the SYMBOL_VECTOR= option. The universal alias name precedes the internal name of the global symbol, separated by a slash (/). In the following example, the global symbol mysub is declared as a universal symbol under the name sub_alias.

```
$ LINK/SHAREABLE MY_SHARE/SYS$INPUT/OPT
SYMBOL_VECTOR=(myadd=procedure,-
               sub_alias/mysub=procedure,-
               mymul=procedure,-
               mydiv=procedure,-
               my_symbol=DATA,-
               my_data=PSECT)
Ctrl/Z
```

You can specify universal alias names for symbols that represent procedures or data; you cannot declare a universal alias name for a symbol implemented as an overlaid program section. In link operations in which the shareable image is included, the calling modules must refer to the universal symbol by its universal alias name to enable the linker to resolve the symbolic reference.

In a privileged shareable image, calls *from within the image* that use the alias name result in a fix-up and subsequent vectoring through the privileged library vector (PLV), which results in a mode change. Calls from within the shareable image that use the internal name are done in the caller's mode. (Calls from external images always result in a fix-up.) For more information about creating a PLV, see the *OpenVMS Programming Concepts Manual*.

4.3.5 Improving the Performance of Installed Shareable Images (Alpha Linking Only)

For Alpha linking, you can improve the performance of an installed shareable image by installing it as a resident image (by using the /RESIDENT qualifier of the Install utility). INSTALL moves the executable, read-only pages of resident images into system space where they reside on huge pages. Executing your image in huge pages improves performance. See Section 1.4 for more information about installing shareable images as resident images. ♦

5
Interpreting an Image Map File

This chapter describes how to interpret the information returned in an image map and describes the combinations of linker qualifiers used to obtain a map.

5.1 Overview

At your request, the linker can generate information that describes the contents of the image and the linking process itself. This information, called an **image map**, can be helpful when locating link-time errors, studying the layout of the image in virtual memory, and keeping track of global symbols.

You can obtain the following types of information about an image from its image map:

- The names of all modules included in the link operation, both explicitly in the LINK command and implicitly from libraries
- The names, sizes, and other information about the image sections that comprise the image
- The names, sizes, and locations of program sections within an image
- The names and values of all the global symbols referenced in the image, including the name of the module in which the symbol is defined and the names of the modules in which the symbol is referenced
- Statistical summary information about the image and the link operation itself

You determine which information the linker includes in a map file by specifying qualifiers in the LINK command line. If you specify the /MAP qualifier, the map file includes certain information by default (called the **default map**). You can also request a map file that contains less information about the image (called a **brief map**) or a map file that contains more information about the image (called a **full map**). Table 5–1 lists the LINK command qualifiers that affect map file production.

Table 5–1 LINK Command Map File Qualifiers

/MAP	Directs the linker to create a map file. This is the default for batch jobs. /NOMAP is the default for interactive link operations.
/BRIEF	When used in combination with the /MAP qualifier, directs the linker to create a map file that contains only a subset of all the possible information.
/FULL	When used in combination with the /MAP qualifier, directs the linker to create a map file that contains all the possible information.
/CROSS_REFERENCE	When used in combination with the /MAP qualifier, directs the linker to replace the Symbols By Name section with a Symbol Cross-Reference section, in which all the symbols in each module are listed with the modules in which they are called. You cannot request this type of listing in a brief map file.

5.2 Components of an Image Map File

The linker formats the information it includes in a map file into sections. Table 5–2 lists the sections of a map file in the order in which they appear in the file. The table also indicates whether the section appears in a brief map, full map, or default map file.

Table 5–2 Image Map Sections

Section Name	Description	Default Map	Full Map	Brief Map
Object Module Synopsis[1]	Lists all the object modules in the image.	Yes	Yes	Yes
†Module Relocatable Reference Synopsis	Specifies the number of .ADDRESS directives in each module.	–	Yes	–
Image Section Synopsis	Lists all the image sections and clusters created by the linker.	–	Yes	–
Program Section Synopsis[1]	Lists the program sections and their attributes.	Yes	Yes	–
Symbols By Name[1]	Lists global symbol names and values.	Yes	Yes	–
Symbol Cross-Reference[1]	Lists each symbol name, its value, the name of the module that defined it, and the names of the modules that refer to it. Replaces the Symbols By Name section when the /CROSS_ REFERENCE qualifier is specified.	Yes	Yes	–
Symbols By Value	Lists all the symbols with their values (in hexadecimal representation).	–	Yes	–
Image Synopsis	Presents statistics and other information about the output image.	Yes	Yes	Yes
Link Run Statistics	Presents statistics about the link run that created the image.	Yes	Yes	Yes

[1]In a full map file, these sections include information about modules that were included in the link operation from libraries but were not explicitly specified on the LINK command line.

†VAX specific

The following sections describe each of the image map sections in detail. The examples of the map sections are taken from the map file created in a link operation of the executable image in Chapter 4.

5.2.1 Object Module Synopsis

The first section that appears in a map file is the Object Module Synopsis. This section lists the name of each module included in the link operation in the order in which it was processed. Note that shareable images included in the link operation are listed here as well. This section of the map file also includes other information about each module, arranged in columns, as in the following example:

```
                          +------------------------+
                          ! Object Module Synopsis !
                          +------------------------+

Module Name ❶  Ident ❷   Bytes ❸ File ❹                          Creation Date ❺   Creator ❻
-----------    -----     -----    -----                          -------------     -------
MY_MATH        V1.0          0 WORK:[PROGS]MY_MATH.EXE;11        14-AUG-1993 12:27  Linker T10-37
MY_MAIN        V1.0        553 WORK:[PROGS]MY_MAIN.OBJ;15        14-AUG-1993 12:27  DEC C X1.1-048E
DECC$SHR       V1.0          0 [SYSLIB]DECC$SHR.EXE;2             5-MAR-1993 07:49  Linker T10-03
SYS$PUBLIC_VECTORS
               X-26          0 [SYSLIB]SYS$PUBLIC_VECTORS.EXE;2   5-MAR-1993 07:34  Linker T10-03
```

❶ Module Name. The name of each object module included in the link operation. The modules are listed in the order in which the linker processed them. If the linker encounters an error during its processing of an object module, an error message appears on the line directly following the line containing the name of that object module.

❷ Ident. The text string in the IDENT field in an object module or in the image header of a shareable image.

❸ Bytes. The number of bytes the object module contributes to the image. Because shareable images are activated at run time, the linker cannot calculate the size of their contributions to the image. Thus, the value 0 (zero) is associated with shareable images.

❹ File. Full file specification of the input file, including device and directory. If the specification is longer than 35 characters, it is shortened by dropping the device portion of the file specification or both the device and directory portions of the file specification.

❺ Creation Date. The date and time the file was created.

❻ Creator. Identification of the language processor or other utility that created the file.

The order in which the modules are listed in this section reflects the order in which the linker processes the input files specified in the link operation. Note that the order of processing can be different from the order in which the files were specified in the command line. For more information about how the linker processes input files, see Chapter 2.

5.2.2 Module Relocatable Reference Synopsis (VAX Linking Only)

VAX

For VAX linking, the information contained in the Module Relocatable Reference Synopsis section varies with the type of image being created. For shareable images, this section lists all of the modules that contain at least one .ADDRESS directive. For executable or system images, this section lists the names of all object modules containing at least one .ADDRESS reference *to a shareable image*. The section lists the modules in the order in which the linker processes them,

including the number of .ADDRESS references found. The linker formats the information as in the following example:

```
                        +--------------------------------------+
                        ! Module Relocatable Reference Synopsis !
                        +--------------------------------------+

Module Name (1)       Number (2) Module Name          Number   Module Name                Number
-----------           ------     -----------          ------   -----------                ------

MAIN1                      1
```

(1) Module Name. The name of each object module included in the link operation. The modules are listed in the order in which the linker processed them.

(2) Number. The number of .ADDRESS references found.

Note that you can reduce linker and image activator processing time by removing .ADDRESS directives from input files. ♦

5.2.3 Image Section Synopsis Section

The Image Section Synopsis section of the linker map file lists the image sections created by the linker. The image sections appear in the order in which the linker created them, which is the same order as the clusters in the linker's cluster list. (For more information about clusters, see Chapter 2.) The section includes other information about these image sections, formatted in columns, as in the following example:

```
                                  +------------------------+
                                  ! Image Section Synopsis !
                                  +------------------------+

   (1)          (2)   (3)      (4)        (5)    (6)      (7)                    (8)              (9)       (10)     (11)
   Cluster      Type  Pglts    Base Addr  Disk VBN PFC    Protection and Paging  Global Sec. Name Match     Majorid  Minorid
   -------      ----  -----    ---------  -------- ---    ---------------------  ---------------- -----     -------  -------

MY_MATH           2     1      00000000-R       0      0  READ WRITE   COPY ON REF   MY_MATH_001      EQUAL     113     5598831
                  2     1      00010000-R       0      0  READ WRITE   COPY ON REF   MY_MATH_002      EQUAL     113     5598831
                  3     1      00020000-R       0      0  READ ONLY                  MY_MATH_003      EQUAL     113     5598831
                  4     1      00030000-R       0      0  READ WRITE   COPY ON REF   MY_MATH_004      EQUAL     113     5598831
                  2     1      00040000-R       0      0  READ WRITE   FIXUP VECTORS MY_MATH_005      EQUAL     113     5598831

DEFAULT_CLUSTER   0     1      00010000         3      0  READ WRITE   NON-SHAREABLE ADDRESS DATA
                  0     1      00020000         4      0  READ ONLY
                  0     1      00030000         5      0  READ WRITE   FIXUP VECTORS
                253    20      7FFF0000         0      0  READ WRITE   DEMAND ZERO

DECC$SHR          2   132      00000000-R       0      0  READ WRITE   COPY ON REF   DECC$SHR_001     LESS/EQUAL  1      0
                  2     4      00020000-R       0      0  READ WRITE   COPY ON REF   DECC$SHR_002     LESS/EQUAL  1      0
                  3    11      00030000-R       0      0  READ ONLY                  DECC$SHR_003     LESS/EQUAL  1      0
                  3   965      00040000-R       0      0  READ ONLY                  DECC$SHR_004     LESS/EQUAL  1      0
                  4     7      000C0000-R       0      0  READ WRITE   COPY ON REF   DECC$SHR_005     LESS/EQUAL  1      0
                  4    71      000D0000-R       0      0  READ WRITE   COPY ON REF   DECC$SHR_006     LESS/EQUAL  1      0
                  4     1    P-000E0000-R       0      0  READ WRITE   COPY ON REF   DECC$SHR_007     LESS/EQUAL  1      0
                  2     9      000F0000-R       0      0  READ WRITE   FIXUP VECTORS DECC$SHR_008     LESS/EQUAL  1      0

SYS$PUBLIC_VECTORS
                  2    15      00000000-R       0      0  READ ONLY                  SYS$PUBLIC_VECTO EQUAL     113     14651409
                  1    24      00004000-R       0      0  READ WRITE   COPY ON REF   SYS$PUBLIC_VECTO EQUAL     113     14651409
                  2     1      00008000-R       0      0  READ WRITE   FIXUP VECTORS SYS$PUBLIC_VECTO EQUAL     113     14651409

          Key for special characters above:
                +--------------------+
                ! R  - Relocatable   !
                ! P  - Protected     !
                +--------------------+
```

ZK–5292A–GE

The items in the following list correspond to the numbered items in the preceding figure:

(1) Cluster. The name of each cluster the linker created, listed in the order in which the linker created them.

❷ Type. The type of image section, expressed as one of the following codes:

Code	Image Section Type
1	Shareable fixed image section
2	Private fixed image section
3	Shareable position-independent image section
4	Private position-independent image section
253	Stack image section

For more information about the types of image sections the linker creates, see Section 3.3.5.

❸ Pages or pagelets. The length of each image section, expressed in pages or pagelets.

❹ Base Address. The base address assigned to the image section. Note that if the cluster is relocatable, the image activator relocates the base address. In this case, the base address entry for each image section in the cluster MY_MATH has the letter ''R'' appended to it, indicating that the base address entry is an offset to be added to the cluster base address assigned by the image activator.

Alpha

For Alpha linking, when images are installed as resident images, the Install utility moves image sections containing code into system space. This invalidates the base addresses listed for these image sections in this section of the map file. Note, however, that the relative positions of the program sections within the image section, listed in the Program Section Synopsis section of the map file, remain valid when the image section is moved into system space. ♦

❺ Disk VBN (virtual block number). The virtual block number of the image file on disk where the image section begins. The number 0 indicates that the image section is not in the image file.

❻ Page fault cluster (PFC). The number of pagelets read into memory by the operating system when the initial page fault occurs for that image section. The number 0 indicates that the system parameter PFCDEFAULT determines this value, rather than the linker.

❼ Protection and Paging. A keyword phrase that characterizes the settings of certain attributes of the image section, such as the attributes that affect protection and paging. The following table lists the keywords used by the linker to indicate these characteristics of an image section:

Keyword	Meaning
COPY ON REF	Indicates that the image section is a copy-on-reference image section. Because a copy-on-reference image section is readable and writable, but not shareable, each process receives a copy of it.
DEMAND ZERO	Indicates that the image section is a demand-zero image section. (For more information, see Section 3.4.3.)
EXECUTABLE	Indicates that the image section contains code.

Keyword	Meaning
FIXUP VECTORS	Indicates that the image section contains the fix-up section. There is always a change-protection fix-up for the fix-up section, so that when the image activator is done, the image activator changes the protection of the image section to READ ONLY.
NON-SHAREABLE ADDRESS DATA	Indicates that the linker set a READ ONLY page in the image section to WRITE so that the image activator can fix up address references (.ADDRESS) in the image section. The linker creates a change-protection fix-up for these image sections that causes the image activator to set the attributes of the image section back to READ ONLY when it finishes processing the address references.
READ ONLY	Indicates that the image section is protected against write access.
READ WRITE	Indicates that the image section allows both read and write access.

The linker may use more than one keyword to describe an image section. For example, to describe an image section that contains code, the linker uses the READ ONLY and EXECUTABLE keywords.

Note that a program section that you may have protected from write access (by setting the NOWRT program section attribute) may appear in the map file as writable (with the READ WRITE keyword). If this program section also has the NON-SHAREABLE ADDRESS DATA keyword (as the first image section in DEFAULT_CLUSTER illustrates), the linker has enabled write access to the program section to allow the image activator to fix up address references in the image section at run time. The image activator resets the program section attributes to READ ONLY after it is finished.

❽ Global Section Name. The name assigned by the linker to each image section comprising a shareable image. The linker creates the names by appending the characters ‘‘_00x’’ after the file name, where ‘‘x’’ is an integer, starting with 1, and incremented for each image section in a shareable image.

❾ Match. The algorithm the image activator uses when comparing identification numbers in a shareable image, expressed by the keyword LESS/EQUAL, EQUAL, or ALWAYS. For more information about this topic, see the description of the GSMATCH= option in Part II.

❿ Majorid. An identification number assigned to the image. The linker assigns the number to the image if it is not specified as part of the link operation in the GSMATCH= option.

⓫ Minorid. An identification number assigned to the image. The linker assigns the number to the image if it is not specified as part of the link operation in the GSMATCH= option.

5.2.4 Program Section Synopsis Section

The Program Section Synopsis section lists the program sections that comprise the image, with information about the size of the program section, its starting- and ending-addresses, and its attributes. The Module Name column in this section lists the modules that contribute to each program section. The following example illustrates this format:

```
                                        +--------------------------+
                                        ! Program Section Synopsis !
                                        +--------------------------+
Psect Name❶     Module Name❷     Base❸    End❹          Length❺          Align❻              Attributes❼
----------      -----------      ----     ---           ------           -----               ----------

$LINK$                          00010000 000100BF 000000C0 (       192.) OCTA  4 NOPIC,CON,REL,LCL,NOSHR,NOEXE,NOWRT,NOVEC,  MOD
                MY_MAIN         00010000 000100BF 000000C0 (       192.) OCTA  4

MY_DATA                         00010010 00010013 00000004 (         4.) OCTA  4 NOPIC,OVR,REL,GBL,NOSHR,NOEXE,  WRT,NOVEC,  MOD
                MY_MATH         00010010 00010010 00000000 (         0.) OCTA  4
                MY_MAIN         00010010 00010013 00000004 (         4.) OCTA  4

$LITERAL$                       000100C0 00010108 00000049 (        73.) OCTA  4   PIC,CON,REL,LCL,  SHR,NOEXE,NOWRT,NOVEC,  MOD
                MY_MAIN         000100C0 00010108 00000049 (        73.) OCTA  4

$READONLY$                      00010110 00010110 00000000 (         0.) OCTA  4 NOPIC,CON,REL,LCL,NOSHR,NOEXE,NOWRT,NOVEC,  MOD
                MY_MAIN         00010110 00010110 00000000 (         0.) OCTA  4

$BSS$                           00020000 00020000 00000000 (         0.) OCTA  4 NOPIC,CON,REL,LCL,NOSHR,NOEXE,  WRT,NOVEC,  MOD
                MY_MAIN         00020000 00020000 00000000 (         0.) OCTA  4

$DATA$                          00020000 00020000 00000000 (         0.) OCTA  4 NOPIC,CON,REL,LCL,NOSHR,NOEXE,  WRT,NOVEC,  MOD
                MY_MAIN         00020000 00020000 00000000 (         0.) OCTA  4

$CODE$                          00020000 0002011B 0000011C (       284.) OCTA  4   PIC,CON,REL,LCL,  SHR,  EXE,NOWRT,NOVEC,  MOD
                MY_MAIN         00020000 0002011B 0000011C (       284.) OCTA  4
```

ZK–5294A–GE

The items in the following list correspond to the numbered items in the preceding figure:

❶ Psect Name. The name of each program section in the image in ascending order of its base virtual address.

❷ Module Name. The names of the modules that contribute to the program section whose name appears on the line directly above in the Psect Name column. If a shareable image appears in this column, the linker processed the program section as a shareable image reference.

❸ Base. The starting virtual address of the program section or of a module that contributes to a program section.

❹ End. The ending virtual address of the program section or of a module that contributes to a program section.

❺ Length. The total length of the program section or of a module that contributes to a program section.

❻ Align. The type of alignment used for the entire program section or for an individual program section contribution. The alignment is expressed in two ways. In the first column, the alignment is expressed using a predefined keyword, such as OCTA. In the second column, the alignment is expressed as an integer that is the power of 2 that creates the alignment. For example, octaword alignment would be expressed as the keyword OCTA and as the integer 4 (because $2^4 = 16$).

If the linker does not support a keyword to express an alignment, it puts the text ‘‘2 **’’ in the column in which the keyword usually appears. When read with the integer in the second column, it expresses these alignments, such as 2 ** 5.

❼ Attributes. The attributes associated with the program section. For a list of all the possible attributes, see Chapter 3.

Alpha

For Alpha linking, the linker includes the MOD attribute in the list of program section attributes (as illustrated in the example). To make room in the display for this attribute, the linker leaves out the Readability (RD /NORD) and User Library (USR/LIB) attributes, which are reserved for future use. ♦

VAX

For VAX linking, the list of attributes includes the Readability (RD/NORD) and User Library (USR/LIB) attributes. The Modified (MOD/NOMOD) attribute, which is not supported for VAX images, is not included. ♦

Note that, if a routine is extracted from the default system library to resolve a symbolic reference, the Program Section Synopsis section in a full map contains information about the program sections comprising that routine. The Program Section Synopsis section in a default map does not.

5.2.5 Symbols By Name Section

The Symbols By Name section lists the global symbols contained in all the modules included in the link operation. The section includes the value of the symbol, in the following format:

```
                                +------------------+
                                ! Symbols By Name !
                                +------------------+

Symbol ❶       Value ❷      Symbol         Value           Symbol         Value          Symbol      Value
------         -----        ------         -----           ------         -----          ------      -----
DECC$EXIT       00001FD0-RX
DECC$GPRINTF    00001710-RX
DECC$MAIN       000007D0-RX
MAIN            00010000-R
MYSUB           00000010-RX
MY_SYMBOL       00000050-RX
SYS$IMGSTA      00000340-RX
__MAIN          00010078-R
```

❶ Symbol. The names of the image's global symbols in alphabetical order.

❷ Value. The value of the symbol, expressed in hexadecimal. The linker appends characters to the end of the symbol value to describe other characteristics of the symbol. For an explanation of these symbols, see Section 5.2.7.

Note that this section is replaced by the Symbol Cross-Reference section when you specify the /CROSS_REFERENCE qualifier in the LINK command. The Symbols by Value section, described in Section 5.2.7, lists the same symbols by value.

5.2.6 Symbol Cross-Reference Section

The Symbol Cross-Reference Section, which is produced in place of the Symbols By Name section when you specify the /CROSS_REFERENCE qualifier, lists all of the symbols referenced in the image, along with the module in which they are defined and with all the modules that reference them. The section formats this information as in the following example:

```
                                        +------------------------+
                                        ! Symbol Cross Reference !
                                        +------------------------+

Symbol ❶        Value ❷         Defined By ❸      Referenced By ... ❹
------          -----           ----------        -----------------
DECC$EXIT     00001FD0-RX       DECC$SHR                MY_MAIN
DECC$GPRINTF  00001710-RX       DECC$SHR                MY_MAIN
DECC$MAIN     000007D0-RX       DECC$SHR                MY_MAIN
MAIN          00010000-R        MY_MAIN
MYSUB         00000010-RX       MY_MATH                 MY_MAIN
MY_SYMBOL     00000050-RX       MY_MATH                 MY_MAIN
SYS$IMGSTA    00000340-RX       SYS$PUBLIC_VECTORS
__MAIN        00010078-R        MY_MAIN
```

❶ Symbol. The name of the global symbol.

❷ Value. The value of the global symbol, expressed in hexadecimal. The linker appends characters to the end of the symbol value to describe other characteristics of the symbol. For an explanation of these symbols, see Section 5.2.7.

❸ Defined By. The name of the module in which the symbol is defined. For example, the symbol `mysub` is defined in the module named MY_MATH.

❹ Referenced By.... The name or names of all the modules that contain at least one reference to the symbol.

5.2.7 Symbols By Value Section

The Symbols By Value section lists all the global symbols in the image in order by value, in ascending numeric order. The linker formats the information into columns, as in the following example:

```
                                          +------------------+
                                          ! Symbols By Value !
                                          +------------------+

Value ❶              Symbols...❷
-----                ----------
00000010     RX-MYSUB
00000050     RX-MY_SYMBOL
00000340     RX-SYS$IMGSTA
000007D0     RX-DECC$MAIN
00001710     RX-DECC$GPRINTF
00001FD0     RX-DECC$EXIT
00010000      R-MAIN
00010078      R-__MAIN
```

❶ Value. The value of each global symbol, expressed in hexadecimal, in ascending numerical order.

❷ Symbols... The names of the global symbols. If more than one symbol has the same value, the linker lists them on more than one line. The characters prefixed to the symbol names indicate other characteristics of the symbol, such as its scope. Table 5–3 lists these codes.

Table 5–3 Symbol Characterization Codes

Code	Meaning
asterisk(*)	Symbol is undefined.
‡A	Symbol is the alias name for a universal symbol.
‡I	Symbol is the internal name of a symbol that has a universal alias name.
U	Symbol is a universal symbol.
R	Symbol is a relocatable symbol.
X	Symbol is an external symbol.
WK	Symbol is a weak symbol. (For more information, see Chapter 2.)

‡Alpha specific

5.2.8 Image Synopsis Section

The Image Synopsis section contains miscellaneous information about the image, such as its name and identification numbers, and a summary of various attributes of the image, such as the number of files used to build the image. The following example illustrates the format of this section of a map file. The list following the example provides more information about items in this section that are not self-explanatory.

```
                                        +----------------+
                                        ! Image Synopsis !
                                        +----------------+

Virtual memory allocated:❶               00010000 0003FFFF 00030000 (196608. bytes, 384. pages)
Stack size:                                    20. pages
Image header virtual block limits:              1.        2. (    2. blocks)
Image binary virtual block limits:              3.        5. (    3. blocks)
Image name and identification:           MY_MAIN V1.0
Number of files:                                7.
Number of modules:                              4.
Number of program sections:                    11.
Number of global symbols:                     944.
Number of cross references:                    13.
Number of image sections:                      20.
User transfer address:                   00010078
Debugger transfer address:               00000340
Number of code references to shareable images:  6.
Image type:                              EXECUTABLE.
Map format:                              FULL WITH CROSS REFERENCE in file WORK:[PROGS]MY_MAIN.MAP;15
Estimated map length:                    148. blocks
```

The following list explains the information returned in each line of the Image Synopsis section:

❶ Virtual memory allocated. This line contains the following information:

- The starting-address of the image (base-address)
- The ending-address of the image
- The total size of the image, expressed in bytes, in hexadecimal radix

The numbers in parentheses at the end of the line indicate the total size of the image, expressed in bytes and in pagelets. Both these values are expressed in decimal.

5.2.9 Link Run Statistics Section

The Link Run Statistics section contains miscellaneous statistical information about the link operation, such as performance indicators, formatted as in the following example:

```
                                  +---------------------+
                                    ! Link Run Statistics !
                                    +---------------------+

Performance Indicators                      Page Faults   CPU Time        Elapsed Time
----------------------                      -----------   --------        ------------
    Command processing:                              93   00:00:00.18     00:00:00.81
    Pass 1:                                         345   00:00:00.55     00:00:12.04
    Allocation/Relocation:                            9   00:00:00.04     00:00:00.30
    Pass 2:                                          29   00:00:00.14     00:00:00.62
    Map data after object module synopsis:            3   00:00:00.05     00:00:00.31
    Symbol table output:                              0   00:00:00.00     00:00:00.10
Total run values:                                   479   00:00:00.96     00:00:14.18

Using a working set limited to 2048 pages and 946 pages of data storage (excluding image)

Total number object records read (both passes):   167
    of which 0 were in libraries and 0 were DEBUG data records containing 0 bytes

Number of modules extracted explicitly             = 0
    with 0 extracted to resolve undefined symbols

5 library searches were for symbols not in the library searched

A total of 0 global symbol table records was written

LINK/MAP/FULL/CROSS MY_MAIN,SYS$INPUT/OPT
my_math/share
```

Part II

LINK Command Reference

LINK

Invokes the OpenVMS Linker utility to link one or more input files into a program image and defines the execution characteristics of the image.

Format

LINK file-spec [,...]

Qualifiers	Defaults
/ALPHA	See reference description.
/BPAGE	See reference description.
/BRIEF	None
/[NO]CONTIGUOUS	/NOCONTIGUOUS
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG[=file-spec]	/NODEBUG
/[NO]DEMAND_ZERO	/DEMAND_ZERO (Alpha linking only)
/[NO]DSF[=file-spec]	/NODSF (Alpha linking only)
/[NO]EXECUTABLE[=file-spec]	/EXECUTABLE
/FULL	None
/[NO]GST	/GST (Alpha linking only)
/HEADER	None
/INCLUDE=(module-name[,...])	None
/[NO]INFORMATIONALS	/INFORMATIONALS
/LIBRARY	None
/[NO]MAP[=file-spec]	/NOMAP
/[NO]NATIVE_ONLY	/NATIVE_ONLY (Alpha linking only)
/OPTIONS	None
/P0IMAGE	None
/PROTECT	None
/[NO]REPLACE	/REPLACE (Alpha linking only)
/[NO]SECTION_BINDING=(CODE,DATA)	/NOSECTION_BINDING (Alpha linking only)
/SELECTIVE_SEARCH	None
/[NO]SHAREABLE[=file-spec]	/NOSHAREABLE
/[NO]SYMBOL_TABLE[=file-spec]	/NOSYMBOL_TABLE
/[NO]SYSEXE	/NOSYSEXE (Alpha linking only)
/[NO]SYSLIB	/SYSLIB
/[NO]SYSSHR	/SYSSHR
/[NO]SYSTEM[=base-address]	/NOSYSTEM
/[NO]TRACEBACK	/TRACEBACK
/[NO]USERLIBRARY[=(table[,...])]	/USERLIBRARY=ALL
/VAX	See reference description.

Parameters

file-spec [,...]
Specifies one or more input files (wildcard characters are not allowed). Input files may be object modules, shareable images, libraries to be searched for external references or from which specific modules are to be included, or options files to be read by the linker. Separate multiple input file specifications with commas (,) or plus signs (+). In either case, the linker creates a single image file.

If you omit the file type in an input file specification, the linker supplies default file types, based on the nature of the input file. For object modules, the default file type is .OBJ. For more information about specifying input files, see Chapter 1.

LINKER Qualifiers

Qualifier Descriptions

This section describes the LINK command qualifiers.

/ALPHA

Directs the linker to produce an OpenVMS Alpha image. The default action, when neither /ALPHA nor /VAX is specified, is to create an OpenVMS VAX image on an OpenVMS VAX system and to create an OpenVMS Alpha image on an OpenVMS Alpha system.

Format

/ALPHA

Qualifier Values

None.

Description

This qualifier is used to instruct the linker to accept OpenVMS Alpha object files and library files to produce an OpenVMS Alpha image.

You must inform the linker where OpenVMS Alpha system libraries and shareable images are located with the logical names ALPHA$LOADABLE_IMAGES and ALPHA$LIBRARY. On an OpenVMS Alpha system, these logicals are already defined to point to the correct directories on the current system disk. On OpenVMS VAX, you must define these logical names so that they translate to the location of an OpenVMS Alpha system disk residing on the system where the Alpha linking is to occur.

For more information on cross-architecture linking, see Section 1.6.

Example

```
$ DEFINE ALPHA$LIBRARY DKB100:[VMS$COMMON.SYSLIB]
$ DEFINE ALPHA$LOADABLE_IMAGES DKB100:[VMS$COMMON.SYS$LDR]
$ LINK/ALPHA ALPHA.OBJ
```

This example, which is performed on an OpenVMS VAX system, shows the definition of logical names to point to the appropriate areas on an OpenVMS Alpha system disk mounted on device DKB100. The qualifier /ALPHA tells the linker to expect the object file, ALPHA.OBJ, to be an OpenVMS Alpha object file and to link it using the OpenVMS Alpha libraries and images on DKB100, if necessary.

/BPAGE

Specifies the page size the linker should use when it creates the image sections that make up an image.

Format

/BPAGE [=page-size-indicator]

Qualifier Values

page-size-indicator
An integer that specifies a page size as the power of 2 required to create a page that size. For example, to get an 8 KB page size, specify the value 13 because 2^{13} equals 8K. The following table lists the page sizes supported by the linker with the defaults:

Value	Page Size	Defaults
9	512 bytes	Default value for VAX links when the /BPAGE qualifier is not specified.
13	8 KB	Default value for VAX links when the /BPAGE qualifier is specified without a value.
14	16 KB	–
15	32 KB	–
16	64 KB	Default value for Alpha links when /BPAGE is not specified or when the /BPAGE qualifier is specified without a value.

Description

The images the linker creates are made up of image sections that the linker allocates on page boundaries. When you specify a larger page size, the origin of image sections increases to the next multiple of that size.

An image linked to a page size that is larger than the page size of the CPU generally runs correctly, but it might consume more virtual address space.

VAX

For VAX linking, linking a shareable image to a larger page size can cause the value of transfer vector offsets to change if they were not allocated in page 0 of the image. Do not link against a shareable image that was created with a different page size. (You cannot determine the page size used in the creation of a VAX image from the image.) ♦

Alpha

For Alpha linking, by default, the linker creates image sections on 64 KB boundaries, thus allowing the images to run properly on any Alpha system, regardless of page size.♦

Example

```
$ LINK/BPAGE=16 MY_PROG.OBJ
```

Including the value 16 with the /BPAGE qualifier causes the linker to create image sections on 64 KB page boundaries.

/BRIEF

Directs the linker to produce a brief image map. For more information, see also the /MAP and /FULL qualifiers.

Format

/MAP/BRIEF

Qualifier Values

None.

Description

A brief map contains the following sections:

- Object Module Synopsis
- Image Section Synopsis
- Link Run Statistics

In contrast, the default image map contains the Object Module Synopsis, Image Synopsis, Link Run Statistics, Program Section Synopsis, and Symbols By Name sections. For more information about image maps, see Chapter 5.

The /BRIEF qualifier must be specified with the /MAP qualifier and is incompatible with the /FULL qualifier and the /CROSS_REFERENCE qualifier.

Example

```
$ LINK/MAP/BRIEF MY_PROG
```

In this example, the linker creates a brief image map with the file name MY_PROG.MAP.

/CONTIGUOUS

Directs the linker to place the entire image in consecutive disk blocks. If sufficient contiguous space is not available on the output disk, the linker reports an error and terminates the link operation.

Format

/CONTIGUOUS

/NOCONTIGUOUS (default)

Qualifier Values

None.

Description

You can use the /CONTIGUOUS qualifier to speed up the activation time of any type of image because images usually activate more slowly if their image disk blocks are not contiguous. Note, however, that in most cases performance benefits do not warrant the use of the /CONTIGUOUS qualifier.

You can also use the /CONTIGUOUS qualifier when linking bootstrap programs for certain system images that require contiguity.

Even when you do not specify the /CONTIGUOUS qualifier, the file system tries to use contiguous disk blocks for images, if sufficient contiguous space is available.

Example

```
$ LINK/CONTIGUOUS MY_PROG
```

This example directs the linker to place the entire image named MY_PROG.EXE in consecutive disk blocks.

/CROSS_REFERENCE

Directs the linker to replace the "Symbols By Name" section in a full or default image map with the Symbol Cross-Reference section.

Format

/MAP/CROSS_REFERENCE

/MAP/NOCROSS_REFERENCE (default)

Qualifier Values

None.

Description

The Symbol Cross-Reference section lists, in alphabetical order, the name of each global symbol, together with the following information about each:

- Its value
- The name of the first module in which it is defined
- The name of each module in which it is referenced

The number of symbols listed in the cross-reference section depends on whether the linker generates a full map or a default map. In a full map, this section includes global symbols from all modules in the image, including those extracted from all libraries. In a default map, this section does not include global symbols from modules extracted from the default system libraries IMAGELIB.OLB and STARLET.OLB. For more information about image map files, see Chapter 5.

The /CROSS_REFERENCE qualifier is incompatible with the /BRIEF qualifier.

Example

```
$ LINK/MAP/CROSS_REFERENCE MY_PROG
```

This example produces an image map file named MY_PROG.MAP that includes a Symbol Cross-Reference section.

/DEBUG

Directs the linker to generate a debugger symbol table (DST) using DBG and TBT object language records and to give the debugger control when the image is run.

Format

/DEBUG[=file-spec]

/NODEBUG (default)

Qualifier Values

file-spec
Identifies a user-written debugger module.

If you specify the /DEBUG qualifier without entering a file specification, the OpenVMS Debugger gains control at run time. Requesting the OpenVMS Debugger does not affect the location of code within the image because the debugger is mapped into the process address space at run time, not at link time. See the *OpenVMS Debugger Manual* for additional information.

If you specify the /DEBUG qualifier with a file specification, the user-written debugger module identified by the file specification gains control at run time. The linker assumes a default file type of .OBJ. Requesting a user-written debugger module does affect the location of code within the image because the debugger module code is processed by the linker together with program code.

Description

The /DEBUG qualifier automatically includes the /TRACEBACK qualifier. If you specify the /DEBUG qualifier and the /NOTRACEBACK qualifier, the linker overrides your specification and includes traceback information.

To debug a shareable image, you must include it in a link operation that creates an executable image. Specify the /DEBUG qualifier when compiling the source modules that comprise the shareable image and specify the /DEBUG qualifier when linking the shareable image.

Example

```
$ LINK/DEBUG MY_PROG
```

This example produces an executable image named MY_PROG.EXE. Upon image activation, control will be passed to the debugger.

/DEMAND_ZERO (Alpha Only)

Alpha

For Alpha linking, enables demand-zero image section production for both executable and shareable images.

Format

/DEMAND_ZERO[=per_page] (default)

/NODEMAND_ZERO

Qualifier Values

per_page
Enables the linker to perform demand zero compression on Alpha images on a per-page basis. If this keyword is not used, the linker performs demand zero compression on an image-section basis only.

Description

On Alpha systems, compilers identify to the linker which program sections have not been initialized by setting the NOMOD program section attribute. The linker collects these uninitialized program sections into demand-zero image sections. (For more information about demand-zero image section production, see Section 3.4.3.)

If you specify the /NODEMAND_ZERO qualifier, the linker still gathers uninitialized program sections into demand-zero image sections but writes them to disk. Thus, the virtual memory layout of an image is the same when the /DEMAND_ZERO qualifier is specified and when the /NODEMAND_ZERO qualifier is specified. (If you specify the /NODEMAND_ZERO qualifier, the linker turns the demand-zero image sections containing the NOMOD program sections into regular image sections and sets the copy-on-reference [CRF] attribute if the write [WRT] attribute is set.)

To force the linker to write a program section to disk, that otherwise would be included in a demand-zero image section, turn off the NOMOD attribute of the program section by using the PSECT_ATTR= option, as in the following example:

```
PSECT_ATTR=psect-name,MOD
```

Note that only language processors can set the NOMOD attribute of a program section.♦

Examples

1. `$ LINK/NODEMAND_ZERO`

   In this example, the linker does not perform demand zero compression.

2. `$ LINK/DEMAND_ZERO`

   In this example, the linker by default performs demand zero compression on a per-image-section basis.

3. `$ LINK/NODEMAND_ZERO=PER_PAGE`

   In this example, the linker performs demand zero compression on both a per-image-section basis and a per-page basis.

/DSF (Alpha Only)

Alpha

For Alpha linking, directs the linker to create a file called a debug symbol file (DSF) for use by the OpenVMS Alpha System-Code Debugger. See *OpenVMS Alpha Device Support: Developer's Guide* for guidelines on using this debugger.

Format

/DSF[=file-spec]

/NODSF default

Qualifier Values

file-spec
Specifies the character string you want the linker to use as the name of the debug symbol file. If you do not include a file type in the character string, the linker appends the .DSF file type to the file name.

If you specify the /DSF qualifier without the file specification, the linker creates a debug symbol file with the file name of the first input file and the default file type .DSF. If you append the /DSF qualifier to one of the input file specifications, the linker creates a debug symbol file with the file name of the file to which the qualifier is appended and with the default file type .DSF.

Description

The /DSF qualifier directs the linker to create a separate file to contain the debug information required by the OpenVMS Alpha System-Code Debugger. The /DSF qualifier can be used with the /NOTRACEBACK qualifier to suppress the appearance of SYS$IMGSTA in the image's transfer array. The /DSF qualifier has no effect on the contents of the image, including the image header.

