VAX/VMS Linker Reference Manual

Order Number: AA–Z420A–TE

September 1984

This manual describes how the VAX/VMS Linker works and how to use it.

Revision/Update Information:	This is a new manual.
Software Version:	VAX/VMS Version 4.0

digital equipment corporation
maynard, massachusetts

September 1984

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a registered trademark of the American Mathematical Society.

LINKER Contents

EXAMPLES

FIGURES

TABLES

Preface

Intended Audience

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

Structure of This Document

The Format Section is an overview of the linker and is intended as a quick reference guide. The format summary contains the DCL command that invokes the linker, listing all command (and/or positional) qualifiers and parameters. The usage summary describes how to invoke and exit from the the linker, how to direct output, and any restrictions you should be aware of.

The Description Section explains how to use the linker. Section 1 presents a conceptual overview of the linker. Section 2 explains how to use an options file. Section 3 discusses shareable images. Section 4 discusses image maps. Section 5 discusses in detail the operations performed by the linker in creating an image. Section 6 describes the VAX object language.

The Qualifier Section describes each DCL command qualifier. Qualifiers appear in alphabetical order.

The Examples Section contains examples of common operations that you perform with the linker.

Associated Documents

For information on including the debugger in the linking operation, and on debugging in general, see the *VAX/VMS Utilities Reference Volume*.

Conventions Used in This Document

Convention	Meaning
RET	A symbol with a one- to three-character abbreviation indicates that you press a key on the terminal, for example, RET .
CTRL/x	The phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/C, CTRL/Y, CTRL/O.
$ SHOW TIME 05-JUN-1985 11:55:22	Command examples show all output lines or prompting characters that the system prints or displays in black letters. All user-entered commands are shown in red letters.

Convention	Meaning
$ TYPE MYFILE.DAT . . .	Vertical series of periods, or ellipsis, mean either that not all the data that the system would display in response to the particular command is shown or that not all the data a user would enter is shown.
file-spec,...	Horizontal ellipsis indicates that additional parameters, values, or information can be entered.
[logical-name]	Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.

New and Changed Features

The following technical changes have been made to the VAX/VMS Linker:

- Link options object records (LNK) are now processed.
- Two new linker options are IDENTIFICATION and NAME. They allow the image name and identification field to be set explicitly in the image header.
- The NEVER parameter to the GSMATCH linker option is no longer allowed.
- The COPY parameter on the positional qualifier /SHAREABLE is no longer allowed. NOCOPY is now mandatory.
- The linker option UNIVERSAL = * is no longer allowed.
- Uninitialized Copy-on-Reference image sections are demand zero, even if they are less than DZRO_MIN.

LINKER

Before a source-language program can be run on VAX/VMS, it must first be assembled or compiled, and then it must be linked. The VAX MACRO assembler and the various VAX language compilers (hereafter referred to as language processors) translate user-written programs into object modules. The VAX/VMS Linker (or, simply, the linker) binds these object modules, together with any other necessary information, into an image, which can be executed by VAX/VMS.

FORMAT

LINK */command-qualifier [,...] file-spec/file-qualifier*

Command Qualifiers	**Defaults**
/BRIEF	*None.*
/[NO]CONTIGUOUS	*/NOCONTIGUOUS*
/[NO]CROSS_REFERENCE	*/NOCROSS_REFERENCE*
/[NO]DEBUG[=file-spec]	*/NODEBUG*
/[NO]EXECUTABLE[file-spec]	*/EXECUTABLE*
/FULL	*None.*
/HEADER	*None.*
/[NO]MAP[=file-spec]	*/NOMAP*
/P0IMAGE	*None.*
/PROTECT	*None.*
/[NO]SHAREABLE[=file-spec]	*/NOSHAREABLE*
/[NO]SYMBOL_TABLE[=file-spec]	*/NOSYMBOL_TABLE*
/[NO]SYSLIB	*/SYSLIB*
/[NO]SYSSHR	*/SYSSHR*
/[NO]SYSTEM[=base-address]	*/NOSYSTEM*
/[NO]TRACEBACK	*/TRACEBACK*
/[NO]USERLIBRARY[=(table[,...])]	*/TRACEBACK*

Positional Qualifiers	**Defaults**
/INCLUDE=(module-name[,...])	*None.*
/LIBRARY	*None.*
/OPTIONS	*None.*
/SELECTIVE_SEARCH	*None.*

Command Parameter

file-spec

Specifies one or more input files. The input files can be object modules to be linked, libraries to be searched for external references or from which specific modules are to be included, shareable images to be included in the output image, or option files to be read by the linker. If you specify multiple input files, separate the file specifications with commas, or plus signs. In either case, the linker creates a single image file.

usage summary

Invoking
Invoke the VAX/VMS Linker by typing the LINK command and one or more input file names at the DCL prompt. You may also include the LINK command in a command procedure.

Exiting
Exit the linker by letting it run to completion.

Directing Output
You can direct output to different types of output files with certain qualifiers, such as the /EXECUTABLE, /SHAREABLE, /MAP, and /SYMBOL_TABLE positional qualifiers. Otherwise, all Linker output and messages go to the current SYS$OUTPUT device.

The names assigned to the image file, the map file, and other output files depend on the first input file name, unless you specify differently. You can specify a different output filename by specifying a name in an /EXECUTABLE, /SHAREABLE, /MAP, or /SYMBOL_TABLE qualifier or by entering one of these qualifiers after a file specification. For more information refer to the Positional Qualifier Section.

Privileges/Restrictions
None.

DESCRIPTION

The Description Section explains how to use the linker. Section 1 presents a conceptual overview of the linker. Section 2 explains how to use an options file. Section 3 discusses shareable images. Section 4 discusses image maps. Section 5 discusses in detail the operations performed by the linker in creating an image. Section 6 describes the VAX object language.

1 Conceptual Overview

This section provides a conceptual overview of the linker by describing the reasons for a linker, operations performed by the linker, and the forms of linker input and output. This information is presented in summary form to benefit readers unfamiliar with the subject. Detailed information on these topics is presented in subsequent sections.

1.1 Reasons for a Linker

Some computer systems do not have linkers as VAX/VMS does. Instead, the language processor performs more of the work needed to resolve symbolic references, and another software component completes the task while loading the program in memory. However, having a linker is advantageous in the following ways:

- A linker simplifies the job of each language processor. For example, the logic needed to resolve symbolic references need not be duplicated in each language processor.
- A linker simplifies modular programming.
- Object modules produced by different language compilers can be combined in a single executable image.

Modular programming is the practice of breaking down a complex task into smaller and simpler subtasks and then writing programs for each of the subtasks. Some advantages of modular programming follow:

- A program consisting of modules of properly designed scope (typically a page or two of coding) is easier to design, write, and test than one that is not modular.
- Breaking down a program into modules makes it easy for more than one programmer to work on the same program.
- Different modules of the same program can be written in different languages for reasons of both programmer preference and the suitability of a particular language to solve a particular task.

Thus, in the VAX/VMS programming environment, individual modules may be separately written and compiled, and then linked together into a single executable image.

1.2 Input to the Linker

This section describes the input that may be included in a linking operation. Any unit of input must be a file. The following kinds of files are acceptable to the linker:

- Object file, which contains one or more object modules
- Shareable image file, which contains one shareable image and one symbol table
- Symbol table file, which contains one symbol table in the form of an object module
- Library file, which contains either one or more object modules or one or more shareable images
- Options file, which may contain file specifications for any of the above kinds of files and/or one or more link options, which are directives to the linker

Aside from link options, which are simply instructions that direct the linker in the linking operation, all the above files contain either object modules or shareable images. These are the two basic forms of input processed by the linker, and they are discussed in depth in Section 1.2.

1.2.1 Object File

When a language processor translates a source language program, it produces as output a file that contains one or more object modules. This file has the default file type OBJ, and it is the primary form of linker input.

Each object module contains records that define its content and memory requirements to the linker. At least one object file must be specified in any linking operation. It may be specified in the command string or in an options file.

The linker processes the entire contents of an OBJ file, that is, every object module in the file. The linker cannot selectively process some modules but not others in the same file. If selective processing of object modules is required, the object modules must be in an object module library file.

1.2.2 Shareable Image File

A shareable image file is the product of a previous linking operation. It is an image that is part of a complete program and is therefore not directly executable; that is, it is not intended to be directly executed by means of the DCL command RUN. To execute, a shareable image must be included as input in a linking operation that produces an executable image. Then, when that executable image is run, the shareable image can execute.

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 interpreted outside the shareable image.

In processing a shareable image file, the linker needs only to read the image header and to process the symbol table.

1.2.3 Symbol Table File

Like a shareable image file, a symbol table file is the product of a previous linking operation. The linker creates a symbol table file when the /SYMBOL_TABLE qualifier is specified in the LINK command.

The contents of a symbol table file vary depending on the kind of image being produced by the linking operation. If the symbol table file is an executable or system image, it contains the names and values of every global symbol in the image. If the symbol table file is a shareable image, it contains the names and values of every universal symbol in the image.

When a symbol table file is specified as input in another linking operation, the linker uses the symbols in that symbol table to resolve undefined symbols in other object modules. It does this by inserting the symbol's value (as listed in the symbol table) in place of the symbol name in the object module where the symbol was undefined.

A major use for specifying a symbol table file as input in a linking operation is to make global symbols in a system image available to a number of other images.

Note, however, that a symbol table file produced in a linking operation that created a shareable image is not adequate input, in a subsequent linking operation, to allow the linker to resolve references to universal symbols in that shareable image. The shareable image itself must be specified as input because the linker requires the value of the symbol (as specified in the symbol table) and other information such as the memory requirements of the shareable image (contained in the image header).

1.2.4 Library File

There are two kinds of library files:

- An object module library file. It contains one or more object modules, the names of the included object module(s), and a symbol table with the names of each global symbol in the library and the name of the module in which they are defined.
- A shareable image library file. It contains the names of the included shareable image(s), a symbol table with the names of each universal symbol in the library and the name of the shareable image in which they are defined.

To simplify the discussion, the term *module* will be used to refer to both an object module in an object library and a shareable image in a shareable image library. Further, the term *symbol* will be used to refer to both a global symbol in an object library or to a universal symbol in a shareable image library.

Libraries contain modules that are useful to many user programs. For example, a user program may call a routine in a library to perform I/O or to calculate a square root. When the user program calls a library routine, the linker locates the routine and includes it in the linking operation.

Using the DCL command LIBRARY, a programmer may create a library and populate it with modules of interest to a number of users. Such a library is called a user library. To make a user library known to the linker, the programmer must do either of the following:

- Specify the library in the LINK command using the /LIBRARY file qualifier.
- Define the library as a default library for the linker to search in the event that it cannot resolve symbolic references using the input modules alone. In this case, the library need not be specified as input in the LINK command. See the description of the /USERLIBRARY command qualifier in the Qualifier Section for information on how to define a user default library.

In addition, VAX/VMS maintains two system libraries that the linker searches by default in the event that unresolved symbolic references remain after all input modules and user libraries (if any) have been processed. These are the system default shareable image library IMAGELIB.OLB and the system default object library STARLET.OLB. Both libraries may be found in the device and directory specified by the system logical name SYS$LIBRARY, that is, their full names are SYS$LIBRARY:IMAGELIB.OLB and SYS$LIBRARY:STARLET.OLB, respectively. These libraries contain system symbol definitions, such as the addresses of entry points for VAX/VMS system services, as well as various routines that perform such functions as I/O for high-level language programs.

Individual modules in a library may be included in the linking operation in either of the following ways:

- Explicitly, by name, using the /INCLUDE positional qualifier in the LINK command.
- Implicitly, when, in searching the library, the linker discovers that the module contains a required symbol definition.

Additional information on how to specify libraries to the linker and how to control the linker's use of libraries may be found in the Qualifier Section, specifically, the /SYSLIB, /SYSSHR, /USERLIBRARY, /INCLUDE, and /LIBRARY qualifiers.

See also Sections 1.4.1.3, 5.3.3.1, and 5.3.3.2 for information on the algorithm used by the linker to process libraries.

1.2.5 Options File

An options file is a file that is specified as input to the linker by means of the /OPTIONS positional qualifier. By using an options file, a programmer may exercise considerable control over the linking operation.

Since Section 2 discusses options files in depth, the scope of the discussion in this section is limited to the contents and uses of options files.

An options file may contain the following types of information:

- One or more input file specifications and associated file qualifiers. Any of the various files mentioned above may be included. Note in particular that a shareable image file to be used as input to the linking operation may be specified only in an options file, never in the command string.

- Special instructions to the linker (called link options) that may not be specified by means of the DCL command language.

An options file is useful (or necessary) in the following ways:

- To specify lengthy and cumbersome command input that must be frequently included in a linking operation.
- To specify command input that exceeds the limit allowed by the DCL command interpreter.
- To include in the linking operation a shareable image that is not contained in a shareable image library.

1.3 Output of the Linker

This section describes the three types of image that the linker creates, as well as the optional image map and symbol table.

1.3.1 Executable Image

An executable image is an image that is executed by the DCL command RUN. As the goal of program development is a program that can be executed on the computer system, the executable image is the end product of program development and the most common type of image created by the linker.