The /DSF and /DEBUG qualifiers are not mutually exclusive. However, the combination is not generally useful. The debug bit in the image header will be set and SYS$IMGSTA will be included in the transfer array, but there will be no information for the symbolic debugger included in the image. The DSF file will be generated as usual.

Example

```
$ LINK/DSF/NOTRACEBACK MY_PROG.OBJ
```

In this example, the linker will create the files MY_PROG.DSF and MY_PROG.EXE.♦

/EXECUTABLE

Directs the linker to create an executable image, as opposed to a shareable image or a system image.

Format

/EXECUTABLE[=file-spec] (default)

/NOEXECUTABLE

Qualifier Values

file-spec
Specifies the character string you want the linker to use as the name of the image file produced by the link operation. If you do not specify a file type in the character string, the linker assigns the .EXE file type by default.

If you do not specify a file name with the /EXECUTABLE qualifier, the linker creates an executable image with the file name of the first input file. If you append the /EXECUTABLE qualifier to an input file specification, the linker creates an executable image with the file name of the file to which the qualifier is appended.

Description

The /NOEXECUTABLE qualifier directs the linker to perform the linking operation but to not create an image file. Use the /NOEXECUTABLE qualifier to have the linker process the input files you specify without creating an image file to check for errors in your LINK command syntax or other link-time errors. You can also use the linker to produce a map file or symbol table file only by specifying the /NOEXECUTABLE qualifier with the /MAP qualifier or the /SYMBOL_TABLE qualifier.

The linker assumes the /EXECUTABLE qualifier as the default unless you specify the /NOEXECUTABLE qualifier, the /SHAREABLE qualifier, or the /SYSTEM qualifier. Note, however, that when used with the /SYSTEM qualifier, you can use the /EXECUTABLE qualifier to specify the name of a system image.

Examples

1. `$ LINK/NOEXECUTABLE MY_PROG`

   This example directs the linker to link the object module in the file MY_PROG.OBJ without creating an image file.

2. `$ LINK/EXECUTABLE MY_PROG`

   This example directs the linker to produce an executable image named MY_PROG.EXE. (The command line $ LINK MY_PROG will yield the same result because the /EXECUTABLE qualifier is the default.)

3. `$ LINK/EXECUTABLE=MY_IMAGE MY_PROG`

   This example directs the linker to produce an executable image with the name MY_IMAGE.EXE instead of the name MY_PROG.EXE.

/FULL

Directs the linker to create a full image map file. For more information, see also the /MAP, /CROSS_REFERENCE, and /BRIEF qualifiers.

Format

/MAP/FULL

Qualifier Values

None.

Description

A full map, which is the most complete image map, contains the following sections:

- Object Module Synopsis
- Module Relocatable Reference Synopsis (VAX linking only)
- Image Section Synopsis
- Program Section Synopsis
- Symbols By Name (and the Symbol Cross-Reference section if the /CROSS_REFERENCE qualifier is specified)
- Symbols By Value
- Image Synopsis
- Link Run Statistics

The full map also contains information about modules included from the default system libraries STARLET.OLB and IMAGELIB.OLB in the Object Module Synopsis, Program Section Synopsis, and Symbols By Name sections. For more information about image map files, see Chapter 5.

The /FULL qualifier is valid only if you also specify the /MAP qualifier in the LINK command. The /FULL qualifier is compatible with the /CROSS_REFERENCE qualifier, but it is not compatible with the /BRIEF qualifier.

Example

```
$ LINK/MAP/FULL MY_PROG
```

This example directs the linker to produce a full image map named MY_PROG.MAP.

/GST (Alpha Only)

Alpha

For Alpha linking, directs the linker to include in the global symbol table (GST) of a shareable image those symbols that have been declared as universal symbols in a symbol vector.

Format

/GST (default)

/NOGST

Qualifier Values

None.

Description

By default, the linker lists in the global symbol table (GST) of a shareable image the global symbols in the image that have been declared universal. By listing these symbols in the GST, the linker allows them to be referenced in link operations where the shareable image is specified as an input file.

To create a shareable image that can be activated by the images that linked against it, but that cannot be used to resolve symbolic references in a link operation, you can specify the /NOGST qualifier. When this qualifier is specified, the linker creates the image with an empty GST. (The linker still creates a GST.) By using the /NOGST qualifier, you can create a run-time version of a shareable image without having to remove the symbol vector from the link operation.

The images that were linked against the shareable image can still access symbols within the image because the /NOGST qualifier does not affect the symbol vector in the image. A symbol vector is an array of linkage pairs that contains the address within the image of the symbols. The value of a universal symbol in the GST is the offset of its entry in the symbol vector. Thus, to the images that were linked against the shareable image, the value of the symbol is the offset of its entry into the symbol vector.

This qualifier is valid only when used with the /SHAREABLE qualifier to create a shareable image.

Example

```
$ LINK/NOGST/SHAREABLE MY_SHARE,UNIVERSALS/OPT
```

This example creates the shareable image MY_PROG.EXE without listing entries for universal symbols in its global symbol table. The image contains an empty global symbol table.♦

/HEADER

When specified with the /SYSTEM qualifier, directs the linker to include an image header in a system image.

Format

/HEADER

Qualifier Values

None.

Description

The linker always creates executable images and shareable images with headers; they are a required component of image files. The linker creates system images without a header by default. To create a system image with a header, you must specify the /HEADER qualifier along with the /SYSTEM qualifier on the command line.

The linker ignores the /HEADER qualifier if it is specified for any other type of image (executable or shareable).

Example

```
$ LINK/SYSTEM/HEADER MY_SYS
```

This example directs the linker to produce a system image named MY_SYS.EXE with an image header. For more information about when to specify the /HEADER qualifier with the /SYSTEM qualifier, see the description of the /SYSTEM qualifier.

/INCLUDE

Identifies the input file specification to which it is appended as a library file and directs the linker to include in the link operation the module or modules specified as the value of the qualifier.

Format

library-name/INCLUDE=(module-name[,...])

Qualifier Values

library-name
Specifies the name of the library from which you want a module or modules extracted. The file name must specify a library file. The linker assumes the default file type of .OLB.

module-name
Specifies the module or modules that you want to extract from the library. To specify more than one module, enclose the list in parentheses and separate the module names with commas.

Description

Note that the /INCLUDE qualifier does not cause the linker to process the library for the definitions of unresolved symbolic references. If you want the linker to search the library for definitions of unresolved symbols, you must also specify the /LIBRARY qualifier before the /INCLUDE qualifier.

Examples

1. `$ LINK MY_PROG,MY_LIB/INCLUDE=(MOD1,MOD2,MOD5)`

   This example directs the linker to include modules MOD1, MOD2, and MOD5 from the library MY_LIB.OLB in the link operation with MY_PROG.

2. `$ LINK MY_PROG,MY_LIB/LIBRARY/INCLUDE=(MOD1,MOD2,MOD5)`

   This example directs the linker to extract modules MOD1, MOD2, and MOD5 from the library MY_LIB.OLB and then to search that library for symbol definitions that are unresolved in all four modules.

/INFORMATIONALS

Directs the linker to output informational messages produced by a link operation.

Format

/INFORMATIONALS (default)

/NOINFORMATIONALS

Qualifier Values

None.

Description

The linker outputs informational messages by default. To suppress informational messages, specify the /NOINFORMATIONALS qualifier.

Example

```
$ LINK/NOINFORMATIONALS MY_PROG
```

When the /NOINFORMATIONALS qualifier is specified, informational messages are suppressed.

/LIBRARY

Identifies the input file specification to which it is appended as a library file and directs the linker to process the library's name table as part of its symbol resolution processing. When the linker finds in the library the definition of a symbol referenced in a previously processed input file, the linker includes from the library the module in which the symbol is defined.

Format

library-name/LIBRARY

Qualifier Values

library-name
Specifies the name of the library to be included in the link operation. You must specify a library file. The linker assumes the default file type of .OLB.

Description

The order in which a library file is specified in the command string (or in an options file) is important because the linker uses the library file to resolve undefined symbols in previously processed (not subsequently processed) modules *only*. For more information about how the linker processes input files to resolve symbolic references, see Chapter 2.

Examples

1. `$ LINK MY_PROG,MY_LIB/LIBRARY,PROG2,PROG3`

   In this example, the linker uses the library MY_LIB.OLB to resolve undefined symbols in MY_PROG, but not in PROG2 or PROG3.

2. `$ LINK MY_PROG,PROG2,PROG3,MY_LIB/LIBRARY`

   In this example, the linker can resolve undefined symbols in MY_PROG, PROG2, and PROG3 from the library MY_LIB.OLB.

/MAP

Directs the linker to create an image map file. For more information, see also the /BRIEF, /CROSS_REFERENCE, and /FULL qualifiers.

Format

/MAP[=file-spec] (default in batch mode)

/NOMAP (default in interactive mode)

Qualifier Values

file-spec
If you enter a file specification with the /MAP qualifier, the linker creates an image map file with that file name. If you do not enter a file type after the file name, the linker assumes a file type of .MAP.

If you do not enter a file specification with the /MAP qualifier, the linker creates an image map file with the file name of the first input file specified on the command line and the file type .MAP. (If there are no input files specified on the command line, the linker names the map file .MAP.)

If you append the /MAP qualifier to one of the input file specifications, the linker creates an image map file with the file name of the file to which the qualifier is appended, using the .MAP file type.

Description

The /MAP qualifier causes the linker to produce a default image map file containing the following sections:

- Object Module Synopsis
- Image Section Synopsis
- Program Section Synopsis
- Symbols By Name
- Link Run Statistics

See Chapter 5 for more information about image map files.

Examples

1. `$ LINK/MAP MY_PROG`

   This example directs the linker to produce an image map with the default name of MY_PROG.MAP.

2. `$ LINK/MAP=MY_MAP MY_PROG`

   This example directs the linker to produce an image map with the name of MY_MAP.MAP instead of the default name of MY_PROG.MAP.

/NATIVE_ONLY (Alpha Only)

Alpha

For Alpha linking, prevents the linker from passing along procedure signature block (PSB) information in special fix-ups to the image activator. The image activator uses this information to build jackets so that native OpenVMS Alpha images can call translated OpenVMS VAX images. Note that this qualifier does not prevent incoming calls from translated OpenVMS VAX images.

Format

/NATIVE_ONLY (default)

/NONATIVE_ONLY

Qualifier Values

None.

Description

Jacket routines reformat data passed in procedure calls between the VAX and Alpha architectures. Jackets are required in images that make calls to translated components.

The linker does not build jackets. It stores PSB data from the compiler in the fixup section. The image activator uses this saved PSB information to build jackets, if they are needed, when the image is activated. Compilers create PSBs when you specify the /TIE qualifier. (For more information, see the OpenVMS Alpha compiler documentation.)

For more information about creating OpenVMS Alpha images that can operate with OpenVMS VAX images, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

Example

```
$ LINK/NATIVE_ONLY MY_PROG
```

In this example, the linker creates an image, named MY_PROG.EXE, that cannot interoperate with translated OpenVMS VAX images.♦

/OPTIONS

Identifies the input file specification to which it is appended as a linker options file.

Format

options-file-name/OPTIONS

Qualifier Values

options-file-name
The file specification of a linker options file. The linker assumes the file type .OPT by default.

Description

A linker options file can contain linker option specifications and input file specifications. For information about creating an options file, see Chapter 1.

Examples

1.
```
$ LINK MY_PROG,MY_OPTIONS/OPTIONS
```

   This example directs the linker to use an options file named MY_ OPTIONS.OPT to produce an executable image named MY_PROG.EXE.

2.
```
$ LINK MY_PROG,SYS$INPUT/OPTIONS
MY_SHARE/SHAREABLE
Ctrl/Z
```

   This example illustrates how to create an options file interactively by specifying SYS$INPUT as the file specification. After entering the options, press Ctrl/Z to end the options file.

/P0IMAGE

Directs the linker to place an executable image entirely in P0 address space. When the /P0IMAGE qualifier is specified, the user stack and OpenVMS RMS buffers, which usually reside in P1 space, are placed in P0 space by the image activator.

Format

/P0IMAGE

Qualifier Values

None.

Description

Note that the address of the stack shown in the map of an image linked with the /P0IMAGE qualifier does not reflect the true address of the stack at run time because, when /P0IMAGE is specified, the virtual address space for the stack is dynamically allocated at the end of P0 space at run time.

/P0IMAGE is used to create executable images that modify P1 address space.

Example

```
$ LINK/P0IMAGE MY_PROG
```

This example directs the linker to set up an executable image named MY_PROG.EXE to be run entirely in the P0 address space.

/PROTECT

Directs the linker to protect an entire shareable image from user-mode write access and supervisor-mode write access. Can be specified only with the /SHAREABLE qualifier.

Format

/PROTECT

/NOPROTECT (default)

Qualifier Values

None.

Description

The /PROTECT qualifier protects an entire shareable image from user-mode write access and supervisor-mode write access. To protect only specific image sections within a shareable image, but not the entire shareable image, use the PROTECT= option. For more information about using the PROTECT= option, see its description later in this section.

The /PROTECT qualifier is commonly used to protect shareable images that are used to implement user-written system services (called privileged shareable images) from user-mode access. If only certain clusters in the shareable image need protection, use the PROTECT= option.

The /PROTECT qualifier is incompatible with the /EXECUTABLE qualifier and the /SYSTEM qualifier.

Example

```
$ LINK/SHAREABLE/PROTECT MY_SHARE
```

This example directs the linker to produce a privileged shareable image named MY_SHARE.EXE.

/REPLACE (Alpha Only)

Alpha

For Alpha linking, specifies that the linker should perform certain optimizations to improve the performance of the resultant image, when instructed by the compiler.

Format

/REPLACE (default)

/NOREPLACE

Qualifier Values

None.

Description

For Alpha linking, it is more efficient to execute a procedure call as a branch, using the BSR (Branch to Subroutine) instruction sequence, than it is to execute the call as a jump, using the JSR (Jump to Subroutine) instruction sequence. In a BSR instruction, the destination can be expressed as an offset, requiring fewer memory fetches than a JSR instruction sequence.

Compilers cannot always take advantage of the efficiency of the BSR instruction because the information needed to calculate the offset is not available until link time, when the linker lays out the image sections that make up the image. To achieve this performance enhancement, compilers flag uses of the JSR instruction sequence and the linker examines each use and attempts to replace it with the BSR instruction sequence wherever possible.

In addition to code replacement, the linker can also specify **hints** to improve the performance of the JSR instructions that remain in the image. A hint is used to index the instruction cache and can improve performance. Hints are generated for JSR instructions within the image and for JSR instructions to shareable images.

For more information about this optimization, see Section 1.4. ♦

/SECTION_BINDING (Alpha Only)

Alpha

For Alpha linking, directs the linker to create an image that can be installed as a resident image and to flag coding practices in the image that would prevent this.

Format

/[NO]SECTION_BINDING[=(CODE,DATA)]

/NOSECTION_BINDING (default)

Qualifier Values

CODE
Prevents the linker from replacing the Jump to Subroutine (JSR) instruction sequence with the more efficient Branch to Subroutine (BSR) instruction sequence when the target of the branch crosses an image section boundary.

DATA
Directs the linker to check for address calculations that create dependencies on the layout of data image sections. The linker reports such occurrences.

When the /SECTION_BINDING qualifier is specified without either the CODE or DATA keyword, the linker performs both types of checking by default.

Description

For Alpha linking, you can improve the performance of an installed image by installing it as a resident image (by using the /RESIDENT qualifier of the Install utility). The Install utility moves portions of resident images into system space where they reside on a large single page with granularity hints set (called a granularity hint region or GHR), thus improving performance.

For an image to be installed as a resident image, it must not contain any dependencies on the layout of image sections, such as branch instructions that cross image section boundaries. The offsets calculated by the linker for such branches depend on the layout of the image sections. The relative position of the code image sections changes when they are moved to system space and the accuracy of the offsets calculated by the linker is destroyed. (These dependencies are created by the linker when it replaces the JSR instruction sequence with the BSR instruction sequence. For more information, see the description of the /REPLACE qualifier.)

When the /SECTION_BINDING qualifier is specified, the linker does not replace JSR instructions when the destination crosses an image section boundary. The linker still replaces the JSR instruction sequence for calls that stay within the boundaries of an image section.

In addition to eliminating image section layout dependencies in code image sections, the linker can also check the data image sections in an image to see if they contain coding practices that produce dependencies on image section layout. The image activator can reposition data image sections to eliminate the gaps in virtual memory left by the code image sections that were moved to system space. However, data image sections can also contain dependencies on image section layout. For example, when an image initializes an address by performing arithmetic on two addresses that reside in two different image

sections, the address calculation creates a dependency on the layout of the data image sections, as in the following example:

```
OWN
   FOO:  INITIAL ( FOO - BAR)
```

If the linker detects the compiler adding or subtracting two intra-image addresses, it assumes that a relative branch is being calculated and displays the following warning:

```
%LINK-W-BINDFAIL, failed to bind reference at %X00030000 between sections
        at locations %X00030000 and %X00010000
        in module X file WORK:[TEST]X.OBJ;6
```

The warning message produced by the linker indicates the two addresses being operated on and the virtual address where it was trying to write the result. To find the source code that is creating the dependency, examine the map file to determine what entities reside at these addresses and then search the source code for places where they are used in calculations. In this example, module X contained a data cell, FOO, initialized with the difference between FOO's address and BAR's (as in the previous code example). In the image map, FOO resides at %X00030000 and BAR at %X00010000. Because these addresses appear in different image sections, the calculation introduces a dependency on the layout of image sections. To fix this dependency, rewrite the source code to remove the calculation or move the two data cells into the same image section by using the COLLECT= option or the PSECT_ATTR= option.

The linker issues a message for each address calculation in data image sections that create dependencies on the layout of image sections, as in the following example:

```
%LINK-W-BINDISABLE, section binding of data has been disabled
%LINK-W-BINDFAIL, failed to bind reference at %X0000865D between sections
        at locations %X00008000 and %X00000000
        in module MKDRIVER file X56Y_RESD$:[DRIVER.OBJ]DRIVER.OLB;1
%LINK-W-BINDFAIL, failed to bind reference at %X00008665 between sections
        at locations %X00008000 and %X00000000
        in module MKDRIVER file X56Y_RESD$:[DRIVER.OBJ]DRIVER.OLB;1
%LINK-W-BINDFAIL, failed to bind reference at %X0000866D between sections
        at locations %X00008000 and %X00000000
        in module MKDRIVER file X56Y_RESD$:[DRIVER.OBJ]DRIVER.OLB;1
```

Example

```
$ LINK/SHARE/SECTION_BINDING MY_PROG
```

In this example, the linker creates the image MY_PROG.EXE and processes it so that it can be installed as a resident image.♦

/SELECTIVE_SEARCH

When this qualifier is appended to an input file specification, the linker processes only those symbols in the input file that have been referenced by previously processed input files.

Format

input-file-name/SELECTIVE_SEARCH

Qualifier Values

input-file-name
The input file you want included in the link operation. The /SELECTIVE_SEARCH qualifier works with object modules and shareable images only. This qualifier is illegal with library files. (To process the modules in a library selectively, you specify the /SELECTIVE qualifier when inserting the files into the library. For more information, see the *OpenVMS Command Definition, Librarian, and Message Utilities Manual*.)

Description

If you do not specify the /SELECTIVE_SEARCH qualifier with an input file, the linker includes all the input file's global symbols in the global symbol table of the image it is creating by default.

Note that the /SELECTIVE_SEARCH qualifier does not affect the size of the resultant image. The entire object module is included in the image, even if only a subset of the symbols in its global symbol table are needed to resolve symbolic references. Specifying the /SELECTIVE_SEARCH qualifier can improve the performance of a link operation and conserve the linker's use of virtual memory.

Examples

1.
```
$ LINK/MAP MY_MAIN,MY_PROG/SELECTIVE_SEARCH
```

   In this example, the linker processes the object module MY_PROG.OBJ selectively. You can verify this processing by checking the list of symbols in the image map file created in this link. The only symbols from the file MY_PROG.OBJ that will appear in the map file are those symbols that were referenced by MY_MAIN.OBJ.

2.
```
$ LINK/MAP=MY_MAIN/EXE=MY_MAIN SYS$INPUT/OPT
CLUSTER=MY_MAIN_CLUS,,,MY_MAIN
MY_SHARE/SHARE/SELECTIVE_SEARCH
Ctrl/Z
```

   In this example, the linker processes the shareable image MY_SHARE.EXE selectively. Note that, to ensure that the linker processes references to symbols in the shareable image before it processes the shareable image selectively, the input file MY_MAIN.OBJ is placed in a named cluster (MY_MAIN_CLUS), using the CLUSTER= option. If the object modules had been specified on the LINK command line, the linker would have put it in the default cluster. The linker processes named clusters before it processes the default cluster.

3.
```
$ LIBRARIAN/INSERT/SELECTIVE MY_LIB MY_PROG
$ LINK MY_PROG,MY_LIB/LIBRARY
```

   In this example, the object module MY_PROG.OBJ is inserted into the library MY_LIB.OLB selectively. When the library is specified in a link operation, the linker processes the object module selectively. This link operation is equivalent to the link operation in example 1.

/SHAREABLE

When specified anywhere on the LINK command line, the /SHAREABLE qualifier directs the linker to create a shareable image. When the /SHAREABLE qualifier is appended to a file specification in a linker options file, it identifies the input file as a shareable image.

Format

/SHAREABLE[=file-spec]

shareable-image-file-name/SHAREABLE

Qualifier Values

file-spec
When the /SHAREABLE qualifier is used to create a shareable image, this parameter specifies the name you want the linker to assign to the shareable image being created. If you do not include a file specification, the linker assigns the shareable image the name of the file to which the /SHAREABLE qualifier is appended in the LINK command line. If the /SHAREABLE qualifier is not appended to an input file specification, the linker assigns to the shareable image the name of the first input file specified on the command line using the extension .EXE.

If you designate a file name but omit the file type, the linker assigns the shareable image the file type .EXE.

shareable-image-file-name
Specifies the name of a shareable image. Note that you can use the /SHAREABLE qualifier to identify a shareable image only in a linker options file. It is illegal to include a shareable image in a link operation by specifying it on the LINK command line.

Description

The linker creates executable images by default; you must specify the /SHAREABLE qualifier to create a shareable image. The /SHAREABLE qualifier is incompatible with the /SYSTEM qualifier.

For more information about creating and using shareable images, see Chapter 4.

Examples

1. `$ LINK/SHAREABLE MY_SHARE,UNIVERSALS/OPT`

   This example directs the linker to produce a shareable image named MY_SHARE.EXE. The options file UNIVERSALS.OPT contains declarations of the universal symbols in the shareable image.

2. `$ LINK/SHAREABLE=MY_APP MY_SHARE, UNIVERSALS/OPT`

   This example directs the linker to produce a shareable image named MY_APP.EXE using the object module MY_SHARE.OBJ as input.

3.
```
$ TYPE MY_OPTIONS.OPT
MY_SHARE/SHAREABLE
$ LINK MY_PROG,MY_OPTIONS.OPT/OPTION
```

   In this example, a shareable image is included in a link operation. The shareable image is specified in the options file MY_OPTIONS.OPT, which is specified as an input file on the LINK command line.

4.
```
$ LINK MY_PROG,SYS$INPUT/OPTION
MY_SHARE/SHAREABLE
Ctrl/Z
```

   This example shows how the shareable image MY_SHARE.EXE is used, together with the object file MY_PROG.OBJ, to create an executable image named MY_PROG.EXE.

   Note how you can specify options interactively at the command line by identifying the logical name SYS$INPUT as an options file. The linker interprets the lines following the LINK command as the contents of an options file, until you terminate the options by entering the Ctrl/Z key sequence.

/SYMBOL_TABLE

Directs the linker to create a symbol table file.

Format

/SYMBOL_TABLE[=file-spec]

/NOSYMBOL_TABLE (default)

Qualifier Values

file-spec
Specifies the character string you want the linker to use as the name of the symbol table file. If you do not include a file type in the character string, the linker appends the .STB file type to the file name.

If you specify the /SYMBOL_TABLE qualifier without the file specification, the linker creates a symbol table file with the file name of the first input file and the default file type .STB. If you append the /SYMBOL_TABLE qualifier to one of the input file specifications, the linker creates a symbol table file with the file name of the file to which the qualifier is appended, with the default file type .STB.

Description

A symbol table file contains a copy of the image's global symbol table, excluding definitions from shareable images, in object module format.

VAX

For VAX linking, a global symbol table produced by a link that creates a shareable image contains only universal symbols. A global symbol table produced by a link that creates an executable image contains all the global symbols in the image.

You can specify symbol table files as input files in link operations if they were produced in an operation in which an executable or system image was created. Symbol table files produced in a link operation in which a shareable image was created do not always contain enough information to be used as input files in link operations. (See Section 1.2.4 for more information.)♦

Alpha

For Alpha linking, you cannot specify symbol table files as input files in a link operation. Symbol table files of Alpha images are intended only as an aid in debugging crash dumps using the OpenVMS Alpha System Dump Analyzer utility (SDA). For more information, see Section 1.2.4.

Note that you can direct the linker to include global symbols in a symbol table file associated with a shareable image by specifying the SYMBOL_TABLE=GLOBALS option. When this option is specified, the linker includes both global symbols as well as universal symbols in the symbol table file. The linker includes only universal symbols in a symbol table file by default.♦

Examples

1. `$ LINK/SYMBOL_TABLE/NOEXE MY_PROG`

   In this example, the linker produces a symbol table file named MY_PROG.STB without producing an executable image.

2.
```
$ LINK/SYMBOL_TABLE=MY_PROG_SYMB_TAB MY_PROG
```

   In this example, the linker produces a symbol table file named MY_PROG_SYMB_TAB.STB. An executable image file named MY_PROG.EXE is also produced.

3.
```
$ LINK/SHAREABLE/SYMBOL_TABLE MY_SHARE,SYS$INPUT/OPT
GSMATCH=lequal,1,1000
SYMBOL_VECTOR=(myproc=PROCEDURE,-
               mydata=DATA,-
               myproc2=PROCEDURE)
SYMBOL_TABLE=GLOBALS
Ctrl/Z
```

   In this example, the linker creates a symbol table file on an Alpha system, named MY_SHARE.STB, that contains both global symbols and universal symbols because the linker option SYMBOL_TABLE=GLOBALS is specified in the options file.

/SYSEXE (Alpha Only)

Alpha

For Alpha linking, directs the linker to process the system shareable image, SYS$BASE_IMAGE.EXE, in a link operation. The linker looks for SYS$BASE_IMAGE.EXE in the directory pointed to by the logical name ALPHA$LOADABLE_IMAGES.

Format

/SYSEXE[=[NO]SELECTIVE]

/NOSYSEXE (default)

Qualifier Values

SELECTIVE

When you specify the SELECTIVE keyword, the linker processes the SYS$BASE_IMAGE.EXE file selectively, including only those symbols from the global symbol table of the SYS$BASE_IMAGE.EXE file that were referenced by input files previously processed in the link operation.

NOSELECTIVE

When you specify the NOSELECTIVE keyword, the linker includes all the symbols from the SYS$BASE_IMAGE.EXE global symbol table in the link operation.

When the /SYSEXE qualifier is specified without a keyword, the linker processes the executive image selectively.

Description

When you specify the /SYSEXE qualifier, the linker processes the SYS$BASE_IMAGE.EXE file selectively *after* processing the system shareable image library, IMAGELIB.OLB, and *before* processing the system object library, STARLET.OLB, and the system service shareable image, SYS$PUBLIC_VECTORS.EXE, which is associated with STARLET.OLB. (By default, the linker processes IMAGELIB.OLB, STARLET.OLB, and SYS$PUBLIC_VECTORS.EXE, in that order, to resolve symbols that remain undefined after all the files specified in the LINK command have been processed and after any user-specified libraries have been processed.) Note that the linker qualifiers that determine whether the linker processes the default system libraries, /SYSSHR and /SYSLIB, do not affect SYS$BASE_IMAGE.EXE processing.

If you want the linker to process SYS$BASE_IMAGE.EXE before processing IMAGELIB.OLB, specify SYS$BASE_IMAGE.EXE in an options file, as you would any other shareable image. If you specify SYS$BASE_IMAGE.EXE in your options file, do not specify the /SYSEXE qualifier in the LINK command.

Note

The linker looks for SYS$BASE_IMAGE.EXE in the directory pointed to by the logical name ALPHA$LOADABLE_IMAGES.

For more information about linking against the OpenVMS executive, see Section 2.4.

Example

```
$  LINK/SHARE/SYSEXE  MY_SHARE, SYS$INPUT/OPT
SYMBOL_VECTOR=(MY_PROC=PROCEDURE)
Ctrl/Z
```

In this example, the linker processes the OpenVMS system executive file, SYS$BASE_IMAGE.EXE, to create a shareable image named MY_SHARE.EXE. ♦

/SYSLIB

Directs the linker to process the default system shareable image library, IMAGELIB.OLB, and the default system object module library, STARLET.OLB, to resolve symbolic references that remain undefined after all specified input files and any default user libraries have been processed.

Format

/SYSLIB (default)

/NOSYSLIB

Qualifier Values

None.

Description

The linker first searches IMAGELIB.OLB, the default system shareable image library, then STARLET.OLB, the default system object library.

Alpha

For Alpha linking, the linker also searches the shareable image SYS$PUBLIC_VECTORS.EXE to resolve references to system services. (For more information about processing SYS$PUBLIC_VECTORS.EXE, see the description of the /SYSEXE qualifier.) The linker looks for these default libraries in the directory pointed to by the logical name ALPHA$LIBRARY.♦

VAX

For VAX linking, the linker looks for these default libraries in the directory pointed to by the logical name SYS$LIBRARY.♦

If you specify the /NOSYSLIB qualifier and the /SYSSHR qualifier, the /SYSSHR qualifier is ignored.

If you want the linker to search IMAGELIB.OLB but not STARLET.OLB, specify the /NOSYSLIB qualifier (to inhibit the default search of both default system libraries) and then specify IMAGELIB.OLB in the LINK command line or in an options file.

Example

```
$ LINK/NOSYSLIB MY_PROG
```

In this example, the linker creates the executable image MY_PROG.EXE without referencing the default system libraries IMAGELIB.OLB or STARLET.OLB.

/SYSSHR

Directs the linker to process the default system shareable image library (IMAGELIB.OLB) to resolve symbolic references that remain undefined after all specified input files and any default user libraries have been processed.

Format

/SYSSHR (default)

/NOSYSSHR

Qualifier Values

None.

Description

To specify that the linker should skip processing the default system shareable image library, IMAGELIB.OLB, but still process the default system object library, STARLET.OLB, specify the /NOSYSSHR qualifier.

See the description of the /SYSLIB qualifier for information about controlling how the linker processes the default system libraries.

Example

```
$ LINK/NOSYSSHR MY_PROG
```

In this example, the linker processes the default system object library (STARLET.OLB), but does not process the default system shareable image library (IMAGELIB.OLB), to resolve symbolic references while producing an executable image named MY_PROG.EXE.

/SYSTEM

Directs the linker to create a system image and optionally allows you to specify the address at which the image should be loaded into memory. A system image cannot be activated with the RUN command; it must be bootstrapped or otherwise loaded into memory.

Format

/SYSTEM[=base-address]

Qualifier Values

base-address
Specifies the address at which the image is to be loaded in virtual memory. You can specify a base address in hexadecimal (%X), octal (%O), or decimal (%D) format. The default base address is %X80000000.

Note that if you specify the /HEADER qualifier, the linker adjusts the base address to the next highest page boundary if it is not already a page boundary. The next highest page boundary is the smallest number that is greater than the value specified in the **base-address** parameter and that is divisible by the default page size (which is architecture specific) or the page size specified using the /BPAGE qualifier.

Description

System images are intended for special purposes, such as standalone operating system diagnostics. When the linker creates a system image, it orders the program sections in alphanumeric order and ignores all program section attributes.

The linker creates the system image with the file name of the first input file and the file type .EXE. If you want a different output file specification, specify that file specification with the /EXECUTABLE qualifier.

If you specify the /SYSTEM qualifier, you cannot specify the /SHAREABLE qualifier or the /DEBUG qualifier.

Example

```
$ LINK/SYSTEM MY_SYS
```

This example directs the linker to produce a system image named MY_SYS.EXE based at address %X80000000.

/TRACEBACK

Directs the linker to include traceback information in the image file. If you specify the /DEBUG qualifier, the linker includes traceback information by default, overriding the /NOTRACEBACK qualifier if it is specified.

Format

/TRACEBACK (default)

/NOTRACEBACK

Qualifier Values

None.

Description

Traceback is a facility that displays information from the call stack when a program error occurs. The output shows which modules were called before the error occurred.

Note that the traceback handler can display traceback information only from the main executable image, not from any shareable images.

Example

```
$ LINK/NOTRACEBACK MY_PROG
```

In this example, the linker does not include traceback information in the executable image named MY_PROG.EXE.

/USERLIBRARY

Directs the linker to process one or more default user libraries to resolve symbolic references that remain undefined after all specified input files have been processed.

Format

/USERLIBRARY[=(table[,...])]

/NOUSERLIBRARY

/USERLIBRARY=ALL (default)

Qualifier Values

table
Specifies the logical name tables that the linker searches for default user libraries. The following keywords are the only acceptable parameter values:

Keyword	Description
ALL	Directs the linker to search the process, group, and system logical name tables for default user library definitions. This is the default.
GROUP	Directs the linker to search the group logical name table for default user library definitions.
NONE	Directs the linker not to search any logical name table; the /USERLIBRARY=NONE qualifier is equivalent to the /NOUSERLIBRARY qualifier.
PROCESS	Directs the linker to search the process logical name table for default user library definitions.
SYSTEM	Directs the linker to search the system logical name table for default user library definitions.

Description

A default user library may be an object module library or a shareable image library.

To define a default user library, you must use the DCL command DEFINE or ASSIGN to equate the logical name LNK$LIBRARY to the file specification of the library, because the linker looks for this logical name to determine if a default user library exists.

Further, to control access to the library, you can define LNK$LIBRARY in the process, group, or system logical name tables by using the /PROCESS qualifier, the /GROUP qualifier, and the /SYSTEM qualifier, respectively, in the DEFINE command.

For example, if you want the library MY_LIB to be your default user library, the library GROUP_LIB to be the default user library of everyone else in your group, and the library ANY_LIB to be the default user library of everyone else on the system, you would issue the following commands:

```
$ DEFINE LNK$LIBRARY DB2:[MARK]MY_LIB
$ DEFINE/GROUP LNK$LIBRARY DB2:[PROJECT]GROUP_LIB
$ DEFINE/SYSTEM LNK$LIBRARY SYS$LIBRARY:ANY_LIB
```

Note that the GRPNAM and SYSNAM privileges are required to use the /GROUP qualifier and the /SYSTEM qualifier, respectively.

If you are defining more than one library in a single logical name table, use the logical names LNK$LIBRARY for the first library, LNK$LIBRARY_1 for the second library, LNK$LIBRARY_2 for the third, and so on, up to the last possible logical name of LNK$LIBRARY_999. However, you must specify these logical names in numeric order without skipping any, because when the linker does not find the next sequential logical name, it stops searching in that logical name table.

The search of default user libraries proceeds as follows:

1. If you specify the /USERLIBRARY=PROCESS qualifier or the /USERLIBRARY qualifier, the linker searches the process logical name table for the name LNK$LIBRARY. If this entry exists, the linker translates the logical name and searches the specified library for unresolved strong references. If you exclude PROCESS from the table list in the /USERLIBRARY qualifier or if no entry exists for LNK$LIBRARY, the linker proceeds to step 4 (searching the group logical name table).
2. If any unresolved strong references remain, the linker searches the process logical name table for the name LNK$LIBRARY_1 and follows the logic of step 1. If no entry exists for LNK$LIBRARY_1, the linker proceeds to step 4 (searching the group logical name table).
3. If any unresolved strong references remain, the linker follows the logic of step 1 for LNK$LIBRARY_2, LNK$LIBRARY_3, and so on, until it finds no match in the process logical name table, at which point it proceeds to step 4.
4. If you specify the /USERLIBRARY=GROUP qualifier or the /USERLIBRARY qualifier, the linker follows the logic in steps 1 through 3 using the group logical name table. If you exclude GROUP from the table list in the /USERLIBRARY qualifier or when any logical name translation fails, the linker proceeds to step 5.
5. If you specify the /USERLIBRARY=SYSTEM qualifier or the /USERLIBRARY qualifier, the linker follows the logic in steps 1 through 3 using the system logical name table. If you exclude SYSTEM from the table list in the /USERLIBRARY qualifier or when any logical name translation fails, the search of default user libraries is complete. By default, the linker proceeds to search the default system libraries if any unresolved references remain.

Search lists are not recognized in LNK$LIBRARY* logical names. Instead, use LNK$LIBRARY_*nnn* with a single library specification.

Example

```
$ LINK/USERLIBRARY=(GROUP) MY_PROG
```

In this example, the linker searches only the group logical name table to translate the logical names LNK$LIBRARY, LNK$LIBRARY_1, LNK$LIBRARY_2, and so on.

/VAX

Directs the linker to produce an OpenVMS VAX image. The default action, when neither /ALPHA nor /VAX is specified, is to create an OpenVMS VAX image on an OpenVMS VAX system and to create an OpenVMS Alpha image on an OpenVMS Alpha system.

Format

/VAX

Qualifier Values

None.

Description

This qualifier is used to instruct the linker to accept OpenVMS VAX object files and library files to produce an OpenVMS VAX image.

You must inform the linker where OpenVMS VAX system libraries and shareable images are located. On an OpenVMS VAX system, you use the logical name SYS$LIBRARY to do this. On an OpenVMS Alpha system, you use the logical name VAX$LIBRARY to do this. Therefore, if the link is to occur on an OpenVMS Alpha system, you must define the logical VAX$LIBRARY so that it translates to the location of an OpenVMS VAX system disk residing on the system where the VAX linking is to occur.

For more information on cross-architecture linking, see Section 1.6.

Example

```
$ DEFINE VAX$LIBRARY DKB200:[VMS$COMMON.SYSLIB]
$ LINK/VAX VAX.OBJ
```

This example, performed on an OpenVMS Alpha system, shows the definition of the logical name VAX$LIBRARY to point to an OpenVMS VAX system disk mounted on device DKB200 in the appropriate area. The qualifier tells the linker to expect the object file, VAX.OBJ, to be an OpenVMS VAX object file and to link it using the OpenVMS VAX libraries and images on DKB200, if necessary.

Option Descriptions

This section describes the linker options that you may specify in a linker options file. For information about creating and using linker options files, see Chapter 1.

You can express numeric parameters in decimal (%D), hexadecimal (%X), or octal (%O) radix by prefixing the number with the corresponding radix operator. If no radix operator is specified, the linker assumes decimal radix.

The default and maximum numeric values in this manual are expressed in decimal numbers, as are the values in any linker messages relating to these options.

Options	Defaults
BASE=	See description. (VAX linking only)
CASE_SENSITIVE=YES/NO	NO
CLUSTER=	See description.
COLLECT=	None
GSMATCH=	See description.
IDENTIFICATION=	See description.
IOSEGMENT=	0,[NO]P0BUFS
ISD_MAX=	Approximately 96
NAME=	Name of the output image file
PROTECT=YES/NO	NO
PSECT_ATTR=	None
RMS_RELATED_CONTEXT=	YES
STACK=	20 pagelets
SYMBOL=	None
SYMBOL_TABLE=GLOBALS/UNIVERSALS	SYMBOL_TABLE=UNIVERSALS (Alpha linking only)
SYMBOL_VECTOR=	None (Alpha linking only)
UNIVERSAL=	None (VAX linking only)

BASE= (VAX Only)

VAX

For VAX linking, specifies the base address (starting address) that you want the linker to assign to the image.

Format

BASE=address

Option Values

address
The address at which you want the image based. You can express the number in decimal (%D), octal (%O), or hexadecimal (%X) notation. If the address specified is not divisible by 512, the linker automatically adjusts it upward to the next multiple of 512, that is, to the next highest page boundary. Do not attempt to base an image linked with a larger page size (specified using the /BPAGE qualifier).

The linker bases shareable images at address 0, by default, and bases system images at address %X80000000, by default.

Description

The BASE= option is illegal in a link operation that produces a system image. To specify a base address for a system image, use the /SYSTEM qualifier.

The BASE= option is not supported for Alpha linking. Note, however, that you can set the base address for an executable image by specifying the address as a parameter to the CLUSTER= option. You cannot create a based *shareable* Alpha image.

In general, the use of the BASE= option to create based images is not recommended. The memory management component of the OpenVMS operating system cannot relocate a based shareable image in the virtual address space, which could result in possible fragmentation of the virtual address space.

The linker processes the BASE= option by assigning the specified base address to the default cluster. If the linker creates additional clusters before it searches the default libraries, which it does if a CLUSTER= or COLLECT= option is specified or if a shareable image is explicitly specified, the following effects may occur:

- If the additional clusters are based (that is, if the CLUSTER= option specifies a base address or if the shareable image is a based shareable image), the linker must check that memory requirements for each based cluster do not conflict. Memory requirements conflict when more than one cluster requires the same section of address space. If they do conflict, the linker issues an error message and aborts the linking operation. If they do not conflict, the linker allocates each cluster the memory space it requests.
- If the additional clusters are not based, there will be no conflicting memory requirements. However, the linker will place each additional cluster at an address higher than that of the default cluster (because the base address of the default cluster must be the base address of the entire image). Consequently, the location of clusters (relative to each other) in the image will differ from what you would expect based on the position of each cluster in the cluster list. (Remember that the additional clusters precede the default

cluster on the cluster list and that the linker typically allocates memory for clusters beginning at the first cluster on the cluster list, then the second, and so on.) For more information about the linker's clustering algorithm, see Chapter 2. For more information about the linker's memory allocation algorithm, see Chapter 3.♦

CASE_SENSITIVE=

Directs the linker to preserve the mixture of uppercase and lowercase characters used in character string arguments to linker options.

Format

CASE_SENSITIVE=YES/NO

Option Values

YES
Enables case sensitivity. You can use any mixture of uppercase and lowercase characters when specifying the keyword YES.

NO
Disables case sensitivity. Note that you must use only uppercase characters when specifying the keyword NO because case sensitivity is enabled and the linker does not accept mixed case in keywords.

Description

Once case sensitivity has been enabled, the linker preserves the case of all succeeding character string arguments to linker options until you explicitly disable it. When the CASE_SENSITIVE= option is disabled (which is the default), the linker changes all the characters in a character string to uppercase before processing the string.

Note that the CASE_SENSITIVE= option only affects how the linker processes arguments to linker options. When it searches object files and shareable image files for symbols that need to be resolved, the linker preserves the case used in the symbol names (created by the language compilers). Also, the names of the linker options (all the characters preceding the equal sign, as in the NAME= option) are unaffected by the case-sensitivity option. The linker changes all the characters in option names to uppercase characters before processing the option, even if case sensitivity has been enabled.

Carefully delimit the section of a linker options file in which you use case sensitivity to avoid unintentional side effects. For example, if you include options in the case sensitive region that accept keyword arguments, such as YES, NO, EXE, and SHR, make sure the keywords are specified using uppercase characters. Because these keywords appear after the equal sign, they are affected by case sensitivity. Similarly, character string arguments used to name a program section, cluster, or image are also affected by case sensitivity.

Example

```
$ link/share/map/full test, sys$input:/opt
case_sensitive=YES
name=ImageName
symbol=OneSymbol,1
case_sensitive=NO
universal=myroutine
Ctrl/Z
```

In the example, the CASE_SENSITIVE= option with the value YES enables case sensitivity in the linker options file. Because case sensitivity has been enabled,

the linker preserves the mix of uppercase and lowercase characters used in character string arguments to all succeeding linker options. In the example, this includes the character string ImageName passed to the NAME= option and the character string OneSymbol passed to the SYMBOL= option.

Specifying the CASE_SENSITIVE= option with the value NO turns off case sensitivity. Note that you must use uppercase characters when specifying the keyword NO. Because case sensitivity has been disabled, the linker changes all the characters in the universal symbol myroutine to uppercase. The following excerpt from the map file produced by this link illustrates how these identifiers were stored by the linker:

```
ImageName
OneSymbol
MYROUTINE
```

CLUSTER=

Directs the linker to create a cluster. (The linker groups input files into clusters before processing their contents.)

Format

CLUSTER=cluster-name,[base-address],[pfc],[file-spec,...]

Option Values

cluster-name
The name you want assigned to the cluster.

base-address
The base virtual address for the cluster. If you omit the base-address value, you must still enter the comma.

Alpha

For Alpha linking, it is illegal to specify a base address for a cluster when creating a shareable image.♦

pfc (page fault cluster)
The number of pagelets read into memory by the operating system when the initial page fault occurs for a page in the cluster. If you do not specify the **pfc** parameter, the operating system uses the default value established by the system parameter PFCDEFAULT. If you omit the page fault cluster value, you must still enter the comma.

file-spec,...
The file you want the linker to place in the cluster. Note that you should not specify in the LINK command itself any file that you specify with the CLUSTER= option (unless you want to include two copies of the file in the final image).

Description

You can use the CLUSTER= option in the following ways:

- To control the order in which the linker processes input files
- To cause specified modules to be placed close together in virtual memory

If you do not specify the CLUSTER= option, the linker always creates at least one cluster, called the default cluster. For more information about how the linker creates clusters, see Chapter 2.

You can also create a cluster by specifying the COLLECT= option

Example

```
$  LINK MY_PROG,SYS$INPUT/OPT
CLUSTER=MY_CLUSTER,,,PROG2,PROG3
```

In this example, the linker creates a cluster, named MY_CLUSTER, that contains the input files named PROG2.OBJ and PROG3.OBJ.

COLLECT=

Directs the linker to place the specified program section (or program sections) into the specified cluster.

Format

COLLECT=cluster-name [/ATTRIBUTES= { RESIDENT / INITIALIZATION_CODE }], psect-name[,...]

Option Values

cluster-name
Name of the cluster.

psect-name[,...]
Name of the program sections (psects) you want placed in the cluster.

Qualifier

/ATTRIBUTES

Alpha

For Alpha linking, directs the linker to mark the cluster 'cluster-name' with the indicated qualifier keyword value. This qualifier is used to build Alpha drivers. See *OpenVMS Alpha Device Support: Developer's Guide* for guidelines for using this qualifier.

Qualifier Values

RESIDENT
Marks the cluster 'cluster-name' as RESIDENT so that the image section created from that cluster has the EISD$V_RESIDENT flag set. This will cause the loader to map the image section into non-paged memory.

INITIALIZATION_CODE
Marks the cluster 'cluster-name' as INITIALIZATION_CODE so that the image section created from that cluster has the EISD$V_INITALCOD flag set. The initialization code will be executed by the loader. This keyword is specifically intended for use with program sections from modules SYS$DOINIT and SYS$DRIVER_INIT in STARLET.OLB.♦

Description

If the specified cluster does not already exist, the linker creates the cluster when it processes the COLLECT= option.

The linker sets the global (GBL) attribute for all the program sections specified, if they do not already have this attribute set. Program sections from a shareable image referenced in the options file with the /SHARE qualifier cannot be specified in the COLLECT= option.

Example

```
$  LINK MY_PROG,SYS$INPUT/OPT
COLLECT=MY_CLUSTER,PSECT2,PSECT3
Ctrl/Z
```

In the example, the linker creates the cluster named MY_CLUSTER, if it does not already exist, and puts the program sections named PSECT2 and PSECT3 in the cluster.

DZRO_MIN=

Specifies the minimum number of contiguous, uninitialized pages that the linker must find in an image section before it can extract the pages from the image section and place them in a newly created demand-zero image section. By creating demand-zero image sections (image sections that do not contain initialized data), the linker can reduce the size of images.

Format

DZRO_MIN=number-of-pages

Option Values

number-of-pages
Specifies the minimum number of contiguous pages.

VAX

For VAX linking, the linker, by default, uses a minimum of 5 pages. Each VAX page equals 512 bytes.♦

Alpha

For Alpha linking, the linker, by default, uses a minimum of 1 page. The size of an Alpha page is CPU-specific. The initial set of Alpha systems uses an 8 KB page.♦

The number of pages must be equal to or greater than the value specified in the parameter.

Description

A demand-zero image section contains uninitialized, writable pages, which do not occupy physical memory in the image file on disk, but which, when accessed during program execution, are allocated memory and initialized with binary zeros by the operating system. (For more information about demand-zero compression, see Chapter 3.)

When specifying a value for this option, be aware that a low value (less than the default value) increases the likelihood that the linker will encounter the required number of contiguous, uninitialized pages and thus may increase the number of demand-zero image sections the linker creates for the image (depending on the contents of the object modules). While this can reduce the size of the image file on disk, it can also decrease the image's paging performance during execution. Conversely, a value higher than the default value decreases the likelihood that the linker will encounter the required number of contiguous, uninitialized pages; decreases the number of demand-zero image sections the linker creates; and may increase the size of the image file on disk but provide better paging performance during execution.

The linker stops creating demand-zero image sections when the total number of image sections in the image reaches the value specified by the ISD_MAX= option or the default value. (For more information, see the description of the ISD_MAX= option.)

The DZRO_MIN= option is illegal in a link operation that produces a system image.

Example

```
$  LINK MY_PROG,SYS$INPUT/OPT
DZRO_MIN=15
Ctrl/Z
```

In this example, the value of the DZRO_MIN= is set to 15.

GSMATCH=

Sets match control parameters for a shareable image and specifies the match algorithm. This option allows you to specify whether executable images that link with a shareable image must be relinked each time the shareable image is updated and relinked.

Format

GSMATCH=keyword,major-id,minor-id

Option Values

keyword

Identifies the match algorithm used by the image activator. Specify one of the following keywords:

Keyword	Meaning
EQUAL	Directs the image activator to allow the executable image to map to the shareable image when the major ID and minor ID in the image section descriptor (ISD) of the executable image are equal to the IDs in the shareable image file.
LEQUAL	Directs the image activator to allow the executable image to map to the shareable image when the major ID is equal to, and the minor ID in the ISD of the executable image is less than or equal to the minor ID in the shareable image file.
ALWAYS	Directs the image activator to allow the executable image to map to the shareable image, regardless of the values of the major ID and minor ID, providing that the image section names are the same. Note that you must still specify values for the major ID and minor ID parameters to satisfy the requirements of the option syntax.

major-id

Specifies the major identification number. The linker uses bits 32 through 47 of the image creation time as the default value of the major ID.

minor-id

Specifies the minor identification number. The linker uses bits 16 through 31 of the image creation time as the default value of the minor ID.

Description

The GSMATCH= option causes a major identification parameter (major-id), a minor identification parameter (minor-id), and a match control keyword to be stored in the image header of the shareable image.

When an executable image is linked with a shareable image, the image header of the executable image contains an image section descriptor (ISD) for the shareable image (as well as an ISD for each image section in the image). The ISD for the shareable image contains a major ID, minor ID, and match control keyword, which the linker copies from the shareable image at link time.

When the executable image is run and the image activator begins processing the ISDs in the image header of the executable image, the image activator encounters the ISD for the shareable image. At this time, the image activator compares the image section name in the ISD to the image section name in the image header of the current shareable image file.

If the image section names do not match, the image activator does not allow the executable image to map to the shareable image, regardless of the GSMATCH parameters.

If the image section names match, the image activator compares the major ID parameters. If they do not match, the image activator does not allow the executable image to map to the shareable image unless GSMATCH=ALWAYS has been specified.

If the major ID parameters match, the image activator compares the minor ID parameters. If the relation between the minor ID parameters does not satisfy the relation specified by the match control keyword, the image activator does not allow the executable image to map to the shareable image. Then the image activator issues an error message stating that the executable image must be relinked.

The match control keyword must be the same in both the shareable and executable image files. If it is different, then the more restrictive match is used. For example, if a shareable image is linked with ALWAYS, and an executable image contains EQUAL (from an earlier version of the shareable image), then the test at activation time will be EQUAL.

Thus, to create an upwardly compatible shareable image, increment only the value of the minor ID and leave unchanged the value of the major ID. If the match control keyword is LEQUAL, the executable image that links against it will run. (If the major ID is changed, executable images can never map to the shareable image unless ALWAYS is specified.) By using this convention, you can ensure that executable images that linked with an older version of the shareable image can map to the newer version.

Examples

1.
```
$  LINK/SHARE MY_SHARE,SYS$INPUT/OPT
GSMATCH=LEQUAL,1,1000
Ctrl/Z
```

   In this example, the GSMATCH= option sets the major and minor identification numbers for this shareable image.

2.
```
$  LINK/SHARE MY_SHARE,SYS$INPUT/OPT
GSMATCH=LEQUAL,1,1001
Ctrl/Z
```

   In this example, the shareable image created in the previous example is relinked and the minor identification number is incremented. Note that executable images that link with the new version cannot map to the old version, whereas executable images that link with the old version can map to the new version.

3.
```
$  LINK/SHARE MY_SHARE,SYS$INPUT/OPT
GSMATCH=ALWAYS,0,0
```
Ctrl/Z

   By specifying the keyword ALWAYS, an executable image can run with any version of the shareable image (newer or older).

IDENTIFICATION=

Sets the image-id field in the image header.

Format

IDENTIFICATION=id-name

Option Values

id-name
The maximum length of the identification character string is 15 characters. If the string contains characters other than uppercase and lowercase A through Z, the numerals 0 through 9, the dollar sign, and the underscore, enclose it in quotation marks.

Description

The linker uses the value of the ID of the first object module processed as the default image ID when producing any kind of image with an image header. Thereafter, as long as the image-id field is not empty, it is not changed unless the linker encounters an object module that has a transfer address on the end-of-module (EOM) object record. (A transfer address is the main entry point for the image.) When it encounters an object module that contains a transfer address, the linker uses the ID from that object module as the value of the image ID.

Because shareable images normally do not have a main entry point, the image ID usually remains as the ID of the first object module processed.

Example

```
$  LINK MY_PROG,SYS$INPUT/OPT
IDENTIFICATION=MY_15_CHAR_NAME
Ctrl/Z
```

IOSEGMENT=

Specifies the amount of space to be allocated for the image I/O segment, which holds the buffers and OpenVMS RMS control information for all files used by the image.

Format

IOSEGMENT=number-of-pages[,[NO]P0BUFS]

Option Values

number-of-pages
Specifies the number of pagelets (512-byte units) to be allocated for the image I/O segment. By default, the operating system uses the value set by the IMGIOCNT system parameter to determine the size of the image I/O space.

[NO]P0BUFS
By default, the operating system allocates the I/O segment in the P1 region of the process space and, if additional space is needed, at the end of the P0 region. If you specify NOP0BUFS, you deny OpenVMS RMS additional pages in the P0 region.

Description

Specifying the value of **number-of-pages** to be greater than the value of IMGIOCNT ensures the contiguity of P1 address space, providing that OpenVMS RMS does not require more pages than the value specified. If OpenVMS RMS requires more pages than the value specified, the pages in the P0 region would be used (by default).

Note that if you specify NOP0BUFS and if OpenVMS RMS requires more pages than have been allocated for it, OpenVMS RMS issues an error message.

Example

```
$  LINK MY_PROG,SYS$INPUT/OPT
IOSEGMENT=100,P0BUFS
Ctrl/Z
```

ISD_MAX=

Specifies the maximum number of image sections allowed in the image.

Format

ISD_MAX=number-of-image-sections

Option Values

number-of-image-sections
The maximum number of image sections that may be created. You can specify the value in hexadecimal (%X), decimal (%D), or octal (%O) radix. The default is decimal radix.

Description

This option is used to control the linker's creation of demand-zero image sections by imposing an upward limit on the number of total image sections. Thus, if the linker is creating demand-zero image sections, and if the total number of image sections reaches the ISD_MAX= value, demand-zero image section creation ceases at that point. (For more information about how the linker creates demand-zero image sections, see Section 3.4.3.)

The ISD_MAX= option may be specified only in a link operation that produces an executable image. The ISD_MAX= option is illegal in a link operation that produces either a shareable or a system image.

The default value for ISD_MAX= is approximately 96. Note that any value you specify is also an approximation. The linker determines an exact ISD_MAX= value based on characteristics of the image, including the different combinations of section attributes. The exact value, however, will be equal to or slightly greater than what you specify; it will never be less.

Example

```
$  LINK MY_PROG,SYS$INPUT/OPT
ISD_MAX=126
Ctrl/Z
```

NAME=

Sets the image-name field in the image header.

Format

NAME=image-name

Option Values

image-name
A character string up to 39 characters in length. If the name contains characters other than uppercase and lowercase A through Z, the numerals 0 through 9, the dollar sign, and the underscore, enclose it in quotation marks.

Description

If the NAME= option is not specified, the string specified with /SHARE or /EXE is used for the image-name field. If no string was specified to /SHARE or /EXE, the name of the first module processed is used.

The NAME= option does not affect the name of the image file.

The image-name field is not used by the linker or librarian.

Alpha

For Alpha linking, the NAME= option also determines the string used in the shareable image list if the image contains a SYMBOL_VECTOR clause with an ALIAS keyword. The use of aliases causes the linker to set up an image with references to itself, including fixups and an entry for itself in the shareable image list.

When the file name is different from the image-name field, the image activator uses the image-name field to identify self-references in the shareable image list. The file name and the image-name field can differ if the image file is renamed, or if they are forced to differ by the NAME= option.♦

Example

```
$  LINK MY_PROG,SYS$INPUT/OPT
NAME=MY_IMAGE
Ctrl/Z
```

PROTECT=

Specifies that the image sections in one or more clusters in a shareable image should be protected from user-mode or supervisor-mode write access.

Format

PROTECT=YES/NO

Option Values

YES
Enables protection of all the clusters defined in subsequent lines in the options file by the CLUSTER= option or the COLLECT= option, up to a line containing another PROTECT= option.

NO
Disables protection for all clusters specified on subsequent lines of a linker options file by the CLUSTER= option or the COLLECT= option, up to the line containing another PROTECT=YES option. Protection is disabled by default.

Description

This option is commonly used to protect clusters that contain privileged code or data in shareable images that implement user-written system services (called privileged shareable images). For more information about creating user-written system services, see the *OpenVMS Programming Concepts Manual*.

Note that the protection applies to the image sections the linker creates from the cluster, *not* the cluster itself. A cluster is an internal construct the linker uses to organize how it processes input files. The linker communicates the actual memory requirements of an image, including its protection, to the image activator as image section specifications.

If the entire shareable image needs to be protected, specify the /PROTECT qualifier.

Example

```
$  LINK MY_PROG,SYS$INPUT/OPT
PROTECT=YES
CLUSTER=A,,,MOD1,MOD2
UNIVERSAL=ENTRY
PROTECT=NO
CLUSTER=B,,,MOD3
PROTECT=YES
COLLECT=B,PSECTX,PSECTY,PSECTZ
Ctrl/Z
```

In this example, the modules MOD1 and MOD2 in cluster A are protected; MOD3 in cluster B is not protected; and program sections PSECTX, PSECTY, and PSECTZ in cluster B are protected. Note that other linker options, such as the UNIVERSAL= option in the example, are not affected.

PSECT_ATTR=

Specifies the attributes of a program section.

Format

PSECT_ATTR=psect-name,attribute-keyword[,...]

Option Values

psect-name
Specifies the name of the program section whose attributes you want to set. The name may be a character string up to 31 characters in length.

attribute-keyword[,...]
One or more attributes, identified by a keyword, separated by commas. Section 3.2 lists the attributes with the keywords you use to specify them.

Description

Attributes not specified in a PSECT_ATTR= option remain unchanged.

If you specify a program section alignment that is greater than the target page size, the linker issues a warning and adjusts the alignment to be equal to the target page size.

Example

```
$  LINK MY_PROG,SYS$INPUT/OPT
PSECT_ATTR=PSECT_1,NOWRT
Ctrl/Z
```

In this example, the linker protects the program section PSECT_1 from write access and leaves all other attributes of PSECT_1 unchanged.

RMS_RELATED_CONTEXT=

Enables or disables RMS related name context processing. This is also known as file specification "stickiness." The default is to have RMS related name context processing enabled. This default applies at the start of each options file regardless of the setting in a previous options file. The related name context itself (the collection of data structures RMS maintains to supply the most recently specified fields) does not carry over from one linker options file to the next. That is, previously specified fields in the previous options file are not used to fill in absent fields for file specifications in the current options file.

Format

RMS_RELATED_CONTEXT=YES/NO

Option Values

YES
Enables RMS related name context processing starting with the context previously saved by a RMS_RELATED_CONTEXT=NO command. If RMS related name context processing is already enabled, this option has no effect.

NO
Disables RMS related name context processing. When RMS related name context processing is disabled, the current context is saved for a possible future RMS_ RELATED_CONTEXT=YES option. If RMS related name context processing is already disabled, specifying RMS_RELATED_CONTEXT=NO has no effect.

Description

When RMS related name processing is enabled (by default at the beginning of each options file), file specifications that do not have all fields of the file specification present will have the missing fields replaced with the corresponding fields most recently specified in earlier file specifications. When disabled, fields in the file specification that are absent are not replaced with corresponding fields of previous file specifications.

When RMS related name processing is disabled, the current related name context is saved. When RMS related name processing is enabled via this option, the saved related name context is restored.

STACK=

Specifies the size of the user-mode stack.

Format

STACK=number-of-pages

Option Values

number-of-pages
Specifies the size of the stack in pagelets (512-byte units).

Description

If you do not specify the STACK= option, the linker allocates 20 pagelets (512-byte units) for the user-mode stack. Note that additional pages for the user-mode stack are automatically allocated, if needed, during program execution.

The STACK= option may be specified only in a link operation that produces an executable image.

Note that the STACK= option is primarily used to set the stack size for images that are linked with the /P0IMAGE qualifier, where the stack could grow into the mapped image and corrupt it.

SYMBOL=

Directs the linker to define an absolute global symbol with the specified name and assign it the specified value.

Format

SYMBOL=symbol-name,symbol-value

Option Values

symbol-name
A character string up to 31 characters in length.

symbol-value
The value you want to assign to the symbol. An absolute global symbol has a fixed numeric value and is therefore not relocatable. Thus, the value must be a number.

Description

The definition of a symbol specified by the SYMBOL= option constitutes the first definition of that symbol, and it overrides subsequent definitions of the symbol in input object modules. In particular:

- If the symbol is defined as relocatable in an input object module, the linker ignores this definition, uses the definition specified by the SYMBOL= option, and issues a warning message.
- If the symbol is defined as absolute in an input object module, the linker ignores this definition and uses the definition specified by the SYMBOL= option; however, it does not issue a warning message.

For more information about symbol resolution, see Chapter 2.

Note

The SYMBOL= option cannot be used with the UNIVERSAL= option or the SYMBOL_VECTOR= option.

Example

```
$  LINK MY_PROG,SYS$INPUT/OPT
SYMBOL=MY_SYMB,15
Ctrl/Z
```

SYMBOL_TABLE= (Alpha Only)

Alpha

For Alpha linking, specifies whether the linker should include global symbols in a symbol table file produced in a link operation in which a shareable image is created. By default, the linker includes only universal symbols in a symbol table file associated with a shareable image.

Format

SYMBOL_TABLE=GLOBALS/UNIVERSALS

Option Values

GLOBALS
Specifies that the linker should include global symbols and universal symbols in the symbol table file associated with the shareable image.

UNIVERSALS
Specifies that the linker should include only universal symbols in the symbol table file associated with the shareable image.

Description

This option may be specified only in the creation of a shareable image. Note that the symbol table file affected by this option cannot be used as input in a subsequent link operation but is intended to be used with the OpenVMS Alpha System Dump Analyzer utility (SDA) as an aid to debugging.

Example

```
$  LINK/SHARE/SYMBOL_TABLE  MY_SHARE,SYS$INPUT/OPT
GSMATCH=lequal,1,1000
SYMBOL_VECTOR=(proc1=PROCEDURE,-
               proc2=PROCEDURE,-
               proc4=PROCEDURE)
SYMBOL_TABLE=GLOBALS
Ctrl/Z
```

In the example, the symbols proc1, proc2, and proc4 are declared as universal symbols. Normally, these symbols would be the only symbols to appear in a symbol table file associated with this shareable image. (The symbol table file duplicates the global symbol table of the shareable image.) However, because the SYMBOL_TABLE=GLOBALS option is specified, the linker also puts all the global symbols in the shareable image into the symbol table file. You must specify the /SYMBOL_TABLE qualifier to obtain a symbol table file.♦

SYMBOL_VECTOR= (Alpha Only)

Alpha

For Alpha linking, declares universal symbols in shareable images.

Format

SYMBOL_VECTOR=([alias-name/]symbol-name=symbol-vector-entry-type)

Option Values

alias-name
Optionally specifies an alias name for the symbol you want to declare universal. When specified, the alias name appears in the GST of the image and values associated with the name specified in the **symbol-name** parameter appear in the symbol vector of the image. Note that you can specify alias names only for symbol vector entries declared using the DATA or PROCEDURE keywords. For more information about symbol vector entry types, see the following table.

symbol-name
Specifies the name of the symbol in the shareable image that you want to declare universal.

symbol-vector-entry-type
Specifies the type of the symbol vector entry. The following table lists the types of symbol vector entries supported by the linker along with the keyword you use to specify them:

Keyword	Function
[1]DATA	Creates a symbol vector entry for data (relocatable or constant). The linker creates an entry for the symbol in the GST of the shareable image.
[1]PROCEDURE	Creates a symbol vector entry for a procedure and creates an entry for the symbol in the GST of the shareable image.
PRIVATE_DATA	Creates a symbol vector entry for data; however, the linker does not create an entry for the data in the GST of the shareable image. Thus, the symbol is not available for other modules to link against.
PRIVATE_PROCEDURE	Creates a symbol vector entry for a procedure; however, the linker does not create an entry for the procedure in the GST of the shareable image. Thus, the symbol is not available for other modules to link against.
PSECT	Creates a symbol vector entry for a program section and creates an entry for the symbol in the GST of the shareable image.

[1]You can specify an alias name for this type of symbol vector entry.

Keyword	Function
SPARE	Use this keyword to create a placeholder when you delete an entry in an existing symbol vector. SPARE allows you to preserve the order of symbol vector entries when you need to create an upwardly compatible shareable image. The SPARE keyword is used alone; it is not preceded by a symbol name and equal sign.

Description

The linker creates an entry in the global symbol table (GST) of a shareable image for each symbol listed in the SYMBOL_VECTOR= option, unless the symbol is declared private, the /NOGST qualifier is specified, or the symbol is the internal name for an alias. Symbols that appear in the GST of a shareable image are available for external modules to link against. For more information about creating and using shareable Alpha images, see Chapter 4.

Example

```
$  LINK/SHARE MY_SHARE,SYS$INPUT/OPT
GSMATCH=lequal,1,1000
SYMBOL_VECTOR=(my_proc=PROCEDURE,-
               my_proc2=PROCEDURE,-
               SPARE,-
               my_data=DATA,-
               my_data_psect=PSECT,-
               report_err=PRIVATE_PROCEDURE)
Ctrl/Z
```

This example creates a symbol vector with entries for procedures, data, and program sections.♦

UNIVERSAL= (VAX Only)

VAX

For VAX linking, declares a symbol in a shareable image as universal, causing the linker to include it in the global symbol table of a shareable image.

Format

UNIVERSAL= symbol-name[,...]

Option Values

symbol-name[,...]
The name of the symbol or symbols you want to declare universal.

Description

This option may be specified only in the creation of a shareable image.

For more information about declaring universal symbols, refer to Chapter 4.

Example

```
$  LINK/SHARE MY_SHARE,SYS$INPUT/OPT
UNIVERSAL=MY_SYMBOL
Ctrl/Z
```

In this example, the linker includes the universal symbol MY_SYMBOL in the global symbol table of the shareable image MY_SHARE.EXE.♦

A

VAX Object Language

VAX

This appendix describes the VAX object language according to Digital software specifications. The object language described is for use with all VAX family software; no subsetting will occur.

The VAX object language describes the contents of object modules to the OpenVMS Linker, as well as to the object module librarian. All language processors that produce code for execution in native mode are free to use any or all of the described object language.

This information is useful primarily to programmers writing compilers or assemblers that must generate object modules acceptable for input to the OpenVMS Linker. These programmers may also find the description of the ANALYZE/OBJECT command in the *OpenVMS DCL Dictionary* useful because it explains how the DCL command ANALYZE/OBJECT may be used to check whether an object module conforms to the requirements of the VAX object language.

This appendix contains nine sections. The first section provides an overview of the object language and lists the main types of records. Each subsequent section discusses a main record including its subrecords and fields.

The $OBJDEF macro, which defines all symbols used in this section, is available to programmers in VAX MACRO and VAX BLISS-32. VAX MACRO programmers will find this macro in the STARLET.MLB object library; VAX BLISS-32 programmers will find it in the STARLET.REQ require file.

A.1 Object Language Overview

Each object module specified as input to the linker must be in the format described by the object language. Thus, object files, object library files, and all symbol table files (which the linker creates) will conform to the format described by the object language.

The object language defines an object module as an ordered set of variable-length records. The following table shows the main record types currently available. Column 1 displays the name of the record, followed by its abbreviation. Column 2 displays the name of the record in symbolic notation; this name is placed in the first byte of the record to identify the record type. Column 3 displays the numerical code corresponding to the name in Column 2; this code may be substituted for the symbolic name in the first byte of the record, though this is not recommended.

Record Type	Symbol	Code
Header (HDR)	OBJ$C_HDR	0

Record Type	Symbol	Code
Global symbol directory (GSD)	OBJ$C_GSD	1
Text information and relocation (TIR)	OBJ$C_TIR	2
End-of-module (EOM)	OBJ$C_EOM	3
Debugger information (DBG)	OBJ$C_DBG	4
Traceback information (TBT)	OBJ$C_TBT	5
Link option specification (LNK)	OBJ$C_LNK	6
End-of-module-with-word-psect (EOMW)	OBJ$C_EOMW	7
Reserved for Digital use		8–100
Reserved always		101–200
Reserved for customer use		201–255

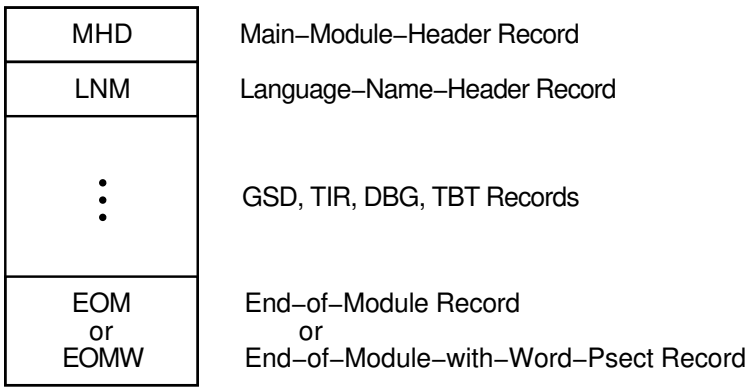
Reserved indicates that the item must not be present because it is reserved for possible future use by the linker and Digital. The linker produces an error if a reserved item is found in an object module.

All legal record types need not appear in a single object module. However, each object module must contain the following:

- One (and only one) main-module-header (MHD) record appearing first in the object module (see Section A.2.1)
- One (and only one) language-name-header (LNM) record appearing second in the object module (see Section A.2.2)
- At least one global-symbol-directory (GSD) record
- Either one end-of-module (EOM) record or one end-of-module-with-word-psect (EOMW) record, but not both, appearing last in the object module

An object module may contain any number of GSD, TIR, DBG, and TBT records, in any order, as long as they are not first or last in the object module. Figure A–1 depicts the correct ordering of records within an object module.

Figure A–1 Order of Records in an Object Module



If a field is currently ignored by the linker, you must still allocate space for it, filling it with zeros to its entire specified length.

Records in the object language may contain the names of program sections, object modules, language processors, utilities, and so on. Two methods of specifying names are implemented in the VAX object language:

- The standard naming method, which uses two fields of the record. The first field is the 1-byte name length field containing the length in characters of the name. The second field is the name field containing the name in ASCII notation.
- The single field naming method, which uses a single field containing the name in ASCII notation. The name is not preceded by a 1-byte name length field.

All name strings except the names specified in header records may be up to 31 characters long.

The following sections contain diagrams of the VAX records and subrecords. Each record or subrecord contains several fields. The left-hand column of a diagram gives, for each field, its name, symbolic representation, and length in bytes. The right-hand column gives the value (which may be a symbolic name), where appropriate, and a description of the field.

Note that many records contain identical fields; if the right-hand column of a diagram does not give a description of a field, that field has already been described in a previous record.

Also note that corresponding numerical codes for record types, subrecord types (in HDR and GSD records), and TIR commands are defined and are given in this section. Though these may be substituted for the symbolic name of the record or subrecord in the appropriate field, this practice is not recommended.

A.2 Header Records

The object language currently provides for the definition of six types of header records. Of the remaining possible types, types 7 to 100 are reserved for use by Digital, and types 101 to 255 are ignored.

The next table lists the various types of header records. Column 1 lists the name of the header type, followed by its abbreviation. Column 2 lists the related symbolic representations and column 3 lists the corresponding numerical codes.

Header Type	Symbol	Code
Main module header (MHD)[1]	MHD$C_MHD	0
Language processor name header (LNM)[1]	MHD$C_LNM	1
Source file header (SRC)[2]	MHD$C_SRC	2
Title text header (TTL)[2]	MHD$C_TTL	3
Copyright header (CPR)[2]	MHD$C_CPR	4
Maintenance status header (MTC)[2]	MHD$C_MTC	5
General text header (GTX)[2]	MHD$C_GTX	6

[1]This record is required by the linker.

[2]This record is currently ignored by the linker.

Header Type	Symbol	Code
Reserved		7–100
Ignored		101–255

The content and format of the MHD and LNM header types, both of which are required in each object module, are described in the following subsections.

Though currently ignored by the linker, the header types SRC, TTL, CPR, MTC, and GTX exist to allow the language processors to provide printable information within the object module for documentation purposes. The format of the SRC, TTL, CPR, MTC, and GTX records consists of a record type field, header type field, and a field containing the ASCII text.

The content and format of the SRC and TTL records are depicted in Section A.2.3 and Section A.2.4. The contents of these records, as well as the MTC record (which contains information about the maintenance status of the object module), are displayed in an object module analysis. (See the description of the ANALYZE /OBJECT command in the *OpenVMS DCL Dictionary*.)

A.2.1 Main Module Header Record (MHD$C_MHD)

The following presents the name, symbolic representation, and length of each field in the main module header record. The listing includes a symbolic value or an explanation of the contents of the field, where appropriate.

RECORD TYPE Name: MHD$B_RECTYP
Length: 1 byte

The record type is OBJ$C_HDR.

HEADER TYPE Name: MHD$B_HDRTYP
Length: 1 byte

The header type is MHD$C_MHD.

STRUCTURE LEVEL Name: MHD$B_STRLVL
Length: 1 byte

The structure level is OBJ$C_STRLVL. Because the format of the MHD record never changes, the structure level field is provided so that changes in the format of other records can be made without recompiling every module that conformed to the previous format.

MAXIMUM RECORD SIZE Name: MHD$W_RECSIZ
Length: 2 bytes

The maximum record size is OBJ$C_MAXRECSIZ, which is limited to 2048 bytes. This field contains the size in bytes of the longest record that can occur in the object module.

MODULE NAME LENGTH Name: MHD$B_NAMLNG
Length: 1 byte

This field contains the length in characters of the module name.

MODULE NAME Name: MHD$T_NAME

Length: variable, 1 to 31 bytes for object modules, 1 to 39 bytes for the module header at the beginning of a shareable image symbol table

This field contains the module name in ASCII format.

MODULE VERSION Name: None

Length: variable, 2 to 32 bytes

This field contains the module version number in standard name format.

CREATION TIME AND DATE Name: None

Length: 17 bytes

This field contains the module creation time and date in the fixed format dd-mmm-yyyy hh:mm, where dd is the day of the month, mmm is the standard 3-character abbreviation of the month, yyyy is the year, hh is the hour (00 to 23), and mm is the minutes of the hour (00 to 59). Note that a space is required after the year and that the total character count for this time format is 17 characters (including hyphens (-), the space, and the colon (:)).

TIME AND DATE OF LAST PATCH Name: None

Length: 17 bytes

This field is currently ignored by the linker and should be padded with 17 zeros.

A.2.2 Language Processor Name Header Record (MHD$C_LNM)

The following presents the name, symbolic representation, and length of each field in the language processor name header record. The listing includes a symbolic value or an explanation of the contents of the field, where appropriate.

RECORD TYPE Name: MHD$B_RECTYP

Length: 1 byte

The record type is OBJ$C_HDR.

HEADER TYPE Name: MHD$B_HDRTYP

Length: 1 byte

The header type is MHD$C_LNM.

LANGUAGE NAME Name: None

Length: variable

This field, which is generated by the language processor, contains the name and version of the source language that the language processor translates into the object language. It consists of a variable-length string of ASCII characters and is not preceded by a byte count of the string.