The linker creates an executable image when the /EXECUTABLE command qualifier is specified with the LINK command or, by default, when neither /SHAREABLE nor /SYSTEM is specified.

An executable image cannot be linked with other images. However, the object modules that make up an executable image can be linked in different combinations or with different link options to produce a different executable image.

1.3.2 Shareable Image

The linker creates a shareable image when the /SHAREABLE command qualifier is specified with the LINK command.

Shareable images are useful in the following ways:

- To provide a means of sharing a single physical copy of a set of procedures and/or data among more than one application program.
- To facilitate the linking of very large applications (say, hundreds of modules) by breaking down the whole into manageable segments.
- To allow the modification of one section of a large program without having to relink the entire program.

1.3.3 System Image

A system image is an image that does not run under the control of the VAX/VMS operating system. It is intended for stand-alone operation on the VAX hardware. The kernel of VAX/VMS, SYS$SYSTEM:SYS.EXE, is a system image.

The linker creates a system image when the /SYSTEM command qualifier is specified with the LINK command.

1.3.4 Image Map

In interactive mode, the linker generates an image map only when the /MAP qualifier is specified in the LINK command. In batch mode, the linker generates an image map by default. The map is written to a map file during the second pass (Pass 2) of the linking operation.

The content of an image map varies depending on which additional qualifiers are specified in the LINK command. The following sections may appear in an image map:

- Object module synopsis
- Module relocatable reference synopsis
- Image section synopsis
- Program section synopsis
- Symbols by name
- Symbols by value
- Image synopsis
- Link run statistics

1.3.5 Symbol Table File

The linker generates a symbol table file only when the /SYMBOL_TABLE qualifier is specified in the LINK command.

A symbol table file may be included as input in a subsequent linking operation.

See Section 2.3 for information on the contents of the symbol table file.

1.4 Linker Functions

This section describes the three major tasks performed by the linker in the process of image creation: resolution of symbolic references, allocation of virtual memory, and image initialization.

1.4.1 Resolution of Symbolic References

This section explains the difference between a symbol definition and a symbol reference, and explains how the linker resolves a symbolic reference. Additional subsections discuss the three types of symbols and the differences between strong and weak global symbols.

A symbol is a name associated with a program location or with a data element. The definition of a symbol is the statement that makes that association explicit. Where a symbol is used as a label to mark a program location (for example, the start of a routine), the definition of the symbol is an address. Where a symbol is used to represent a data element, the definition of the symbol is a value.

For example, the following VAX MACRO statement defines the symbol ROUTINEA to be the location of the specified instruction:

```
ROUTINEA::  MOVL #FIELDA,FIELDB
```

The following VAX MACRO statement defines the symbol FIELDA to be the data value 100:

```
FIELDA == 100
```

A symbolic reference is the use of a symbol in a statement that is not its definition.

In the first of the previous two examples, FIELDA and FIELDB are references. In the following VAX MACRO statement, ROUTINEA is a symbolic reference:

```
JMP ROUTINEA
```

The linker maintains a global symbol table (GST) in which it stores the name of every global symbol and its definition (if known).

To resolve a symbolic reference, the linker searches its GST for a definition of the symbol. If it finds one, it substitutes the value of the symbol (its definition) for the symbol itself.

Thus, to resolve the reference to FIELDA, the linker substitutes the value 100 for the symbol in the MOVL #FIELDA,FIELDB instruction. To resolve the reference to ROUTINEA in the JMP ROUTINEA instruction, the linker substitutes the address of the MOVL #FIELDA,FIELDB instruction for the symbol ROUTINEA in the JMP ROUTINEA instruction.

The linker, however, does the work of symbol resolution only for two of the three types of symbols, namely, global and universal. Language processors perform symbol resolution for local symbols. The next section describes these three types of symbols.

1.4.1.1 Types of Symbols

Symbols are designated as local, global, or universal on the basis of their scope, that is, the range of object modules over which they can be recognized and properly interpreted.

A local symbol is a symbol that cannot be interpreted outside the object module that contains its definition. If a reference to a local symbol is made in an outside module, an error results. Since local symbols are defined and referenced within a single object module, the language processor resolves local symbolic references.

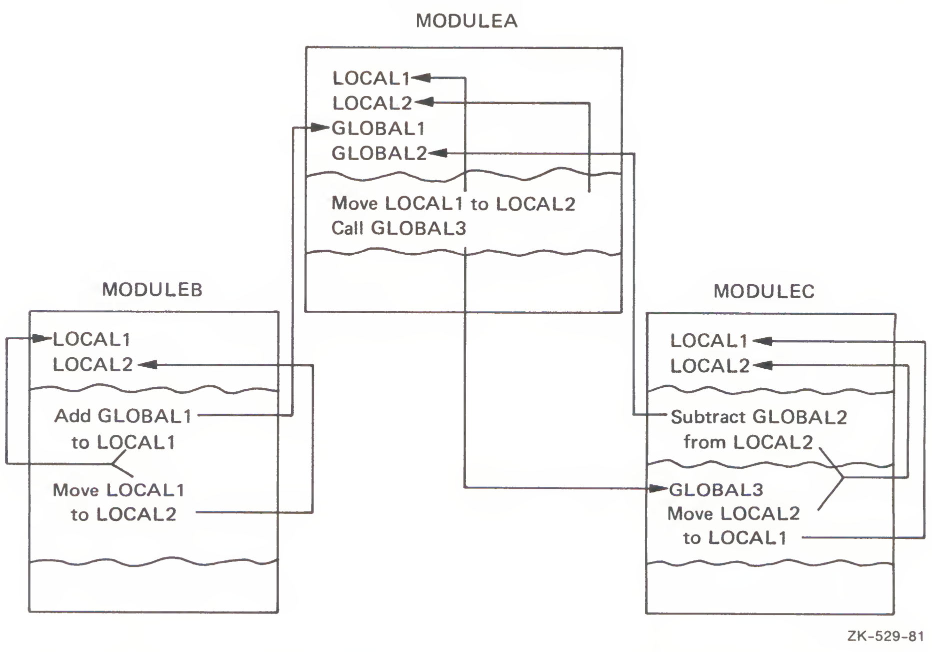
A global symbol is a symbol that can be interpreted outside the object module that contains its definition. Language processors cannot resolve a global symbolic reference if the definition of the global symbol being referenced is not included in the module they are processing. The linker, therefore, must resolve such a global symbolic reference.

A universal symbol is a global symbol found only in shareable images. 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 interpreted outside the shareable image.

The reason that another type of symbol is needed for shareable images concerns the nature of shareable images. A shareable image may have been created from modules that themselves contained global symbols. However, now that these symbols are all defined within a single shareable image, it may not be necessary for object modules outside the shareable image to refer to them. In other words, some of them now function as local symbols within the shareable image. Consequently, the term *universal* is applied to those global symbols that can be referred to by outside modules. This differentiates those global symbols from global symbols that cannot be (and do not need to be) so referred to.

Figure LINK–1 illustrates some of the concepts discussed so far: modular programming, symbol reference and symbol definition, and local and global symbols. Arrows point from a symbol reference to a symbol definition. (The statements do not reflect a specific programming language.)

Figure LINK–1 Symbol Resolution



1.4.1.2 Designation of Local, Global, and Universal Symbols

By default, language processors determine whether a symbol is local or global. For example, the VAX FORTRAN compiler designates statement numbers as local and module entry points as global.

In some languages, the programmer can explicitly specify whether a symbol is local or global by including or excluding particular attributes in the symbol definition. For example, in VAX PL/I the attribute EXTERNAL specifies a global symbol. In VAX MACRO, the use of a single colon (:) in defining a label makes the label a local symbol, while a double colon (::) makes it a global symbol.

A symbol is designated as universal by specifying it in the UNIVERSAL= option in an options file.

1.4.1.3 Weak and Strong Global Symbols

In VAX MACRO, VAX BLISS-32, and VAX PASCAL 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.

It is not possible to make a weak definition or a weak reference in any of the other VAX languages.

In VAX MACRO, VAX BLISS-32, and VAX PASCAL 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 VAX MACRO), the WEAK attribute (in VAX BLISS-32), or the WEAK_GLOBAL or WEAK_EXTERNAL (in VAX 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.

Strong Reference

A strong reference can be made to a weakly defined symbol or to a strongly defined symbol. In either case, the linker will go about resolving the reference during its first pass through the input, as described in detail in Section 5.3.3 and its subsections.

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.

Weak Reference

A weak reference can be made to a weakly defined symbol or to a strongly defined symbol. In either case, the linker will go about resolving 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 an as yet unwritten module). The solution is to inform the linker that the resolution of this particular global symbol is not crucial to the linking operation by making the reference to the symbol weak.

Strong Definition

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

The important point is that, for a strongly defined symbol, if the module (in which the symbol is defined) is contained in a library, the symbol will be included in the library symbol table; if the symbol is weakly defined, it will not.

Weak Definition

A symbol with a weak definition is not included in the symbol table of a library. 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.

1.4.2 Virtual Memory Allocation

Virtual memory allocation is the assignment of virtual memory space to an image. Specifically, it is the algorithmic process of placing different parts of the program at different memory locations to satisfy the requirements of different program segments and of VAX/VMS memory management.

The linker, rather than the language processor, performs virtual memory allocation for the following two reasons:

- Modular programming would not be possible if the language processor allocated virtual memory.
- Language processors do not know the memory requirements of many of the external modules that are called by the module they are processing.

1.4.3 Image Initialization

After it resolves references and allocates virtual memory, the linker fills in the actual contents of the image. This image initialization consists mainly of copying the binary data and code that was written by the compiler or assembler. However, the linker must perform two additional functions to initialize the image contents:

- It must insert addresses into instructions that refer to externally defined fields. For example, if a module contains an instruction moving FIELDA to FIELDB and if FIELDB is defined in another module, the linker must determine the virtual address of FIELDB and insert it into the instruction.
- It must compute values that depend on externally defined fields. For example, if a module initializes location X to contain Y plus Z and if Y and Z are defined in an external module, the linker must compute the value of Y plus Z and insert it in X.

1.5 Image Activation

Image activation is the preparation of an image for execution; it occurs, among other ways, in response to any DCL command (such as RUN) that causes an image to execute. Image activation is performed by the image activator, a module within the executive of VAX/VMS.

The major work of image activation involves setting up the process page tables to accurately reflect the state of all pages in the image file. To accomplish this, the image activator performs a number of specialized functions depending on the kind of image it is activating. However, in general, the image activator *activates* an image file in the following ways:

1. It opens the image file created by the linker, thus allowing the system to perform its file protection checks and logical name translation.
2. It reads into memory the image header, which contains, among other information, a series of image section descriptors (ISDs), each of which describes a portion of the virtual address space.
3. It sets up the process page tables by examining each ISD contained in the image header, determining the type of image section being described there (private, demand-zero, or global), and calling the appropriate memory management system service to perform the actual mapping of the described image section.
4. It calls the appropriate system services to allocate space for the user stack and the image I/O segment, which is an area of memory used by VAX RMS to manipulate files during image execution. The size of both the image I/O segment and the user stack may be established by the user at link time by means of the IOSEGMENT and STACK link options, respectively.
5. It calculates the privileges that will be in effect while this image is executing.

When the image activator finishes its work, it executes an REI instruction, which passes control to the transfer address of the image. At this point the image begins execution.

2 Options Files

An options file is an input file that is specified by means of the /OPTIONS positional qualifier. Using an options file, a programmer can exercise considerable control over the linking operation, as well as simplify the specification of complex input.

An options file may contain the following types of information:

- Input file specifications and associated positional qualifiers, in addition to any that you enter in the LINK command itself
- Special instructions to the linker that are not available through the DCL command language

This section discusses when to use an options file, how to create an options file, and what link options and positional qualifiers may be specified in an options file.

2.1 When to Use an Options File

An options file is useful in the following ways:

- To give the linker a series of file specifications and file qualifiers that you use frequently in linking operations
- To identify a shareable image as an input file in a linking operation
- To enter a longer list of files and positional qualifiers than the VAX/VMS command interpreter can hold in its command input buffer

2.1.1 Entering Frequently Used Input Specifications

For convenience and flexibility, you can create an options file containing a group of file specifications and positional qualifiers that you link frequently, and you can specify this options file as input to the linker.

Consider the following two examples:

1. You want to create an executable image named PAYROLL containing modules named PAYCALC, FICA, FEDTAX, STATETAX, and OTHERDED. You also want to be able to make changes to any of the modules and conveniently relink the image.

   To accomplish these goals, you can use the EDIT or CREATE command to create the file PAYROLL.OPT, containing the following line:

   ```
   PAYCALC,FICA,FEDTAX,STATETAX,OTHERDED
   ```

   Then, to link the image initially or to relink it any time thereafter, you can simply enter the following:

   ```
   $ LINK PAYROLL/OPTIONS
   ```

If you did not use an options file, you would have to enter the following command each time you linked the modules:

```
$ LINK/EXECUTABLE=PAYROLL PAYCALC,FICA,FEDTAX,STATETAX,OTHERDED
```

The more file specifications and positional qualifiers you must specify, the greater is the convenience of using an options file.

2 Two programmers, one writing PROGX and the other PROGY, need to include the modules MODA, MODB, and MODC, and to search the library LIBZ. One programmer can create an options file (say, [G15]GROUP15.OPT) containing the file specifications for MODA, MODB, and MODC, and the specification for LIBZ followed by /LIBRARY. At link time, then, each programmer must specify only the name of his or her module and the options file. For example:

```
$ LINK/MAP PROGX,[G15]GROUP15/OPTIONS
```

2.1.2 Identifying a Shareable Image as Input

To identify as input a shareable image that is not in a library, you must use an options file.

The /SHAREABLE positional qualifier, which is used to identify an input file as a shareable image file, can be used only in an options file; otherwise, the linker interprets it as a command qualifier rather than a positional qualifier.

See the Command Qualifier and the Positional Qualifier Sections for more information on the /SHAREABLE qualifier.

2.1.3 Entering Very Long Commands

Whenever you need to link a series of input files and positional qualifiers that exceeds the buffer capacity of the command interpreter (256 characters), use an options file.

The number of file and qualifier specifications that the command interpreter buffer can hold depends on the length of the specific entries themselves and how much of each line is used. However, as a general guideline, if the LINK command statement exceeds six or seven lines, the command interpreter may not be able to process it. In this case, you must put some or all of the input file specifications and positional qualifiers in an options file.

2.1.4 Entering Link Options

Table LINK–1 lists the link options that may be specified only in an options file. For each link option, Table LINK–1 states the specification format, the default value (if applicable), and a brief explanation. Section 2.3 provides a detailed description of each link option.

Table LINK–1 Link Options

Format	Default Value	Explanation
BASE=n	%X200 for executable, 0 for shareable, and %X80000000 for system	Sets the base virtual address for the image
CLUSTER=cluster-name,-[base-address],-[pfc],[file-spec,...]	(See Section 2.3)	Defines a cluster of image sections
COLLECT=cluster-name,-psect-name[,...]		Moves the named program sections to the specified cluster
DZRO_MIN=n	5	Sets the minimum number of uninitialized pages for demand-zero compression
GSMATCH=keyword,-major-id,minor-id	EQUAL,x,y (See Section 2.3)	Sets match control parameters for a shareable image
IDENTIFICATION=id-name	(See Section 2.3)	Sets the image ID field in the image header
IOSEGMENT=n,-[[NO]P0BUFS]	0, P0BUFS	Sets the number of pages for the image I/O segment
ISD_MAX=n	Approximately 96 (See Section 2.3)	Sets the maximum number of image sections
NAME=file-name	Filename of the output image file	Sets the image name field in the image header
PROTECT= YES NO	NO	Protects clusters in shareable images
PSECT_ATTR=psect-name,-attribute[,...]		Sets program section attributes
STACK=n	20	Sets the initial number of pages for the user mode stack
SYMBOL=name,value		Defines a symbol as global and assigns it a value
UNIVERSAL=symbol-name [,...]		Makes the named global symbol(s) universal

2.2 How to Create and Specify an Options File

To create an options file, use the EDIT command to create a file with any valid file name and a file type of OPT. (You can use any file type, but the linker uses a default file type of OPT with the /OPTIONS qualifier.)

The options file can contain input file specifications and associated positional qualifiers, and/or any link option listed in Table 2–1.

Follow these rules when entering input in an options file:

1. You must separate input files with a comma (,).
2. You cannot enter command qualifiers.
3. You can enter the /INCLUDE, /LIBRARY, /SELECTIVE_SEARCH, and /SHAREABLE positional qualifiers, that is, all positional qualifiers except /OPTIONS.
4. You can enter only one link option per line.
5. You can continue any line by entering the continuation character, the hyphen (-), at the end of the line.
6. You can enter comments after an exclamation point (!).
7. You can abbreviate the name of a link option to as few letters as needed to make the abbreviation unique (for example, UNIVERSAL=ENTRY can be abbreviated UNI=ENTRY).

The following example shows a file named PROJECT3.OPT containing both input file specifications and link options:

```
            PROJECT3.OPT
MOD1,MOD7,LIB3/LIBRARY,-
LIB4/LIBRARY/INCLUDE=(MODX,MODY, MODZ),-
MOD12/SELECTIVE_SEARCH
STACK=75
SYMBOL=JOBCODE,5
```

To include all the specifications and options in this example at link time, you need specify only the file name followed by /OPTIONS. For example:

```
$ LINK/MAP/CROSS_REFERENCE PROGA, PROGB,-
   PROGC, PROJECT3/OPTIONS
```

If you have entered the DCL command SET VERIFY, the contents of the options file are displayed as the file is processed.

You can specify one or more options files in a LINK command statement.

If you want the LINK command to be in a command procedure and you want to specify an options file in the LINK command, specify SYS$INPUT: as the options file. In this way, the DCL command interpreter interprets the lines following the line containing the LINK command as lines in the options file.

For example, a command procedure LINKPROC.COM might contain the following lines:

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

It is advantageous to use a command procedure to invoke the LINK command (as shown in the pevious example) because a single file contains both the LINK command and all input file specifications, including any options files. Thus, to perform the linking operation using all of the input in the previous example, you need only enter the following command:

```
$ @LINKPROC
```

2.3 Link Options

This section discusses in detail each link option in alphabetical order. Each option has the general format:

```
option_name=parameter[,...]
```

If the parameter is a number (indicated by "n"), you can express it 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, as are the values in any linker messages relating to these options.

Link Options

BASE=n

Directs the linker to assign the image a base (starting) virtual address equal to the value of the parameter n.

The BASE= option is illegal in a linking operation that produces a system image and will elicit a warning message. To specify a base address for a system image, use the /SYSTEM[=base-address] command qualifier.

If the address specified in the BASE= option is not divisible by 512, the linker automatically adjusts it upward to the next multiple of 512 (that is, the next highest page boundary).

If the BASE= option is not specified, the linker uses the following default base addresses: hexadecimal 200 (decimal 512) for an executable image; 0 for a shareable image; and hexadecimal 80000000 for a system image.

In general, the use of the BASE= option to create based images is not recommended. VAX/VMS memory management cannot relocate a based 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 (since 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 here 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. See Section 5.3 and its subsections for more information on the linker's clustering and memory allocation algorithms).

CLUSTER=cluster-name,[base-address],[pfc],[file-spec,...]
Directs the linker to create a cluster.

The CLUSTER= option is used for either or both of the following reasons:

- 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 creates one (the default) or more clusters as described in Section 5.3 and its subsections.

You must specify a cluster name in the CLUSTER= option; the other parameters are optional. However, if you omit the base address or the page fault cluster (pfc) or both, you must still enter the comma after each omitted parameter. For example:

```
CLUSTER=AUTHORS,,,TWAIN,DICKENS
```

The optional base-address parameter specifies the base virtual address for the cluster.

The optional page-fault-cluster (pfc) parameter specifies the number of pages to be read into memory when a fault occurs for a page in the cluster. If you do not specify the pfc parameter, VAX/VMS memory management uses the default value established by the SYSGEN parameter PFCDEFAULT.

The file-spec parameter(s) specifies the file(s) 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 two copies of that file included in the final image).

Typically, you specify files to be included in the cluster. However, it is possible to create an empty cluster and to fill it later with program sections by means of the COLLECT= option (see Section 5.3.2).

COLLECT=cluster-name,psect-name[,...]
Directs the linker to place the named program section(s) in the named cluster. If the named cluster has not yet been defined by a CLUSTER= option, the linker creates the cluster when it processes the COLLECT= option.

The linker gives all named program sections the global (GBL) attribute if they do not already have it. Program sections contained in an input shareable image cannot be specified in the COLLECT= option. See Section 5.3.2 for information on the use of the COLLECT= option.

DZRO_MIN=n

Directs the linker to perform demand-zero compression on an image section only when the number of contiguous, uninitialized, writeable pages in that image section is equal to or greater than the value specified by the parameter n.

The DZRO_MIN= option is illegal in a linking operation that produces a system image and elicits a warning message.

Demand-zero compression is the extracting of contiguous, uninitialized, writeable pages from an image section and the placing of these pages into a newly created demand-zero image section.

A demand-zero image section contains uninitialized, writeable 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.

If the DZRO_MIN= option is not specified, the linker uses a default value of five. This means that an uninitialized, writeable portion of an image section is not eligible for demand-zero compression unless it contains at least five contiguous pages.

A DZRO_MIN= value less than five might cause the linker to compress more sections and thus create a greater number of demand-zero image sections (depending on the contents of the object modules). The effect is a reduction in the size of the image file on disk but a decrease in the image's paging performance during execution.

On the other hand, a DZRO_MIN value greater than five might cause the linker to compress fewer sections and thus create fewer demand-zero image sections. The possible effect might be to 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 established by the ISD_MAX= option (or the default value). See the description of the ISD_MAX= option in this section for more information.

Also see Section 5.3.6.1 for a discussion of how and when the linker performs demand-zero compression and for information about additional requirements for demand-zero compression.

GSMATCH=keyword,major-id,minor-id

Sets match control parameters for a shareable image. Its use in the creation of a shareable image allows you to specify whether or not executable images that link with that shareable image must be relinked each time the shareable image is updated and relinked.

The GSMATCH= option causes a major identification parameter (a "major id"), a minor identification parameter (a "minor id"), and a match control keyword to be stored in the image header of the shareable image.

These GSMATCH parameters are used in the following way. When an executable image is linked with a shareable image, the image header of the executable image will contain an image section descriptor (ISD) for each image section in the image, as well as a single ISD for the shareable image. The ISD for the shareable image will contain a major id, minor id, and match control keyword, which the linker copies from the current (at link time) shareable image.

Subsequently, 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 will encounter 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, no matter what GSMATCH parameters are in effect.

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, in which case it allows the mapping.

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 to the effect that the executable image must be relinked. The match control keywords may be EQUAL, LEQUAL, or ALWAYS.

- EQUAL directs the image activator to allow the executable image to map to the shareable image only if the minor id in the ISD of the executable image is equal to the minor id in the shareable image file.
- LEQUAL directs the image activator to allow the executable image to map to the shareable image only if 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 and minor-id parameters, providing that the image section names are the same. However, the syntax of this option requires that you specify the major- and minor-id parameters anyway.

By convention, when a programmer updates and relinks a shareable image, he or she always specifies the GSMATCH= option to increase the value of the minor id and to leave unchanged the value of the major id and the match control keyword. (If the major id is changed, executable images can never map to the shareable image.)

By means of this convention, a programmer who updates and relinks a shareable image can ensure that executable images that linked with the older version of the shareable image can map to the newer version by using the GSMATCH= option in the following way:

- When creating the first version of the shareable image, specify a GSMATCH= option such as the following:

  ```
  GSMATCH=LEQUAL,1,1000
  ```

- When updating and relinking the older (first) version, specify a GSMATCH= option such as the following:

  ```
  GSMATCH=LEQUAL,1,1001
  ```

Note that, in the above example, 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.

To allow an executable image that links with any version to map to any other version (newer or older), you can specify the following when you create a shareable image:

```
GSMATCH=ALWAYS,0,0
```

If you do not specify the GSMATCH= option in the creation of a shareable image, executable images that link with the shareable image must be relinked whenever the shareable image is updated and relinked. For, the default value for the GSMATCH= option is the following, where x and y together are the middle 32 bits of the 64-bit creation time stored in the shareable image file header:

```
GSMATCH=EQUAL,x,y
```

Thus, since the values of x and y will never be the same again (x and y increase with time) and since the keyword EQUAL requires that the minor-id parameters be the same, this default value will require that executable images that link with a shareable image always be relinked whenever that shareable image is updated and relinked.

See Sections 3.5.1 and 3.5.2 for examples of the use of the GSMATCH= option.

IDENTIFICATION=id-name

Sets the image id field in the image header. The maximum length of the field is 15 characters. If id-name contains characters other than uppercase and lowercase A through Z, the numerals 0 through 9, the dollar sign, and the underscore, enclose it with quotation marks.

The image id field is initially taken from the id of the first object module processed 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 an object module is encountered which has a transfer address on the end-of-module (EOM, refer to EOM object record) object record. This transfer address is the main entry point for the image.

For executable images, the image id comes from the id of the object module that contains the main entry point for the image.

For shareable images, the image id usually remains as the id of the first object module processed, since shareable images normally do not have a main entry point.

IOSEGMENT=n[,[NO]P0BUFS]

Specifies the number of pages to be allocated for the image I/O segment, which holds the buffers and VAX RMS control information for all files used by the image.

The required parameter n specifies the number of pages to be allocated for the image I/O segment.

The optional parameter P0BUFS specifies that any additional pages needed by VAX RMS be allocated in the P0 region, while the optional parameter NOP0BUFS denies VAX RMS additional pages in the P0 region.

If the IOSEGMENT= option is not specified, VAX/VMS allocates 32 pages in the P1 region of the process virtual address space for the image I/O segment and allocates additional pages (if needed) at the end of the P0 region of the process address space. Thus, the default setting is, in effect, IOSEGMENT=32,P0BUFS.

Specifying the value of n to be greater than 32 guarantees the contiguity of P1 address space, providing that VAX RMS does not require more pages than the value specified. If VAX RMS does require 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 VAX RMS requires more pages than have been allocated for it, VAX RMS issues an error message.