A.2.3 Source Files Header Record (MHD$C_SRC)

The following presents the name, symbolic representation, and length of each field in the source files header record. The listing includes a symbolic value or an explanation of the contents of the field, where appropriate. The contents of this record, though ignored by the linker, are displayed in an object module analysis. (See the description of the ANALYZE/OBJECT command in the *OpenVMS DCL Dictionary*.)

RECORD TYPE Name: MHD$B_RECTYP
Length: 1 byte

The record type is OBJ$C_HDR.

HEADER TYPE Name: MHD$B_HDRTYP
Length: 1 byte

The header type is MHD$C_SRC.

SOURCE FILES Name: None
Length: variable

This field, which is generated by the language processor, contains the list of file specifications from which the object module was created. It consists of a variable-length string of ASCII characters and is not preceded by a byte count of the string.

A.2.4 Title Text Header Record (MHD$C_TTL)

The following presents the name, symbolic representation, and length of each field in the title text header record. The listing includes a symbolic value or an explanation of the contents of the field, where appropriate. The contents of this record, though ignored by the linker, are displayed in an object module analysis. (See the description of the ANALYZE/OBJECT command in the *OpenVMS DCL Dictionary*.)

RECORD TYPE Name: MHD$B_RECTYP
Length: 1 byte

The record type is OBJ$C_HDR.

HEADER TYPE Name: MHD$B_HDRTYP
Length: 1 byte

The header type is MHD$C_TTL.

TITLE TEXT Name: None
Length: variable

This field, which is generated by the language processor, contains a brief description of the object module. It consists of a variable-length string of ASCII characters and is not preceded by a byte count of the string.

A.3 Global Symbol Directory Records

GSD records contain information that the linker uses to build the global symbol table and the program section table. Using this information, the linker allocates virtual address space and combines program sections into image sections.

At least one GSD record must appear in an object module.

The first field in a GSD record is the record type GSD$B_RECTYP, whose value is OBJ$C_GSD. Subsequent fields describe one or more GSD subrecords, each of which begins with the GSD type field GSD$B_GSDTYP.

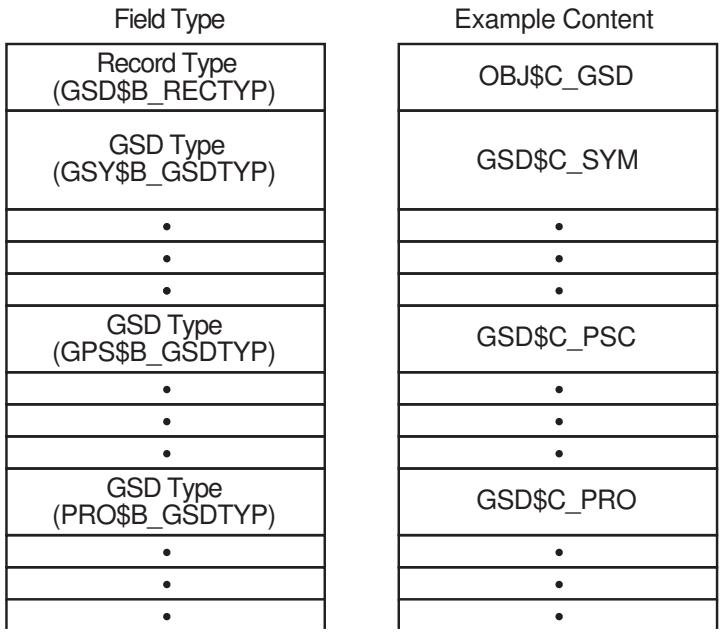
Table A–1 lists each type of GSD subrecord together with its symbolic representation and its corresponding numerical code.

Table A–1 Types of GSD Subrecords

GSD Subrecord	Symbol	Code
Program section definition	GSD$C_PSC	0
Global symbol specification	GSD$C_SYM	1
Entry point symbol and mask definition	GSD$C_EPM	2
Procedure with formal argument definition	GSD$C_PRO	3
Symbol definition with word psect	GSD$C_SYMW	4
Entry point definition with word psect	GSD$C_EPMW	5
Procedure definition with word psect	GSD$C_PROW	6
Entity ident consistency check	GSD$C_IDC	7
Environment definition/reference	GSD$C_ENV	8
Module-local symbol definition/reference	GSD$C_LSY	9
Module-local entry point definition	GSD$C_LEPM	10
Module-local procedure definition	GSD$C_LPRO	11
Program section definition in a shareable image	GSD$C_SPSC	12

Again, a single GSD record may contain one or more of the above types of subrecords. Figure A–2 displays the general format of a GSD record that contains multiple subrecords. Column 1 displays the field names; column 2 displays possible values for those fields. Note that the RECORD TYPE field appears only once at the beginning. Each subrecord begins with the GSD TYPE field.

Figure A–2 GSD Record with Multiple Subrecords



The following subsections describe the format and content of each GSD subrecord. For each subrecord, the name, length, value, and description of each field are given, where appropriate.

Note that the RECORD TYPE field is not listed with the GSD subrecords. Remember that this field must always appear first in the GSD record and that it appears only once, regardless of how many GSD subrecords are included in the GSD record.

A.3.1 Program Section Definition Subrecord (GSD$C_PSC)

The linker assigns program sections an identifying index number as it encounters their respective GSD subrecords, that is, the GSD$C_PSC records. The linker assigns these numbers in sequential order, assigning 0 to the first program section it encounters, 1 to the second, and so on, up to the maximum allowable limit of 65,535 (2^{16} –1) within any single object module.

Program sections are referred to by other object language records by means of this program section index. For example, the global symbol specification subrecord (GSD$C_SYM) contains a field that specifies the program section index. This field is used to locate the program section containing a symbol definition. Also, TIR commands use the program section index.

Of course, care is required to ensure that program sections are defined to the linker (and thus assigned an index) in proper order so that other object language records that reference a program section by means of the index are in fact referencing the correct program section.

The following presents the name, symbolic representation, and length of each field in the program section definition subrecord. The listing includes a symbolic value or an explanation of the contents of the field, where appropriate. Note that the names of fields in this subrecord begin with GPS rather than PSC.

GSD TYPE Name: GPS$B_GSDTYP
Length: 1 byte

The GSD type is GSD$C_PSC.

ALIGNMENT Name: GPS$B_ALIGN
Length: 1 byte

This field specifies the virtual address boundary at which the program section is placed. Each module contributing to a particular program section may specify its own alignment unless the program section is overlaid, in which case each module must specify the same alignment. An overlaid program section is one in which the value of flag bit 2 (GPS$V_OVR) is not equal to 0.

The contents of the alignment field is a number from 0 to 9, which is interpreted as a power of 2; the value of this expression is the alignment in bytes. The value 9 in the alignment field indicates alignment on page boundaries, which is the limit for program section alignment. Table A–2 illustrates some common alignment field values.

Table A–2 Alignment Field Values

Value	Alignment
0	1 (BYTE)
1	2 (WORD)
2	4 (LONGWORD)
3	8 (QUADWORD)
9	PAGE

FLAGS Name: GPS$W_FLAGS
Length: 2 bytes

This field is a word-length bit field, each bit indicating (when set) that the program section has the corresponding attribute. (See Section 3.2 for a description of program section attributes.) The next table lists the numbers, names, and corresponding meanings of each bit in the field:

Bit	Name	Meaning if Set
0	GPS$V_PIC	Program section is position independent.
1	GPS$V_LIB	Program section is defined in the symbol table of a shareable image, to which this image is bound. This bit is used by the linker and should not be set in user-defined program sections.
2	GPS$V_OVR	Contributions to this program section by more than one module are overlaid.
3	GPS$V_REL	Program section is relocatable. If this bit is not set, the program section is absolute and therefore contains only symbol definitions. Note that memory is not allocated for absolute program sections.
4	GPS$V_GBL	Program section is global.
5	GPS$V_SHR	Program section is shareable between two or more active processes.
6	GPS$V_EXE	Program section is executable.
7	GPS$V_RD	Program section is readable.
8	GPS$V_WRT	Program section is writable.
9	GPS$V_VEC	Program section contains change mode dispatch vectors or message vectors.
10–15		Reserved.

ALLOCATION Name: GPS$L_ALLOC
Length: 4 bytes

This field contains the length in bytes of this module's contribution to the program section. If the program section is absolute, the value of the allocation field must be zero.

PSECT NAME LENGTH Name: GPS$B_NAMLNG
Length: 1 byte

This field contains the length in characters of the program section name.

PSECT NAME Name: GPS$T_NAME
Length: variable, 1 to 31 bytes

This field contains the name of the program section in ASCII format.

A.3.2 Global Symbol Specification Subrecord (GSD$C_SYM)

The global symbol specification subrecord is used to describe the nature of a symbol (global or universal, relocatable or absolute) and how it is being used (definition or reference, weak or strong). This information is specified in the FLAGS field of the subrecord.

There are two formats for a global symbol specification subrecord, one for a symbol definition and one for a symbol reference. A symbol definition is indicated when bit 1 (GSY$V_DEF) in the FLAGS field is set, that is, when GSY$V_DEF = 1. A symbol reference is indicated when GSY$V_DEF = 0.

Section A.3.2.1 describes the format of the global symbol specification subrecord for symbol definitions; Section A.3.2.2 does the same for symbol references. Note that the PSECT INDEX and VALUE fields are present only for symbol definitions, not for symbol references.

A.3.2.1 GSD Subrecord for a Symbol Definition

The following presents the name, symbolic representation, and length of each field in the global symbol specification subrecord for a symbol definition. The listing includes a symbolic value or an explanation of the contents of the field, where appropriate.

GSD TYPE Name: SDF$B_GSDTYP
Length: 1 byte

The GSD type is GSD$C_SYM.

DATA TYPE Name: SDF$B_DATYP
Length: 1 byte

This field describes the data type of the global symbol. The data type is encoded as described in the *OpenVMS Programming Interfaces: Calling a System Routine*. The linker currently ignores this field.

FLAGS Name: SDF$W_FLAGS
Length: 2 bytes

This field is a 2-byte bit field, whose bits describe the strong global symbol. Only bits 0 through 3 are used. The following table lists the numbers, names, and meanings of each bit in the field:

Bit	Name	Meaning
0	GSY$V_WEAK	When this bit is set, a weak symbol definition is indicated; when clear, a strong symbol definition.
1	GSY$V_DEF	This bit is set for a symbol definition.
2	GSY$V_UNI	When this bit is set, a universal symbol definition is indicated; when clear, a global symbol definition. Note that when this bit is set, the value of GSY$V_WEAK is ignored.

Bit	Name	Meaning
3	GSY$V_REL	When this bit is set, the symbol is defined as relocatable; when clear, as absolute. When it is relocated, the value of a relocatable symbol is augmented by the base address of the module's contribution to the program section.
4–15		Reserved.

PSECT INDEX Name: SDF$B_PSINDX
Length: 1 byte

This field contains the program section index, described in Section A.3.1. This field identifies the program section that contains the symbol definition. It may contain a number from 0 through 255 (2^8 –1).

VALUE Name: SDF$L_VALUE
Length: 4 bytes

This field contains the value assigned to the symbol by the language processor.

NAME LENGTH Name: SDF$B_NAMLNG
Length: 1 byte

This field contains the length in characters of the symbol name.

SYMBOL NAME Name: SDF$T_NAME
Length: variable, 1 to 31 bytes

This field contains the symbol name in ASCII format.

A.3.2.2 GSD Subrecord for a Symbol Reference

The following presents the name, symbolic representation, and length of each field in the global symbol specification subrecord for a symbol reference. The listing includes a symbolic value or an explanation of the contents of the field, where appropriate.

GSD TYPE Name: SRF$B_GSDTYP
Length: 1 byte

The GSD type is GSD$C_SYM.

DATA TYPE Name: SRF$B_DATYP
Length: 1 byte

This field describes the data type of a global symbol. The data type is encoded as described in the *OpenVMS Programming Interfaces: Calling a System Routine*. The linker currently ignores this field.

FLAGS Name: SRF$W_FLAGS
Length: 2 bytes

This field is a 2-byte bit field, whose bits describe the global symbol. Only bits 0 through 3 are used. The following are the numbers, names, and corresponding meanings of each bit in the field:

Bit	Name	Meaning
0	GSY$V_WEAK	When this bit is set, a weak symbol definition is indicated; when clear, a strong symbol definition.
1	GSY$V_DEF	This bit is set for a symbol definition.
2	GSY$V_UNI	The linker ignores the value of this bit for a symbol reference.
3	GSY$V_REL	The linker ignores the value of this bit for a symbol reference.
4–15		Reserved.

NAME LENGTH Name: SRF$B_NAMLNG
Length: 1 byte

This field contains the length in characters of the symbol name.

SYMBOL NAME Name: SRF$T_NAME
Length: variable, 1 to 31 bytes

This field contains the symbol name in ASCII format.

A.3.3 Entry-Point-Symbol-and-Mask-Definition Subrecord (GSD$C_EPM)

The following presents the name, symbolic representation, and length of each field in the entry-point-symbol-and-mask-definition subrecord. The listing includes a symbolic value or an explanation of the contents of the field, where appropriate.

GSD TYPE Name: EPM$B_GSDTYP
Length: 1 byte

The GSD type is GSD$C_EPM.

DATA TYPE Name: EPM$B_DATYP
Length: 1 byte

This field describes the data type of a global symbol. The data type is encoded as described in the *OpenVMS Programming Interfaces: Calling a System Routine*. The linker currently ignores this field.

FLAGS Name: EPM$W_FLAGS
Length: 2 bytes

This field is a 2-byte bit field, whose bits describe the strong global symbol. Only bits 0 through 3 are used. The following are the numbers, names, and corresponding meanings of each bit in the field:

Bit	Name	Meaning
0	GSY$V_WEAK	When this bit is set, a weak symbol definition is indicated; when clear, a strong symbol definition.
1	GSY$V_DEF	This bit is set for a symbol definition.
2	GSY$V_UNI	When this bit is set, a universal symbol definition is indicated; when clear, a global symbol definition. Note that when this bit is set, the value of GSY$V_WEAK is ignored.

Bit	Name	Meaning
3	GSY$V_REL	When this bit is set, the symbol is defined as relocatable; when clear, as absolute. When it is relocated, the value of a relocatable symbol is augmented by the base address of the module's contribution to the program section.
4–15		Reserved.

PSECT INDEX Name: EPM$B_PSINDX
Length: 1 byte

This field contains the program section index, described in Section A.3.2. This field identifies the program section that contains the symbol definition. It may contain a number from 0 through 255 (2^8 –1).

VALUE Name: EPM$L_ADDRS
Length: 4 bytes

This field contains the value assigned to the symbol by the language processor.

ENTRY MASK Name: EPM$W_MASK
Length: 2 bytes

The point of entry to a procedure invoked by a CALLS or CALLG instruction has an entry mask. Transfer vectors to these procedures also use entry masks. The language processor uses a TIR command to direct the linker to insert the mask at the procedure entry point or at the transfer vector.

NAME LENGTH Name: EPM$B_NAMLNG
Length: 1 byte

This field contains the length in characters of the symbol name.

SYMBOL NAME Name: EPM$T_NAME
Length: variable, 1 to 31 bytes

This field contains the symbol name in ASCII format.

A.3.4 Procedure-with-Formal-Argument-Definition Subrecord (GSD$C_PRO)

The following presents the name, symbolic representation, and length of each field in the procedure-with-formal-argument-definition subrecord. The listing includes a symbolic value or an explanation of the contents of the field, where appropriate.

GSD TYPE Name: PRO$B_GSDTYP
Length: 1 byte

The GSD type is GSD$C_PRO.

DATA TYPE Name: PRO$B_DATYP
Length: 1 byte

This field describes the data type of a global symbol. The data type is encoded as described in the *OpenVMS Programming Interfaces: Calling a System Routine*. The linker currently ignores this field.

FLAGS Name: PRO$W_FLAGS

Length: 2 bytes

This field is a 2-byte bit field, whose bits describe the strong global symbol. Only bits 0 through 3 are used. The following are the numbers, names, and corresponding meanings of each bit in the field:

Bit	Name	Meaning
0	GSY$V_WEAK	When this bit is set, a weak symbol definition is indicated; when clear, a strong symbol definition.
1	GSY$V_DEF	This bit is set for a symbol definition.
2	GSY$V_UNI	When this bit is set, a universal symbol definition is indicated; when clear, a global symbol definition. Note that when this bit is set, the value of GSY$V_WEAK is ignored.
3	GSY$V_REL	When this bit is set, the symbol is defined as relocatable; when clear, as absolute. When it is relocated, the value of a relocatable symbol is augmented by the base address of the module's contribution to the program section.
4–15		Reserved.

PSECT INDEX Name: PRO$B_PSINDX

Length: 1 byte

This field contains the program section index, described in Section A.3.2. This field identifies the program section that contains the symbol definition. It may contain a number from 0 through 255 (2^8 –1).

VALUE Name: PRO$L_ADDRS

Length: 4 bytes

This field contains the value assigned to the symbol by the language processor.

ENTRY MASK Name: PRO$W_MASK

Length: 2 bytes

The point of entry to a procedure invoked by a CALLS or CALLG instruction has an entry mask. Transfer vectors to these procedures also use entry masks. The language processor uses a TIR command to direct the linker to insert the mask at the procedure entry point or at the transfer vector.

NAME LENGTH Name: PRO$B_NAMLNG

Length: 1 byte

This field contains the length in characters of the symbol name.

SYMBOL NAME Name: PRO$T_NAME

Length: variable, 1 to 31 bytes

This field contains the symbol name in ASCII format.

MINIMUM ACTUAL ARGUMENTS Name: FML$B_MINARGS

Length: 1 byte

This field specifies the minimum number of arguments required for a valid call to this procedure. Permissible values are 0 to 255. The number must include the function return value if it exists.

MAXIMUM ACTUAL ARGUMENTS Name: FML$B_MAXARGS
Length: 1 byte

This field specifies the maximum number of arguments that may be included in a valid call to this procedure. Permissible values are 0 to 255. Note that the linker does not perform argument validation; however, it will issue a warning message if the value of MINIMUM ACTUAL ARGUMENTS is greater than the value of MAXIMUM ACTUAL ARGUMENTS.

FORMAL ARG1 DESCRIPTOR Name: None
Length: variable, 2 to 256 bytes

This field specifies a single formal argument descriptor. There is a FORMAL ARG DESCRIPTOR field for each formal argument specified. This field contains three subfields; its format is displayed at the end of this section.

FORMAL ARGn DESCRIPTOR Name: None
Length: variable, 2 to 256 bytes

This field specifies the last (*n*) formal argument descriptor and is identical in format to previous formal argument descriptor fields. Note that if there is a function return value, this field specifies it.

Each FORMAL ARG DESCRIPTOR field contains three subfields. The content and format are as follows:

ARG VAL CTL ARG$B_VALCTL
Length: 1 byte

This field is the argument validation control byte. Bits 0 and 1 together define the argument passing mechanism (ARG$V_PASSMECH). Bits 2 through 7 are ignored. There are four possible values for ARG$V_PASSMECH corresponding to the four possible values (0 through 3) resulting from the combination of the values of bits 0 and 1:

ARG$V_PASSMECH	Name	Description
0	ARG$K_UNKNOWN	Unspecified
1	ARG$K_VALUE	By value
2	ARG$K_REF	By reference
3	ARG$K_DESC	By descriptor

REM BTYE CNT Name: ARG$B_BYTECNT
Length: 1 byte

This field contains the length in bytes of the remainder of the argument descriptor. Permissible values are 0 through 255. Because the linker does not perform argument validation, it uses the value of this field only to determine how many subsequent bytes to ignore.

DETAILED ARGUMENT DESCRIPTION Name: None

Length: variable, 0 to 255 bytes

This field contains a detailed description of the argument. The linker currently ignores this field.

Note that if bits 2 through 7 in ARG$B_VALCTL are not equal to 0 or the value of ARG$B_BYTECNT is not equal to 0, or both, then recompiling the object module may be necessary if that argument validation is implemented in a future version of the linker.

A.3.5 Symbol-Definition-with-Word-Psect Subrecord (GSD$C_SYMW)

This subrecord is identical in format to the global symbol definition subrecord described in Section A.3.2.1, with the exception that the PSINDX field in this subrecord is 2 bytes long.

The field names in this record begin with SYMW, instead of SYM as in the global symbol definition subrecord. For example, in this subrecord the name of the GSD TYPE is SYMW$B_GSDTYP.

Note that the name of the PSECT INDEX field in this subrecord is SYMW$W_ PSINDX.

A.3.6 Entry-Point-Definition-with-Word-Psect Subrecord (GSD$C_EPMW)

This subrecord is identical in format to the entry-point-symbol-and-mask-definition subrecord described in Section A.3.3, with the exception that the PSINDX field in this subrecord is 2 bytes long.

The field names in this record begin with EPMW, instead of EPM as in the entry-point-symbol-and-mask-definition subrecord. For example, in this subrecord the name of the GSD TYPE is EPMW$B_GSDTYP.

Note that the name of the PSECT INDEX field in this subrecord is EPMW$W_ PSINDX.

A.3.7 Procedure-Definition-with-Word-Psect Subrecord (GSD$C_PROW)

This subrecord is identical in format to the procedure with formal argument definition subrecord described in Section A.3.4, with the exception that the PSINDX field in this subrecord is 2 bytes long.

The field names in this record begin with PROW, instead of PRO as in the procedure-with-formal-argument-definition subrecord. For example, in this subrecord the name of the GSD TYPE is PROW$B_GSDTYP.

Note that the name of the PSECT INDEX field in this subrecord is PROW$W_ PSINDX.

A.3.8 Entity-Ident-Consistency-Check Subrecord (GSD$C_IDC)

This subrecord allows for the consistency checking of an entity at link time. Using this subrecord, a compiler may emit code to check the consistency of any type of entity that has either an ASCIC or binary ident string associated with it.

The following presents the name, symbolic representation, and length of each field in the entity ident consistency check subrecord. The listing includes a symbolic value or an explanation of the contents of the field, where appropriate.

GSD TYPE Name: IDC$B_GSDTYP

Length: 1 byte

The GSD type is GSD$C_IDC.

FLAGS Name: IDC$W_FLAGS
Length: 2 bytes

The FLAGS field is a 2-byte bit field, of which only the first five bits are used. When bit 0 (IDC$V_BINIDENT) is set, that is, when IDC$V_BINIDENT = 1, the ident is a 32-bit binary value; when clear, the ident is an ASCIC string.

Bits 1 and 2 (IDC$V_IDMATCH) specify the ident match control for 32-bit binary idents and are thus only significant when IDC$V_BINIDENT = 1. IDC$V_ MATCH may take two values: 0 (IDC$C_LEQ) or 1 (IDC$C_EQUAL).

When IDC$V_MATCH = IDC$C_LEQ, the binary ident of the entity specified in the subrecord must be less than or equal to the binary ident of the entity that is listed in the entity name table.

When IDC$V_MATCH = IDC$C_EQUAL, the binary ident of the entity specified in the subrecord must be equal to the binary ident of the entity that is listed in the entity name table. Remaining values of IDC$V_MATCH, that is, the numbers 2 to 8, are reserved.

Bits 3 to 5 (IDC$V_ERRSEV) specify error message severity levels. When the value of IDC$V_ERRSEV is 0, the message severity is warning; when 1, success; when 2, error; when 3, informational; when 4, severe.

Bits 6 to 15 in the FLAGS field are reserved.

NAME LENGTH Name: IDC$B_NAMLNG
Length: 1 byte

This field contains the length in characters of the entity name.

ENTITY NAME Name: IDC$T_NAME
Length: variable, 1 to 31 bytes

This field contains the entity name in ASCII format.

IDENT LENGTH Name: None
Length: 1 byte

This field contains the length in bytes of the ident string. For binary idents, this field contains the value 4.

IDENT STRING Name: None
Length: variable, 1 to 31 bytes

This field contains the ident string. The ident string may be an ASCIC string or a 32-bit binary value. If this string specifies a 32-bit binary value, it consists of 24 bits of minor ident and 8 bits of major ident, analogous to the global section match values for a shareable image. If this string specifies an ASCIC string, its length is variable.

OBJECT NAME LENGTH Name: None
Length: 1 byte

This field contains the length in bytes of the name of the entity in the entity name table.

OBJECT NAME Name: None
Length: 1 to 31 bytes

This field contains the name of the entity in the entity name table.

When this GSD subrecord is processed during Pass 1, the linker searches the entity name table (which is a single name table for all entity types) for an entity of the same name. If the linker locates such an entity, it compares the idents. If the idents do not satisfy the specified match control value, the linker issues a warning message.

A.3.9 Environment-Definition/Reference Subrecord (GSD$C_ENV)

The following presents the name, symbolic representation, and length of each field in the environment and reference subrecord. The listing includes a symbolic value or an explanation of the contents of the field, where appropriate.

GSD TYPE Name: ENV$B_GSDTYP
Length: 1 byte

The GSD type is GSD$C_ENV.

FLAGS Name: ENV$W_FLAGS
Length: 2 bytes

This is a 2-byte bit field.

Bit 0 (ENV$V_DEF) is a bit mask. When ENV$V_DEF = 1, the subrecord describes an environment definition; when clear, an environment reference.

Bit 1 (ENV$V_NESTED) is set to indicate that the current environment is nested within another environment. The parent environment index is ENV$W_ ENVINDX.

Bits 2 through 15 are not used.

ENVIRONMENT INDEX Name: ENV$W_ENVINDX
Length: 2 bytes

This field contains the environment index, a number from 0 through 65,535. As with a program section, each environment is assigned a number (its index) that the TIR records and GSD records use to refer to it.

If the current environment is contained within another environment (for example, a nested environment), then this field contains the index of the surrounding or ‘‘parent’’ environment. Otherwise, this field is 0. However, because a 0 could also be interpreted as the current environment being contained within environment 0, the ENV$V_NESTED bit may be tested to clear up this ambiguity.

NAME LENGTH Name: ENV$B_NAMLNG
Length: 1 bytes

This field contains the length in characters of the environment name.

ENVIRONMENT NAME Name: ENV$T_NAME
Length: variable, 1 to 31 bytes

This field contains the environment name.

The linker reports any undefined environments at the end of Pass 1. Note that a total of 65,535 environments may be defined or referenced in any single object module.

A.3.10 Module-Local Symbol Definition/Symbol Reference Subrecord (GSD$C_LSY)

This subrecord, as with the global symbol specification subrecord described in Section A.3.2, has two formats: one for a symbol definition and one for a symbol reference. The following subsections describe each of these formats.

A.3.10.1 Module-Local Symbol Definition

The format of a module-local symbol definition is identical to the format of the symbol-definition-with-word-psect subrecord described in Section A.3.5, with the following exceptions:

- The field names in this record begin with LSDF instead of SYMW as in the symbol-definition-with-word-psect subrecord. For example, in this subrecord the name of the GSD TYPE is LSDF$B_GSDTYP.
- The module-local symbol definition subrecord contains an additional field, directly following the FLAGS field and preceding the PSINDX field: the ENVIRONMENT INDEX field (LSDF$W_ENVINDX).
- Only two of the four bits in the FLAGS field are used in this subrecord. Because this is a definition, LSY$V_DEF must be set. Bit 3 (LSY$V_REL) is set or not set depending on whether the module-local symbol is relocatable or not relocatable, respectively. Bit 0 (LSY$V_WEAK) and bit 2 (LSY$V_UNI) are ignored because a module-local symbol may not be defined as either weak or universal.

A.3.10.2 Module-Local Symbol Reference

The format of a module-local symbol reference is identical to the format of the global symbol reference subrecord described in Section A.3.2.2, with the following exceptions:

- The field names in this record begin with LSRF instead of SRF as in the global symbol reference subrecord. For example, in this subrecord the name of the GSD TYPE is LSRF$B_GSDTYP.
- The module-local symbol reference subrecord contains an additional field directly following the FLAGS field and preceding the NAME LENGTH field: the ENVIRONMENT INDEX field (LSRF$W_ENVINDX).
- Only bit 1 (LSY$V_DEF) in the FLAGS field is used. Because this is a reference, LSY$V_DEF must be reset. Bits 0, 2, and 3 are ignored.

A.3.11 Module-Local Entry-Point-Definition Subrecord (GSD$C_LEPM)

This subrecord is identical in format to the entry-point-definition-with-word-psect subrecord described in Section A.3.6, with the following exceptions:

- The field names in this record begin with LEPM instead of EPMW as in the entry-point-definition-with-word-psect subrecord. For example, in this subrecord the name of the GSD TYPE is LEPM$B_GSDTYP.
- The module-local entry point definition subrecord contains an additional field directly following the FLAGS field and preceding the PSINDX field: the ENVIRONMENT INDEX field (LEPM$W_ENVINDX).

A.3.12 Module-Local Procedure-Definition Subrecord (GSD$C_LPRO)

This subrecord is identical in format to the procedure-definition-with-word-psect subrecord described in Section A.3.7, with the following exceptions:

- The field names in this record begin with LPRO instead of PROW as in the procedure-definition-with-word-psect subrecord. For example, in this subrecord the name of the GSD TYPE is LPRO$B_GSDTYP.
- The module-local procedure definition subrecord contains an additional field directly following the FLAGS field and preceding the PSINDX field: the ENVIRONMENT INDEX field (LEPM$W_ENVINDX).

A.3.13 Program-Section-Definition-in-Shareable-Image Subrecord (GSD$C_SPSC)

This subrecord is identical in format to the program section definition subrecord described in Section A.3.1, with the following exceptions:

- This subrecord is generated only by the linker and is reserved to the linker.
- This subrecord is only legal in the global symbol table (GST) of a shareable image.
- This subrecord contains an additional 4-byte field directly following the ALLOCATION field and preceding the PSECT NAME LENGTH field: the BASE field (SGPS$L_BASE). The BASE field contains the base address of this program section in the shareable image.
- The field names in this subrecord begin with SGPS instead of GPS as in the program section definition subrecord. For example, in this subrecord the name of the GSD TYPE is SGPS$B_GSDTYP.

A.3.14 Vectored-Symbol-Definition Subrecord (GSD$C_SYMV)

This subrecord is identical in format to the global symbol definition subrecord described in Section A.3.2.1, with the exception of an additional longword field, SDFV$L_VECTOR.

The field names in this record begin with SDFV instead of SDF as in the global symbol definition subrecord. For example, in this subrecord the name of the GSD type field is SDFV$B_GSDTYPE instead of SDF$B_GSDTYPE.

This subrecord is reserved for use by Digital only.

A.3.15 Vectored-Entry-Point-Definition Subrecord (GSD$C_EPMV)

This subrecord is identical in format to the entry point symbol and mask definition subrecord described in Section A.3.3, with the exception of an additional longword field, EPMV$L_VECTOR.

The field names in this record begin with EPMV instead of EPM as in the entry-point-symbol-and-mask-definition subrecord. For example, in this subrecord the name of the GSD type field is EPMV$B_GSDTYPEl instead of EPM$B_ GSDTYPEl.

This subrecord is reserved for use by Digital only.

A.3.16 Vectored-Procedure-Definition Subrecord (GSD$C_PROV)

This subrecord is identical in format to the procedure definition subrecord described in Section A.3.4, with the exception of an additional longword field, PROV$L_VECTOR.

The field names in this record begin with PROV instead of PRO as in the procedure definition subrecord. For example, in this subrecord the name of the GSD type field is PROV$B_GSDTYP instead of PRO$B_GSDTYP.

This subrecord is reserved for use by Digital only.

A.3.17 Symbol-Definition-with-Version-Mask Subrecord (GSD$C_SYMM)

This subrecord is identical in format to the global symbol definition subrecord described in Section A.3.2.1, with the exception of an additional longword field, SDFM$L_VERSION_MASK.

The field names in this record begin with SDFM instead of SDF as in the global symbol definition subrecord. For example, in this subrecord the name of the GSD type field is SDFM$B_GSDTYPE instead of SDF$B_GSDTYPE.

This subrecord is reserved for use by Digital only.

A.3.18 Entry-Point-Definition-with-Version-Mask Subrecord (GSD$C_EPMM)

This subrecord is identical in format to the entry-point-symbol-and-mask-definition subrecord described in Section A.3.3, with the exception of an additional longword field, EPMM$L_VERSION_MASK.

The field names in this record begin with EPMM instead of EPM as in the entry-point-symbol-and-mask-definition subrecord. For example, in this subrecord the name of the GSD type field is EPMM$B_GSDTYPE instead of EPM$B_GSDTYPE.

This subrecord is reserved for use by Digital only.

A.3.19 Procedure-Definition-with-Version-Mask Subrecord (GSD$C_PROM)

This subrecord is identical in format to the procedure definition subrecord described in Section A.3.4, with the exception of an additional longword field, PROV$L_VECTOR.

The field names in this record begin with PROV instead of PRO as in the procedure definition subrecord. For example, in this subrecord the name of the GSD type field is PROV$B_GSDTYP instead of PRO$B_GSDTYP.

This subrecord is reserved for use by Digital only.

A.4 Text Information and Relocation Records (OBJ$C_TIR)

A text information and relocation record contains commands and data that the linker uses to compute and record the contents of the image.

The linker's creation of the binary content of an image file is controlled by the language processor using the commands contained in TIR records.

A TIR record consists of the RECORD TYPE field (TIR$B_RECTYP) followed by one COMMAND field and one DATA field for each TIR command in the record. Because a TIR record may contain many TIR commands, it may be quite long. It may not, however, exceed the record size limit for the object module. This limit is set in the MAXIMUM RECORD SIZE field (MHD$W_RECSIZ) in the main module header record (MHD$C_MHD).

The fields in a TIR record are described in this section. Note that the description given for the first COMMAND and first DATA field applies to all TIR commands but one, the STORE IMMEDIATE command, while the description given for the second COMMAND and second DATA field applies only to the STORE IMMEDIATE command. This does not imply that the STORE IMMEDIATE command must follow other TIR commands; TIR commands may appear within the TIR record in any order.

RECORD TYPE Name: TIR$RECTYP
Length: 1 byte

The record type is OBJ$C_TIR.

COMMAND Name: None
Length: 1 byte

This field designates the TIR command. This description of the COMMAND field applies to all TIR commands except the STORE IMMEDIATE command, which is described in the following COMMAND field. There are 85 TIR commands (excluding the STORE IMMEDIATE command), and each has a positive number ranging from 0 to 84 that is used to encode the command in the field.

DATA Name: ENV$T_NAME
Length: variable

This field contains the data upon which the previously specified (in the COMMAND field) TIR command operates. The length of this field is implied by the command itself. For example, if the previous COMMAND field specifies a stack-byte command, the length of this DATA field is 1 byte.

COMMAND Name: None
Length: 1 byte

This field contains the name of a TIR command. This description of the COMMAND field applies only to the STORE IMMEDIATE command. The STORE IMMEDIATE command is designated by any negative number (bit 7 is set) in the COMMAND field. The absolute value of the COMMAND field is the length in bytes of the following DATA field. The STORE IMMEDIATE command directs the linker to write the contents of the DATA field directly to the output image file, without using the internal stack. Thus, from 1 to 128 bytes of data may be immediately stored by means of this command.

DATA Name: ENV$T_NAME
Length: variable

This field contains the data upon which the previously specified TIR command operates. The length of this field is given by the command itself. When the previous COMMAND field contains a STORE IMMEDIATE command, the length of this DATA field is the absolute value of the COMMAND field.

Most TIR commands operate on values on the linker's internal stack, which is longword-aligned at all times. Values placed on the stack by TIR commands are retained during processing of other record types; however, the stack must be completely collapsed when the EOM or EOMW record is processed. The minimum stack space available is never less than 25 longwords.

TIR commands fall into four categories:

- Stack commands place data on the stack.

- Store commands pop data from the stack and write it to the output image file. The only exception is the STORE IMMEDIATE command, which writes data directly to the image file without using the stack.
- Operator commands perform arithmetic operations on data currently on the stack.
- Control commands reposition the linker's location counter.

In the interest of linker performance, a few implementation decisions and their possible side effects should be noted.

- The linker does not execute a STORE REPEATED command when the value being stored is zero. Such a command is, in effect, a null operation. The reason for this is twofold. First, the pages of an image are guaranteed to be zero anyway, unless otherwise initialized by the compiler. Second, demand-zero compression works only on pages that have not been initialized; thus, not allowing a STORE REPEATED command to initialize a page with zeros permits the linker to compress that page.
- The linker is a two-pass processor of object modules. It ignores TIR records on its first pass but processes them on its second pass to produce the output image file. TIR records are not processed if the linking operation is aborted because of a command or link-time error before the linker's second pass. Consequently, user or compiler errors (such as truncation errors) that are usually detectable during the linker's second pass are not detected in this case.

  TIR commands are described in the following four subsections. Section A.4.1 discusses the stack commands; Section A.4.2, the store commands; Section A.4.3, operator commands; and Section A.4.4, control commands. The commands are presented in numerical order, based on their equivalent numerical codes (in decimal), except for the STORE IMMEDIATE command, which does not have a specific numerical code. The STORE IMMEDIATE command has been described under the second TIR COMMAND.

A.4.1 Stack Commands

The stack commands place bytes, words, and longwords on the stack. Byte and word stack commands (except those that stack the values of global symbols or addresses) have signed-extension-to-longword format and zero-extension-to-longword format.

The data placed on the stack is taken from one of the following sources:

- The DATA field directly following the COMMAND field
- A global symbol
- A computation derived from the base address of a program section

Table A–3 lists the stack commands and related codes, together with a brief description of each command.

Table A–3 Stack Commands

Code	Command	Description
0	TIR$C_STA_GBL (Stack Global)	Data is the name of the global symbol in standard name format. The command stacks the 32-bit binary value of the stacks symbol.
1	TIR$C_STA_SB (Stack Signed Byte)	Data is a 1-byte constant, which is sign-extended to 32 bits.
2	TIR$C_STA_SW (Stack Signed Word)	Data is a 2-byte constant, which is sign-extended to 32 bits.
3	TIR$C_STA_LW (Stack Longword)	Data is a 4-byte constant.
4	TIR$C_STA_PB (Stack Psect Base Plus Byte Offset)	Data is a 1-byte program section index followed by a single signed byte offset[1]. The psect base and byte offset are added and stacked.
5	TIR$C_STA_PW (Stack Psect Base Plus Word Offset)	Data is a 1-byte program section index followed by a single signed word offset[1]. The psect base and word offset are added and stacked.
6	TIR$C_STA_PL (Stack Psect Base Plus Longword Offset)	Data is a 1-byte program section index followed by a single signed longword offset[1]. The psect base and longword offset are added and stacked.
7	TIR$C_STA_UB (Stack Unsigned Byte)	Same as Stack Signed Byte except that the value is zero-extended to 32 bits.
8	TIR$C_STA_UW (Stack Unsigned Word)	Same as Stack Signed Word except that the value is zero-extended to 32 bits.
9	TIR$C_STA_BFI (Stack Byte From Image)	This command is reserved for use by Digital. The top longword on the stack is used as an address, in the image, from which to retrieve a byte. The byte is zero-extended and replaces the top longword on the stack.
10	TIR$C_STA_WFI (Stack Word From Image)	This command is reserved for use by Digital. It is the word equivalent of the previous command.
11	TIR$C_STA_LFI (Stack Longword From Image)	This command is reserved for use by Digital. It is the longword equivalent of the previous two commands.
12	TIR$C_STA_EPM (Stack Entry Point Mask)	This command has the same format as the Stack Global command. However, it stacks the entry point mask (unsigned stack's word) that accompanies the symbol definition, rather than the symbol value. An error results if the symbol is not an entry point.