ISD_MAX=n

Specifies the maximum number of image sections allowed in the image. The parameter n may be a number in hexadecimal (%X), decimal (%D), or octal (%O) radix. The default is decimal radix.

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 compressing the image by creating demand-zero sections and if the total number of image sections reaches the ISD_MAX= value, compression ceases at that point.

The ISD_MAX= option may be specified only in a linking operation that produces an executable image, since the linker only performs demand-zero compression when it is creating an executable image. The ISD_MAX= option is illegal in a linking operation that produces either a shareable or a system image and will elicit a warning message.

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 certain 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.

See the explanation of the DZRO_MIN= option in this section and Section 5.3.6.1, for more information about demand-zero compression.

NAME=image-name

Sets the image name field in the image header. The maximum length of the string is 15 characters. If image-name contains characters other than uppercase and lowercase A through Z, the numerals 0 through 9, the dollar sign, and the underscore, enclose it with quotation marks.

PROTECT= YES/NO

Directs the linker to protect one or more clusters in a shareable image from user-mode or supervisor-mode write access. This option may be specified only in the creation of a shareable image.

PROTECT=YES specifies that clusters defined (by means of the CLUSTER= or COLLECT= options) on subsequent lines (up to the line containing another PROTECT= option, if any) in the options file be protected.

PROTECT=NO, the default value, specifies that clusters defined (by means of the CLUSTER= or COLLECT= options) on subsequent lines (up to the line containing another PROTECT= option, if any) in the options file not be protected.

The following is an example of an options file containing the PROTECT= option. Modules MOD1 and MOD2 in cluster A are protected; MOD3 in cluster B is not protected; program sections PSECTX, PSECTY, and PSECTZ in cluster B are protected.

```
PROTECT=YES
CLUSTER=A,,,MOD1,MOD2
UNIVERSAL=ENTRY
PROTECT=NO
CLUSTER=B,,,MOD3
PROTECT=YES
COLLECT=B,PSECTX,PSECTY,PSECTZ
```

Note that other link options may be interspersed among any PROTECT= options and are not affected by any PROTECT= options.

This option is used in the creation of a privileged shareable image, parts of which need to be protected from nonprivileged users and other parts not. If the entire shareable image needs to be protected, specify the /PROTECT command qualifier. See Section 3.3.3 for more information on privileged shareable images.

PSECT_ATTR=psect-name,attribute[,...]

Directs the linker to assign one or more attributes to a program section. This option is used to change the attributes assigned to a program section by a language processor.

Attributes not mentioned in the PSECT_ATTR= option remain unchanged. For example, the following directs the linker to make the program section ALPHA not writeable instead of writeable and to leave all other attributes of ALPHA unchanged:

```
PSECT_ATTR=ALPHA,NOWRT
```

STACK=n

Directs the linker to allocate the specified number of pages for the user mode stack.

Note that additional pages for the user mode stack are automatically allocated, if needed, during program execution.

If you do not specify the STACK= option, the linker allocates 20 pages for the user mode stack.

The STACK= option may only be specified in a linking operation that produces an executable image. The STACK= option is illegal in a linking operation that produces either a shareable or a system image and will elicit a warning message.

SYMBOL=name,value

Directs the linker to define an absolute global symbol with the specified name and to assign it the specified value.

An absolute global symbol is a global symbol with an absolute, numerical value. That is, it is not a relocatable symbol. Thus, the parameter "value" in the SYMBOL= option must be a number.

The definition of a symbol specified by the 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, uses the definition specified by the SYMBOL= option, but does not issue a warning message.

UNIVERSAL=symbol-name[,...]
Directs the linker to make the specified global symbol in a shareable image a universal symbol. This option may only be specified in the creation of a shareable image.

Making a global symbol a universal symbol ensures that the symbol may be referred to by outside object modules, that is, by object modules other than those that were linked to create the shareable image.

Refer to Section 1.4.1.1 for more information on universal symbols.

3 Shareable Images

This section explains the benefits and uses of shareable images, explains how to write source programs for shareable images, discusses the use of the LINK command and pertinent options, explains how to use shareable images, and provides two detailed examples.

Familiarity with the information in Section 5, Linker Operations, is important to a complete understanding of the information in this section. In addition, Section 1 contains relevant information about shareable image files and libraries, and universal symbols.

3.1 Benefits and Uses of Shareable Images

This section discusses the benefits of using shareable images and some applications in which the use of shareable images is essential or important.

Note that some of the benefits of using shareable images can only be realized if the shareable image is installed using the Install Utility. Installing a shareable image makes the shareable image known to VAX/VMS (that is, makes it a "known" image) and results in its image sections being promoted to global sections. The installation of images is usually done by the system manager.

For additional information on installing images (both shareable and executable) see the Install Utility in the *VAX/VMS Utilities Reference Volume.*

3.1.1 Conserving Disk Storage Space

All programs executed under the VAX/VMS system must be disk resident. When a shareable image is linked with an executable image, the physical contents of the shareable image are typically not copied into the executable image file. Hence, many executable images that have been linked with the same shareable image may be disk resident, but only one copy of the shareable image need exist on disk. This significantly reduces the requirements for disk storage space.

Thus, the use of a shareable image conserves disk storage space, whether or not that shareable image has been installed using the Install Utility.

If, however, the shareable image has not been installed, the image activator copies, at run time, that shareable image into the address space of any process that runs an executable image to which that shareable image has been linked. Therefore, if the shareable image is not installed, multiple copies of that shareable image may exist at run time.

3.1.2 Conserving Main Physical Memory

Main physical memory is one of the prime resources that any operating system has to control. When a shareable image is installed, VAX/VMS creates a set of global sections in physical memory—one for each image section in the shareable image—and allows these global sections to be mapped into the address space of many processes. Mapping the same physical pages of a global section into many processes eliminates the need for each process to have its own copy of those pages, thus significantly reducing the requirements for main physical memory.

If a shareable image is not installed, multiple processes cannot share it at run time; instead, each process receives a private copy of the shareable image at run time. Thus, to conserve main physical memory, the shareable image must be installed.

3.1.3 Reducing Paging I/O

Paging occurs when a process attempts to access a virtual address that is not in the process working set. When such a page fault occurs, the page either is in a disk file (in which case paging I/O is required) or is already in main physical memory. One of the reasons a page can be resident (that is, in main physical memory) when a fault occurs is that it is a shared page, already faulted by some other process that is sharing it. In this case, no I/O operation is required before mapping the page into the working set of a subsequent process. Thus, if many processes are using an installed shareable image, it is more likely that its pages are already physically resident. This reduces the need for paging I/O.

Note that, since a shareable image must be installed if it is to be shared by more than one process at run time, the benefit of reduced paging I/O is possible only if the shareable image is installed.

3.1.4 Sharing Memory-Resident Databases

There are many applications, particularly in data acquisition and control systems, in which response times are so critical that control variables and data readings must remain in main memory. Frequently, many programs must make use of this data.

Shareable images help to simplify the implementation of such applications. The shared database may be, for example, a named VAX FORTRAN COMMON area built into a shareable image. The shareable image may also include routines to synchronize access to such data. When programs of the application bind with the shareable image, they have easy access to the data (and routines) at the VAX FORTRAN level.

It is possible, moreover, for such data bases to contain initial values, and for the most recent values to be written back to disk on system shutdown or at regular intervals. Recording the values at regular intervals makes it possible for a system restart to use the most recent values of the variables of an online process.

Note that since more than one application program (or executable image) is manipulating at run time the same physical pages of the memory-resident, shareable-image data base, the shareable image must be installed.

3.1.5 Making Software Updates Compatible

A major problem in maintaining a large software installation is that of incorporating a new version of a software component in all programs that use it. Packaging software facilities as shareable images can help alleviate the problem.

By carefully organizing a shareable image and by using transfer vectors and position independent coding techniques, you can make significant changes and enhancements to the content of the shareable image and yet eliminate the need to relink all the images bound with it.

3.2 Writing Source Programs for Shareable Images

A shareable image, by its nature, is meant to be linked with many executable images so that it can be executed simultaneously by one or more of these executable images. For this reason, when you write code for shareable images, you should consider shareability, position independence, and the use of transfer vectors.

Sections 3.2.1 and 3.2.2 discuss the topics of shareability and position independence for the benefit, in particular, of VAX MACRO and VAX BLISS-32 programmers, since these languages allow considerable control in this area. Programmers in other high-level languages need not be concerned about these issues since their respective VAX compilers generate code according to the guidelines discussed in these sections.

Sections 3.2.3 and 3.2.4 discuss the use of transfer vectors and the rules for creating upwardly compatible shareable images, topics of interest to programmers in all VAX languages.

3.2.1 Shareability

The sharing of routines between two or more processes must address the issue of whether each process has access to data that one or more other processes are using. Sometimes this sharing is a requirement, as in the case of industrial data acquisition applications. At other times, however, it cannot be allowed. For example, if a piece of data used by a routine is a loop counter, each process must have a separate counter; otherwise the routine cannot be shared simultaneously. This situation is part of the problem known as reentrancy.

Sections of a program that cannot be shared among multiple users (such as the loop counter described above) should be placed in program sections having the WRT and NOSHR attributes. This can be accomplished in VAX MACRO with the .PSECT assembler directive, which serves to place a section of code in a program section with specified attributes.

Alternatively, program sections may be assigned these attributes at link time by means of the PSECT_ATTR option. In either case, the linker places all program sections with the WRT and NOSHR attributes in copy-on-reference image sections or demand-zero image sections.

When an image is activated at run time, a copy-on-reference image section is initialized to contain the contents of the shareable image file. Thereafter, the copy-on-reference image section is treated just like a user-private image section, that is, each process maintains its own copy with its own values. In sum, each user receives a separate physical copy of each copy-on-reference image section contained in a shareable image. Pages of these sections are stored in the system paging file when they are removed from the working set.

On the other hand, if an image section is not copy-on-reference, each user has access to the same physical copy. If the image section is shareable and writeable (SHR and WRT), then when a page of that image section is removed from all user working sets, it is eventually written back into the shareable image file on disk. If such a page had been modified by one or more users, the shareable image file on disk will contain the latest copy. It is this, in fact, which makes it possible to rerun such applications as data acquisition or transaction processing with the most recent values of shareable, modifiable data.

Note here that the cooperating user programs in applications like these are responsible for synchronizing access to such data. Further, when such an application has run, the data will no longer contain its initial values.

Note the following two points about shareability when writing code for a shareable image:

- If an image section is not writeable (has the NOWRT attribute), all processes can use the same copy regardless of whether it is shareable (SHR) or not shareable (NOSHR) since no form of data privacy or security is currently implemented.
- If an image section in a shareable image contains code that includes the .ADDRESS or .ASCID assembler directives, that image section is not shareable.

  At run time when references made using these directives are resolved (see Section 5.3.6.2), an actual address will be inserted for each of these directives. Since these addresses will be correct only for a single process (other processes that want to access the image section simultaneously may have a different memory allocation), the image section containing these directives is not shareable.

3.2.2 Position Independence

Position-independent code executes correctly no matter where it is placed in the virtual address space. Since shareable images by nature are meant to be executed by multiple processes, it is highly desirable that they be position independent.

Position independence is advantageous in shareable images for two main reasons:

- A position-independent shareable image can be linked into the address space of many users without fragmenting their address space. In other words, the image activator can place the shareable image at varying locations in each user image so that the memory allocation for each image is contiguous. This aspect has various performance advantages such as the conservation of page table space for each user image.
- A position-independent shareable image can be enlarged without such modification requiring that all executable images bound to it be relinked. This advantage is made possible by virtue of the fact that a position-independent shareable image is allocated virtual memory at run time, not link time (as for a based shareable image).

Adhere to the following coding guidelines to ensure the position independence of a shareable image. (Note that the .ASCID directive contains an implicit .ADDRESS directive, and therefore the following guidelines for the use of the .ADDRESS directive apply as well to the .ASCID directive.)

- Never refer to numeric virtual addresses when you can refer to their symbolic representation.
- Use the .LONG directive to make a data reference only when the data is absolute. When the linker encounters a .LONG directive in a reference to relocatable data, it must assign a virtual address to resolve the reference. In this case, the linker issues a warning message because such a reference makes the shareable image position-dependent.
- Use the .ADDRESS directive to make a reference to relocatable data or to absolute data. As described in Section 6.3.6.2, the linker processes such references to maintain the position independence of the shareable image.
- Use PC relative addressing mode in a code reference when the target of the reference is contained in your image.
- Use general addressing mode in a code reference when the target of that reference is not contained in your image, as, for example, when you are making a call to a routine in LIBRTL or to various optional software products. Again, to be safe, always use general addressing mode for any external reference.

Although the linker and image activator work together to make code containing .ADDRESS directives position independent, the use of this directive should nevertheless be avoided because it incurs linker and image activator overhead, that is, additional processing time. Further, an image section containing this directive is not shareable.

As mentioned, if a shareable image contains .LONG references to relocatable data, the linker reports each occurrence when the shareable image is linked. To suppress these messages, you can either replace each occurrence of .LONG with .ADDRESS, or you can specify the shareable image to the linker as a based shareable image by using the BASE= option.

In sum, if shareable images are to be most useful among many processes, they should be position independent. Since the VAX instruction set and addressing modes lend themselves to convenient generation of position-independent code, their use, together with strict adherence to the above guidelines, will enable you to write position-independent shareable image code in VAX MACRO or VAX BLISS-32. All high-level language VAX compilers supported by VAX/VMS produce position-independent code.

3.2.3 Transfer Vectors

In writing a source program for a shareable image, you use transfer vectors to enable user programs that are eventually linked with that shareable image to successfully call routines within the shareable image no matter where the shareable image is located in virtual memory.

In its simplest form, a transfer vector is a labeled virtual memory location that contains an address of, or a displacement to, a second location in virtual memory. The second location is the start of the instruction stream (usually a routine) of interest.

The following subsections discuss the advantages of and creation of transfer vectors.

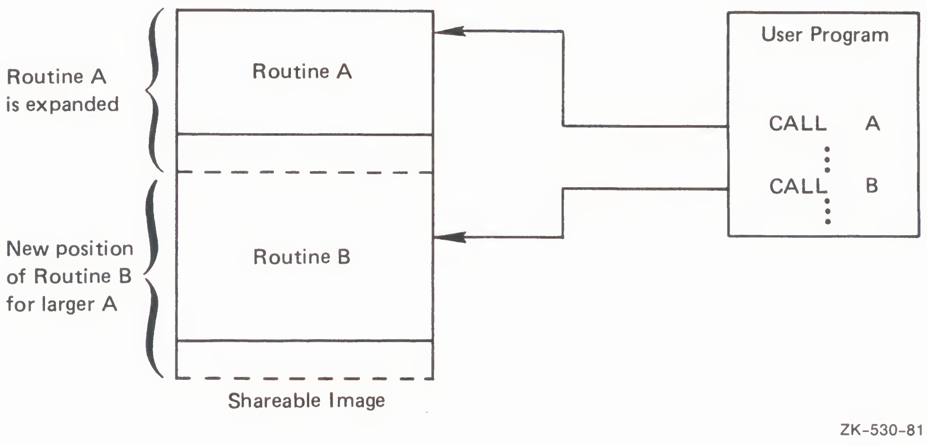
3.2.3.1 Advantages of Transfer Vectors

There are two main reasons for transfer vectors in shareable images:

- They make it easy to modify and enhance the contents of the shareable image.
- They allow you to avoid relinking user programs bound to the shareable image in the event that the shareable image is modified.

For example, in Figure LINK–2 the two routines A and B are bound into a shareable image, which is then bound into a user program. No transfer vectors are used. The user program calls both A and B. Thus, the user program contains a representation of the addresses of both A and B.

Figure LINK–2 Shareable Image Without Transfer Vectors

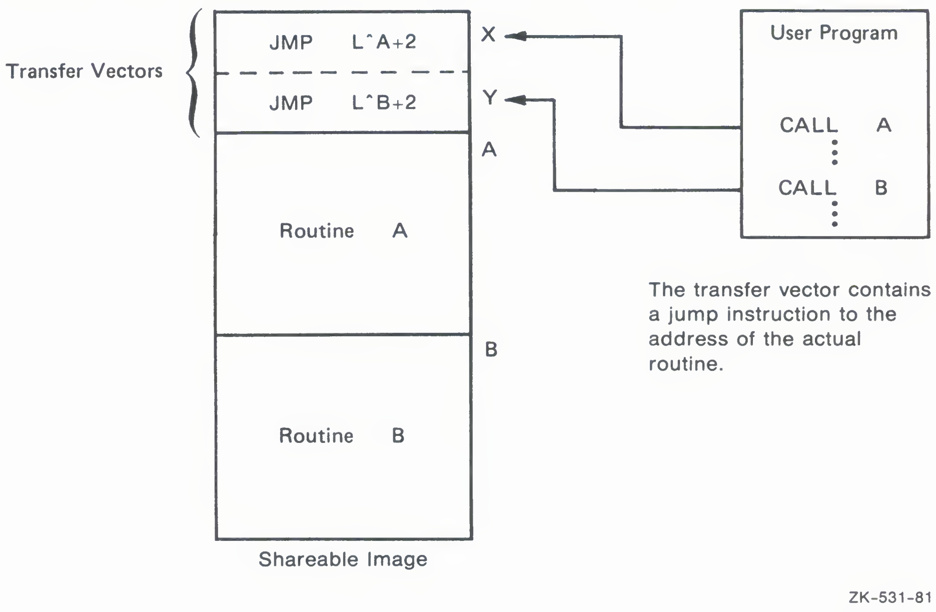


Referring to the example in Figure LINK–2, assume that it becomes necessary to add more code to routine A. When the shareable image is relinked, routine A will have the same address but because routine A has increased in size, routine B must be given a "higher" address—higher by the amount of code added to A.

Thus, if the user program that is bound with the shareable image is not relinked, it can successfully call A, since A's address has not changed. However, the call to B would result in a transfer of control to the old address of B (which is now somewhere in the enlarged routine A), and the desired result would not occur.

Figure LINK–3 depicts the situation when the shareable image contains transfer vectors. If routine A is enlarged and the shareable image relinked, the user program still calls the same transfer vector for routine B, which now contains the new address of routine B. Thus, the desired result is achieved and will always be achieved so long as the user program calls the correct vector and the vector address does not change.

Figure LINK–3 Shareable Image With Transfer Vectors



The use of transfer vectors also allows you to add new routines to a shareable image without needing to relink programs that use existing routines. For instance, in the above example, if a third routine C were added, it would be desirable not to have to relink a user program that used only A and B. Without transfer vectors, it would be necessary to link the three routines in the sequence A,B,C to prevent the addresses of routines A and B from changing, thus necessitating that all user programs linked to the shareable image be relinked. With transfer vectors, however, you can allocate a new vector location to C (after those for A and B) and then link the three routines in any order.

3.2.3.2 Creating Transfer Vectors

Transfer vectors may only be created in VAX MACRO. This section describes how to code a transfer vector for a routine that is entered by a procedure call (a CALLS or CALLG instruction) or by a subroutine call (a JSB or BSB instruction).

By default, VAX compilers generate procedure calls. Therefore, calls to routines in shareable images will generally be made via CALLS or CALLG instructions. For this reason, you should code the transfer vector for a routine in a shareable image using the format for procedure calls shown in Example LINK–1. If, however, you are certain that the routine is coded to be called via a subroutine call, you should use the format for subroutine calls shown in Example LINK–2.

In either case, DIGITAL recommends that you code a transfer vector to be eight bytes long. This may necessitate padding the transfer vector, as is done in Example LINK–2.

Note that a BSBB or BSBW instruction should not be used in a transfer vector to call a routine because these instructions branch to the routine via displacements from the PC. As a result, the linker must assign a virtual address to the start of the routine, thus making the shareable image position-dependent.

Example LINK–1 illustrates the VAX MACRO definition of a transfer vector for a routine FOO entered by means of a CALLG or CALLS instruction.

Example LINK–1 Transfer Vector Coded for a Procedure Call

```
.TRANSFER     FOO       ;Begin transfer vector to FOO
.MASK         FOO       ;Store register save mask
JMP           L^FOO+2   ;Jump to routine, beyond the
                        ;register save mask
```

Example LINK–2 illustrates the VAX MACRO definition of a transfer vector for a routine FOO entered by means of a JSB or BSB instruction.

Example LINK–2 Transfer Vector Coded for a Subroutine Call

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

When the linker encounters the .TRANSFER directive in the above examples, it first verifies that it is creating a shareable image; that is, that the /SHAREABLE qualifier was specified in the LINK command. If so, the linker performs the following actions:

1 It makes the symbol FOO a universal symbol.

2 It resolves all "internal" references to FOO by assigning FOO the address of the routine, not the address of the transfer vector.

3 It resolves all "external" references to FOO by assigning FOO the address of the .TRANSFER directive, rather than the address of the routine that FOO represents. Note that this step (step 3) occurs after step 2.

Thus, the .TRANSFER directive does not cause any memory to be allocated and does not generate any binary code. It merely generates instructions to the linker.

Note: **Because all references to FOO from object modules outside the shareable image are resolved using the address of the transfer vector, this address should never change. To ensure this, you should code all transfer vectors within a single object module and link this module at the beginning of the shareable image (for example, by means of the CLUSTER= option). Then the vector addresses will remain fixed even if the source code for the routines is modified.**

In Example LINK–1, the .MASK directive follows the .TRANSFER directive. This is required when the .TRANSFER directive is used with procedures entered by the CALLS or CALLG instructions (according to the VAX procedure calling standard). The .MASK directive directs the linker to allocate two bytes of memory, find the register save mask accompanying entry point FOO, and store that mask in the allocated memory. Thus, the register save mask for each routine entered via a transfer vector is saved in the transfer vector itself.

Following the .MASK directive is a jump (JMP) instruction to the start of the routine. This instruction occurs immediately after the register save mask. Note that if the procedure is entered via a CALLS or CALLG instruction, it is necessary to jump to the entry point plus 2 in order to skip over the register save mask.

In conclusion, by means of transfer vectors, any entry point to a routine in a shareable image can be made a universal symbol, thus enabling that routine to be called by any user program outside the shareable image in the same way as an ordinary object module. Further, such routines in a shareable image can be modified without requiring that user programs bound to the shareable image be relinked.

Example LINK–2 shows how a transfer vector for a routine in VAX FORTRAN is linked with that routine to produce a shareable image.

For additional information on VAX MACRO assembler directives, see the *VAX MACRO and Instruction Set Reference Volume.*

3.2.4 Rules for Creating Upwardly Compatible Shareable Images

To be able to make changes to shareable images and not have to relink the images using that shareable image, you must observe the following rule:

Transfer vectors must not be rearranged or removed. Thus, if a routine is deleted from a shareable image for any reason, its transfer vector should point to a dummy routine to ensure that user programs bound to the shareable image do not fail in unforeseen ways. It is also a good idea to allow for the addition of future transfer vectors by reserving extra space at the end of your transfer vector object module. You should always add new transfer vectors at the end. Never insert them between existing transfer vectors.

For based shareable images, you must also observe the following additional rule:

> The new shareable image must not be larger than the old one.

To meet the above guidelines for upward compatibility for based shareable images, it is useful to reserve expansion space in a shareable image at the time of creation to allow that image to grow. So long as modifications to the shareable image do not cause it to grow beyond the expansion space, user programs bound to the shareable image need not be relinked.

However, since there is a substantial overhead in increasing the size of a shareable image (one entry in the system's global page table per shareable page), you should reduce the expansion area when the shareable image is no longer being developed. These restrictions on the upward compatibility of based shareable images may be avoided by coding the shareable image for position independence.

3.3 Creating a Shareable Image

This section discusses information relevant to a linking operation that results in the creation of a shareable image. Much of this information is discussed in detail elsewhere in this manual or in other manuals. In either case, you will be instructed where to look for additional information.

To create a shareable image, use the /SHAREABLE[=file-spec] command qualifier with the LINK command.

One or more object modules (but at least one), as well as one or more shareable images, may be specified as input in the creation of a shareable image.

To specify a shareable image as input in the creation of a shareable image (in fact, any image), use the /SHAREABLE positional qualifier. /SHAREABLE, as a positional qualifier, is only legal in options files.

The Format Section discusses the syntax of the LINK command, and the Command Qualifier and Positional Qualifier Sections discuss the use of /SHAREABLE as a command and positional qualifier.

3.3.1 Using the UNIVERSAL=Option

Universal symbols are the global symbols of a shareable image that are of use to programs that subsequently link with the shareable image. They are, therefore, the only symbols contained in the symbol table of a shareable image.

Any number of the global symbols of a shareable image may be made universal. Typically, however, only a very small set of the global symbols of a shareable image need to be made universal because few of them are of interest outside the shareable image. Normally, all the entry points (routine names) provided in a shareable image are made universal, and sometimes other constants are also made universal.

High-level languages do not provide a way of characterizing a symbol as universal. Aside from the VAX MACRO .TRANSFER directive, the UNIVERSAL= option is the only way to designate a global symbol as universal.

You may specify one or more global symbols by name in the UNIVERSAL= option. See Section 2.3 for more information on the format and syntax of the UNIVERSAL= option.

3.3.2 Using The GSMATCH=Option

The GSMATCH= option sets match control parameters for a shareable image. Its use in the creation of a shareable image allows you to specify whether or not executable images that link with that shareable image must be relinked each time the shareable image is updated and relinked.

If you do not specify the GSMATCH= option in the creation of a shareable image, executable images that link with the shareable image must be relinked whenever the shareable image is updated and relinked.

For a thorough discussion of the GSMATCH= option, see Section 2.3.

3.3.3 Creating Privileged Shareable Images

A privileged shareable image differs from a typical shareable image in the following ways:

- A privileged shareable image may contain dispatchers that handle change-mode-to-kernel and change-mode-to-executive instructions.
- Portions (or all) of the privileged shareable image are protected from user-mode and supervisor-mode write access.
- A privileged shareable image must be installed using the Install Utility (INSTALL), whereas a typical shareable image need not be installed.

Thus, a privileged shareable image allows executable images to call user-written procedures that run in a more privileged mode in the same way that they call system services.

Both the PROTECT= option and the /PROTECT command qualifier are used to create privileged shareable images, though not both at the same time. The PROTECT= option is used when some clusters require protection and some do not. The /PROTECT command qualifier is used when all clusters require protection. Note that an individual program section may be protected by means of the VEC program section attribute, providing it has the correct vector format.