[1]Although this command provides for a signed offset value, negative offsets are rarely correct. Note too that the base address is that of this module's contribution to the program section.

(continued on next page)

Table A–3 (Cont.) Stack Commands

Code	Command	Description
13	TIR$C_STA_CKARG (Compare Procedure Argument and Stack for TRUE or FALSE)	This command checks to see whether the argument passing mechanism (ARG$V_PASSMECH) in the formal argument descriptor matches the argument passing mechanism in the actual argument descriptor. The DATA field for this command consists of the ASCIC symbol name in standard name format (1 count byte followed by the symbol name (1 to 31 bytes)). This is followed by the 1-byte argument index and the actual argument descriptor. The format of the actual argument descriptor is the same as the format of the formal argument descriptor described in Section A.3.4. The linker compares the values of ARG$V_PASSMECH for the formal and actual argument descriptors. If these values agree or if there is no formal argument descriptor, the linker places the TRUE value on top of the stack; otherwise, it stacks the FALSE value.
14	TIR$C_STA_WPB (Stack Psect Base Plus Byte Offset with Word Psect)	Same as TIR$C_STA_PB except the program section index is a word rather than a byte.
15	TIR$C_STA_WPW (Stack Psect Base Plus Word Offset with Word Psect)	Same as TIR$C_STA_PW except the program section index is a word rather than a byte.
16	TIR$C_STA_WPL (Stack Psect Base Plus Longword Offset with Word Psect)	Same as TIR$C_STA_PL except the program section index is a word rather than a byte.
17	TIR$C_STA_LSY (Stack Local Symbol Value)	Data is a 2-byte environment index followed by the ASCIC symbol name in standard name format.
18	TIR$C_STA_LIT (Stack Literal)	Data is a 1-byte index of the literal to be stacked. If the literal has not been defined, the linker stacks zero and issues an error message.
19	TIR$C_STA_LEPM (Stack Local Symbol Entry Point Mask)	This command has the same format as the Stack Local Symbol Value command and the same action as the Stack Entry Point Mask command.

A.4.2 Store Commands

The store commands pop the top longword from the stack and write it to the output image file. Several store commands provide validation of the quantity being stored, with the possibility of issuing truncation errors during the operation. After a store command is executed, the location counter is pointing to the next byte in the output image.

Table A–4 lists the store commands and related codes, together with a brief description of each command.

Table A–4 Store Commands

Code	Command	Description
20	TIR$C_STO_SB (Store Signed Byte)	Low byte is written to image. Bits 31:7 must be identical.
21	TIR$C_STO_SW (Store Signed Word)	Low word is written to image. Bits 31:15 must be identical.
22	TIR$C_STO_LW (Store Longword)	One longword is written to image.
23	TIR$C_STO_BD (Store Byte Displaced)	Current location counter plus 1 is subtracted from the top longword on the stack. Low byte of resulting value is written to image. Bits 31:7 must be identical.
24	TIR$C_STO_WD (Store Word Displaced)	Current location counter plus 2 is subtracted from the top longword on the stack. Low word of resulting value is written to image. Bits 31:15 must be identical.
25	TIR$C_STO_LD (Store Longword Displaced)	Current location counter plus 4 is subtracted from the top longword on the stack. Resulting value is written to image.
26	TIR$C_STO_LI (Store Short Literal)	Low byte of top longword on the stack is written to image. Bits 31:6 must be zero.
27	TIR$C_STO_PIDR (Store Position-Independent Data Reference)	The longword on top of the stack is the address of a data item. If the address is absolute, the longword is written to the image. If the address is relocatable, the linker stores information in the image file to allow the image activator to initialize the location when the image is run.
28	TIR$C_STO_PICR (Store Position-Independent Code Reference)	The purpose of this command is to generate a position-independent code reference. The top longword on stack is the address of an item to which a position-independent instruction makes reference. If the item is absolute, the byte 9F hexadecimal (the operand specifier for absolute addressing mode) followed by the top longword on the stack are written to the image, and the location counter is incremented. If the item is relocatable, the byte EF hexadecimal (the operand specifier for longword relative addressing mode) is written to the image; the top longword on the stack is Store Longword Displaced (see TIR$C_STO_LD); and the location counter is incremented. If the item is relocatable and contained in a shareable image, the byte FF (the operand specifier for longword relative deferred addressing mode) is written to the image; the top longword on the stack is Store Longword Displaced (target is in the EXIT vector); and the location counter is incremented.

(continued on next page)

Table A–4 (Cont.) Store Commands

Code	Command	Description
29	TIR$C_STO_RSB (Store Repeated Signed Byte)	The longword on top of the stack is used as a repeat count. The low byte of the next longword on the stack is then written to the image the indicated number of times. Both longwords are removed from the stack upon completion. See note in Section A.4 regarding the use of this command with zeros.
30	TIR$C_STO_RSW (Store Repeated Signed Word)	Same as above command except that a word rather than a byte is written.
31	TIR$C_STO_RL (Store Repeated Longword)	Same as previous two commands except that a longword is written.
32	TIR$C_STO_VPS (Store Arbitrary Field)	This command writes a bit field to the image. The data field consists of an unsigned byte containing the value p, followed by another unsigned byte containing the value s. Bits 0 to s–1 of the top longword on the stack are written to the image starting at bit p of the current location. Only the specified bits of the image are altered. After the operation, the location counter is the address of the byte containing bit (p+s) of the location modified. Note that the value of p+s must be greater than zero and less than or equal to either 32 or ((P+8)/8)8–1, whichever is less. In other words, the bit field must be contained within a single byte.
33	TIR$C_STO_USB (Store Unsigned Byte)	Same as TIR$C_STO_SB except that bits 31:8 must be zero.
34	TIR$C_STO_USW (Store Unsigned Word)	Same as TIR$C_STO_SW except that bits 31:16 must be zero.
35	TIR$C_STO_RUB (Store Repeated Unsigned Byte)	Same as TIR$C_STO_RSB except that bits 31:8 of the stored byte must be zero.
36	TIR$C_STO_RUW (Store Repeated Unsigned Word)	Same as TIR$C_STO_RSW except that bits 31:16 of the stored word must be zero.
37	TIR$C_STO_B (Store Byte)	This command writes the low byte of the top longword on the stack to the image file. It thus permits any 8-bit value (from –128 to 255) to be written to the image. If the top longword on the stack is negative, then bits 31:7 must be 1; if positive, then bits 31:8 must be zero.
38	TIR$C_STO_W (Store Word)	This command writes the low word of the top longword on the stack to the image file. It thus permits any 16-bit value (from –32768 to 65535) to be written to the image. If the top longword on the stack is negative, bits 31:15 must be 1; if positive, the bits 31:16 must be zero.

(continued on next page)

Table A–4 (Cont.) Store Commands

Code	Command	Description
39	TIR$C_STO_RB (Store Repeated Byte)	The top longword on the stack is used as a repeat count. The low byte of the next longword on the stack is then written to the image the indicated number of times. This is the repeated version of Store Byte (see TIR$C_STO_B).
40	TIR$C_STO_RW (Store Repeated Word)	This is the word version of the Store Repeated Byte command.
41	TIR$C_STO_RIVB (Store Repeated Immediate Variable Bytes)	Data is a 1-byte count *n* field followed by *n* bytes of data. These *n* bytes of data are written to the image the number of times specified by the top longword on the stack (which is removed from the stack upon completion of the command). If the top longword on the stack is zero, nothing is stored.
42	TIR$C_STO_PIRR (Store Position-Independent Reference Relative)	The longword (longword 1) on the top of the stack is the address of a data item. If the data item is absolute, the command is the same as the Store Longword command except that the next longword on the stack (following the top one) is also removed from the stack upon completion of the command. If the data item is relocatable, the value of the second longword (longword 2) on the stack is checked. If its value is –1, the current value of the location counter is substituted for longword 2, and the value stored is longword 1 minus longword 2. Both longwords are removed from the stack upon completion of the command.
43–49		Reserved.

A.4.3 Operator Commands

Operator commands perform arithmetic operations on the top one or two longwords on the stack. Upon completion of the operation, the result is the top longword on the stack.

The linker evaluates expressions in Post Fix Polish form. All arithmetic operations are performed in signed 32-bit two's complement integers. There is no provision for floating-point, string, or quadword computation. Attempts to divide by zero produce a zero result and a nonfatal warning message.

Table A–5 lists the operator commands and related codes, together with a brief description of each command.

Table A–5 Operator Commands

Code	Command	Description
50	TIR$C_OPR_NOP (No-Operation)	No operation results.
51	TIR$C_OPR_ADD (Add)	Top two longwords on the stack are added.
52	TIR$C_OPR_SUB (Subtract)	Top longword on the stack is subtracted from the next longword on the stack.
53	TIR$C_OPR_MUL (Multiply)	Top two longwords on the stack are multiplied.
54	TIR$C_OPR_DIV (Divide)	Top longword on the stack is divided into the next longword on the stack.
55	TIR$C_OPR_AND (Logical AND)	Logical AND of top two longwords.
56	TIR$C_OPR_IOR (Logical Inclusive OR)	Inclusive OR of top two longwords.
57	TIR$C_OPR_EOR (Logical Exclusive OR)	Exclusive OR of top two longwords.
58	TIR$C_OPR_NEG (Negate)	Top longword is negated.
59	TIR$C_OPR_COM (Complement)	Top longword is complemented.
60	TIR$C_OPR_INSV (Insert Field)	This command is reserved. It is similar to TIR$C_STO_VPS except that the bit field is written to the next longword on the stack instead of to the image file. The location counter is therefore unaffected. After completion of the command, only the top longword on the stack is removed.
61	TIR$C_OPR_ASH (Arithmetic Shift)	The top longword on the stack specifies the shift count and direction to be applied to the next longword on the stack. When the top longword is negative, bits in the next longword are shifted right with replication of the sign bit. When the top longword is positive, bits in the next longword are shifted left with zeros moved into low-order bits.
62	TIR$C_OPR_USH (Unsigned Shift)	Same as previous command except that, for a shift right, zeros are always moved into the high bits.
63	TIR$C_OPR_ROT (Rotate)	The top longword on the stack specifies the rotate count and direction to be applied to the next longword on the stack. When the top longword is positive, the next longword is rotated left; when negative, right. The top longword must have an absolute value of 0 to 32.

(continued on next page)

Table A–5 (Cont.) Operator Commands

Code	Command	Description
64	TIR$C_OPR_SEL (Select)	This command manipulates the top three longwords on the stack. If the top longword has the value TRUE (low bit set), it and the next (second) longword on the stack are removed, leaving the third longword (unchanged) on top of the stack. If the top longword has the value FALSE (low bit clear), the value of the next (second) longword is copied to the following (third) longword, and the top and second longwords are removed, leaving the third (now having the value of the second) on top of the stack. Thus, the command collapses three longwords on the stack to a single longword that has the value of the second or third based on the value of the first.
65	TIR$C_OPR_REDEF (Redefine Symbol to Current Location)	This command is used only in the creation of shareable images. Data for the command consists of a symbol name in standard name format, that is, 1 count byte followed by a variable-length (1 to 31 bytes) ASCII string. The value of the symbol, as listed in the shareable image's symbol table, is made equal to the value of the current location counter (at the time the command is processed). Values of the symbol within the image itself are not affected by the command. The value is not assigned until after all image binary has been written to the image output file. If no binary is generated (or is aborted), the value is not assigned. The symbol is also made universal.
66	TIR$C_OPR_DFLIT (Define a Literal)	Data is a 1-byte field that indicates the literal (0 to 255) to be defined. The literal is assigned the value of the top longword on the stack, which is removed upon completion of the command. Note that this command does not stack a result.
67–79		Reserved.

A.4.4 Control Commands

Control commands manipulate the linker's location counter. Table A–6 lists the control commands and related codes, together with a brief description of each command.

Table A–6 Control Commands

Code	Command	Description
80	TIR$C_CTL_SETRB (Set Relocation Base)	The top longword on the stack is placed into the location counter and then removed from the stack.
81	TIR$C_CTL_AUGRB (Augment Relocation Base)	Data consists of a signed longword. The value of this longword is added to the current location counter.

(continued on next page)

Table A–6 (Cont.) Control Commands

Code	Command	Description
82[1]	TIR$C_CTL_DFLOC (Define Location)	The value of the top longword on the stack is used as an index. The value of the current location counter is then saved under this index. Upon completion of the command, the top longword is removed from the stack.
83[1]	TIR$C_CTL_STLOC (Set Location)	The value of the top longword on the stack is an index (from a previous Define Location command) that is used to locate a previously saved location counter. The value of the previously saved location counter is then set as the value of the current location counter. Upon completion of the command, the top longword is removed from the stack.
84[1]	TIR$C_CTL_STKDL (Stack Debug)	The value of the top longword on the stack is an index (from a previous Define Location command). The top longword is removed from the stack, and the saved location counter, located by means of the index, is placed on top of the stack.
85–127		Reserved.

[1]This command is legal only in debugger information (DBG) and traceback information (TBT) records. For each object module, a list of debugger indexes is kept. These commands operate on the list for the object module in which the DBG or TBT record occurs.

A.5 End-of-Module Record

The end-of-module (EOM) record declares the end of the module. Either this record or the end-of-module-with-word-psect (EOMW) record must be the last record in the object module.

If the module does not contain a program section that contains the transfer address, the EOM record is 2 bytes long, consisting of only the RECORD TYPE and ERROR SEVERITY fields.

If the module does contain a program section that contains the transfer address, the EOM record can be either 7 or 8 bytes long, depending on whether the optional TRANSFER FLAGS field is included.

The fields in an EOM record are described in the following table.

RECORD TYPE Name: EOM$B_RECTYP
Length: 1 byte

The record type is OBJ$C_EOM.

ERROR SEVERITY Name: EOM$B_COMCOD
Length: 1 byte

This field contains completion codes, which are generated by the language processor. This field may contain a value from 0 to 3, where each number corresponds to a completion code. Values from 4 to 10 are reserved, and values from 11 to 255 are ignored. The following table lists the name, corresponding value, and meaning of each of the four completion codes.

Value	Name	Meaning
0	EOM$C_SUCCESS	Successful compilation or assembly; no errors detected.
1	EOM$C_WARNING	Language processor generated warning messages. The linker issues a warning message and proceeds with the linking operation.
2	EOM$C_ERROR	Language processor generated severe errors. The linker issues an error message, proceeds with the linking operation, but does not produce an output image file.
3	EOM$C_ABORT	Language processor generated fatal errors. The linker aborts the linking operation.
4–10		Reserved.
11–255		Ignored.

PSECT INDEX Name: EOM$B_PSINDX

Length: 1 byte

This field contains the program section index of the program section within the module that contains the transfer address. Note that this field is present only if the module contains a program section that contains the transfer address.

TRANSFER ADDRESS Name: EOM$L_TRFADR

Length: 4 bytes

This field contains the location of the transfer address. This location is expressed as an offset from the base of this module's contribution to the program section that contains the transfer address. Note that this field is present only if the module contains a program section that contains the transfer address.

TRANSFER FLAGS Name: EOM$L_TFRFLG

Length: 1 byte

This field is a 1-byte bit mask that contains information about the transfer address. When bit 0 is set (EOM$V_WKTFR = 1), a weak transfer address is indicated; when clear (EOM$V_WKTFR = 0), a strong transfer address is indicated. If bit 0 is set and a transfer address has already been defined, no error results. Bits 1 to 7 are reserved and must contain zeros. Note that this field may be present only if the module contains a program section that contains the transfer address, and even then it is optional.

A.6 End-of-Module-with-Word-Psect Record

The end-of-module-with-word-psect record is identical in format to the end-of-module record (OBJ$C_EOM), with the following exceptions:

- The field names in the EOMW record begin with EOMW instead of EOM as in the end-of-module record. For example, in the EOMW record, the RECORD TYPE field has the name EOMW$B_RECTYP.
- The PSECT INDEX field for the EOMW record is 2 bytes long instead of 1 byte as in the EOM record.

A.7 Debugger Information Records

The purpose of debugger information records is to allow the language processors to pass compilation information, such as descriptions of local variables, to the debugger. The transmission of this information may make use of all the functions (commands) available in the TIR command set.

The command stream in DBG records generates a debugger symbol table (DST). The DST immediately follows the binary of the user image, and the image header contains a descriptor of where in the file such data is written. The production of the DST in memory makes use of a separate location counter within the linker. This location counter is initialized as if the DST is the highest addressed part of the program region of the image. Note, however, the DST is not mapped into the user image.

The linker produces a DST only if the /DEBUG qualifier is specified at link time.

A.8 Traceback Information Records

Traceback information records are the means by which language processors pass information to the facility that produces a traceback of the call stack. From the point of view of the linker and its processing of these records, they are identical to DBG records. That is, they may be mixed with DBG records, and all data generated goes into the DST as if they are DBG records.

The purpose of separating the information contained in DBG records is to allow inclusion of a DST containing only traceback data when no debugging is requested at link time. If the production of traceback information is disabled at link time, then these records are ignored.

A.9 Link Option Specification Records

The link option specification records are defined to allow the language processor to provide the linker with additional input files to be searched for symbol resolution at link time.

As a result, the file specifications in the link option records must be correct at link time. Also, because the files in the LNK records are encountered during the first pass of the linking operation, no related name defaulting can be performed for file specifications.

The linker can, however, apply default file types if none are present in the file specifications in the LNK records:

OBJ Indicates object files

OLB Indicates object libraries and shareable image libraries

EXE Indicates shareable images

The first field in a LNK record is the record type LNK$B_RECTYP, whose value is OBJ$C_LNK. The next field describes the LNK subrecord type, LNK$B_LNKTYP.

The next table lists each LNK subrecord type, its symbolic representation, and its numeric code value.

LNK Subrecord	Symbol	Code
Object library file specification	LNK$C_OLB	0
Shareable image library file specification	LNK$C_SHR	1
Object library with inclusion list	LNK$C_OLI	2
Object file or symbol table file	LNK$C_OBJ	3
Shareable image file	LNK$C_SHA	4

FLAGS Name: LNK$W_FLAGS
Length: 2 bytes

This field follows the subrecord type and is a word-length bit field. Currently, only the two flag bits described in the next table are used with LNK$W_FLAGS.

Bit	Name	Meaning if Set
0	LNK$V_SELSER	Selectively searches object module or symbol table. This bit is valid only for LNK$C_OBJ subrecords.
1	LNK$V_LIBSRCH	After module inclusion, searches this library for resolution of currently undefined symbols. The need for this bit arises out of an ambiguity between the usage of the two record types LNK$C_OLI and LNK$C_OLB. The use of this bit is best illustrated by the /LIBRARY and /INCLUDE file qualifiers. Note that an input file specification such as A /INCLUDE=(B,C) corresponds to a LNK$C_OLI type, and an input file specification such as A/LIB corresponds to a LNK$C_OLB type. However, an input file such as A/LIB /INCLUDE=(B,C) is indicated by a linker options record type of LNK$C_OLI with the LNK$V_LIBSRCH bit set. This bit is valid only for LNK$C_OLI subrecords.

FILE NAME LENGTH Name: LNK$W_NAMLNG
Length: 2 bytes

This field is one word in length and is the length of the file name string. For LNK$C_OLI subrecord types, this length does not include the length of the list of modules to be included.

FILE NAME Name: LNK$T_NAME
Length: LNK$W_NAMLNG

This field is the file specification of the file to be included.

Note that for all subrecord types except LNK$C_OLI, this is the end of the LNK record. For LNK$C_OLI records, the modules to be included are described as a series of ASCII counted strings and appear immediately after the file name LNK$T_NAME. The end of the module inclusion list is indicated by 1 byte of zero. ♦

B

Alpha Object Language

Alpha

This appendix defines Structure Level 2 of the Alpha object language. The object language describes the contents of object modules to the OpenVMS Linker utility (the linker), as well as to the OpenVMS Librarian utility. All language processors that produce code for execution in native mode are free to use any or all of the object language components.

This information is useful primarily to programmers writing compilers or assemblers that must generate object modules acceptable for input to the linker. These programmers may also find the description of the ANALYZE/OBJECT command in the *OpenVMS DCL Dictionary* useful. ANALYZE/OBJECT will parse the object module and perform limited integrity checking on the object.

This appendix contains seven sections. The first section provides an overview of the object language and lists the main types of records. Each subsequent section discusses a main record and its subrecords, as well as the context in which it must be used.

The symbols used in this section are available to BLISS-32 programmers in STARLET.REQ and STARLET.L32. These files also contain definitions for the VAX object language defined in Appendix A. For C programmers, these symbols are defined in EOBJRECDEF.H on Alpha systems and in OBJRECDEF.H on VAX systems.

B.1 Object Language Overview

Each object module (or compiler-generated symbol table file) specified as input to the linker must be in the format described in this appendix. The object language defines an object module as an ordered set of variable-length records. Note that for VAX object language, the record length can be determined only by the value returned by OpenVMS RMS on each read operation, but that for Alpha object language, the total length of the record is recorded in the size field (EOBJ$W_SIZE). Table B–1 shows the main record types currently available.

Table B–1 Object Record Types

Record Type	Symbol	Value
Header (HDR)	EOBJ$C_EMH	8
End-of-module (EOM)	EOBJ$C_EEOM	9
Global symbol directory (GSD)	EOBJ$C_EGSD	10
Text information and relocation (TIR)	EOBJ$C_ETIR	11
Debugger information (DBG)	EOBJ$C_EDBG	12

(continued on next page)

Table B–1 (Cont.) Object Record Types

Record Type	Symbol	Value
Traceback information (TBT)	EOBJ$C_ETBT	13
Reserved to Digital		All others

The term **reserved** indicates that the value must not be used in the EOBJ$W_ RECTYP field because it is reserved for possible future use by Digital. The linker will issue a warning if it encounters an illegal value.

All six legal record types do not have to appear in a single object module. However, each object module must contain the following:

- One (and only one) main module header record (EOBJ$C_EMH, subtype EMH$C_MHD) appearing first in the object module (see Section B.2.1)
- One (and only one) language name header record (EOBJ$C_EMH, subtype EMH$C_LNM) appearing second in the object module (see Section B.2.2)
- At least one global symbol directory record (EOBJ$C_EGSD)
- One end-of-module (EOBJ$C_EEOM) record at the end of the object module

An object module may contain any number of global symbol directory (EOBJ$C_ EGSD), text information and relocation (EOBJ$C_ETIR), debugger (EOBJ$C_ EDBG), or traceback (EOBJ$C_ETBT) records as long as they are not first or last in the object module. An object file may contain any number of object modules, delimited by header and end-of-module records. Figure B–1 depicts the correct ordering of records within an object module.

If a field is currently ignored by the linker, you must still allocate space for it, filling it with zeros to its entire specified length. Some structures require padding at the end with zeros in order to achieve quadword alignment.

Object records may contain the names of program sections, object modules, language processors, utilities, and so on. Most records implement names as counted strings (1-byte name length field followed by the indicated number of ASCII characters). Others, such as the EOBJ$C_EMH subrecord EOBJ$C_ LNM, place the name at the end of the record and use the record's size field to extrapolate the size of the name.

Table B–2 shows the relationships between structure definitions. Each column in the table lists a structure prefix in bold letters. The meaning of the prefix is listed in Table B–3. The values in the column below the prefix represent the key on which a substructure is based. The next column in that row is another structure prefix in bold letters. For example, a simple global symbol definition record is described with four structures. The EOBJ$ structure defines the main record. The EGSD$ structure defines the envelope around the subrecords, of which the EGSY$ and ESDF$ structures define the particulars of the symbol definition.

Figure B–1 Order of Records in an Object Module

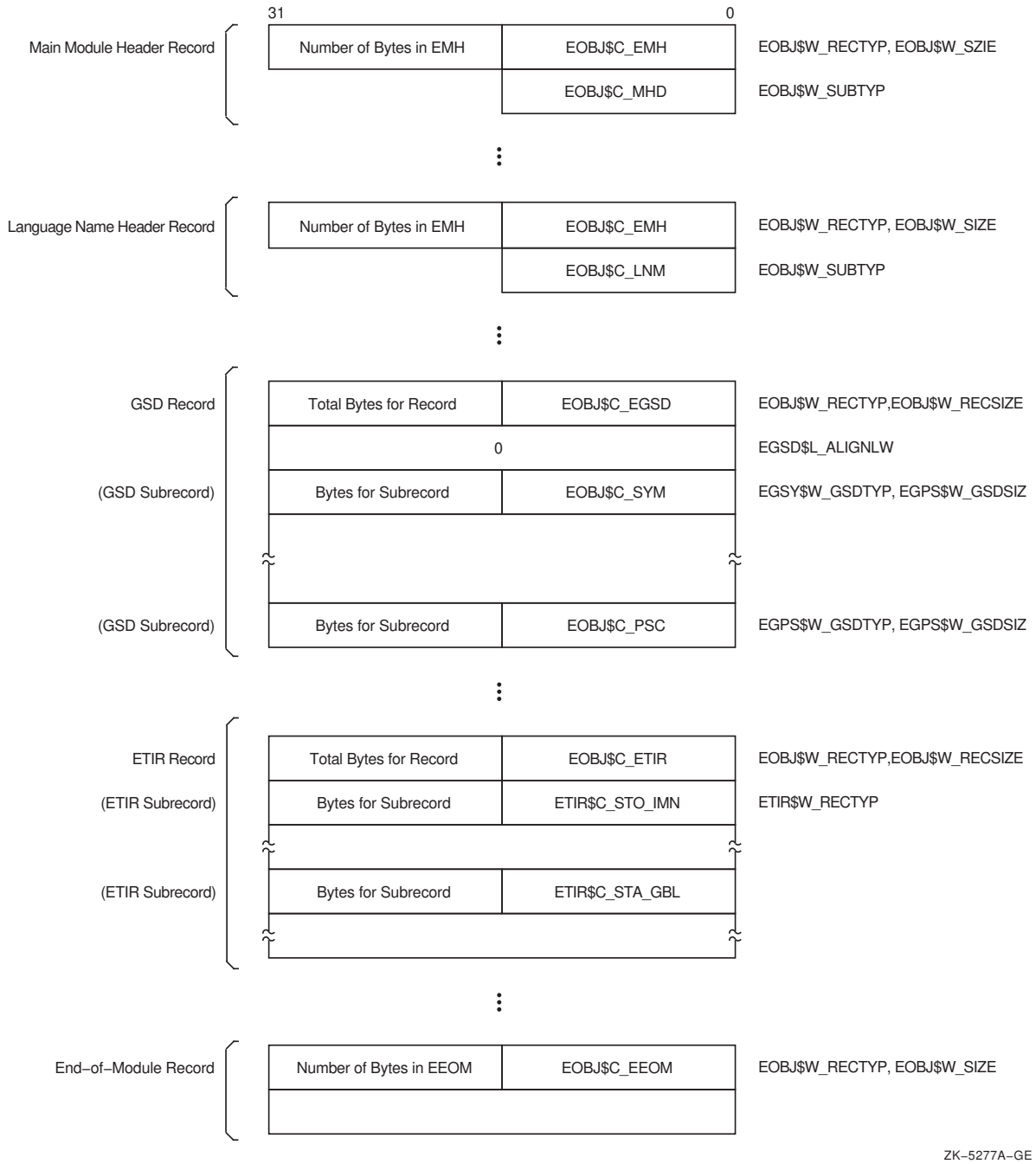


Table B–2 Relationships of Structures in the Alpha Object Language

EOBJ$			
EOBJ$C_EMH	**EMH$**		
EOBJ$C_EEOM	**EEOM$**		
EOBJ$C_EGSD	**EGSD$**		
	EGSD$C_PSC	**EGPS$**	
	EGSD$C_SYM	**EGSY$**	
		EGSY$V_DEF=1	**ESDF$**
		EGSY$V_DEF=0	**ESRF$**
	EGSD$C_IDC	**EIDC$**	
	EGSD$C_SPSC[1]	**ESGPS$**	
	EGSD$C_SYMV[1]	**ESDFV$**	
	EGSD$C_SYMM[1]	**ESDFM$**	
	EGSD$C_SYMG[1]	**EGST$**	
EOBJ$C_ETIR	**ETIR$**		
EOBJ$C_EDBG	**EDBG$**		
EOBJ$C_ETBT	**ETBT$**		

[1]This record is reserved for use by the linker.

Table B–3 Key to Structure Prefixes

EOBJ$	Object Record Defines the type and size fields of a variable-length record, and the object record types.
EMH$	Module Header Record Defines the module header record and its subrecord types.
EEOM$	End Of Module Record Defines the end-of-module record.
EGSD$	Global Symbol Directory Record Defines a global symbol directory record and its subrecord types.
EGPS$	Psect Definition Defines the GSD subrecord for a psect definition.
EGSY$	Symbol Definition or Reference Defines the common fields in a GSD subrecord for a symbol definition or reference.
ESDF$	Symbol Definition Defines the fields in the GSD subrecord for a symbol definition that follows the common fields defined by EGSY$.
ESRF$	Symbol Reference Defines the fields in the GSD subrecord for a symbol reference that follows the common fields defined by EGSY$.
EIDC$	Entity Identity Check Defines the GSD subrecord for an entity ident consistency check.

(continued on next page)

Table B–3 (Cont.) Key to Structure Prefixes

ESGPS$[1]	Shareable Psect Definition Defines the GSD subrecord for a shareable image psect definition.
ESDFV$[1]	Vectored Symbol Definition Defines the GSD subrecord for a vectored symbol.
ESDFM$[1]	Masked Symbol Definition Defines the GSD subrecord for a masked symbol.
EGST$[1]	Universal Symbol Definition Defines a GSD subrecord for a universal symbol definition.
ETIR$	Text Information and Relocation Record Defines a text information and relocation command.
EDBG$	Debugger Record Defines a record used to build a debugger symbol table. Contains TIR subrecords defined by ETIR$.
ETBT$	Traceback Record Defines a record used to build a debugger symbol table for use with the traceback facility. Contains TIR subrecords defined by ETIR$.

[1]This record is reserved for use by the linker.

The following sections contain descriptions and diagrams of the Alpha object language records and subrecords.

B.2 Module Header Records (EOBJ$C_EMH)

The Alpha object language currently defines seven types of header records. Each type is assigned a symbolic code with values between 0 and 6. All other values are illegal in an Alpha object module.

Table B–4 lists the various types of header records.

Table B–4 Module Header Subrecords

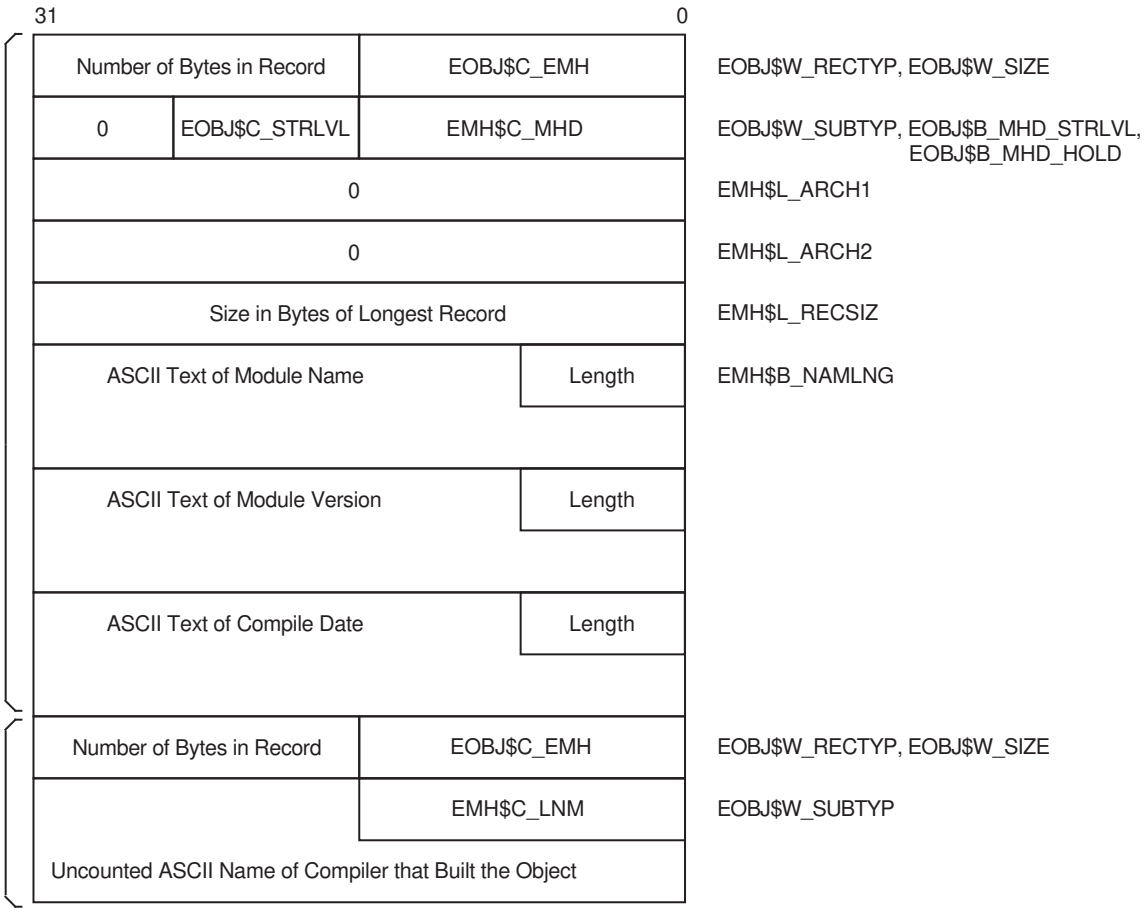
Subtype	Symbol	Value
Main module header (MHD)[1]	EMH$C_MHD	0
Language processor name header (LNM)[1]	EMH$C_LNM	1
Source file header (SRC)[2]	EMH$C_SRC	2
Title text header (TTL)[2]	EMH$C_TTL	3
Copyright header (CPR)[2]	EMH$C_CPR	4
Maintenance status header (MTC)[2]	EMH$C_MTC	5
General text header (GTX)[2]	EMH$C_GTX	6
Reserved to Digital		All others

[1]This record is required by the linker.

[2]This record is currently ignored by the linker.

The content and format of the MHD and LNM header subtypes, both of which are required in each object module, are described in the following subsections. Figure B–2 depicts a module header with MHD and LNM subrecords.

Figure B–2 Module Header Record with Subrecords



ZK–5278A–GE

Though currently ignored by the linker, the header subtypes SRC, TTL, CPR, MTC, and GTX exist to allow the language processors to provide printable information within the object module for documentation purposes.

The format of the LNK, SRC, TTL, CPR, MTC, and GTX records consists of a record type field (EOBJ$W_RECTYP) containing the value EOBJ$C_EMH, a record size field (EOBJ$W_SIZE), a header subtype field (EOBJ$W_SUBTYP), and a field containing the *uncounted* ASCII text.

The content and format of the SRC and TTL records are depicted in subsections B.2.3 and B.2.4, respectively. The contents of these records, as well as the MTC record (which contains information about the maintenance status of the object module), are displayed in an object module analysis. (See the description of the ANALYZE/OBJECT command in the *OpenVMS DCL Dictionary*.)

B.2.1 Main Module Header Record (EMH$C_MHD)

The main module header record, depicted in Figure B–2, is composed of the following fields. The name, symbolic representation, and length of each field are presented, followed by a symbolic value or an explanation of the contents of the field, where appropriate.

RECORD TYPE

Name: EMH$W_RECTYP

Length: 2 bytes

The field EMH$W_RECTYP redefines EOBJ$W_RECTYP. It must contain the value EOBJ$C_EMH.

RECORD SIZE

Name: EMH$W_SIZE

Length: 2 bytes

The field EMH$W_SIZE redefines EOBJ$W_SIZE. It is the size of the entire record, including the preceding record type field.

HEADER TYPE

Name: EMH$W_HDRTYP

Length: 2 bytes

The field EMH$W_HDRTYP redefines EOBJ$W_SUBTYP. It must contain a header subtype such as EMH$C_MHD (main module header) or EMH$C_LNM (language name and version).

STRUCTURE LEVEL

Name: EMH$B_STRLVL

Length: 1 byte

The structure level is EOBJ$C_STRLVL. Because the format of the MHD record never changes, the structure level field is provided so that changes in the format of other records can be made without recompiling every module that conformed to the previous format.

ALIGNMENT BYTE

Name: EMH$B_TEMP

Length: 1 byte

Alignment, must be zero.

ARCHITECTURE

Name: EMH$L_ARCH1

Length: 4 bytes

Currently unused, must be zero.

ARCHITECTURE

Name: EMH$L_ARCH2

Length: 4 bytes

Currently unused, must be zero.

MAXIMUM RECORD SIZE

Name: EMH$L_RECSIZ

Length: 4 bytes

This field contains the size in bytes of the longest record that can occur in the object module. This value may not exceed the maximum size of a record that is defined by the constant EOBJ$C_MAXRECSIZ, which is 8192 bytes.

MODULE NAME LENGTH

Name: EMH$B_NAMLNG

Length: 1 byte

This field contains the length in bytes of the module name.

MODULE NAME

Name: EMH$T_NAME

Length: Variable, 1 to 31 bytes for object modules, 1 to 39 bytes for the module header at the beginning of a shareable image's global symbol table

This field contains the module name in ASCII format.

MODULE VERSION Name: None

Length: Variable, 1 to 32 bytes, including the length byte

This field contains the module version number as an ASCII counted string. The length byte must be present and contain 0 if there is no module version.

CREATION TIME AND DATE Name: None

Length: 17 bytes

This field contains the module creation time and date in the fixed format dd-mmm-yyyy hh:mm, where dd is the day of the month, mmm is the standard 3-character abbreviation of the month, yyyy is the year, hh is the hour (00 to 23), and mm is the minutes of the hour (00 to 59). Note that a space is required after the year and that the total character count for this time format is 17 characters (including hyphens (-), the space, and the colon (:)).

B.2.2 Language Processor Name Header Record (EMH$C_LNM)

The language processor name header record is composed of the following fields:

RECORD TYPE Name: EMH$W_RECTYP

Length: 2 bytes

EMH$W_RECTYP redefines EOBJ$W_RECTYP. The record type is EOBJ$C_EMH.

RECORD SIZE Name: EMH$W_SIZE

Length: 2 bytes

The field EMH$W_SIZE redefines EOBJ$W_SIZE. It is the size of the entire record, including the preceding record type field.

HEADER TYPE Name: EMH$W_HDRTYP

Length: 2 bytes

EMH$W_HDRTYP redefines EOBJ$W_SUBTYP. The header type is EMH$C_LNM.

LANGUAGE NAME Name: None

Length: Variable

This field, which is generated by the language processor, contains the name and version of the compiler that wrote the object module. It consists of a variable-length string of ASCII characters and is *not* preceded by a byte count of the string.

B.2.3 Source Files Header Record (EMH$C_SRC)

The contents of the source files header record, although ignored by the linker, are displayed in an object module analysis. (See the description of the ANALYZE /OBJECT command in the *OpenVMS DCL Dictionary*.) The source files header record is composed of the following fields:

RECORD TYPE Name: EMH$W_RECTYP

Length: 2 bytes

EMH$W_RECTYP redefines EOBJ$W_RECTYP. The record type is EOBJ$C_EMH.

RECORD SIZE Name: EMH$W_SIZE
Length: 2 bytes

The field EMH$W_SIZE redefines EOBJ$W_SIZE. It is the size of the entire record, including the preceding record type field.

HEADER TYPE Name: EMH$W_HDRTYP
Length: 2 bytes

EMH$W_HDRTYP redefines EOBJ$W_SUBTYP. The header type is EMH$C_SRC.

SOURCE FILES Name: None
Length: Variable

This field, which is generated by the compiler, contains the list of file specifications from which the object module was created. It consists of a variable-length string of ASCII characters and is *not* preceded by a byte count of the string.

B.2.4 Title Text Header Record (EMH$C_TTL)

The contents of the title text header record, although ignored by the linker, are displayed in an object module analysis. (See the description of the ANALYZE /OBJECT command in the *OpenVMS DCL Dictionary*.) The title text header record is composed of the following fields:

RECORD TYPE Name: EMH$W_RECTYP
Length: 2 bytes

EMH$W_RECTYP redefines EOBJ$W_RECTYP. The record type is EOBJ$C_EMH.

RECORD SIZE Name: EMH$W_SIZE
Length: 2 bytes

The field EMH$W_SIZE redefines EOBJ$W_SIZE. It is the size of the entire record, including the preceding record type field.

HEADER TYPE Name: EMH$W_HDRTYP
Length: 2 bytes

EMH$W_HDRTYP redefines EOBJ$W_SUBTYP. The header type is EMH$C_TTL.

TITLE TEXT Name: None
Length: Variable

This field, which is generated by the language processor, contains a brief description of the object module. It consists of a variable-length string of ASCII characters and is *not* preceded by a byte count of the string.

B.3 Global Symbol Directory Records (EOBJ$C_EGSD)

GSD records contain information that the linker uses to build link-time structures describing symbol references, symbol definitions, procedure definitions, and psect definitions. These structures are used to build the global symbol table, the debugger symbol table, and image sections, including the fix-up section.

At least one GSD record must appear in each object module.

A GSD record consists of a type field (EGSD$W_RECTYP), a size field (EGSD$W_RECSIZ), a quadword-alignment field (EGSD$L_ALIGNLW), and one or more GSD subrecords. Each subrecord consists of a type field, a size field, and one or more fields that differ depending on the type value. Each GSD subrecord must start on a quadword boundary. The beginning of the GSD record is filled out to a quadword by the EGSD$L_ALIGNLW field. This places the first subrecord on a quadword boundary. Each subrecord must be filled out to a quadword boundary by padding with zeros so that the following subrecord is also quadword aligned. Any padding must be reflected in the record's total byte count.

Table B–5 lists each type of GSD subrecord together with its symbolic representation and its corresponding numeric value.

Table B–5 Types of GSD Subrecords

GSD Subrecord	Symbol	Value
Program section definition	EGSD$C_PSC	0
Global symbol specification	EGSD$C_SYM	1
Random identity check	EGSD$C_IDC	2
Reserved to Digital		3–4
Shareable program section definition[1]	EGSD$C_SPSC	5
Vectored symbol definition[2]	EGSD$C_SYMV	6
Masked symbol definition[2]	EGSD$C_SYMM	7
Universal symbol definition[1]	EGSD$C_SYMG	8
Reserved to Digital		All others

[1]This record is reserved to the linker for building global symbol tables.

[2]This record is reserved to the linker for building the VMS executive.

A single GSD record must contain at least one of the above types of subrecords. The number of subrecords in a GSD record is limited by the maximum record size specified in the header field EMH$L_RECSIZ. Figure B–1 shows the general format of a GSD record that contains two subrecords. Note that the RECORD TYPE and RECORD SIZE fields appear only once at the beginning of the record, regardless of how many subrecords there are. The RECORD SIZE field counts all of the bytes in the record, including the RECORD TYPE and RECORD SIZE fields themselves, and all of the GSD subrecords. Each GSD subrecord includes a GSD SUBRECORD TYPE and GSD SUBRECORD SIZE field. The RECORD TYPE and RECORD SIZE fields for the GSD record are not listed in the following sections, which describe each subrecord, and are not shown in the diagrams.

B.3.1 Program Section Definition Subrecords (EGSD$C_PSC, EGSD$C_SPSC)

The linker assigns program sections an identifying index number as it processes each successive psect definition, that is, each EGSD$C_PSC subrecord. The linker assigns these numbers in sequential order, assigning 0 to the first program section it encounters, 1 to the second, and so on, up to the maximum allowable limit of 65,535 (2^{16} –1) within any single object module.

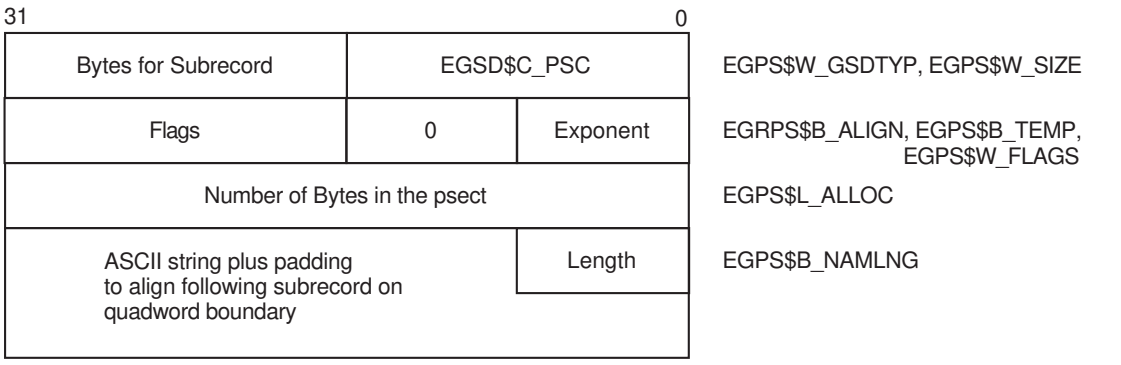
Program sections are referred to by other object language records by means of this program section index. For example, the global symbol specification subrecord (EGSD$C_SYM) contains a field that specifies the program section index. This field is used to locate the program section containing storage for the symbol. Text information and relocation (TIR) commands also use the program section index.

Care is required to ensure that program sections are defined to the linker (and thus assigned an index) in the proper order so that other object language records that reference a program section by means of the index are in fact referencing the correct program section. Program sections may be referenced before they are defined, although it is good practice to define the program sections first when possible.

B.3.1.1 Normal Program Section Definition Subrecord (EGSD$C_PSC)

Figure B–3 depicts the format of a program section definition subrecord, showing the fields it contains and providing a description of each.

Figure B–3 GSD Subrecord for a Program Section Definition



ZK–5283A–GE

GSD SUBRECORD TYPE Name: EGPS$W_GSDTYP
Length: 2 bytes

The GSD type is EGSD$C_PSC.

GSD SUBRECORD SIZE Name: EGPS$W_SIZE
Length: 2 bytes

This field contains the size of the entire subrecord, including the preceding type field and any padding used to quadword align the following record.

PSECT ALIGNMENT Name: EGPS$B_ALIGN

Length: 1 byte

This field specifies the virtual address boundary at which the program section is placed. Each contribution to a particular program section may specify its own alignment. If the contributions have different alignments, the greatest alignment is used to align the entire program section. The flags field of an overlaid program section has the EGPS$V_OVR bit set.

The alignment field contains a value between 0 and 16, which is interpreted as a power of 2; the value of this expression is the alignment in bytes between 1 and 64K. Table B–6 illustrates some common alignment field values.

Table B–6 Alignment Field Values

Value	Alignment in Bytes	
0	1	(BYTE)
1	2	(WORD)
2	4	(LONGWORD)
3	8	(QUADWORD)
4	16	(OCTAWORD)
9	512	(PAGELET)
13	8K	(8K BIG PAGE)
16	64K	(64K BIG PAGE)

ALIGNMENT BYTE

Name: EGPS$B_TEMP

Length: 1 byte

Field alignment byte. Must be 0.

FLAGS

Name: EGPS$W_FLAGS

Length: 2 bytes

This field is a word-length bit field, each bit indicating (when set) that the program section has the corresponding attribute. (See Section 3.2 for a description of program section attributes.) The following table lists the numbers, names, and corresponding meanings of each bit in the field:

Bit	Name	Meaning if Set
0	EGPS$V_PIC	Not meaningful. For compatability reasons, the linker still sorts psects on this attribute when building image sections. This bit should never be the only distinguishing attribute between two different psects. If two psects have exactly the same attributes except for EGSP$V_PIC, then the psect attributes should be changed so that they match.
1	EGPS$V_LIB	Program section is defined in the symbol table of a shareable image, to which this image is bound. This bit is used by the linker and should not be set in user-defined program sections.

Bit	Name	Meaning if Set
2	EGPS$V_OVR	Contributions to this program section by more than one module are overlaid. If this bit is set, EGPS$V_REL and EGPS$V_GBL must also be set. An overlaid program section must not be referenced by the ESDF$L_PSINDX field of a symbol definition.
3	EGPS$V_REL	Program section is relocatable. If this bit is not set, the program section is absolute and therefore contains only symbol definitions. Note that memory is not allocated for absolute program sections.
4	EGPS$V_GBL	Program section is global.
5	EGPS$V_SHR	Program section is shareable between two or more active processes.
6	EGPS$V_EXE	Program section is executable.
7	EGPS$V_RD	Program section is readable. This attribute is currently ignored by the linker.
8	EGPS$V_WRT	Program section is writable.
9	EGPS$V_VEC	Program section contains change mode dispatch vectors or message vectors. This bit is normally left clear and set by the programmer with the linker option PSECT_ATTRIBUTE=psect-name,VEC
10	EGPS$V_NOMOD	Program section contains unmodified data and can be included in a demand-zero image section.
11	EGPS$V_COM	Program section defines conditional storage for a symbol with the EGSD$V_COMM bit set. If this bit is set, EGPS$V_OVR, EGPS$V_REL, and EGPS$V_GBL must also be set.
12–15		Reserved to Digital.

ALLOCATION

Name: EGPS$L_ALLOC

Length: 4 bytes

This field contains the length in bytes of this module's contribution to the program section. If the program section is absolute, the value of the allocation field must be 0.

PSECT NAME LENGTH

Name: EGPS$B_NAMLNG

Length: 1 byte

This field contains the length in bytes of the program section name.

PSECT NAME

Name: EGPS$T_NAME

Length: Variable, 1 to 31 bytes

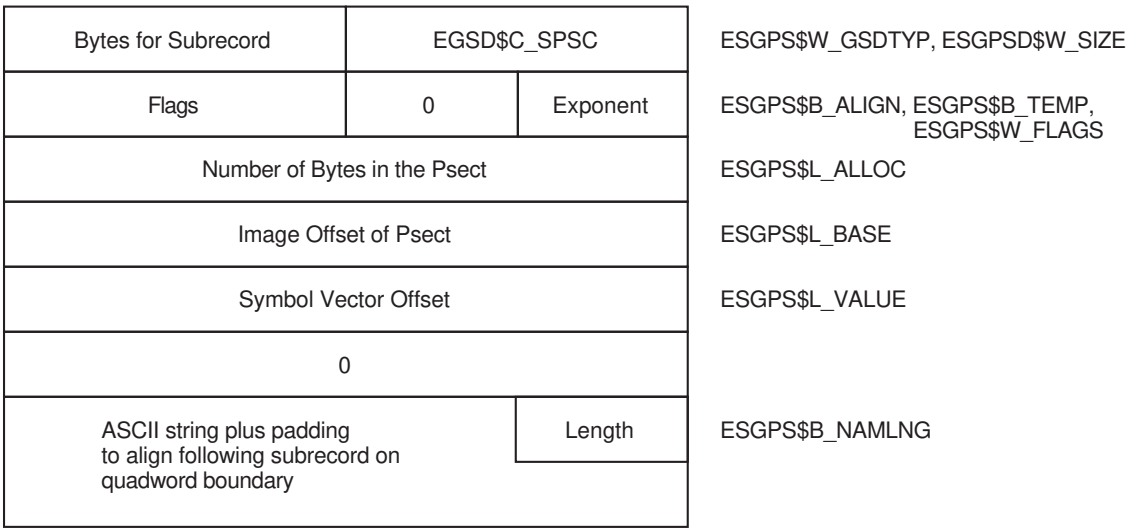
This field contains the name of the program section in ASCII format. Note that program section names are limited to 31 bytes, while symbol names are limited to 64 bytes. Compilers that implement global symbols as overlaid program sections (as opposed to global symbol definitions with storage allocated by a concatenated program section) must be aware of this restriction.

B.3.1.2 Program-Section-Definition-in-Shareable-Image Subrecord (GSD$C_SPSC)

This subrecord is reserved to the linker. It is generated in the GST when a program section is made universal with the SYMBOL_VECTOR keyword PSECT. When a main image links against a shareable image that contains these special universal program sections, the linker matches program sections from the main image to those in the shareable image and overlays them. The overlay is done only if the program sections have the same name, the attributes EGSY$V_OVR, EGSY$V_REL, and EGSY$V_GBL, and the same allocation. References to the program section in the main image are fixed up and thereby converted to references to the corresponding program section in the shareable image.

Shareable program sections have indexes, but are never referenced by symbol definitions. All symbol definitions in the GST point to an absolute program section, regardless of whether they are relocatable, that is, regardless of whether the EGSY$V_REL bit is set or clear. Figure B–4 depicts the format of a definition subrecord, followed by a short description of each field.

Figure B–4 GSD Subrecord for a Shareable Image Program Section Definition



ZK–5287A–GE

GSD SUBRECORD TYPE Name: ESGPS$W_GSDTYP
Length: 2 bytes

GSD type is EGSD$C_SPSC.

GSD SUBRECORD SIZE Name: ESGPS$W_SIZE
Length: 2 bytes

This field contains the size of the entire subrecord, including the preceding type field and any padding used to quadword align the following record.

PSECT ALIGNMENT Name: ESGPS$B_ALIGN
Length: 1 byte

This field specifies the virtual address boundary at which the program section is placed. The alignment for the entire program section is the largest one specified by any of its contributions. Valid values for this field are 0 to 16.

ALIGNMENT BYTE Name: ESGPS$B_TEMP
Length: 1 byte

Field alignment byte. Must be 0.

FLAGS Name: ESGPS$W_FLAGS
Length: 2 bytes

This field is a word-length bit field, each bit indicating (when set) that the program section has the corresponding attribute. The next table lists the numbers, names, and corresponding meanings of each bit in the field:

Bit	Name	Meaning if Set
0	ESGPS$V_PIC	Reserved, must be 1.
1	ESGPS$V_LIB	Reserved, must be 0.
2	ESGPS$V_OVR	Always set. Even if the contributions to this program section were concatenated, this attribute is set on the GST entry for the universal program section definition.
3	ESGPS$V_REL	Program section is relocatable. Must be 1.
4	ESGPS$V_GBL	Program section is global. Must be 1.
5	ESGPS$V_SHR	Program section is shareable between two or more active processes. Propagated from the contributing psects.
6	ESGPS$V_EXE	Reserved, must be 0.
7	ESGPS$V_RD	Reserved, must be 0.
8	ESGPS$V_WRT	Program section is writable. Propagated from the contributing psects.
9	ESGPS$V_VEC	Reserved, must be 0.
10	ESGPS$V_NOMOD	Reserved, must be 0.
11	ESGPS$V_COM	Reserved, must be 0.
12–15		Reserved to Digital.

ALLOCATION Name: ESGPS$L_ALLOC
Length: 4 bytes

This field contains the length in bytes of this module's contribution to the program section. The value is always nonzero.

BASE Name: ESGPS$L_BASE
Length: 4 bytes

This field contains a copy of the low-order longword of the second half of the symbol vector entry. It is the image offset of the program section.

VALUE Name: ESGPS$L_VALUE
Length: 4 bytes

This field contains the offset into the symbol vector of an entry for this program section.

PSECT NAME LENGTH Name: ESGPS$B_NAMLNG
Length: 1 byte

This field contains the length in bytes of the program section name.

PSECT NAME Name: ESGPS$T_NAME
Length: Variable, 1 to 31 bytes

This field contains the name of the program section in ASCII format.

B.3.1.3 Standard Program Section Names and Attributes

Table B–7 describes the standard program section names and attributes used by the Alpha compilers provided by Digital.

Table B–7 Standard Program Sections

Name	Attributes	Use
ABS	NOPIC CON ABS LCL SHR NOEXE NORD NOWRT	Contains the absolute program section, which is used by global constants. It has no allocation. Note that the linker builds the global symbol table with an absolute program section named .$$ABS$$., which has the attributes PIC, LIB, and RD.
BSS	NOPIC CON REL LCL NOSHR NOEXE RD WRT NOMOD	Contains unmodified data. Used to construct demand-zero image sections.
$CODE$	PIC CON REL LCL SHR EXE NORD NOWRT	Contains executable instructions.
$DATA$	NOPIC CON REL LCL NOSHR NOEXE RD WRT [NOMOD]	Contains read/write, statically initialized data.
$LINK$	NOPIC CON REL LCL NOSHR NOEXE RD NOWRT	Contains linkage consisting of procedure descriptors, linkage pairs, and .ADDRESS references.
$LITERAL$	PIC CON REL LCL SHR NOEXE RD NOWRT	Contains read-only literals
Common Data	NOPIC OVR REL GBL NOSHR NOEXE RD WRT [NOMOD]	Contains common data. These psects are usually named after the variable they represent (for example, FORTRAN common blocks). They are no longer marked SHR by default, as on VAX systems.
$READONLY$	PIC CON REL LCL SHR NOEXE RD NOWRT	Contains read-only data.

B.3.2 Global Symbol Specification Subrecords (EGSD$C_SYM, EGSD$C_SYMG)

The global symbol specification subrecords are used to describe the nature of a symbol (global or universal, relocatable or absolute) and how it is being used (definition or reference, weak or strong). This information is specified in the FLAGS field of the subrecord. Unlike the GSD$C_SYM record on VAX systems, this record type is also used to define procedures (by setting the EGSY$V_NORM bit in the FLAGS field). There is no Alpha object language equivalent to the GSD$C_EPM record type, which is used on VAX systems to define an entry point mask for a global procedure.

There are two formats for a global symbol specification subrecord, one for a symbol definition and one for a symbol reference. A symbol definition is indicated when the EGSY$V_DEF bit in the FLAGS field is set. A symbol reference is indicated when the EGSY$V_DEF bit is clear.

Section B.3.2.1 describes the format of the global symbol specification subrecord for symbol definitions; Section B.3.2.3 does the same for symbol references.

B.3.2.1 GSD Subrecord for a Global Symbol Definition (EGSD$C_SYM with EGSY$V_DEF Set)

Figure B–5 depicts a global symbol specification subrecord that defines data. Figure B–6 depicts a global symbol specification that defines a procedure. The structures of each of these definitions are identical and have the same names, but some of the fields are interpreted differently based on the setting of the EGSY$V_NORM bit in the EGSY$W_FLAGS field. The Alpha object language does not have a separate subrecord type for procedures (for example, the VAX entry-point-symbol-and-mask-definition [GSD$C_EPM] subrecord).

Figure B–5 GSD Subrecord for a Global Symbol Definition (Data)

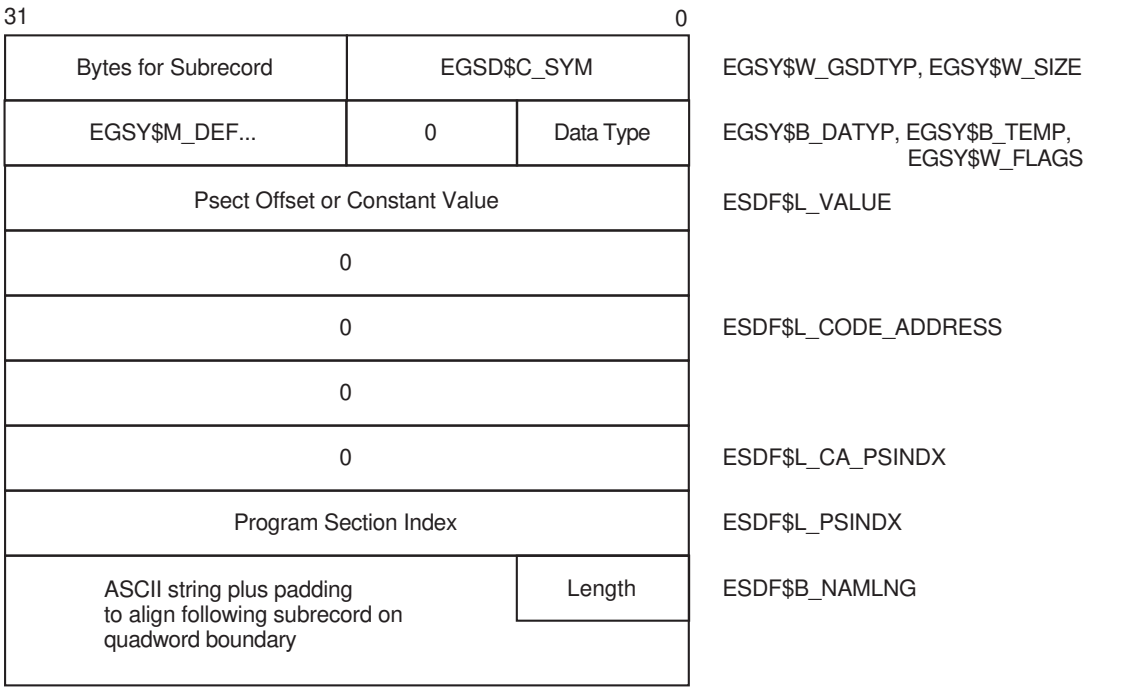
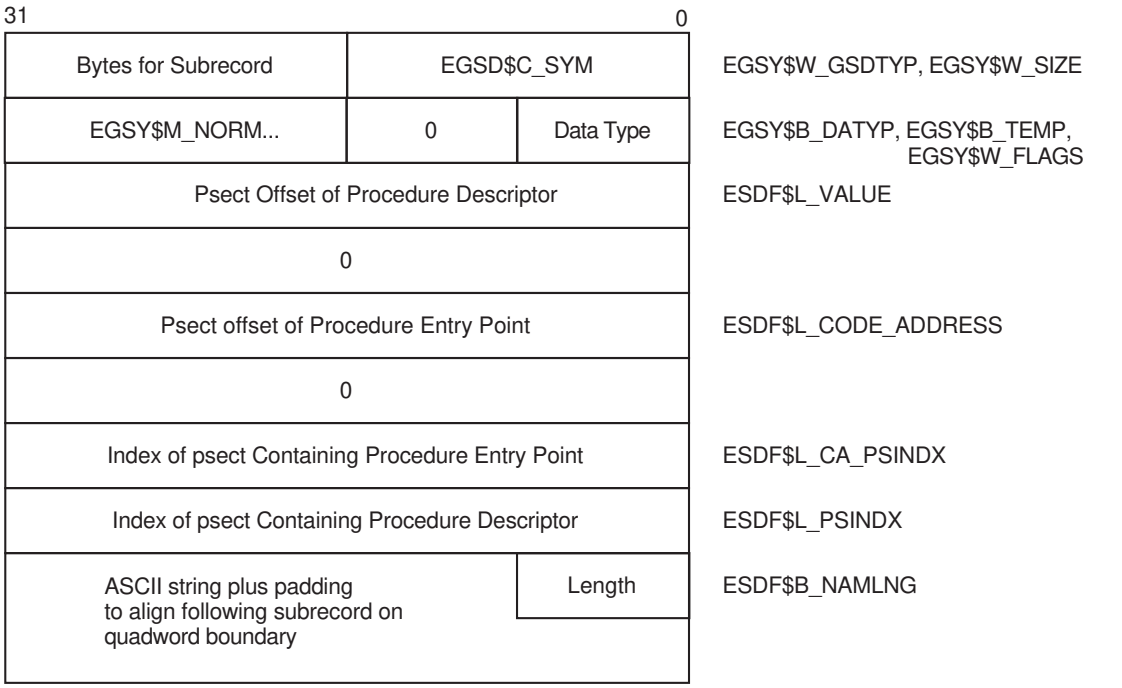


Figure B–6 GSD Subrecord for a Global Symbol Definition (Procedure)



ZK–5282A–GE

The following is a list of the fields in a global symbol definition subrecord. Fields that have different meanings depending on the value of the FLAGS field are noted.

GSD SUBRECORD TYPE Name: ESDF$W_GSDTYP
Length: 2 bytes

This field redefines EGSY$W_GSDSIZ. GSD type is EGSD$C_SYM.

GSD SUBRECORD SIZE Name: ESDF$W_SIZE
Length: 2 bytes

This field redefines EGSY$W_GSDSIZ. It is the size of the entire subrecord, including the preceding type field and any padding used to quadword align the following record.

DATA TYPE Name: ESDF$B_DATYP
Length: 1 byte

This field redefines EGSY$B_DATYP. It contains the data type of the global symbol. The data type is encoded as described in the *OpenVMS Programming Interfaces: Calling a System Routine*. The linker currently ignores this field.

ALIGNMENT BYTE Name: ESDF$B_TEMP
Length: 1 byte

This field redefines EGSY$B_TEMP. Alignment field must be 0.

FLAGS Name: ESDF$W_FLAGS
Length: 2 bytes

This field redefines EGSY$W_FLAGS. It is a 2-byte bit field, but only bits 0 through 6 are used. The following table lists the numbers, names, and meanings of each bit in the field:

Bit	Name	Meaning
0	EGSY$V_WEAK	When this bit is set, a weak symbol definition or reference is indicated; when clear, a strong symbol definition or reference.
1	EGSY$V_DEF	This bit is set for a symbol definition.
2	EGSY$V_UNI	Reserved, must be 0.
3	EGSY$V_REL	When this bit is set, the symbol is defined as relocatable; when clear, as absolute. When it is relocated, the value of a relocatable symbol is augmented by the base address of the module's contribution to the program section.
4	EGSY$V_COMM	Conditional Symbol Definition. If set, then EGSY$V_ REL and EGSY$V_WEAK must be set, and the program section that contains the storage for the symbol must have EGPS$V_COM set.
5	EGSY$V_VECEP	Reserved, must be 0.
6	EGSY$V_NORM	Indicates Normal Procedure Definition. If set, then EGSY$V_REL must be set.
7–15		Reserved to Digital.

VALUE Name: ESDF$L_VALUE
Length: 4 bytes

This field contains the value assigned to the symbol by the language processor. If the EGSY$V_NORM bit is set, the value of this field is the offset into the program section (indicated in ESDF$L_PSINDX) of the procedure descriptor.

PROCEDURE ENTRY POINT OFFSET Name: ESDF$L_CODE_ADDRESS
Length: 4 bytes

If the EGSY$V_NORM bit is set, the value in this field is the offset into the program section (indicated by ESDF$L_CA_PSINDX) of the entry point of the procedure. If the EGSY$V_NORM bit is clear, this field must be 0.

PROCEDURE ENTRY POINT PSECT INDEX Name: ESDF$L_CA_PSINDX
Length: 4 bytes

If the EGSY$V_NORM bit is set, the value in this field contains the program section index of the program section that contains the code address. If the EGSY$V_NORM bit is clear, this field must be 0.

PSECT INDEX Name: ESDF$L_PSINDX
Length: 4 bytes

This field contains the program section index, described in Section B.3.1. If EGSY$V_REL is set, the value in this field identifies a relocatable program section (one with EGPS$V_REL set) that contains the storage for the data or procedure descriptor. If the EGSY$V_REL bit is clear (the symbol is a constant), the PSECT INDEX must point to an absolute program section (one with EGPS$V_REL clear).

SYMBOL NAME LENGTH Name: ESDF$B_NAMLNG
Length: 1 byte

This field contains the length in bytes of the symbol name.

SYMBOL NAME Name: ESDF$T_NAME
Length: Variable, 1 to 64 bytes

This field contains the symbol name in ASCII format.

B.3.2.2 GSD Subrecord for a Universal Symbol Definition (EGSD$C_SYMG)

This record is reserved for use by the linker. It is used solely in global symbol tables in shareable images and global symbol table files (.STB) generated by the linker. This record type is not used in compiler-generated symbol table files. Figure B–7 depicts the universal symbol specification subrecord for a data definition.

Figure B–7 GSD Subrecords for Universal Data Definition

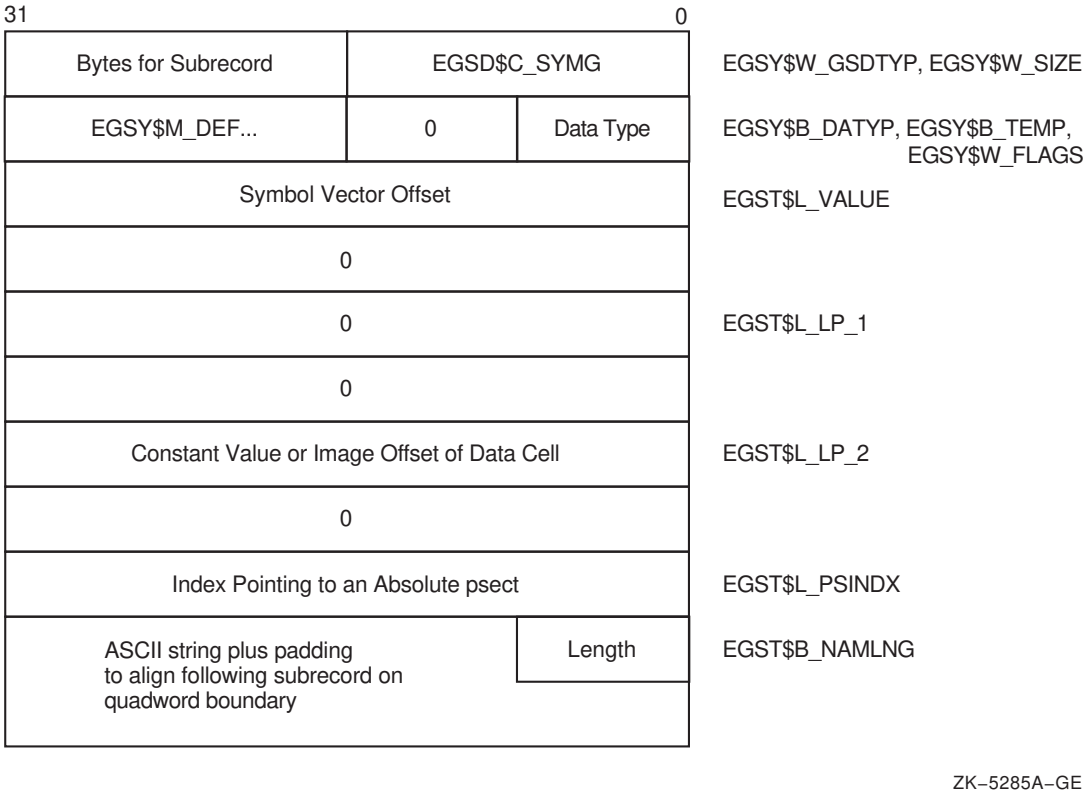
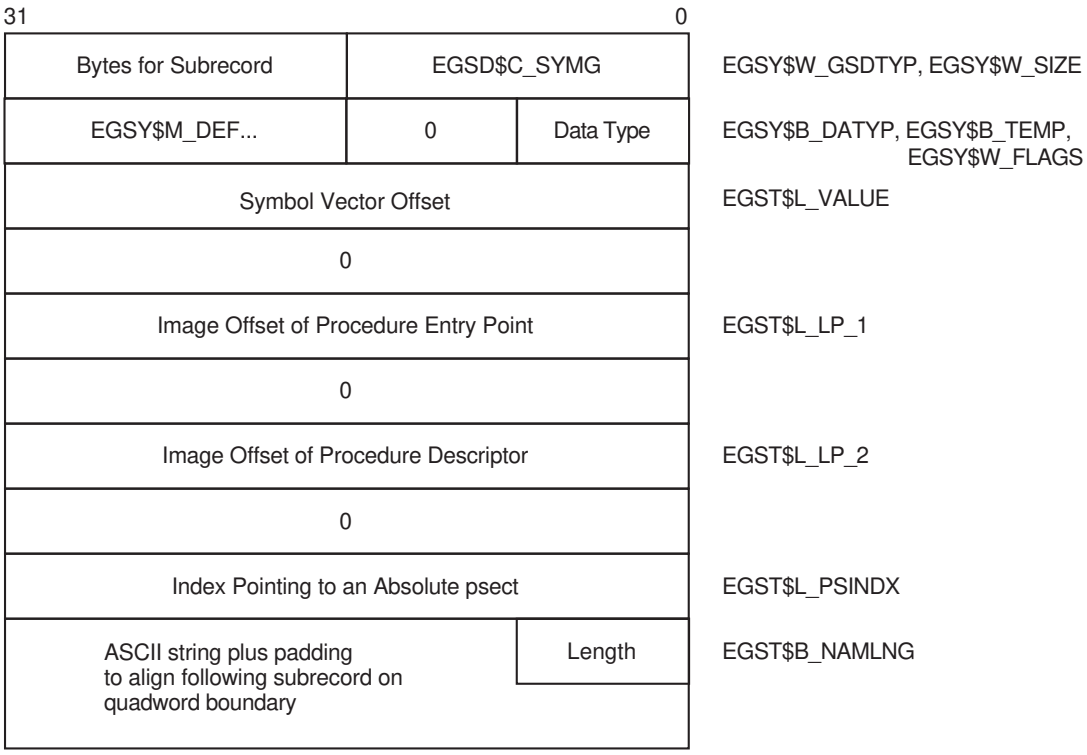


Figure B–8 depicts the universal symbol specification subrecord for a procedure definition.

Figure B–8 GSD Subrecord for a Universal Procedure Definition



ZK–5286A–GE

The following is a list of the fields in a universal symbol definition subrecord.

GSD SUBRECORD TYPE Name: EGST$W_GSDTYP
Length: 2 bytes

This field redefines EGSY$W_GSDTYP. The GSD type is EGSD$C_SYMG.

GSD SUBRECORD SIZE Name: EGST$W_SIZE
Length: 2 bytes

This field redefines EGSY$W_GSDSIZ. It is the size of the entire subrecord, including the preceding type field and any padding used to quadword align the following record.

DATA TYPE Name: EGST$B_DATYP
Length: 1 byte

This field redefines EGSY$B_DATYP. Reserved, must be 0.

ALIGNMENT BYTE Name: EGST$B_TEMP
Length: 1 byte

This field redefines EGSY$B_TEMP. Used for alignment, must be 0.

FLAGS Name: EGST$W_FLAGS
Length: 2 bytes

This field redefines EGSY$W_FLAGS. It is a 2-byte bit field, whose bits describe the universal symbol.

Bit	Name	Meaning
0	EGSY$V_WEAK	Reserved, must be 0.
1	EGSY$V_DEF	Indicates definition, must be 1.
2	EGSY$V_UNI	Indicates universal symbol, must be 1.
3	EGSY$V_REL	When this bit is set, the symbol is defined as relocatable; when clear, it is absolute. EGSY$V_REL should be set if the symbol identifies a procedure (EGSY$V_NORM is set) or if the symbol identifies a relocatable data cell. If the symbol is a constant, the EGSY$V_REL bit should be clear.
4	EGSY$V_VECEP	Reserved, must be 0.
5	EGSY$V_COMM	Reserved, must be 0.
6	EGSY$V_NORM	Normal Procedure Definition.
7–15		Reserved to Digital.

VALUE Name: EGST$L_VALUE
Length: 4 bytes

This field contains the offset into the symbol vector of the symbol vector entry for this symbol. If the symbol is a constant, the VALUE field still contains an offset into the symbol vector; the constant value resides in the symbol vector itself. External references to a constant symbol will receive fix-ups. Unlike VAX systems, there are no link-time constants on Alpha systems.

FIRST HALF OF SYMBOL VECTOR ENTRY Name: EGST$L_LP_1
Length: 4 bytes

This field contains the low-order 32 bits of of the first quadword of the symbol's entry in the symbol vector. If the EGSY$V_NORM bit is set, this is the offset into the image of the procedure entry point. Otherwise, this field contains zeros.

SECOND HALF OF SYMBOL VECTOR ENTRY Name: EGST$L_LP_2
Length: 4 bytes

This field contains the low-order 32 bits of of the second quadword of the symbol's entry in the symbol vector. If the EGSY$V_REL bit is clear, the symbol is a constant and this field contains the constant value. If EGSY$V_REL bit is set and the EGSY$V_NORM bit is clear, this field contains the image offset of data. When the EGSY$V_NORM bit is set, the GSD subrecord defines a procedure and the field contains the image offset of a procedure descriptor.

PSECT INDEX Name: EGST$L_PSINDX
Length: 4 bytes

This must be the index of an absolute psect. Normally there is only one absolute psect, named ''.$$ABS$$.'', and its index is 0.

SYMBOL NAME LENGTH Name: EGST$B_NAMLNG
Length: 1 byte

This field contains the length in bytes of the symbol name.