When creating a privileged shareable image, you should protect the clusters containing code or data that privileged-mode code must access, and not protect the clusters that user-mode code must access. Thus, the /PROTECT command qualifier should only be used when the entire shareable image needs to be protected. The VEC program section attribute should only be used for the program section that contains the change mode dispatch vectors.

See Section 2.3 for a discussion of the PROTECT= option and the Command Qualifier Section for a discussion of the /PROTECT command qualifier.

For more infomation on installing privileged shareable images, see the Install Utility in the *VAX/VMS Utilities Reference Volume*.

3.4 Using Shareable Images

To use a shareable image, you include the shareable image as input in a linking operation that results in the creation of an executable image.

To specify a shareable image as input to the linker, you must use an options file, unless the shareable image is in a shareable image library. In the options file, you specify the shareable image file as input with the /SHAREABLE positional qualifier.

If the shareable image is in a shareable image library, such as SYS$LIBRARY:IMAGELIB.OLB, you use the /INCLUDE file qualifier to specify a shareable image for extraction from the library. See the Positional Qualifier Section and Section 2 for more information about the /SHAREABLE and /INCLUDE positional qualifiers, and about options files, respectively.

Usually shareable images are installed by the system manager to make them available to cooperating users at run time. Note that images with writeable, shareable data, such as COMMON areas, must always be installed /WRITEABLE.

When an executable image that is linked with a shareable image is run, the image activator opens the shareable image file and checks the global section match. If the match succeeds, the image activator maps the shareable image into the assigned virtual address space. One of two things happens depending on whether the shareable image has been installed with the /SHARE qualifier.

If the shareable image has been installed with the /SHARE qualifier, all processes share the same copy of the shareable image in physical memory. Thus, if the executable image references a page of the shareable image that is not currently in physical memory, that page is read in from the shareable image. If the executable image references a page that is already in physical memory, that page is used. Note that once a page of a shareable image is read into physical memory for one process, any other process can use the same page in physical memory.

If the shareable image has been installed without the /SHARE qualifier, or if the shareable image has not been installed, or if the global section has the copy-on-reference attribute, the image activator creates a private copy of the shareable image. In this case, the private copy of the shareable image is treated as part of your executable image. Each process that is linked with the shareable image must have its own copy of the shareable image in physical memory.

If the match fails, the image activator displays an error message indicating that the required global sections are not available.

If the image activator cannot find the shareable image and if the executable image has a private copy of the shareable image, that copy is used. But if the executable image does not have a private copy, the image activator displays an error message indicating that the shareable image is not available.

If the image activator finds a shareable image but the match fails, it will not use a private copy even if one is present in the executable image.

If the image activator finds a shareable image, the match succeeds, and the executable image already has a private copy of the shareable image, the image activator uses the copy in the shareable image file.

Note that by default the image activator locates a shareable image section (or global section, if the shareable image was installed) in the following way: (1) it strips the file name from the global section name, for example, LBRSHR is the file name derived from the global section name LBRSHR_001, and (2) it combines that file name with the default device and file type specification SYS$SHARE:.EXE, yielding, for example, the full file specification SYS$SHARE:LBRSHR.EXE.

Therefore, if you want to use a shareable image in a directory other than the default directory SYS$SHARE:, you must define a logical name for that shareable image to enable the image activator to locate it. That is, you must assign the full file specification of the shareable image to the name of the shareable image, as follows:

```
$ DEFINE LBRSHR DISK$WORKDISK:[MYDIR]LBRSHR
```

3.5 Examples of Shareable Images

This section contains two examples that serve to demonstrate much of what has been discussed in this section. Both examples are command procedures that create, compile, link, and run programs.

Both examples demonstrate the following general aspects of program development using shareable images:

- Using a command procedure to facilitate many aspects of program development, such as the compiling, linking, and running of programs
- Writing source code for shareable images, including the use of transfer vectors
- Creating shareable images in linking operations
- Using options files, particularly in command procedures
- Installing shareable images

In addition, each example demonstrates a more complex use to which shareable images may be put. The first example below shows how resource allocation procedures may be shared among separate shareable images within the same process, while the second example shows the correct way to use multiple shareable images that reference the same COMMON area.

Example LINK–3 contains the command procedure SHREXAMP1.COM. Note that a full image map of MAIN1, the executable image generated by this command procedure, is displayed and described in Section 4.

Numbers in the programming examples are keyed to numbered notes that follow each example.

Example LINK–3 Sharing Resource Allocation Procedures Among Shareable Images

```
$ V ='F$VERIFY(0)
$ !
$ ! This command procedure demonstrates that two separate
$ ! shareable images (each of which calls the resource
$ ! allocation procedure LIB$GET_EF), when linked together
$ ! into an executable image, will share the OWN (local)
$ ! storage, rather than use two separate areas of local
$ ! storage, one per shareable image.
$ !
$ !
$ ! If the first parameter to this procedure is not null, the
$ ! procedure cleans up from a previous invocation and exits.
$ !
$ !
$ DELETE MAIN1.*.*,A1.*.*,B1.*.*     ❶
$ IF P1 .NES. "" THEN EXIT
$ !
$ ! Create the source files
$ !
$ !
$ ! The main program
$ !
$ CREATE MAIN1.MAR     ❷
        .title  main1
        .ident  /v03-001/
;
; The following cell (beginning at the first .PSECT directive and
; ending at the line before the next .PSECT directive) is not
; used by this program.  The cell contains a .ADDRESS directive
; and is included here merely to demonstrate how the linker
; records its occurrence in the Module Relocatable Reference
; Synopsis section of the image map.  See Section 4 for a
; description of this section of the image map, as well as for
; the actual image map generated by this program.
;
        .psect  $data$,noexe,wrt,noshr,long
addr_data:
        .address lib$get_vm
        .psect  $code$,exe,nowrt,shr,long
a_name: .ASCIC  /A/              ;Name strings for output
b_name: .ASCIC  /B/
```

(Continued on next page)

Example LINK–3 (Cont.) Sharing Resource Allocation Procedures Among Shareable Images

```
        .entry  start,^M<R11>       ;Program entry point
        movl    #3,r11              ;Loop three times
        subl2   #4,sp               ;Allocate temp on the stack
10$:    pushal  (sp)                ;Stack address to return value
                                    ;into
        calls   #1,G^a              ;Call A
        bsbb    err_check           ;Check for error from A
        pushl   (sp)                ;Stack allocated EFN number
        pushal  a_name              ;Stack routine name
        calls   #2,w^fao_and_output   ;Print EFN allocated
        pushal  (sp)                ;Stack address to return value
                                    ;into
        calls   #1,G^b              ;Call B
        bsbb    err_check           ;Check for errors
        pushl   (sp)                ;Stack allocated EFN number
        pushal  b_name              ;Stack routine name
        calls   #2,w^fao_and_output   ;format and output EFN
                                      ;allocated
        sobgtr  r11,10$             ;Loop for all
        movl    #1,r0               ;Exit success
        ret                         ;Return from image
err_check:
        blbs    r0,10$              ;Branch if no error
        pushl   r0                  ;Error---stack code
        calls   #1,G^lib$signal     ;Signal the error
10$:    rsb                         ;Return
fao_ctrstr:
        .ASCID  /Procedure !AC allocated event flag !UL./
fao_and_output:
        .word   ^M<R2>
        subl2   #138,sp             ;Allocate space for
                                    ;descriptor,buffer
        movl    #132,(sp)           ;Create a string descriptor
        moval   8(sp),4(sp)         ;...
        movl    sp,r2               ;Save address of descriptor
        $FAO_S  ctrstr=fao_ctrstr,-  ;Format the output line
                outlen=(r2),-
                outbuf=(r2),-
                P1=4(ap),-
                P2=8(ap)
        bsbb    err_check           ;Check for FAO error
        pushal  (sp)                ;Stack descriptor address
        calls   #1,G^lib$put_output  ;Output formatted line
        bsbb    err_check           ;Check for lib$put_output error
        ret                         ;All done
       .end     start
$
```

(Continued on next page)

Example LINK–3 (Cont.) Sharing Resource Allocation Procedures Among Shareable Images

```
$ !
$ ! Subroutine A
$ !
$ CREATE A1.MAR     ❸
        .title  a_1
        .psect  a_transfer,exe,nowrt,pic,shr,gbl
        .transfer a               ;Transfer vector for
                                  ;  shareable image A

        .mask   a
        jmp     l^a+2             ;Skip the entry mask

       .psect   code,nowrt,pic,shr
       .entry   a,0               ;Entry point for routine A
       callg    (ap),G^lib$get_ef ;Call the RTL routine
                                  ;  LIB$GET_EF
       ret
       .end
$
$ !
$ ! Subroutine B
$ !
$ CREATE B1.MAR      ❹
        .title  b_1

        .psect  b_transfer,exe,nowrt,pic,shr,gbl
        .transfer b               ;Transfer vector for
                                  ;  shareable image B
        .mask  b
        jmp    l^b+2              ;Skip the entry mask
        .psect code,nowrt,pic,shr
        .entry b,0                ;Entry point for routine B
        callg  (ap),G^lib$get_ef  ;Call the RTL routine
                                  ;  LIB$GET_EF
        ret
        .end
$
$ SET VERIFY
$ !
$ ! Compile and link
$ !
$ MACRO MAIN1
$ MACRO A1      ❺
$ MACRO B1
$
```

(Continued on next page)

Example LINK–3 (Cont.) Sharing Resource Allocation Procedures Among Shareable Images

```
$ LINK/MAP/FULL/SHARE A1,SYS$INPUT/OPTIONS
!
! Options for shareable image A
!                                                    ❻
GSMATCH = LEQUAL,1,0
UNIVERSAL = A
CLUSTER = A_TRANSFER
COLLECT = A_TRANSFER,A_TRANSFER
$
$ LINK/MAP/FULL/SHARE B1,SYS$INPUT/OPTIONS
!
! Options for shareable image B
!                                                    ❼
UNIVERSAL = B
CLUSTER = B_TRANSFER
COLLECT = B_TRANSFER,B_TRANSFER
$
$ LINK/MAP/FULL MAIN1,SYS$INPUT/OPTIONS
!
! Options for executable image MAIN                  ❽
!
A1/SHARE,B1/SHARE
$ !
$ ! Define logical names for the shareable images so they do
$ ! not need to be moved to SYS$SHARE:
$ !
$ DEFINE /USER A1 SYS$DISK:[]A1        ❾
$ DEFINE /USER B1 SYS$DISK:[]B1
$ !
$ ! Run the program.  If the test works, six different event flag
$ ! numbers will have been assigned.  If the test fails and each
$ ! routine has its own data base, then two sets of the same
$ ! three event flag numbers will have been assigned.
$ !
$ RUN MAIN1     ❿
$ V = 'F$VERIFY(V)
$ EXIT     ⓫
```

The following comments annotate the preceding command procedure and explain its execution. Each comment has a number that corresponds to the circled number embedded in the text.

❶ Deletes all files created by any previous execution of this command procedure.

❷ Creates the file MAIN1.MAR, a VAX MACRO program that calls the two subroutines A and B.

❸ Creates the file A1.MAR, a VAX MACRO subroutine A that calls the resource allocation procedure LIB$GET_EF. Note the use of a transfer vector.

❹ Creates the file B1.MAR, a VAX MACRO subroutine B that calls the resource allocation procedure LIB$GET_EF. Note the use of a transfer vector.

❺ Assembles MAIN1, A1, and B1.

❻ Links A1 as a shareable image—at the same time requesting a *full map* and specifying by means of an options file that (1) GSMATCH values be established, (2) the symbol A be made universal, (3) the cluster A_transfer be created as an empty cluster, and (4) the program section A_transfer be placed in the cluster A_transfer.

❼ Links B1 as a shareable image—at the same time requesting a full map and specifying by means of an options file that (1) the symbol B be made universal, (2) the cluster B_transfer be created as an empty cluster, and (3) the program section B_transfer be placed in the cluster B_transfer. Note that because the GSMATCH= option has not been specified, MAIN1 will have to be relinked if B1 is ever relinked.

❽ Links MAIN1 as an executable image—at the same time requesting a full map and specifying by means of an options file that the shareable images A1 and B1 be included as input.

❾ Defines logical names for each of the shareable image files, which are in the user's default device and directory, so that the image activator will be able to find them at run time. (Remember that by default the image activator searches for shareable images using the filename and the default file specification SYS$SHARE:.EXE.)

❿ Runs MAIN1.

⓫ Exits.

Example LINK–3 demonstrates, among other things, how resource allocation procedures can be shared among separate shareable images within the same process.

The shareable images A and B each call the resource allocation routine LIB$GET_EF in the Run-Time Library (VMSRTL). This routine allocates local event flags from a process-wide pool. If a flag is available, its number is returned to the caller. If no flags are available, an error is returned.

MAIN1 calls A three times and B three times. Therefore, assuming the availability of event flags, six event flag numbers will have been returned when MAIN1 finishes executing.

The sharing of the routine LIB$GET_EF by A and B means that they both use (call) the same copy of the routine. As a result, the routine LIB$GET_EF will return six different event flag numbers—for example, 63,62,61,60,59,58.