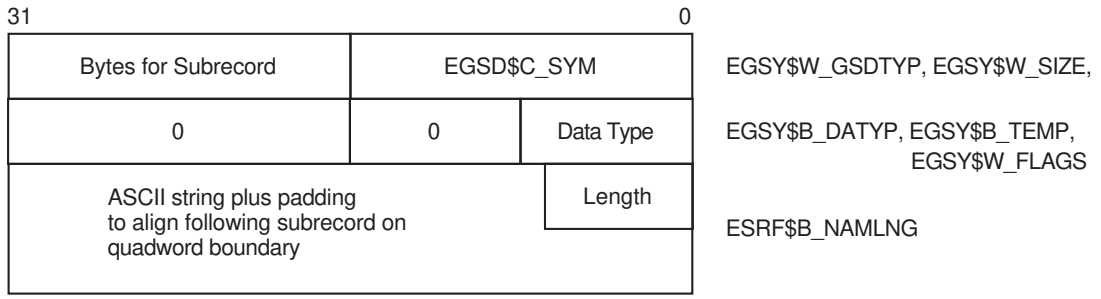
SYMBOL NAME Name: EGST$T_NAME
Length: Variable, 1 to 64 bytes

This field contains the symbol name in ASCII format.

B.3.2.3 GSD Subrecord for a Symbol Reference

Figure B–9 depicts the global symbol specification subrecord for a symbol reference. It is followed by a short description of each field.

Figure B–9 GSD Subrecord for a Global Symbol Reference (EGSD$C_SYM with EGSY$V_DEF Clear)



GSD SUBRECORD TYPE Name: ESRF$W_GSDTYP
Length: 2 bytes

This field redefines EGSY$W_GSDTYP. The GSD type is EGSD$C_SYM.

GSD SUBRECORD SIZE Name: ESRF$W_SIZE
Length: 2 bytes

This field redefines EGSY$W_GSDSIZ. It is the size of the entire subrecord, including the preceding type field and any padding used to quadword align the following record.

DATA TYPE Name: ESRF$B_DATYP
Length: 1 byte

This field redefines EGSY$B_DATYP. The linker currently ignores this field. Should be 0.

ALIGNMENT BYTE Name: ESRF$B_TEMP
Length: 1 byte

This field redefines EGSY$B_TEMP. Used for alignment, must be 0.

FLAGS Name: ESRF$W_FLAGS
Length: 2 bytes

This field redefines EGSY$W_FLAGS. The bits in this 2-byte bit field describe the global symbol. Only bits 0 through 1 are used. The following table lists the numbers, names, and corresponding meanings of each bit in the field.

Bit	Name	Meaning
0	EGSY$V_WEAK	When this bit is set, a weak symbol definition or reference is indicated; when clear, a strong symbol definition or reference.
1	EGSY$V_DEF	This bit must be 0.
2–15		Reserved to Digital.

SYMBOL NAME LENGTH Name: ESRF$B_NAMLNG
Length: 1 byte

This field contains the length in bytes of the symbol name.

SYMBOL NAME Name: ESRF$T_NAME
Length: Variable, 1 to 64 bytes

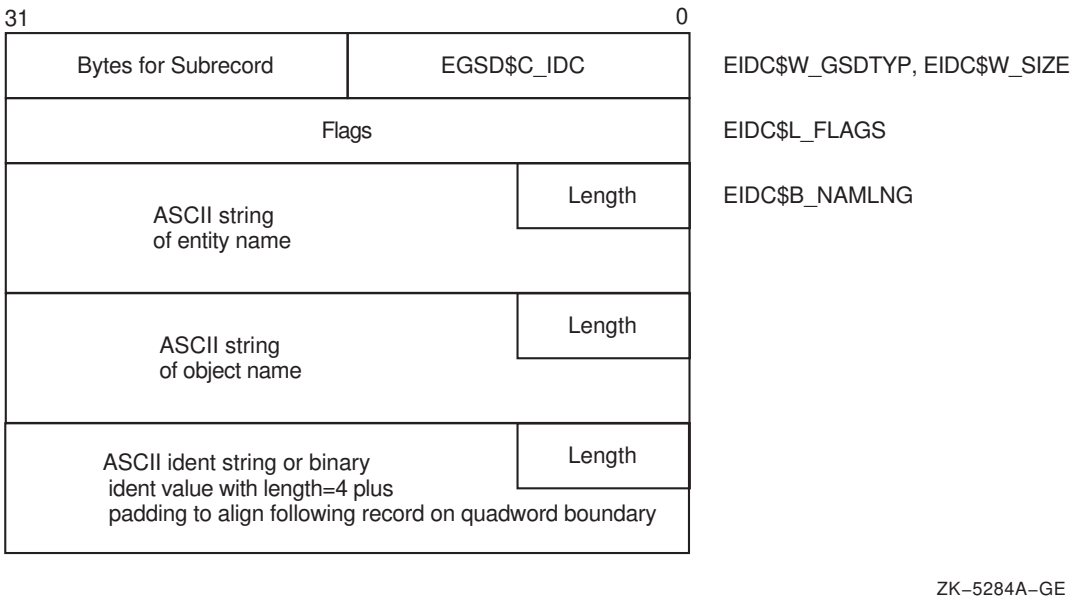
This field contains the symbol name in ASCII format.

B.3.3 Entity-Ident-Consistency-Check Subrecord (EGSD$C_IDC)

This subrecord allows for the consistency checking of an entity (qualified by its object) at link time. Using this subrecord, a compiler may emit code to check the consistency of any type of entity/object combination that has either an ASCII or a binary ident string associated with it.

Figure B–10 depicts the format of an entity-ident-consistency-check subrecord.

Figure B–10 GSD Subrecord for an Entity Ident Consistency Check



When this GSD subrecord is processed during Pass 1, the linker searches its entity name table (which is a single name table for all entity types) using the concatenated entity and object strings (including the length bytes) as the lookup key. This process differs from VAX systems, where the key is the entity name alone. If the linker does not find an entry using the concatenated key, it creates one. If the linker finds an entry for the key, it compares the idents. If the idents do not satisfy the specified match control value, the linker issues a message, with the severity specified by the EIDC$V_ERRSEV field.

The following is a short description of each field in the entity-ident-consistency-check subrecord.

GSD SUBRECORD TYPE Name: EIDC$W_GSDTYP
Length: 2 bytes

The GSD type is EGSD$C_IDC.

GSD SUBRECORD SIZE Name: EIDC$W_SIZE
Length: 2 bytes

This field contains the size of the entire subrecord, including the preceding type field and any padding used to quadword align the following record.

FLAGS Name: EIDC$L_FLAGS
Length: 4 bytes

Only the first five bits of this 4-byte bit field are used. When the EIDC$V_ BINIDENT bit is set, the IDENT STRING has a length of 4 and is a 32-bit binary value; when clear, the IDENT STRING is a counted ASCII string.

EIDC$V_IDMATCH points to the first bit of a two-bit substructure in the FLAGS field. The match bits specify the match control for the IDENT STRING when the EIDC$V_BINIDENT flag is set, that is, when the idents are 32-bit binary values. The match bits may have two values: 0 (EIDC$C_LEQ) or 1 (EIDC$C_EQUAL). If EIDC$V_BINIDENT is clear, then the ASCII idents must be equal.

When the match bits are clear (EIDC$C_LEQ), the binary ident of the entity specified in the subrecord must be less than or equal to the binary ident of the entity listed in the entity name table.

When the first match bit is set (EIDC$C_EQUAL), the binary ident of the entity specified in the subrecord must be equal to the binary ident of the entity listed in the linker's entity name table. Remaining values for the two-bit substructure pointed to by EIDC$V_IDMATCH (the values 2 through 3) are reserved.

EIDC$V_ERRSEV points to the first bit of a three-bit substructure in the FLAGS field. The severity bits determine the severity of the message issued if the IDENT STRING fields do not meet the match criteria. When the severity bits equal 0, the message severity is warning; when 1, success; when 2, error; when 3, informational; when 4, severe.

Bits 6 to 31 in the FLAGS field are reserved.

ENTITY NAME LENGTH Name: EIDC$B_NAMLNG
Length: 1 byte

This field contains the length in bytes of the entity name.

ENTITY NAME Name: EIDC$T_NAME
Length: Variable, 1 to 31 bytes

This field contains the entity name in ASCII format.

OBJECT NAME LENGTH Name: None
Length: 1 byte

This field contains the length in bytes of the object name.

OBJECT NAME Name: None
Length: Variable, 1 to 31 bytes

This field contains the object name in ASCII format.

IDENT STRING LENGTH Name: None
Length: 1 byte

This field contains the length in bytes of the ident string. For binary idents, this field contains the value 4.

IDENT STRING Name: None
Length: Variable, 1 to 31 bytes

This field contains the ident string. The ident string may be a counted ASCII string or a 32-bit binary value, determined by the flags field. If this string specifies a 32-bit binary value, it consists of 24 bits of minor ident and 8 bits of major ident, analogous to the global section match values for a shareable image. If this string specifies a counted ASCII string, its length is variable.

B.3.4 GSD Subrecords Reserved to the OpenVMS Operating System (EGSD$C_SYMV, EGSD$C_SYMM)

The following sections define subrecords reserved to the OpenVMS Alpha operating system for building the executive. Vectored symbol definitions (EGSD$C_SYMV) occur in the global symbol tables of execlets and are used for loading. Version mask subrecords (EGSD$C_SYMM) are currently found only in the system image SYS$BASE_IMAGE.EXE. The mechanisms used by the linker to generate these record types are not documented. This section is provided for readers who need to understand more about the internals of the OpenVMS Alpha operating system.

B.3.4.1 Vectored-Symbol-Definition Subrecord (EGSD$C_SYMV)

This subrecord is identical in format to the global symbol definition subrecord described in Section B.3.2.1, with the following exceptions:

- The field names begin with ESDFV$ instead of ESDF$ as in the global symbol definition subrecord.
- This subrecord contains an additional longword field, ESDFV$L_VECTOR, which contains an offset into the symbol vector of either SYS$BASE_IMAGE.EXE or SYS$PUBLIC_VECTORS.EXE. The target image is determined by the name of the absolute program section pointed to by the field ESDFV$L_PSINDX. If the name of the psect is ‘‘.$$SYS$BASE_IMAGE$$.’’ then the symbol vector of SYS$BASE_IMAGE.EXE is modified. If the name of the absolute program section is ‘‘.$$SYS$PUBLIC_VECTORS$$.’’ then the symbol vector of SYS$PUBLIC_VECTORS.EXE is modified. Once

the symbol vector is modified, it points (directly or indirectly) to the loaded data or procedure.

- This subrecord does not contain the ESDF$L_CODE_ADDESS or ESDF$L_CA_PSINDX fields.
- The VALUE field contains an image offset, not a symbol vector offset. Execlets do not have symbol vectors; therefore, the VALUE field is treated differently.

This subrecord is reserved for use by Digital only.

B.3.4.2 Symbol-Definition-with-Version-Mask Subrecord (EGSD$C_SYMM)

This subrecord is identical in format to the global symbol definition subrecord described in Section B.3.2.1, with the following exceptions:

- The field names in this record begin with ESDFM$ instead of ESDF$ as in the global symbol definition subrecord.
- This subrecord contains an additional longword field, ESDFM$L_VERSION_MASK. This mask is used to index a list of system components on which the symbol is dependent. This mask is used by the linker to create the system version array in an image that links against SYS$BASE_IMAGE.EXE.
- This subrecord does not contain the ESDF$L_CODE_ADDESS or ESDF$L_CA_PSINDX fields.

This subrecord is reserved for use by Digital only.

B.4 Text Information and Relocation Records (EOBJ$C_ETIR)

A text information and relocation (TIR) record contains commands and data that the linker uses to manipulate its internal stack, perform calculations, and initialize the image.

A TIR record consists of the RECORD TYPE field (EOBJ$W_RECTYP, set to EOBJ$C_ETIR), the record size (EOBJ$W_SIZE), and one or more TIR commands. Each TIR command consists of a type (ETIR$W_RECTYP), the command size (ETIR$W_SIZE), and the command arguments. A TIR record may contain many TIR commands, but it must not exceed the record size limit for the object module as defined in the MAXIMUM RECORD SIZE field (EMH$L_RECSIZ) of the main module header record.

RECORD TYPE Name: EOBJ$W_RECTYP
Length: 2 bytes

The record type is EOBJ$C_ETIR.

RECORD SIZE Name: EOBJ$W_SIZE
Length: 2 bytes

The record size must include the record size and type fields themselves, as well as the size of any TIR commands that the record contains.

TIR COMMAND TYPE Name: ETIR$W_RECTYP
Length: 2 bytes

This field designates the TIR command type.

TIR COMMAND SIZE Name: ETIR$W_SIZE
Length: 2 bytes

The command size must include the command size and type fields themselves, as well as the aggregate length of any command arguments.

There are 69 text information and relocation commands, divided into five classes with the following associated ranges:

Command Type	Minimum Value	Maximum Value
Stack	ETIR$C_MINSTACOD (0)	ETIR$C_MAXSTACOD (6)
Store	ETIR$C_MINSTOCOD (50)	ETIR$C_MAXSTOCOD (65)
Operator	ETIR$C_MINOPRCOD (100)	ETIR$C_MAXOPRCOD (116)
Control	ETIR$C_MINCTLCOD (150)	ETIR$C_MAXCTLCOD (154)
Store Conditional	ETIR$C_MINSTCCOD (200)	ETIR$C_MAXCTLCOD (214)

TIR commands either manipulate the linker's stack or initialize storage in the image. Stack commands cause the linker to push values onto the stack. The linker's stack is 64 bits wide. Each value pushed onto the stack is converted to a 64-bit value as required. Store commands are used to fetch global values or to pop the stack and store the results in the image. Operator commands perform operations on values that were pushed onto the stack with previous operations. The result is always stored back on the stack. Control commands set the linker's location counter in preparation for the next store command. Conditional store commands are used to set up conditional linkages and to optimize the instruction stream.

The most common store command, STORE IMMEDIATE (ETIR$C_STO_IMM), is used to write the specified binary directly into the image. This is how the instruction stream is written. The linker is unaware of the instructions written by a STORE IMMEDIATE command. The only way for the linker to interact with the instruction stream is by means of the conditional store commands. The linker does not execute a STORE IMMEDIATE command (either ETIR$C_STO_IMM or ETIR$C_STO_IMMR) when the value being stored is zero. Such a command is, in effect, a null operation. The pages of an image file are guaranteed to be zero until they are specifically initialized by the compiler.

When the linker finishes processing the TIR commands for a given module, that is, when it processes the EEOM record, the stack must be completely collapsed. Otherwise, the linker will issue a warning, LINK-W-EOMSTCK.

TIR commands are described in the following five subsections. Section B.4.1 discusses the stack commands; Section B.4.2, the store commands; Section B.4.3, operator commands; Section B.4.4, control commands; and Section B.4.5, conditional store commands. The commands are presented in numerical order, based on their equivalent numerical values (in decimal).

B.4.1 Stack Commands

The stack commands place longwords and quadwords on the stack. The value placed on the stack is taken from one of the following sources:

- The arguments directly following the ETIR$W_SIZE field
- A global symbol
- A computation derived by adding an offset to the base address of a program section

Each stack command increments the linker's stack pointer. Context is saved to indicate whether the value on the stack is relocatable or defined by a shareable image and therefore external to the image. This context is used later by the operator and store commands. Table B–8 lists the stack commands and their related values, together with a brief description of each command.

Table B–8 Stack Commands

Value	Command	Description
0	**ETIR$C_STA_GBL** (Stack Global)	Argument is a counted ASCII string containing the name of a global symbol. The command stacks the 32-bit binary value of the symbol. The value is not sign extended. The high-order 32 bits are always 0. The saved context identifies the value as relocatable if the EGSY$V_REL bit in the symbol definition was set. The saved context identifies the value as external if the symbol was defined by a shareable image.
1	**ETIR$C_STA_LW** (Stack Longword)	Argument is a 4-byte constant. The value is sign extended to 64 bits.
2	**ETIR$C_STA_QW** (Stack Quadword)	Argument is an 8-byte constant. If the high-order 32 bits are nonzero, they are ignored in calculations performed with subsequent operator commands.
3	**ETIR$C_STA_PQ** (Stack Psect Base Plus Byte Offset)	Arguments are a longword program section index and a quadword whose low-order 32 bits represent a signed byte offset from the program section base.[1] The quadword argument is pushed onto the stack and context is saved to indicate whether the program section is relocatable or defined by another shareable image. A subsequent ETIR$C_STO_ OFF adds the 32-bit offset to the program section base and sign-extends the result to 64 bits. This command should always be followed by an ETIR$C_ STO_OFF command.
4	**ETIR$C_STA_LI** (Stack Literal)	Not supported in structure level 2.
5	**ETIR$C_STA_MOD** (Stack Module)	Not supported in structure level 2.

[1]For VAX object language, the TIR$C_STA_PB command provides for a signed offset value, to be used with absolute psects to express constants. For Alpha object language, this is illegal. The result will be relocated, which is inappropriate for constants. Use the ETIR$C_STA_QW and ETIR$C_STA_LW commands with constants.

(continued on next page)

Table B–8 (Cont.) Stack Commands

Value	Command	Description
6	**ETIR$C_STA_CKARG** (Compare Procedure Argument and Stack for TRUE or FALSE)	Not supported in structure level 2.
7–49		Reserved to Digital.

B.4.2 Store Commands

Store commands instruct the linker to write a stream of bytes at the current image location counter. The image location counter is set implicitly by a previous store command or explicitly with a control command (for example, ETIR$C_CTL_SETRB). After a store command is executed, the image location counter is pointing to the next byte in the output image. Some store commands pop values from the linker's stack (decrementing the linker's stack pointer), and others do not.

The context for a value that is saved by a stack command indicates whether the value is relocatable and must be treated as an address by subsequent operator and store commands. A **relocation** is generated if the image is relocatable (for example, linked /SHARE) and the saved context indicates that the address occurs within the image. When the image activator processes relocations, it adds the base address of the image to the value stored at the location indicated by the linker.

If the saved context indicates that the value is defined by a shareable image and is therefore external to the image being created, then a **fix-up** is generated. Fix-ups are generated without consideration of whether the value is relocatable. For VAX images, references to constants that were defined externally were resolved at link time, and therefore they did not require fixing up. For Alpha images, the linker generates fix-ups for references to externally defined constants. The references are finally resolved by the image activator. This makes it easier to construct an upwardly compatible Alpha shareable image. When the image activator processes fix-ups, it moves an entry from the symbol vector of the shareable image that defined the symbol into the linkage section of the image that referenced it.

Table B–9 lists the store commands and a brief description of each command. Commands that pop the stack and thereby decrement the linker's stack pointer are noted.

Table B–9 Store Commands

Value	Command	Description
50	**ETIR$C_STO_B** (Store Byte)	The stack is popped. The low byte is written to the image. If the value on the stack is a constant contributed by a shareable image, the linker issues an error message indicating that byte fix-ups are not supported.

(continued on next page)

Table B–9 (Cont.) Store Commands

Value	Command	Description
51	**ETIR$C_STO_W** (Store Word)	The stack is popped. The low word is written to the image. If the value on the stack is a constant contributed by a shareable image, the linker issues an error message indicating that word fix-ups are not supported.
52	**ETIR$C_STO_LW** (Store Longword)	The stack is popped. The low longword is written to the image. A relocation or fix-up is generated based on the saved context.
53	**ETIR$C_STO_QW** (Store Quadword)	The stack is popped. A quadword is written to the image. A relocation or fix-up is generated based on the saved context.
54	**ETIR$C_STO_IMMR** (Store Immediate Repeated)	Arguments are a longword count of bytes and a stream of binary data. The stack is popped. The low-order 32 bits of the popped value are used as a repeat count. The data stream is successively stored the number of times the count indicates.
55	**ETIR$C_STO_GBL** (Store Global)	Argument is a counted ASCII string containing the name of a global symbol. If the global value is not external and not relocatable then the 32-bit value field is sign extended to 64 bits. Otherwise, the high-order 32 bits are 0. A quadword is written to image. If the global value is relocatable and the image is relocatable, a relocation is generated. If the global value is contributed by a shareable image, a fix-up is generated.
56	**ETIR$C_STO_CA** (Store Code Address)	Argument is a counted ASCII string containing the name of a global symbol. The address of the entry point of the procedure named by the string is written into the image. The address is written as a quadword with the high-order 32 bits set to 0. A relocation is generated if the image is relocatable and the global symbol is not external. A fix-up is generated if the procedure is defined by a shareable image.
57	**ETIR$C_STO_RB** (Store Relative Branch)	Not supported in structure level 2.
58	**ETIR$C_STO_AB** (Store Absolute Branch)	Not supported in structure level 2.

(continued on next page)

Table B–9 (Cont.) Store Commands

Value	Command	Description
59	**ETIR$C_STO_OFF** (Store Offset to Psect)	The quadword offset is popped from the stack and the low-order 32 bits are added to the base of the saved program section. The result is sign extended to 64 bits and written to the image. The value is always treated as an address and relocated if the image is relocatable, or fixed up if the program section is contributed by a shareable image. This command should always be preceded by an ETIR$C_STA_PQ command.
60		Reserved to Digital.
61	**ETIR$C_STO_IMM** (Store Immediate)	Arguments are a longword count of bytes and a stream of binary data. The data stream is stored directly into the image at the current image location counter.
62		Reserved to Digital.
63	**ETIR$C_STO_LP_PSB** (Store LP with Procedure Signature)	Not supported in structure level 2.
64	**ETIR$C_STO_BR_GBL** (Store Branch Global)	Arguments are a longword program section index and quadword offset pointing to the instruction to be replaced, a longword program section and quadword offset pointing to a base address, and a counted string containing the name of a procedure. If the longword displacement from the base address to the code address associated with the procedure name can be expressed as a signed 21-bit integer, then store the displacement in the low-order 21 bits of the instruction pointed to by the first program section index and quadword offset.

(continued on next page)

Table B–9 (Cont.) Store Commands

Value	Command	Description
65	**ETIR$C_STO_BR_PS** (Store Branch Psect + Offset)	Arguments are a longword program section index and quadword offset pointing to the instruction to be replaced; a longword program section and quadword offset pointing to a base address; and a longword program section and quadword offset pointing to a target address. If the longword displacement from the base address to the target address can be expressed as a signed 21-bit integer, then store the displacement in the low-order 21 bits of the instruction pointed to by the first program section index and quadword offset.
66–99		Reserved to Digital.

B.4.3 Operator Commands

Operator commands perform 32-bit arithmetic operations on the low-order longwords of operands popped from the stack. The linker evaluates expressions in Post Fix Polish form. All arithmetic operations are performed in signed 32-bit two's complement integers. There is no provision for floating-point, string, or quadword computation. Attempts to divide by zero produce a zero result and a nonfatal warning message. Upon completion of the operation, the result is pushed back on the stack. The contents of the high-order 32 bits of the result are not predictable. Therefore the result should be popped from the stack with a longword store command (ETIR$C_STO_LW) rather than a quadword store command (ETIR$C_STO_QW).

The operator commands check the context saved by previous stack or operator commands. If one of the operands is defined by a shareable image, then the other operand must be a constant and the operation must be addition or subtraction. The context saved for the result is the context of the relocatable operand. In those cases where both operands are relocatable, neither is allowed to be external, so the saved context for the result simply indicates that it is relocatable and not external.

Table B–10 lists the operator commands and related values, together with a brief description of each command.

Table B–10 Operator Commands

Value	Command	Description
100	**ETIR$C_OPR_NOP** (No-Operation)	No operation results.
101	**ETIR$C_OPR_ADD** (Add)	Stack is popped twice. The two popped values are added, and the result is pushed back onto the stack. Stack pointer is decremented.

(continued on next page)

Table B–10 (Cont.) Operator Commands

Value	Command	Description
102	**ETIR$C_OPR_SUB** (Subtract)	Stack is popped twice. First value popped is subtracted from the second, and the result is pushed back onto the stack. Stack pointer is decremented.
103	**ETIR$C_OPR_MUL** (Multiply)	Stack is popped twice. The values are multiplied and the result is pushed back onto the stack. Stack pointer is decremented.
104	**ETIR$C_OPR_DIV** (Divide)	Stack is popped twice. The second value popped is divided by the first value popped, and the result is pushed back onto the stack. Stack pointer is decremented.
105	**ETIR$C_OPR_AND** (Logical AND)	Stack is popped twice. The result is the logical AND of the two operands. The result is pushed back onto the stack. Stack pointer is decremented.
106	**ETIR$C_OPR_IOR** (Logical Inclusive OR)	Stack is popped twice. The result is the inclusive OR of the two operands. The result is pushed back onto the stack. Stack pointer is decremented.
107	**ETIR$C_OPR_EOR** (Logical Exclusive OR)	Stack is popped twice. The result is the exclusive OR of the two operands. The result is pushed back onto the stack. Stack pointer is decremented.
108	**ETIR$C_OPR_NEG** (Negate)	Stack is popped once. The value is negated and pushed back onto the stack. Stack pointer is unchanged.
109	**ETIR$C_OPR_COM** (Complement)	Stack is popped once. The value is complemented and pushed back onto the stack. Stack pointer is unchanged.
110	**ETIR$C_OPR_INSV** (Insert Field)	Not supported in structure level 2.
111	**ETIR$C_OPR_ASH** (Arithmetic Shift)	Stack is popped twice. The second value popped specifies the shift count and direction to be applied to the first value popped. When the shift count is negative, the bits are shifted right with replication of the sign bit. When the shift count is positive, the bits are shifted left with zeros moved into low-order bits. The result is pushed back onto the stack. Stack pointer is decremented.
112	**ETIR$C_OPR_USH** (Unsigned Shift)	Not supported in structure level 2.
113	**ETIR$C_OPR_ROT** (Rotate)	The stack is popped twice. The second value popped specifies the rotation count and direction to be applied to first value popped. When the rotation count is positive, the bits in the first value popped are rotated left; when negative, right. The rotation count must have an absolute value of 0 to 32. The result is pushed back onto the stack. Stack pointer is decremented.

(continued on next page)

Table B–10 (Cont.) Operator Commands

Value	Command	Description
114	**ETIR$C_OPR_SEL** (Select)	Stack is popped two or three times. If the first value popped evaluates to TRUE (low bit set), another value is popped, leaving the top value on the resulting stack unchanged. If the first value popped evaluates to FALSE (low bit clear), the stack is popped two more times. The value popped by the second of the three pops is pushed back onto the stack, replacing the value removed by the third pop. In both cases, the stack pointer is decremented by two.
115	**ETIR$C_OPR_REDEF** (Redefine Symbol to Current Location)	Not supported in structure level 2.
116	**ETIR$C_OPR_DFLIT** (Define a Literal)	Not supported in structure level 2.
117–149		Reserved to Digital.

B.4.4 Control Commands

Control commands manipulate the linker's location counter. Table B–11 lists the control commands and related values, together with a brief description of each command.

Table B–11 Control Commands

Value	Command	Description
150	**ETIR$C_CTL_SETRB** (Set Relocation Base)	The stack is popped. The low-order longword of the value is placed in the image location counter. The stack pointer is decremented.
151	**ETIR$C_CTL_AUGRB** (Augment Relocation Base)	Argument is a signed longword. The value of this longword is added to the current image location counter. Stack pointer is unchanged.
152[1]	**ETIR$C_CTL_DFLOC** (Define Location)	The stack is popped. The low-order longword of the popped value is used as an index. The value of the current DST location counter is then saved under this index.
153[1]	**ETIR$C_CTL_STLOC** (Set Location)	The stack is popped. The low-order longword of the value is used as an index to locate a DST location counter saved by a previous DEFINE LOCATION command. The current DST location counter is updated with the saved DST location counter. The stack pointer is decremented.
154[1]	**ETIR$C_CTL_STKDL** (Stack Defined Location)	The stack is popped. The low-order longword of the popped value is used as an index to locate a DST location counter saved by a previous DEFINE LOCATION command. The saved DST location counter is pushed back on the stack. The stack pointer is unchanged.

[1]This command is legal only in debugger information (DBG) and traceback information (TBT) records. For each object module, a list of debugger indexes is kept. These commands operate on the list for the object module in which the DBG or TBT record occurs.

(continued on next page)

Table B–11 (Cont.) Control Commands

Value	Command	Description
155–199		Reserved to Digital.

B.4.5 Conditional Store Commands

There are two types of conditional store commands. The first type *declares conditional linkage*, and assigns the linkage a unique index. The second type *enables the linker to replace an instruction* in the code stream. If an instruction is replaced, the linker is able to identify the conditional linkage that the original instruction relied on and eliminate the relocation that would otherwise have been generated for it. This is called **linkage retirement**. Linkage retirement leaves the storage allocated for the linkage intact but eliminates the relocations. Linkage retirement is not supported for structure level 2; the linker unconditionally generates relocations for all linkages.

B.4.5.1 Defining Conditional Linkage with Address-Related Commands

Table B–12 lists the commands for allocating conditional linkage with the arguments that follow the size field.

The ETIR$C_STC_LP_PSB command is used to pass signature information associated with the procedure to the linker. The linker propagates this information to the fix-up section with a special type of fix-up when the /NONATIVE_ONLY switch is specified. The signature information is used by the image activator to build jackets at activation time if the routine resides in a translated image. The signature is opaque to the linker. For more information about the format of OpenVMS procedure signatures, see the *OpenVMS Calling Standard*.

Table B–12 Summary of Store Conditional Commands for Linkage

Value	Command	Arguments
200	**ETIR$C_STC_LP** (Store Conditional Linkage Pair)	Not supported in structure level 2
201	**ETIR$C_STC_LP_PSB** (Store Conditional Linkage Pair Plus Signature)	Linkage Index (longword) Procedure Name (counted string) Signature Length (byte, may be 0) Signature (only if length is positive)
202	**ETIR$C_STC_GBL** (Store Conditional Global)	Linkage Index (longword) Global Name (counted string)
203	**ETIR$C_STC_GCA** (Store Conditional Code Address)	Linkage Index (longword) Procedure Name (counted string)
204	**ETIR$C_STC_PS** (Store Conditional Psect Plus Offset)	Linkage Index (longword) Psect index (longword) Offset (quadword)

The linkage index argument common to all of these commands is a positive integer value that must be unique to the module. Each index identifies a quadword of linkage. The commands used to declare a conditional linkage pair (ETIR$C_STC_LP and ETIR$C_STC_LP_PSB) reserve two index values. For example, if the index value 3 is declared, the index value 4 is implicitly declared. The linkage index is used by subsequent instruction replacement commands to identify the linkage on which the original instruction was dependent.

Linkage that is declared with a conditional linkage command and a nonzero linkage index can be used only by instructions that are identified by one of the instruction-related conditional store commands listed in Section B.4.5.2. Linkage that is declared with a conditional linkage command and a zero linkage index is not conditional. Improper use of conditional linkage cannot be detected by the linker and will result in incorrect program execution.

The linker assumes that its image location pointer points to the linkage being declared when the conditional store command is processed. If the linkage is needed, the linker will fill it with different values depending on the command as indicated in Tables B–13 and B–14. (Note that for structure level 2, the linker always fills in and relocates linkages.) The term **symbol vector offset** designates an offset into the symbol vector of the image that defined the symbol. The type of relocation or fix-up applied to the value in the linkage pair is specified in parentheses.

Table B–13 Contents of Linkage When Symbol Is Local to the Image

Command	First Quadword	Second Quadword
ETIR$C_STC_LP	Not supported in structure level 2	
ETIR$C_STC_LP_PSB	Image offset of procedure entry (Quadword relocation)	Image offset of procedure descriptor (Quadword relocation)
ETIR$C_STC_GBL	Image offset of global data (Quadword relocation)	Not applicable
ETIR$C_STC_GCA	Image offset of procedure entry (Quadword relocation)	Not applicable
ETIR$C_STC_PS	Image offset of (psect + offset) (Quadword relocation)	Not applicable

Table B–14 Contents of Linkage When Symbol Is External to the Image

Command	First Quadword	Second Quadword
ETIR$C_STC_LP	Not supported in structure level 2	
ETIR$C_STC_LP_PSB	Symbol vector offset (Linkage pair with signature fix-up)	Zero
ETIR$C_STC_GBL[1]	Symbol vector offset (Quadword .ADDRESS fix-up)	Not applicable
ETIR$C_STC_GCA[1]	Symbol vector offset (Code address fix-up)	Not applicable
ETIR$C_STC_PS[1] (Only if psect is OVR,REL,GBL)	Symbol vector offset (Quadword .ADDRESS fix-up)	Not applicable

[1]Not supported for structure level 2.

B.4.5.2 Optimizing Instructions with Instruction-Related Commands

At the direction of the compiler, the linker is able to optimize portions of the instruction stream. The compiler identifies replacements with one of the instruction-related commands listed in Table B–15. These commands supply a reference to the linkage (previously defined with one of the commands listed in Table B–12), a replacement instruction, and some conditions for the linker to test.

Figure B–11 shows the offsets used by the linker to test for replacement and the replacement instructions that are used. The following commands might be used to replace a sequence of instructions used to call procedure B from inside procedure A:

- ETIR$C_STC_NOP
- ETIR$C_STC_LDA
- ETIR$C_STC_BSR

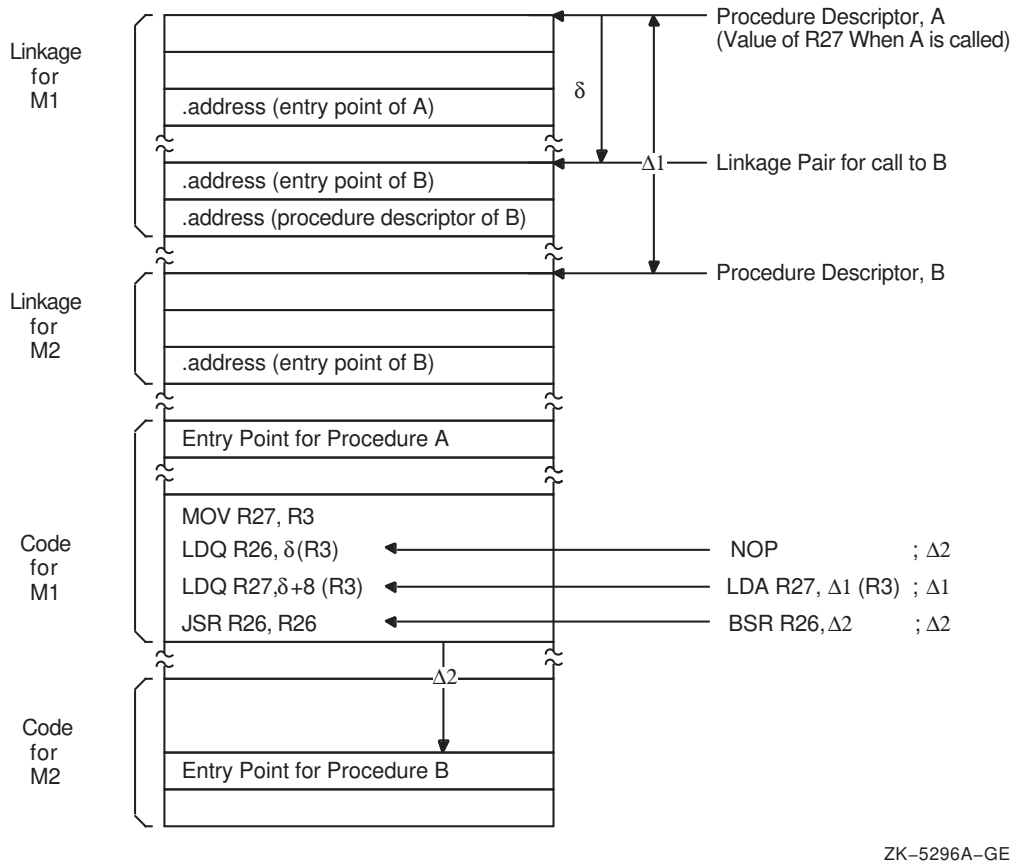
The ETIR$C_STC_NOP passes the addresses used to calculate the offset $\Delta 2$. If $\Delta 2$ can be expressed as a signed 21-bit integer, the linker replaces the first LDQ instruction with the instruction specified by the command. (For example, the compiler might specify BIS R31,R31,R31, an instruction that does nothing and uses very few CPU cycles). The replacement of the first LDQ instruction anticipates the replacement of the JSR instruction and eliminates a memory reference, that is, the reference to the linkage pair that contains the address of the procedure entry for B. If $\Delta 2$ is too large to be expressed as a signed 21-bit integer, the linker does not do the replacement. The linker cannot delete the storage that contained the LDQ instruction (instead of replacing it with a NOP instruction) because the offset δ that was calculated by the compiler would no longer be valid.

The ETIR$C_STC_LDA command passes the addresses used to calculate the offset $\Delta 1$. If $\Delta 1$ can be expressed as a signed 16-bit integer, the linker replaces the second LDQ instruction with the LDA instruction supplied by the command and fills in the 16-bit offset field with $\Delta 1$. This replacement eliminates a memory reference to the second half of the linkage pair, which contains the address of the procedure descriptor for B. If $\Delta 1$ is too large to be expressed as a signed 16-bit integer, the linker does not do the replacement. This replacement can happen independently of the replacement of the other LDQ and JSR instructions.

Finally, the ETIR$C_STC_BSR command passes the same addresses as ETIR$C_STC_NOP (so that they will fail or succeed together), and the same test is done again. If the test succeeds, the linker replaces the JSR instruction with the BSR instruction supplied by the command and fills in the 21-bit offset field with $\Delta 2$.

Table B–15 lists the commands for optimizing the instruction stream. The default instruction stream must be placed in memory (typically with an ETIR$C_STO_IMM command) before the instruction-related commands are issued. The linker cannot detect attempts to optimize instructions that have not previously been placed in memory. Each command in the table has the following arguments (excluding the ETIR$C_STC_NBH_PS and ETIR$C_STC_NBH_GBL commands, which are not supported in structure level 2). The argument abbreviations are used by the commands in Table B–15 to describe the operation of the command.

Figure B–11 Optimization of a Standard Call



Argument	Abbreviation	Size
Linkage Index	LI	Longword
Psect Index 1	PS1	Longword
Offset 1	OFF1	Quadword
Replacement instruction	REPINS	Longword
Psect Index 2	PS2	Longword
Offset 2	OFF2	Quadword
Global Name (_GBL commands only)	GNAM	Counted String
Psect Index 3 (_PS commands only)	PS3	Longword
Offset 3 (_PS commands only)	OFF3	Quadword

The **Linkage Index** argument must be nonzero and point to the linkage declared with an address-related conditional store command. The **PS1** and **OFF1** arguments are used to calculate the location of the instruction that needs to be replaced. For the commands that deal with NOP and BSR instructions, the **PS2** and **OFF2** arguments point to the base of the calculation, typically, $JSR + 4$. The commands for replacing the LDA instruction use **PS2** and **OFF2** to point to the base of the linkage section. The **PS3** and **OFF3** arguments point to the

procedure entry point for the NOP and BSR commands and to the procedure descriptor for the LDA commands.

Table B–15 Summary of Store Conditional Commands for Instruction Replacement

Value	Command	Description
205	**ETIR$C_STC_NOP_GBL** (Store Conditional NOP at Global Address)	Store a NOP (specified in REPINS) at (PS1 + OFF1) if the longword displacement from (PS2 + OFF2) to the procedure entry associated with GNAM can be expressed as a signed 21-bit integer.
206	**ETIR$C_STC_NOP_PS** (Store Conditional NOP at Psect Plus Offset)	Store a NOP (specified in REPINS) at (PS1 + OFF1) if the longword displacement from (PS2 + OFF2) to (PS3 + OFF3) can be expressed as a signed 21-bit integer.
207	**ETIR$C_STC_BSR_GBL** (Store Conditional BSR at Global Address)	If the longword displacement from (PS2 + OFF2) to the procedure entry associated with GNAM can be expressed as a signed 21-bit integer, then insert the displacement into the low-order 21 bits of the BSR (specified in REPINS) and store it at (PS1 + OFF1).
208	**ETIR$C_STC_BSR_PS** (Store Conditional BSR at Psect Plus Offset)	If the longword displacement from (PS2 + OFF2) to (PS3 + OFF3) can be expressed as a signed 21-bit integer, then insert the displacement into the low-order 21 bits of the BSR (specified in REPINS) and store it at (PS1 + OFF1).
209	**ETIR$C_STC_LDA_GBL** (Store Conditional LDA at Global Address)	If the byte displacement from (PS2 + OFF2) to the procedure descriptor associated with GNAM can be expressed as a signed 16-bit integer, then insert the displacement into the low-order 16 bits of the LDA instruction (specified in REPINS) and store it at (PS1 + OFF1).
210	**ETIR$C_STC_LDA_PS** (Store Conditional LDA at Psect Plus Offset)	If the byte displacement from (PS2 + OFF2) to (PS3 + OFF3) can be expressed as a signed 16-bit integer, then insert the displacement into the low-order 16 bits of the LDA instruction (specified in REPINS) and store it at (PS1 + OFF1).
211	**ETIR$C_STC_BOH_GBL** (Store Conditional BSR or Hint at Global Address)	If the longword displacement from (PS2 + OFF2) to the procedure entry associated with GNAM can be expressed as a signed 21-bit integer, then insert the displacement into the low-order bits of the instruction specified in REPINS and store it at (PS1 + OFF1). If the displacement is too large or the global name is defined by a shareable image, insert a hint into the low-order 14 bits of the instruction at (PS1 + OFF1).

(continued on next page)

Table B–15 (Cont.) Summary of Store Conditional Commands for Instruction Replacement

Value	Command	Description
212	**ETIR$C_STC_BOH_PS** (Store Conditional BSR or Hint at Psect Plus Offset)	If the longword displacement from (PS2 + OFF2) to (PS3 + OFF3) can be expressed as a signed 21-bit integer, then insert the displacement into the low-order bits of the instruction specified in REPINS and store it at (PS1 + OFF1). If the displacement is too large, insert a hint into the low-order 14 bits of the instruction at (PS1 + OFF1).
213	**ETIR$C_STC_NBH_GBL** (Store Conditional (NOP and BSR) or Hint at Global Address)	Not supported in structure level 2.
214	**ETIR$C_STC_NBH_PS** (Store Conditional (NOP and BSR) or Hint at Psect Plus Offset)	Not supported in structure level 2.

Instruction replacement will fail when any of the following conditions occur:

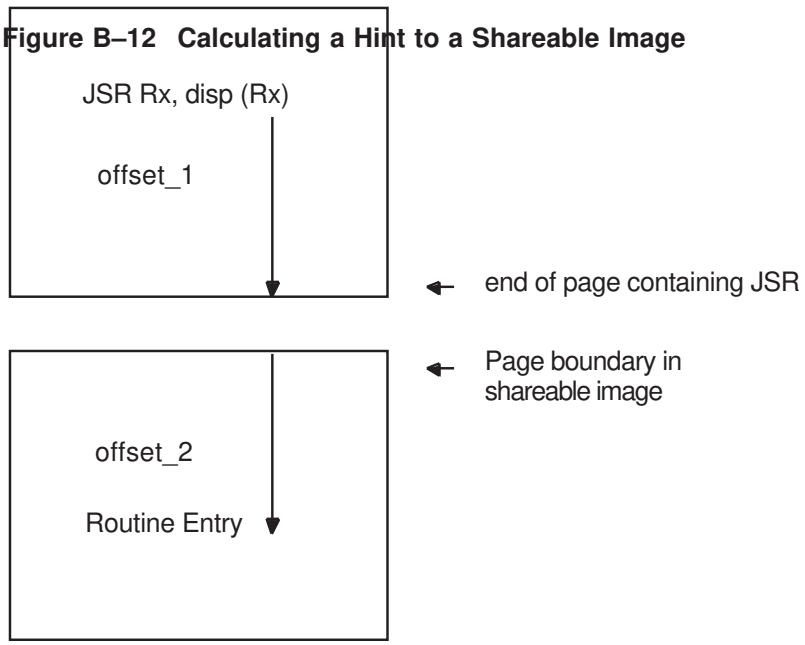
- The linker qualifier /NOREPLACE is specified.
- The displacement cannot be expressed in the number of bits specified for each command listed in Table B–15.
- The linker qualifier /SECTION_BINDING=CODE is specified and the call and the destination are in different image sections.
- A command with the _GBL suffix is used and the global name is defined by a shareable image.

B.4.5.2.1 Calculating JSR Hints When the conditions for replacement cannot be met, the ETIR$C_STC_BOH_PS and ETIR$C_STC_BOH_GBL commands calculate hints and insert them into the low-order 14 bits of the instruction in the default instruction stream.

The hint field in the JSR instruction is used to index the instruction cache. The hint is correct if the low-order 16 bits of the result of $JSR + 4 + (4 * HINT < 13:0>)$ is equal to the low-order 16 bits of the address of the destination routine's entry point. Program execution is not affected by a wrong hint, but performance may be. Even given a correct hint, the instruction cache may not contain the desired instructions, which may then have to be fetched from memory or disk.

Hints to shareable images may no longer be correct at run time if the shareable image is relinked after the main image is created. Hints to shareable images installed with the /RESIDENT qualifier will still be correct because the position of the procedure entry relative to the beginning of a page boundary will not have changed. Hints generated for calls to the system executive, for example, calls to SYS$BASE_IMAGE.EXE or SYS$PUBLIC_VECTORS.EXE, will not be correct. The image offsets contained in their global symbol tables cannot be converted at link time into the real page offsets of the procedures in the loaded execlets.

Figure B–12 Calculating a Hint to a Shareable Image



If the destination routine is inside the image being linked, the linker calculates the hint as the longword displacement between $JSR + 4$ and the procedure entry point. If the displacement is negative, the algorithm still works; the low-order 16 bits of $JSR + 4 + \left(4 * HINT < 13 : 0 >\right)$ is still equal to the low-order 16 bits of the address of the destination routine. It is irrelevant that the high-order 16 bits of the result may not equal the high-order 16 bits of the destination routine.

If the destination routine is outside the image, the linker adds the byte displacement from the JSR instruction to the end of the current page (offset_ 1) to the byte displacement of the destination routine from the beginning of the page (offset_2), as shown in Figure B–12. The low-order 16 bits of the result are shifted right two bits to generate the hint. The linker uses the EGST$L_LP_1 field in the definition of the destination routine (from the global symbol table) and the image page size (from the image header) to determine the offset into the page of the procedure entry.

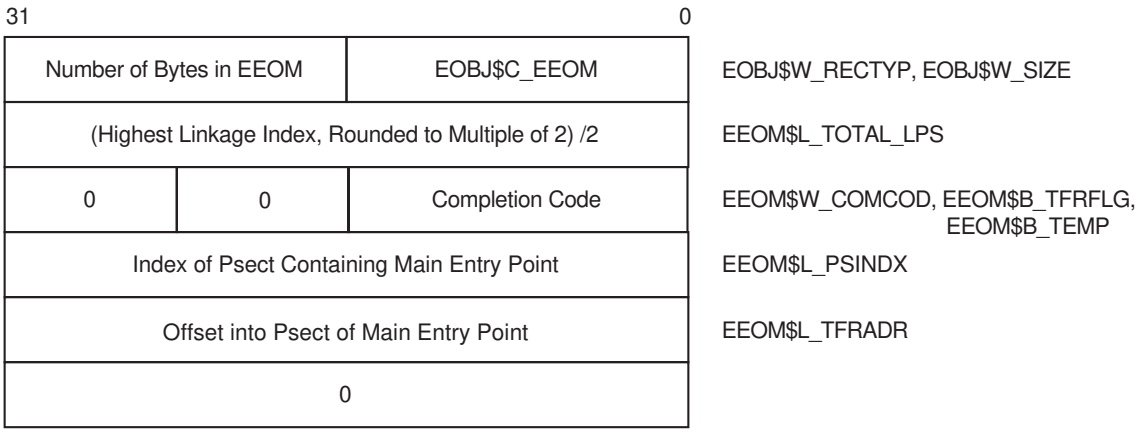
B.5 End-of-Module Record (EOBJ$C_EEOM)

The end-of-module (EOBJ$C_EEOM) record declares the end of the module. It must be the last record in the object module.

If the module does not contain a program section that contains the transfer address, the end-of-module (EOM) record is 10 bytes long, consisting of only the RECORD TYPE, RECORD SIZE, LINKAGE COUNT, and COMPLETION CODE fields.

If the module does contain a program section that contains the transfer address, the EOM record is 24 bytes long. A full EOM record is shown in Figure B–13.

Figure B–13 End-of-Module Record



ZK–5279A–GE

The fields in an EEOM record are described in the following list.

RECORD TYPE Name: EEOM$W_RECTYP
Length: 2 bytes

The field EEOM$W_RECTYP redefines EOBJ$W_RECTYP. It must contain the value EOBJ$C_EEOM.

RECORD SIZE Name: EEOM$W_SIZE
Length: 2 bytes

The field EEOM$W_SIZE redefines EOBJ$W_SIZE. It is the size of the entire record, including the preceding record type field.

TOTAL LINKAGE Name: EEOM$L_TOTAL_LPS
Length: 4 bytes

This field contains the highest linkage index, rounded up to an even number, and divided by two. It is used by the linker to allocate table space after Pass 1 and to check the range of linkage indexes passed in TIR commands processed in Pass 2. The linker does not detect improper use of the highest index when rounding occurs. For example, if the highest linkage declared was 9 (yielding a value of 5 for the EEOM$L_TOTAL_LPS field), and a TIR command specifies a linkage index of 10 (which appears to be in range), unpredictable results will occur.

COMPLETION CODE Name: EEOM$B_COMCOD
Length: 1 byte

This field contains completion codes, which are generated by the language processor. This field may contain a value from 0 to 3, where each number corresponds to a completion code. Values from 4 to 255 are reserved by Digital. The following table lists the name, corresponding value, and meaning of each of the four completion codes.

Value	Name	Meaning
0	EEOM$C_SUCCESS	Successful compilation or assembly; no errors detected.
1	EEOM$C_WARNING	Language processor generated warning messages. The linker issues a warning message and proceeds with the linking operation.
2	EEOM$C_ERROR	Language processor generated severe errors. The linker issues an error message, proceeds with the linking operation, but does not produce an output image file.
3	EEOM$C_ABORT	Language processor generated fatal errors. The linker aborts the linking operation.
4–255		Reserved to Digital.

When the linker is creating a shareable image, it performs a logical OR on the completion codes from all of the modules and shareable images that contributed to it. The result is then propagated to the COMPLETION CODE field of the end-of-module record of the resulting image's global symbol table. If any module or shareable image specified a completion code other than EEOM$C_SUCCESS, any programmer who links against the resulting image will receive a warning.

TRANSFER FLAGS Name: EEOM$B_TFRFLG
Length: 1 byte

This field is a 1-byte bit mask that contains information about the transfer address. When the EEOM$V_WKTFR bit is set, a weak transfer address is indicated; when clear, a strong transfer address is indicated. If bit EEOM$V_ WKTFR is set and a transfer address has already been defined, no error results. Bits 1 to 7 are reserved and must contain zeros. Note that this field may be present only if the module contains a program section that contains the transfer address.

ALIGNMENT BYTE Name: EEOM$B_TEMP
Length: 1 byte

Alignment byte, must be 0.

PSECT INDEX Name: EEOM$L_PSINDX
Length: 4 bytes

This field contains the program section index of the program section within the module that contains the transfer address. Note that this field is present only if the module contains a program section that contains a transfer address.

TRANSFER ADDRESS Name: EEOM$L_TFRADR
Length: 4 bytes

This field contains the location of the transfer address. The location is expressed as an offset from the base of this module's contribution to the program section that contains the transfer address. Note that this field is present only if the module contains a program section that contains the transfer address.

B.6 Debugger Information Records (EOBJ$C_EDBG)

The purpose of debugger information records is to allow the language processors to pass compilation information, such as descriptions of local variables, to the debugger. The transmission of this information may make use of all the functions (commands) available in the TIR command set, except for the instruction-related conditional store commands described in Section B.4.5.2.

The command stream in DBG records generates a debugger symbol table (DST). The DST immediately follows the binary of the user image, and the image header contains a descriptor of where in the file such data is written. The production of the DST in memory makes use of a separate location counter within the linker. Note that the linker does not produce an image section descriptor for the DST and that the DST is not mapped into the user's address space by the image activator. The debugger must read and map the DST.

The linker uses ETIR$C_EDBG and ETIR$C_ETBT records to produce a DST if the image is linked with the /DEBUG qualifier. If the image is only linked with the /TRACEBACK qualifier (the default), then the linker skips the ETIR$C_ EDBG records and builds a DST using only the ETIR$C_ETBT records. The linker will not process the ETIR$C_EDBG records and will skip the ETIR$C_ ETBT records; if you specify /DEBUG/NOTRACEBACK, the linker ignores the /NOTRACEBACK qualifier.

B.7 Traceback Information Records (EOBJ$C_ETBT)

Traceback information records are the means by which language processors pass information to the facility that produces a traceback of the call stack. From the point of view of the linker and its processing of these records, they are identical to DBG records. That is, they may be mixed with DBG records, and all data generated goes into the DST as if they were DBG records.

The purpose of separating the information contained in DBG records is to allow inclusion of a DST containing only traceback data when no debugging is requested at link time. If the production of traceback information is disabled at link time, these records are ignored. ♦

Index

A

ABS program section name
  definition, B–16
.ADDRESS directive
  count in image map file, 5–4
  image activator's processing of, 3–24
  linker's processing of, 3–24
Address ranges
  aligning on page boundaries, 3–23
Address-related commands
  Alpha object language, B–36
Alias names
  specifying for universal symbols, 4–11, LINK–67
Alignment
  See Data alignment
Allocating
  virtual memory for images, 3–17
Alpha
  images
    creating, 1–20
      specifying in link operations, LINK–5
ALPHA$LIBRARY logical name, 1–9, 1–20
ALPHA$LOADABLE_IMAGES logical name, 1–21, 2–19, LINK–35
Alpha object language, B–1
  address-related commands, B–36
  conditional store commands, B–36, B–40
  control commands, B–35
  data structures, B–2
  debugger information record format, B–46
  end-of-module record format, B–43
  instruction-related commands, B–38
  operator commands, B–33
  stack commands, B–29
  store commands, B–30
  text information and relocation records, B–27
  traceback information record format, B–46
/ALPHA qualifier, 1–21, LINK–5
ANALYZE/IMAGE command
  examining image files, 1–12
  listing the image sections in an image, 3–20
ANALYZE/OBJECT command
  examining object modules, 1–6
Architectures
  linker options, 1–20
ASSIGN command
  defining the LNK$LIBRARY logical name, LINK–41
/ATTRIBUTES qualifier, LINK–50

B

BASE= option, LINK–45
Base addresses
  defaults for images, LINK–45
  specifying using the CLUSTER= option, LINK–49
  system image, LINK–39
Based images
  creation of, LINK–45
  memory allocation for, LINK–45
  processing of, 3–9
Based shareable images
  creating, 4–8
/BPAGE qualifier, LINK–6
Brief image map files, LINK–7
/BRIEF qualifier, LINK–7
BSR instruction
  substituting for the JSR instruction, 1–15, LINK–26
BSS program section name
  definition, B–16

C

Case sensitivity
  in options file, LINK–47
CASE_SENSITIVE= option, LINK–47
CLUSTER= option, LINK–49
  basing images, LINK–45
  controlling image section creation, 3–22
  controlling the order of input file processing, 2–18
  fixing position of transfer vector in image, 4–7
Clustering of input files
  controlling image section creation, 3–22
  effect on image creation, 3–9
  in a based image, LINK–45
  processing based images, 3–9
  using the COLLECT= option, LINK–50

Clusters
See Clustering of input files
$CODE$ program section name
definition, B–16
COLLECT= option, LINK–50
controlling image section creation, 3–22
controlling the order of input file processing, 2–18
Conditional store commands
Alpha object language, B–36, B–40
/CONTIGUOUS qualifier, LINK–8
Control commands
Alpha object language, B–35
VAX object language, A–30
$CRMPSC system service
See SYS$CRMPSC system service
Cross-architecture
linking, 1–20, 1–21
logical names, 1–20
Cross-reference section of image map files, LINK–9
format, 5–8
/CROSS_REFERENCE qualifier, LINK–9

D

$DATA$ program section name
definition, B–16
Data alignment
specifying alignment of program sections, 3–4
Debugger information records
Alpha object language, B–46
VAX object language, A–33
Debugging
Alpha object language records, B–46
enabling at link time, LINK–10
including debugger information in an image, 3–24
including global symbols in a symbol table file, LINK–66
including traceback information, LINK–40
object language record format, B–46
specifying a user-written debugger, LINK–10
VAX object language records, A–33
/DEBUG qualifier, LINK–10
Debug symbol file
See also /DSF qualifier
creating, LINK–13
Debug symbol files
creating, 1–15
DEFINE command
defining the LNK$LIBRARY logical name, LINK–41
Demand-zero compression, 3–25
controlling, 3–25, LINK–52
Demand-zero image sections
creating, 3–25, LINK–11, LINK–52
definition, LINK–52
disabling creation of, LINK–11
maximum number of, LINK–59
/DEMAND_ZERO qualifier, LINK–11
DSF files
See Debug symbol files
/DSF qualifier, LINK–13
DZRO_MIN= option, LINK–52
controlling demand-zero compression, 3–25

E

End-of-module record format
Alpha object language, B–43
VAX object language, A–31
Executable images
creating, 1–13
definition, 1–1
specifying a base address, LINK–45
/EXECUTABLE qualifier, LINK–14
Executive images
linking against, LINK–35

F

Fix-ups
creation of, 3–24
definition, 1–3
Full image map files
creating, LINK–15
/FULL qualifier, LINK–15

G

GBL program section attribute
effect on image creation, 3–9
implicit setting by linker, 2–18
GHRs (granularity hint regions)
improving the performance of shareable images, 1–16
Global sections
linker-assigned names of, 5–6
Global symbol directories
See GSDs
Global symbols
declaring as universal symbols, 4–1
defining with the SYMBOL= option, LINK–65
definition, 2–1
determining the address of, 3–18
implemented as overlaid program sections, 2–1
including in a symbol table file, LINK–66
object module reference specification record, A–11
object module specification format, B–23
version mask, B–27
object module specification subrecord, B–16

Global symbols
object module specification subrecord (cont'd)
version mask, B–27
strong reference to, 2–20
weak reference to, 2–20
Global symbol tables
See GSTs
Granularity hint regions
See GHRs
GSDs (global symbol directories)
entity-ident-consistency-check subrecord, A–16, B–24
entry-point-and-mask-definition subrecord, A–12
entry-point-definition-with-version-mask subrecord, A–21
entry-point-definition-with-word-psect subrecord, A–16
environment-definition/reference subrecord, A–18
in object modules, 2–1
module-local entry-point definition subrecord, A–19
module-local procedure definition subrecord, A–20
module-local symbol definition subrecord, A–19
module-local symbol reference subrecord, A–19
procedure-definition-with-version-mask subrecord, A–21
procedure-definition-with-word-psect subrecord, A–16
procedure-with-formal-argument-definition subrecord, A–13
program-section-definition-in-shareable-image subrecord, A–20, B–14
program section definition subrecords, B–11
record format in object module, A–6, B–10
symbol definition subrecord format, A–10, B–17
symbol-definition-with-version-mask subrecord, A–21, B–27
symbol-definition-with-word-psect subrecord, A–16
symbol reference subrecord, B–23
symbol specification subrecords, A–10, B–16
text information and relocation records, A–21
universal symbol definition record, B–20
vectored-entry-point-definition subrecord, A–20
vectored-procedure-definition subrecord, A–21
vectored-symbol-definition subrecord, A–20, B–26
GSMATCH= option, LINK–54
/GST qualifier, LINK–16
creating run-time kits with, 4–10
GSTs (global symbol tables)
controlling contents of, 4–10, LINK–16, LINK–68
GSTs (global symbol tables) (cont'd)
creating, 4–8
definition, 2–1
deleting entries in, 4–10

H

/HEADER qualifier, LINK–17
Hints
calculating JSR hints, B–42

I

IDENTIFICATION= option, LINK–57
Image activator
description, 1–4
GSMATCH processing, LINK–55
performing image optimizations, 4–11
shareable image ID processing, LINK–55
Image I/O segments, LINK–58
IMAGELIB.OLB file, 2–13, LINK–15
included in image map files, LINK–9
order of processing, 2–19
processing by linker, LINK–37, LINK–38
Image map files
brief, 5–2, LINK–7
components of, 5–2
creating, 1–14, 5–1, LINK–21
default, 5–2
full, 5–2, LINK–15
image section synopsis, 3–20, 5–4
image synopsis, 5–10
linker's writing of, 3–24
link run statistics, 5–11
listing symbols by name, 5–8
listing symbols by value, 5–9
naming, LINK–21
object module synopsis, 5–3
verifying order of processing, 2–18
program section synopsis, 5–6
symbol characterization codes, 5–9
symbol cross-reference section, LINK–9
symbols cross-reference section, 5–8
Images
See also Executable images; Shareable images
activation of, LINK–54
allocating memory for, 3–17
base address of, in map, 5–10
building for Alpha and VAX architectures, 1–21
creating an image map file, LINK–15, LINK–21
creating resident images, LINK–27
default page size, LINK–6
enhancing performance of installed images, 1–16
I/O segment, LINK–58
initializing, 1–2, 3–23
length of, in map, 5–10

Images (cont'd)
naming, LINK–14
operating with translated VAX images, LINK–22
optimizing performance, 1–3, 1–15, LINK–26
Alpha object language commands, B–38
calculating JSR hints, B–42
reducing the size of, 3–25, LINK–11
resident, 1–16
specifying Alpha in link operations, LINK–5
specifying identification character string, LINK–57
specifying stack size, LINK–64
specifying value of name field in image header, LINK–60
specifying VAX in link operations, LINK–43
storing in contiguous disk blocks, LINK–8
synopsis of in image map file, 5–10
using ANALYZE/IMAGE command to examine, 1–12
Image sections
attributes, 3–18
demand-zero attribute, 3–18
determined by program section attributes, 3–12
binding address to, LINK–28
controlling creation of, 3–21
creating, 3–9
creating from program sections, 3–10
demand-zero, LINK–11, LINK–52
determining the address of, 3–18
determining the program sections in, 3–16
examining with the ANALYZE/IMAGE utility, 3–20
filling with binary information, 3–23
fix-up, 3–24
listed in map file, 3–15, 3–20, 5–4
maximum number of, LINK–59
order, in cluster, 3–13
protection of, LINK–61
specifying the base address of, LINK–49
type designations, 3–19
using CLUSTER= option to control, 3–22
IMGIOCNT system parameter
overriding at link time, LINK–58
/INCLUDE qualifier, LINK–18
effect on symbol resolution processing, 2–13
specified with the /LIBRARY qualifier, 2–13
specifying libraries as linker input, 1–9
/INFORMATIONALS qualifier, LINK–19
Initializing images, 1–2, 3–23
Input files
types of, 1–4
Installing images
enhancing performance of, 1–16
link-time considerations, 1–16
resident images, LINK–27
Installing shareable images, 1–8
creating resident shareable images, 1–16
enhancing performance of, 4–11
Instruction-related commands
Alpha object language, B–38
IOSEGMENT= option, LINK–58
ISD_MAX= option, LINK–59
controlling demand-zero compression, 3–25

J

Jacket routines
link-time considerations, LINK–22
JMP instruction
in transfer vectors, 4–6
JSR instruction
calculating hints for, 1–16, LINK–26, B–42
replacing with the BSR instruction, 1–15, LINK–26

K

Kitting shareable images, LINK–16
controlling universal symbol declarations, 4–10

L

Library files
containing object modules, 1–8
containing shareable images, 1–8
creating, 1–8
default system libraries
order of processing, 2–19
processing, 2–13, LINK–37, LINK–38
examining contents of, 1–8
name table, 2–21
processing during symbol resolution, 2–11
selective processing of, 2–15
specifying as linker input, 1–9, LINK–18, LINK–20
specifying default user libraries, 2–13, LINK–41
types of libraries accepted as linker input, 1–8
/LIBRARY qualifier, LINK–20
effect on symbol resolution processing, 2–12
specified with the /INCLUDE qualifier, 2–13
specifying libraries as linker input, 1–9
$LINK$ program section name
definition, B–16
LINK command
clustering of input files, 2–16, 2–18, LINK–49
in command procedure, 1–12
invoking, LINK–3
qualifiers, 1–17
specifying input files, LINK–3
specifying library files, LINK–20

Linker utility (linker)
how to invoke, 1–3
Link operations
obtaining statistical information about, 5–11
$LITERAL$ program section name
definition, B–16
LNK$LIBRARY logical name, LINK–42
processing of, 2–14
LNK$OPEN_LIB logical name
open systems library processing, 2–14

M

Major ID
specifying value of, LINK–54
Map files
See Image map files
Mapping virtual memory
using SOLITARY program section attribute, 3–23
/MAP qualifier, LINK–21
Memory allocation
absolute program section, 3–4
for based images, LINK–45
for images, 1–2
information about, in map, 5–10
relocatable program section, 3–4
Memory resident databases
implementing as shareable image, 1–7
$MGBLSC system service
See SYS$MGBLSC system service
Minor ID
specifying value of, LINK–54

N

NAME= option, LINK–60
Naming images, LINK–14
Naming shareable images, LINK–31
NAS (Network Application Support)
open systems library processing, 2–14
/NATIVE_ONLY qualifier, LINK–22
Network Application Support
See NAS
NOMOD program section attribute
resolving conflicts, 3–25

O

Object languages
Alpha object language, B–1
VAX object language, A–1
Object modules
as linker input file, 1–5
end-of-module records, A–31
end-of-module-with-psect record, A–32
Object modules (cont'd)
entity-ident-consistency-check subrecord, A–16, B–24
entry-point-definition-with-version-mask subrecord, A–21
entry-point-definition-with-word-psect subrecord, A–16
entry-point-symbol-and-mask-definition subrecord, A–12
environment-definition/reference subrecord, A–18
global symbol definition subrecord, B–17
global symbol directory records, A–6, B–10
global symbol reference subrecord, B–23
global symbol specification subrecord, A–10, B–16
including in a link operation from a library, LINK–18, LINK–20
in libraries, 1–8
in symbol resolution processing, 2–6
language processor name header record, A–5, B–8
listed in map file, 5–3
main module header record, A–4, B–6
module header records, A–3, B–5
module-local entry-point definition subrecord, A–19
module-local procedure definition subrecord, A–20
module-local symbol definition subrecord, A–19
module-local symbol reference subrecord, A–19
normal program section definition subrecord format, B–11
order of records in, A–1, B–2
procedure-definition-with-version-mask subrecord, A–21
procedure-definition-with-word-psect subrecord, A–16
procedure-with-formal-argument-definition subrecord, A–13
program-section-definition-in-shareable-image subrecord, A–20, B–14
program section definition record format, A–8, B–11
in shareable image, B–14
source files header record, A–5, B–8
symbol definition subrecord format, A–10
symbol-definition-with-version-mask subrecord, A–21, B–27
symbol-definition-with-word-psect subrecord, A–16
symbol reference GSD subrecord, A–11
text information and relocation records, A–21
title text header record, A–6, B–9
universal symbol definition subrecord, B–20
using ANALYZE/OBJECT utility to examine, 1–6
vectored-entry-point-definition subrecord, A–20

Object modules (cont'd)
vectored-procedure-definition subrecord, A–21
vectored-symbol-definition subrecord, A–20, B–26
vectored symbol specification subrecords, B–26
Open systems library
support for NAS in linker, 2–14
OpenVMS Alpha System-Code Debugger
creating debug symbol file for, LINK–13
Operator commands
Alpha object language, B–33
VAX object language, A–28
Optimizing images
Alpha object language commands, B–38
calcuating JSR hints, B–42
Optimizing performance, 1–3, 1–15
Options
case sensitivity of option arguments, LINK–47
summary of linker options, 1–19
use of radix operators, LINK–44
Options files
as linker input, 1–10
case sensitivity of option arguments, LINK–47
creating, 1–11
specifying in a link operation, 1–11, LINK–23
specifying on the command line, 1–11
/OPTIONS qualifier, LINK–23

P

/P0IMAGE qualifier, LINK–24
Page faults
specifying page fault clusters, LINK–49
Page sizes
specifying in link operations, LINK–6
Performance
optimizing, 1–3, 1–15
PFCDEFAULT system parameter
overriding default value, LINK–49
PLV (privileged library vector), 4–11
Privileged library vector
See PLV
Privileged shareable images
declaring universal symbols in, 4–11
protecting, LINK–25
protecting image sections in, LINK–61
Procedure signature blocks
See PSBs
Program sections
absolute, 3–4
alignment of, 3–4
as universal symbols, 4–4
attributes, 3–3
conflicting, 3–25
determining image section attributes, 3–12
effects on image section creation, 3–11
modifying, 3–22
Program sections (cont'd)
collecting into image sections, 3–10, 3–22, LINK–50
concatenated, 3–17
creation of, 3–3
declaring as universal symbols, 4–9
determining image section location, 3–16
determining the address of, 3–18
implicit setting of GBL attribute by linker, 2–18
in ANALYZE/OBJECT listing, 3–6
in shareable images
object module format, B–14
isolating in an image section, 3–23
listed in map file, 3–16, 5–6
modifying program section attributes, 3–22
NOMOD attribute
resolving conflicts, 3–25
object module definition format, A–8, B–11
overlaid, 2–1, 3–17, 4–4
relocatable, 3–4
SHR attribute, 4–4
significant attributes of, 3–13
SOLITARY attribute, 3–23
sorting by attributes, 3–11
specifying values of attributes, LINK–62
standard program section names (Alpha systems only), B–16
PROTECT= option, LINK–61
Protecting image sections
using the PROTECT= option, LINK–61
Protecting shareable images, LINK–25
/PROTECT qualifier, LINK–25
PSBs (procedure signature blocks), LINK–22
PSECT_ATTR= option, LINK–62
controlling image section creation, 3–22

R

Radix operators
used with linker options, LINK–44
$READONLY$ program section name
definition, B–16
Relocating symbols
definition, 1–2
/REPLACE qualifier, LINK–26
effect on debugging, 1–16
Resident images
creating, 1–16, LINK–27
effect on data image sections, LINK–27
effect on image map file, 5–5
link-time considerations, 1–16
RMS_RELATED_CONTEXT option, LINK–63
Run-time kitting
creating shareable images for, LINK–16

S

SDA (System Dump Analyzer utility)
  using with symbol table files, 1–10, LINK–33
/SECTION_BINDING qualifier, LINK–27
  creating resident shareable images, 1–16
  improving the performance of installed shareable images, 4–11
Selective searches
  specifying the /SELECTIVE_SEARCH qualifier, LINK–29
/SELECTIVE_SEARCH qualifier, 2–15
Shareable images
  activating, LINK–54
  as linker input files, 1–6
  benefits of, 1–6
  creating, 1–13, 4–1, LINK–31
  creating a based shareable image, 4–8
  creating a run-time kit, 4–10, LINK–16
  creating resident shareable images, 1–16
  debugging, LINK–10
  declaring alias names for universal symbols, 4–11
  declaring universal symbols on Alpha systems, LINK–67
  declaring universal symbols on VAX systems, 4–2
  default base address, LINK–45
  definition, 1–1
  enhancing performance of, 1–16
  ensuring upward compatibility, LINK–55
    deleting universal symbols, 4–10
    guidelines, 4–6
    on Alpha systems, 4–9
    on VAX systems, 4–4
  implicit processing of, 2–11
  improving the performance of, 4–11
  in libraries, 1–8
    default location, 1–9
    specifying as linker input, LINK–18, LINK–20
  installing, 1–7, 1–8
  naming, LINK–31
  privileged, 4–11
  protecting, LINK–25, LINK–61
  resident images
    effect on image map file, 5–5
  specifying as linker input, 1–7, LINK–31
    in libraries, LINK–20
  specifying identification numbers, LINK–54
  symbol vector program section, 3–3
  use of GSMATCH= option, LINK–55
/SHAREABLE qualifier, LINK–31
  creating shareable images, 4–1
STACK= option, LINK–64
Stack commands
  Alpha object language, B–29
  VAX object language, A–23
STARLET.OLB file, 2–13, LINK–15
  included in image map files, LINK–9
  order of processing, 2–19
  processing by linker, LINK–37
Store commands
  Alpha object language, B–30
  VAX object language, A–25
Strong symbol
  definition, 2–20
  reference, 2–20
SYMBOL= option, LINK–65
Symbol resolution processing
  definition, 1–2
  description, 2–2
  handling undefined symbols, 2–5
  of object modules, 2–6
  ordering of input files, 2–16
  overview, 2–1
  processing default libraries, 2–13
  processing files selectively, 2–15
  specifying selective processing, LINK–29
  types of input files included, 2–5
Symbols
  See also Global symbols; Symbol resolution processing; Universal symbols
  cross-referenced in image map file, 5–8
  declaring universal symbols on Alpha systems, 4–8
  declaring universal symbols on VAX systems, 4–2
  global, 2–1
    determining the address of, 3–18
  implemented as overlaid program sections, 2–1
  listed by name in image map file, 5–8
  listed by value in image map file, 5–9
  local, 2–1
  strong, 2–1, 2–20
  strong definition of, 2–21
  symbol resolution processing, 2–2
  types of, 2–1
  universal, 2–1
  vectored symbol definition subrecords, B–26
  weak, 2–1, 2–20
  weak definition of, 2–21
Symbol table files
  as linker input files, 1–10
  controlling the contents of, LINK–66
  creating, 1–14, LINK–33
  naming, LINK–33
  using with SDA utility, 1–10, LINK–33
Symbol vectors
  creating, 4–8, LINK–67
  declaring alias names for universal symbols, 4–11

Symbol vectors (cont'd)
ensuring upward compatibility on Alpha systems, 4–9
guidelines, 4–9
in program section, 3–3
run-time flow of control, 4–9
SYMBOL_TABLE= option, LINK–66
/SYMBOL_TABLE qualifier, LINK–33
SYMBOL_VECTOR= option, LINK–67
declaring universal symbols, 4–8
$SYMVECT program section, 3–3
SYS$BASE_IMAGE.EXE file
linking against, 2–19
order of processing, 2–19, LINK–35
SYS$CRMPSC system service
using SOLITARY program section attribute with, 3–23
SYS$LIBRARY logical name, 1–9, 1–21, 2–13
SYS$MGBLSC system service
using SOLITARY program section attribute with, 3–23
SYS$PUBLIC_VECTORS.EXE file
order of processing, 2–19, LINK–35
processing, 2–14, LINK–37
SYS.STB file
linking against, 2–19
/SYSEXE qualifier, LINK–35
linking against the executive image, 2–19
/SYSLIB qualifier, LINK–37
effect on default library processing, 2–19
/SYSSHR qualifier, LINK–38
effect on default library processing, 2–19
System Dump Analyzer utility
See SDA
System images
creating, 1–13, LINK–39
creating a header for, LINK–17
default base address, LINK–45
definition, 1–1
naming, LINK–39
System library files
including in image map files, LINK–9, LINK–15
linker processing of, 2–13, LINK–37
order of processing, 2–19
open systems support library, 2–14
/SYSTEM qualifier, LINK–39
System services
resolving references to, 2–14, 2–19, LINK–35, LINK–37
user-written, 4–11

T

Text information and relocation (TIR) records
Alpha object language, B–27
VAX object language, A–21
Traceback facility
link-time considerations, LINK–40
Traceback information records
Alpha object language, B–46
VAX object language, A–33
/TRACEBACK qualifier, LINK–40
Transfer vectors
comparison to UNIVERSAL= option, 4–4
creating, 4–5
ensuring upward compatibility, 4–6
example program, 4–6
including data in, 4–5
including in a link operation, 4–7
providing upward compatibility, 4–4

U

UNIVERSAL= option, LINK–69
comparison to transfer vectors, 4–4
declaring universal symbols, 4–2
specifying, 4–8
Universal alias names
specifying, 4–11, LINK–67
Universal symbols
declaring alias names for, 4–11
declaring on Alpha systems, 4–8
declaring on VAX systems, 4–2, LINK–69
definition, 2–1
for symbols that represent data, 4–5
for symbols that represent procedures, 4–4
object module definition format, B–20
User library files
limiting scope of linker processing, LINK–41
linker's search of, LINK–42
specifying, 2–13, LINK–41
/USERLIBRARY qualifier, LINK–41
User-written system services
implemented as privileged shareable images, 4–11

V

VAX
images
creating, 1–20
specifying in link operations, LINK–43
VAX$LIBRARY logical name, 1–20
VAX object language, A–1
control commands, A–30
debugger information record format, A–33
operator commands, A–28
stack commands, A–23

VAX object language (cont'd)
store commands, A–25
traceback information record format, A–33
/VAX qualifier, 1–21, LINK–43
Virtual memory
allocating for images, 1–2, 3–17

W

Weak symbol
definition, 2–20
reference, 2–20