If, on the other hand, the routine LIB$GET_EF were not shared by A and B (that is, they each had their own copy of the routine), the routine would return two sets of the same three event flag numbers—for example, 63,63,62,62,61,61—one set (63,62,61) to A and one set to B.

Sharing of the routine LIB$GET_EF is made possible by the fact that the linker includes VMSRTL (which contains LIB$GET_EF) in the executable image only once, despite the fact that two shareable images call it.

Example LINK–4 contains the command procedure SHREXAMP2.COM.

Example LINK–4 Using Complex Shareable Images

```
$ V = 'F$VERIFY(0)
$ !
$ ! This example demonstrates how to create and link multiple
$ ! shareable images, each of which references an identical
$ ! COMMON area.  To do this, one shareable image containing
$ ! the COMMON blocks is created and linked.  Then, the
$ ! shareable images that reference the COMMON area are created
$ ! and linked with the shareable image containing the
$ ! the COMMON area.
$ !
$ ! Note that the shareable image SHARE2B does not
$ ! use a transfer vector.  Therefore, when SHARE2B
$ ! is updated and relinked, any executable image bound to it
$ ! must be relinked.
$ !
$ DELETE ACOM2.*;*,SHARE2A.*;*,XFR2A.*;*,SHARE2B.*;*,-    ❶
  MAIN2.*;*
$ IF P1 .NES. "" THEN EXIT
$ CREATE ACOM2.FOR    ❷
C
C       FORTRAN Block data
C
C
C       This module contains only the declarations for the
C       COMMON blocks referenced by the shareable images.
C
        BLOCK DATA ACOM

        COMMON  /COMMON1/ integer_array(67)
        COMMON  /COMMON2/ integer_array2(48)

        END
$
$ CREATE SHARE2A.FOR    ❸
C
C       FORTRAN subroutine share2a
C
        SUBROUTINE share2a

        COMMON  /COMMON1/ integer_array(67)
        COMMON  /COMMON2/ integer_array2(48)

        integer_array(1) = 67
        integer_array2(1) = 48

        RETURN
        END
$
```

(Continued on next page)

Example LINK–4 (Cont.) Using Complex Shareable Images

```
$ CREATE XFR2A.MAR       ❹
        .title  xfr2a - Transfer vector for SHARE2A
        .ident  /v01-001/
        .psect  $$xfrvectors,exe,nowrt
        .transfer share2a
        .mask   share2a
        jmp     l^share2a+2
        .end
$
$ CREATE SHARE2B.FOR       ❺
C
C       FORTRAN subroutine share2b
C
        SUBROUTINE share2b
        COMMON  /COMMON1/ integer_array(67)
        COMMON  /COMMON2/ integer_array2(48)
        integer_array(1) = 48
        integer_array2(1) = 67
        RETURN
        END
$
$ CREATE MAIN2.FOR       ❻
C
C       Main program
C
        PROGRAM MAIN2
        COMMON  /COMMON1/ integer_array(67)
        COMMON  /COMMON2/ integer_array2(48)
        CALL share2a
        TYPE 10,integer_array(1),integer_array2(1)
10      FORMAT(' SHARE2A---',I,I)
        CALL share2b
        TYPE 20,integer_array(1),integer_array2(1)
20      FORMAT(' SHARE2B---',I,I)
        STOP
        END
$
```

(Continued on next page)

Example LINK–4 (Cont.) Using Complex Shareable Images

```
$ SET VERIFY
$ !
$ ! Compile and link
$ !
$ FORTRAN ACOM2
$ FORTRAN SHARE2A
$ MACRO XFR2A           ❼
$ FORTRAN SHARE2B
$ FORTRAN MAIN2
$
$ LINK /SHARE /MAP /FULL ACOM2       ❽
$
$ LINK /SHARE /MAP=SHARE2A /FULL -
  SHARE2A, SYS$INPUT/OPTION                         ❾
!
! Options input for SHARE2A
!
ACOM2/SHARE
GSMATCH=LEQUAL,1,0
CLUSTER=TRANSFER_VECTOR,,,XFR2A
$
$ LINK /SHARE /MAP /FULL SHARE2B,SYS$INPUT/OPTION
!
! Options input for SHARE2B                           ❿
!
UNIVERSAL=SHARE2B
GSMATCH=LEQUAL,1,0
ACOM2/SHARE
$
$ LINK /MAP /FULL MAIN2, SYS$INPUT/OPTION
!
! Options input for MAIN2                     ⓫
!
SHARE2A/SHARE,SHARE2B/SHARE
$
$ !
$ ! Now run the program
$ !
$ !
$ ! First install the shareable image containing the
$ ! COMMONS.
$ !
$ RUN SYS$SYSTEM:INSTALL                ⓬
SYS$DISK:[]ACOM2 /OPEN /SHARE /WRITE
$ DEFINE /USER ACOM2 SYS$DISK:[]ACOM2
$ DEFINE /USER SHARE2A SYS$DISK:[]SHARE2A       ⓭
$ DEFINE /USER SHARE2B SYS$DISK:[]SHARE2B
$ RUN MAIN2     ⓮
$ !
$ ! Remove the global section
$ !
$ RUN SYS$SYSTEM:INSTALL    ⓯
SYS$DISK:[]ACOM2 /DELETE
$ V = 'F$VERIFY(V)
$ EXIT      ⓰
```

The following comments annotate the preceding command procedure and explain its execution. Each comment has a number that corresponds to the circled number embedded in the text.

❶ Deletes all files created by any previous execution of this command procedure.

❷ Creates the file ACOM2.FOR, a VAX FORTRAN block data program that defines two arrays as COMMON areas.

❸ Creates the file SHARE2A.FOR, a VAX FORTRAN subroutine that assigns a value to the first cell in each of the COMMON areas defined by ACOM2.FOR.

❹ Creates the file XFR2A.MAR, a VAX MACRO definition of a transfer vector for the VAX FORTRAN subroutine SHARE2A.FOR.

❺ Creates the file SHARE2B.FOR, a VAX FORTRAN subroutine that assigns a value (the reverse of those assigned by SHARE2A.FOR) to the first cell in each of the COMMON areas defined by ACOM2.FOR.

❻ Creates the file MAIN2.FOR, the main VAX FORTRAN program that (1) calls the subroutine SHARE2A and prints the values it assigns, and (2) calls the subroutine SHARE2B and prints the values that it assigns.

❼ Compiles each of the above four VAX FORTRAN programs and assembles the VAX MACRO transfer vector definition.

❽ Links ACOM2 as a shareable image—at the same time requesting a full map. Note that because the GSMATCH= option has not been specified, SHARE2A, SHARE2B, and MAIN2 will have to be relinked if ACOM2 is ever relinked.

❾ Links SHARE2A as a shareable image containing a transfer vector—at the same time requesting a full map and specifying (by means of an options file) the shareable image ACOM2 as input and the GSMATCH parameter LEQUAL with a major id of 1 and a minor id of 0.

Note that because the transfer vector definition XFR2A is specified first on the command line, it is necessary to explicitly specify that the shareable image and its image map be named SHARE2A, instead of XFR2A; this is done by specifying the =SHARE2A parameter with the /SHARE and /MAP qualifiers. Note too that it is not necessary to specify UNIVERSAL=SHARE2A to make SHARE2A a universal symbol; the .TRANSFER directive in XFR2A makes SHARE2A universal.

❿ Links SHARE2B as a shareable image—at the same time requesting a full map and specifying (by means of an options file) the shareable image ACOM2 as input, the entry point SHARE2B as universal, and the GSMATCH parameter LEQUAL with a major id of 1 and a minor id of 0. Note that since a transfer vector was not included in this linking operation, SHARE2B may have to be relinked in the event it is modified.

⓫ Links MAIN2 as an executable image—at the same time requesting a full map and specifying (by means of an options file) the shareable images SHARE2A and SHARE2B as input.

⓬ Installs ACOM2 as writeable and shareable.

⓭ Defines logical names for each of the shareable image files, which are in the user's default device and directory, so that the image activator will be able to find them at run time (remember that by default the image activator searches for shareable images using the filename and the default file specification of SYS$SHARE:.EXE).

⓮ Runs MAIN2.

⓯ Deletes the global section that was created by the previous installation of ACOM2.

⓰ Exits

The following are some of the more complex aspects of the use of shareable images that are demonstrated by this example:

- This example demonstrates how to create and link multiple shareable images each of which references an identical COMMON area. This situation is trickier than that involving multiple subroutines within a single image each of which references an identical COMMON area.

  Since SHARE2A and SHARE2B are independent shareable images, each would normally have its own COMMON area. Thus, the main program MAIN2, which wishes to manipulate a single COMMON area by means of the shareable images SHARE2A and SHARE2B would not be able to do so. However, by the use of a third shareable image ACOM2, the desired result is achieved in the following way.

  When both shareable images that modify the COMMON area—SHARE2A and SHARE2B—are linked (created), a third shareable image that defines the COMMON area is included as input. Then, SHARE2A and SHARE2B are included as input in the linking of the main program MAIN2. In this way, when MAIN2 calls SHARE2A and SHARE2B to modify the COMMON area, the identical COMMON is modified (since both SHARE2A and SHARE2B have been linked with ACOM2).

  It is important to realize that SHARE2A and SHARE2B are both independent shareable images and each one may be independently modified without requiring that the other or that MAIN2 be relinked.

  In contrast, the following link sequence would result in a single COMMON area, but the shareable images SHARE2A and SHARE2B would no longer be independent:

  ```
  $ LINK/SHARE/MAP/FULL SHARE2A,SYS$INPUT/OPTION
  GSMATCH=LEQUAL,1,0
  $ LINK/SHARE/MAP/FULL SHARE2B,SYS$INPUT/OPTION
  SHARE2A/SHARE GSMATCH=LEQUAL,1,0
  ```

- This example demonstrates, as a convenient side effect of the previously described advantage, that it is necessary to install only ACOM2 as writeable, not SHARE2A or SHARE2B.

4 Image Map

If you so request, the linker produces an image map containing information about the contents of the image and about the linking process itself. You can print a copy of the map with the DCL command PRINT and use it to help locate link-time errors, to study the layout of the image in virtual memory, to keep track of global symbols, and so on.

To obtain a map in interactive mode, you must specify the /MAP[=file-spec] qualifier in the LINK command. If you specify the optional file-spec parameter, the linker writes the map to a file with that file specification. If you do not specify this parameter, then by default the linker writes the map to a file having the file name of the first input file and the file type MAP.

There are several types of map. Section 4.1 discusses these map types and the LINK command qualifiers used to obtain them. Section 4.2 discusses the sections that comprise the various types of map. Section 4.3 shows an example map and explains some of the information it contains.

4.1 Types of Image Map

The following are the three types of image map:

- Brief map
- Default map
- Full map

Of these three, the full map is by far the most useful. To get a full map, specify the /MAP and /FULL command qualifiers in the LINK command; to get a brief map, specify /MAP and /BRIEF; to get a default map, simply specify /MAP.

With the default and full map, you can also request that a Symbol Cross-Reference section replace the Symbols by Name section by specifying the /CROSS_REFERENCE command qualifier.

Table LINK–2 shows the five possible types of map output and the LINK command qualifiers required to produce each type.

Table LINK–2 Types of Image Map

Command	Type of Map Produced
$ LINK/MAP/BRIEF	Brief map
$ LINK/MAP	Default map
$ LINK/MAP/CROSS_REFERENCE	Default map with symbol cross-reference
$ LINK/MAP/FULL	Full map
$ LINK/MAP/FULL/CROSS_REFERENCE	Full map with symbol cross-reference

4.2 Image Map Sections

The number of sections contained in an image map depends on the type of map. A full map contains eight sections; a default map, five; and a brief map, three.

Column 1 of Table LINK–3 lists each of the possible image map sections in the order in which they appear in the image map. Column 2 lists the type(s) of map in which each of these sections appears. Column 3 provides a brief explanation of the contents of each image map section.

The term "all" in Column 2 means that the corresponding map section appears in all three types of map—full, default, and brief—while the terms "default" and "full" in Column 2 mean that the corresponding map section appears in the default and full maps, respectively. Note that the term "brief" does not appear in Table LINK–3; map sections contained in a brief map are designated by the term "all" in Column 2.

Not only does a full map contain more sections than a default or brief map, but some of its sections may also contain more information than those same sections would in a default or brief map.

The following are the four map sections that may contain more information if they appear in a full map than they would if they appeared in a default or brief map:

- Object module synopsis
- Program section synopsis
- Symbols by name
- Symbol cross-reference

In a full map, these sections may contain information about modules or shareable images that were implicitly included (but not explicitly specified) in the linking operation. For example, if a routine were extracted from the default system library to resolve a symbol reference, the Program Section Synopsis section in a full map would contain information about the program sections comprising that routine, whereas the Program Section Synopsis section in a default map would not.

Table LINK–3 Image Map Sections

Section Name	Appearance	Explanation
Object module synopsis	All	Object modules in the image
Module relocatable Reference synopsis	Full	Number of .ADDRESS directives in each module
Image section synopsis	Full	Image sections and clusters
Program section synopsis	Default	Program sections and the full modular contributions

Table LINK–3 (Cont.) Image Map Sections

Section Name	Appearance	Explanation
Symbols by name or Symbol cross reference	Default Full	Symbols by Name lists global symbol names and values. However, if you specify /CROSS_REFERENCE, symbol cross reference appears instead, listing symbol names values, defining modules, and referring modules.
Symbols by value	Full	Hexadecimal symbol values and names of symbols with those values
Image synopsis	All	Statistics and other information about the output image
Link run statistics	All	Statistics about the link run that created the image

Thus, a default or brief map contains information only about modules and shareable images that are explicitly included as input in the linking operation, that is, modules or shareable images that are specified in the command string or in an options file. Information about modules or shareable images included as a result of the linker's search of default libraries is not included in the default or brief map.

A full map, on the other hand, contains information about all modules and shareable images included in the linking operation, both those explicitly specified and those implicitly included as a result of the linker's search of default libraries.

4.3 Example of a Full Map

This section provides an annotated illustration of a full image map. As previously noted, a full map contains eight image map sections. These sections are shown in the order in which the linker generates them. Brief and default maps do not have all of these sections, but the sections that they do have are in the order shown here.

This map was generated by the following LINK command, which appears in the shareable image example in Section 3.5.1:

```
$ LINK/MAP/FULL MAIN1,SYS$INPUT/OPTIONS
```

The options file SYS$INPUT contains the following line:

```
A1/SHARE,B1/SHARE
```

To the casual reader, the full map shown here may be taken as a representative example of the format and content of any full map. However, the sophisticated reader may want to take advantage of the fact that the source code, compile commands, and link commands that preceded the generation of this map are available for study in Section 3.5.1.

Further, careful study of this map will illustrate much of the information presented elsewhere in this manual. For example, the linker's clustering algorithm, discussed in Section 5, is illustrated in the Image Section Synopsis, enabling the reader to see how the linker arranges user-defined clusters (defined by the COLLECT= and CLUSTER= options), clusters it creates by default (when shareable images are extracted from IMAGELIB), and the default cluster.

Headings and items in each illustration are explained only if they are not self-explanatory.

All numbers are in hexadecimal radix unless they are followed by a period (.), in which case they are in decimal radix.

```
                                                         1-AUG-1984 18:14        VAX-11 Linker V3A-18          Page    1

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

Module Name    Ident         Bytes      File                           Creation Date      Creator
-----------    -----         -----      -----                          -------------      -------
A1             0                 0 DISK$STARWORK03:[LEAGUE]A1.EXE;1       1-AUG-1984 18:14  VAX-11 Linker V3A-18
B1             0                 0 DISK$STARWORK03:[LEAGUE]B1.EXE;1       1-AUG-1984 18:14  VAX-11 Linker V3A-18
MAIN1          V03-001         183 DISK$STARWORK03:[LEAGUE]MAIN1.OBJ;1    1-AUG-1984 18:14  VAX/VMS Macro V04-00
SYS$P1_VECTOR  V03-041           0 SYS$COMMON:[SYSLIB]STARLET.OLB;2      30-JUL-1984 23:04  VAX/VMS Macro V04-00
LIBRTL         V04-FT2           0 SYS$COMMON:[SYSLIB]LIBRTL.EXE;1       31-JUL-1984 03:10  VAX-11 Linker V3A-18
```

The Module Name column contains the name of each object module in the order in which it is processed by the linker. 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.

The ident column contains identification information for object modules. This information is taken from the .IDENT field in the object module header. The ident for shareable images consists of the file type and version number. For example, the ident for the shareable image A1 is the file type EXE and Version number 1.

The Byte column contains the number of bytes that the object module contributes to the image.

The File column shows the device, directory, and file containing the object module.

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

Module Name                   Number  Module Name                  Number  Module Name                  Number
-----------                   ------  -----------                  ------  -----------                  ------
MAIN1                              1
```

The module relocatable reference synopsis section contains information to enable you to locate .ADDRESS directives in the event that you want to remove these directives. Removing .ADDRESS directives reduces linker and image activator processing time.

If the linker is creating a shareable image, the Module Name column shows the name of each object module containing at least one .ADDRESS directive. If the linker is creating an executable or system image, the Module Name column shows the name of each object module containing at least one .ADDRESS reference to a shareable image.

The Number column lists the number of .ADDRESS occurrences found in the object module whose name appears in the Module Name column.

```
DISK$STARWORK03:[LEAGUE]MAIN1.EXE;1                        1-AUG-1984 18:14       VAX-11 Linker V3A-18          Page    2
                                                 +------------------------+
                                                 ! Image Section Synopsis !
                                                 +------------------------+
   Cluster       Type Pages   Base Addr  Disk VBN PFC Protection and Paging       Global Sec. Name  Match      Majorid   Minorid
   -------       ---- -----   ---------  -------- --- --------------------       ----------------  -----      -------   -------
A1                  3     1    00000000-R        0   0 READ ONLY                    A1_001          LESS/EQUAL        1         0
                    3     1    00000200-R        0   0 READ ONLY                    A1_002          LESS/EQUAL        1         0
                    2     1    00000400-R        0   0 READ WRITE   FIXUP VECTORS A1_003          LESS/EQUAL        1         0
B1                  3     1    00000000-R        0   0 READ ONLY                    B1_001          EQUAL           111   3182177
                    3     1    00000200-R        0   0 READ ONLY                    B1_002          EQUAL           111   3182177
                    2     1    00000400-R        0   0 READ WRITE   FIXUP VECTORS B1_003          EQUAL           111   3182177
DEFAULT_CLUSTER     0     1    00000200          2   0 READ WRITE   COPY ON REF
                    0     1    00000400          3   0 READ ONLY
                    0     1    00000600          4   0 READ WRITE   FIXUP VECTORS
                  253    20    7FFFD800          0   0 READ WRITE DEMAND ZERO
LIBRTL              3   111    00000000-R        0   0 READ ONLY                    LIBRTL_001      LESS/EQUAL        1        11
                    4     1    0000DE00-R        0   0 READ WRITE   COPY ON REF    LIBRTL_002      LESS/EQUAL        1        11

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

The Cluster column shows the name of each cluster in the order in which it is processed by the linker.

The Type column contains information of interest only to the linker and image activator with one exception—type 253 designates the image section that is the user stack.

The Pages column contains the length in pages of each image section. For example, cluster A1 has 3 image sections, each of which is 1 page long.

The Base Address column contains the base address assigned to the image section. Note that if the cluster is relocatable, the image activator assigns the base address to the cluster. In this case, the base address entry for each image section in the cluster has the letter *R* appended to it, indicating that the base address entry is to be interpreted as an offset to be added to the cluster base address assigned by the image activator.

The Disk VBN (virtual block number) column contains the virtual block number of the image file on disk wherein the image section resides. The number zero (0) indicates that the image section is not in the image file.

The PFC (page fault cluster) column indicates how many pages will be read into memory by VAX/VMS when a page fault occurs for that image section. The number zero (0) indicates that VAX/VMS memory management, rather than the linker, determines this value.

The Protection and Paging column shows the protection applied to each image section. The term FIXUP VECTORS indicates that the corresponding image section is a fix-up image section. The term DEMAND ZERO indicates that the image section is a demand-zero image section. The term COPY ON REF indicates that the image section is a copy-on-reference image section. Since a copy-on-reference image section is readable, writeable, but not shareable, each process gets a private copy of it.

The Global Section Name column contains the name assigned by the linker to each shareable image section.

The Match, Majorid, and Minorid columns contain global section match information for shareable image sections. See the explanation of the GSMATCH= option in Section 3.3 for detailed information. Note that since the GSMATCH= option was not specified *in the creation of shareable image* B1 (cluster B1), default values were assigned.

```
DISK$STARWORK03:[LEAGUE]MAIN1.EXE;1                          1-AUG-1984 18:14        VAX-11 Linker V3A-18            Page    3
                                                +--------------------------+
                                                ! Program Section Synopsis !
                                                +--------------------------+
Psect Name      Module Name       Base      End           Length            Align             Attributes
----------      -----------       ----      ---           ------            -----             ----------
$DATA$                            00000200 00000203 00000004 (          4.) LONG 2 NOPIC,USR,CON,REL,LCL,NOSHR,NOEXE,  RD,  WRT,NOVEC
                MAIN1             00000200 00000203 00000004 (          4.) LONG 2
$CODE$                            00000400 000004B2 000000B3 (        179.) LONG 2 NOPIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT,NOVEC
                MAIN1             00000400 000004B2 000000B3 (        179.) LONG 2
```

The Psect Name column lists the name of each program section in the image, in increasing order of the base virtual address allocated.

Information about the program section as a whole may be found on the same line as the program section name by reading across the page. For example, under the Base and End columns are the starting and ending virtual addresses of the program section; under the Length column is the total length; under the Align column is the alignment (see Section A.3.1 for information on how to interpret the number shown); and under the Attributes column are the attributes.

The Module Name column lists the name(s) of modules that contribute to the program section whose name appears on the line directly above in the Psect Name column. Information about a particular module's contribution to a program section may be found by reading across the page, as described above for the program section as a whole.

```
DISK$STARWORK03:[LEAGUE]MAIN1.EXE;1                          1-AUG-1984 18:14        VAX-11 Linker V3A-18            Page    4
                                                     +-----------------+
                                                     ! Symbols By Name !
                                                     +-----------------+
Symbol          Value              Symbol          Value              Symbol          Value              Symbol          Value
------          -----              ------          -----              ------          -----              ------          -----
A               00000648-RX
B               00000654-RX
LIB$GET_VM      00000668-RX
LIB$PUT_OUTPUT  00000660-RX
LIB$SIGNAL      00000664-RX
START           00000404-R
SYS$FAO         7FFEDF50
SYS$IMGSTA      7FFEDF68
```

The symbols by name section is replaced by the Symbol Cross Reference section if the /CROSS_REFERENCE qualifier is specified in the LINK command.

The Symbols column lists the names of each global symbol in the image in alphabetical order.

The Value column lists the hexadecimal values of each global symbol in the image, as well as additional information about the nature of the symbol. For example, the letter *R* appended to the symbol value designates a relocatable symbol; the letter *X* designates an external symbol, that is, one defined in another image; the letter *U* designates a universal symbol; and the asterisk (*) designates an undefined symbol. Note that the linker assigns a value of zero (0) to any undefined symbol.

```
DISK$STARWORK03:[LEAGUE]MAIN1.EXE;1                          1-AUG-1984 18:14        VAX-11 Linker V3A-18            Page    5
                                                     +-------------------+
                                                     ! Symbols By Value !
                                                     +-------------------+
Value                                        Symbols...
-----                                        ----------
00000404     R-START
00000648    RX-A
00000654    RX-B
00000660    RX-LIB$PUT_OUTPUT
00000664    RX-LIB$SIGNAL
00000668    RX-LIB$GET_VM
7FFEDF50       SYS$FAO
7FFEDF68       SYS$IMGSTA
```

```
  Key for special characters above:
+-------------------+
! *  - Undefined    !
! U  - Universal    !
! R  - Relocatable  !
! X  - External     !
+-------------------+
```

The Value column lists the hexadecimal values of each global symbol in ascending numerical order.

The Symbols column lists the corresponding names of the global symbols whose values appear in the Value column, as well as information about the nature of each symbol (as described above in the symbols by name section).

```
DISK$STARWORK03:[LEAGUE]MAIN1.EXE;1                     1-AUG-1984 18:14       VAX-11 Linker V3A-18          Page    6
                                              +----------------+
                                              ! Image Synopsis !
                                              +----------------+

Virtual memory allocated:                      00000200 000007FF 00000600 (1536. bytes, 3. pages)
Stack size:                                          20. pages
Image header virtual block limits:                    1.          1. (     1. block)
Image binary virtual block limits:                    2.          4. (     3. blocks)
Image name and identification:                 MAIN1 V03-001
Number of files:                                      7.
Number of modules:                                    5.
Number of program sections:                           8.
Number of global symbols:                           250.
Number of image sections:                            12.
User transfer address:                         00000404
Debugger transfer address:                     7FFEDF68
Number of address fixups:                             1.
Number of code references to shareable images:        5.
Image type:                                    EXECUTABLE.
Map format:                                    FULL in file DISK$STARWORK03:[LEAGUE]MAIN1.MAP;1
Estimated map length:                          54. blocks
```

The image synopsis section contains miscellaneous information about the image, most of which is self-explanatory.

The virtual memory allocated line lists the base and ending addresses of the image, as well as the total length of memory allocated expressed in hexadecimal (and in parentheses in decimal bytes and decimal pages).

The number of code references to shareable images section may also be interpreted as the number of external references.

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

Performance Indicators                         Page Faults CPU Time Elapsed Time
----------------------                         ----------- -------- ------------
    Command processing:                                107 00:00:00.35 00:00:01.16
    Pass 1:                                            147 00:00:00.60 00:00:01.36
    Allocation/Relocation:                              55 00:00:00.15 00:00:00.50
    Pass 2:                                             48 00:00:00.26 00:00:00.72
    Map data after object module synopsis:              22 00:00:00.13 00:00:00.13
    Symbol table output:                                 5 00:00:00.02 00:00:00.11
Total run values:                                      384 00:00:01.51 00:00:03.98
Using a working set limited to 900 pages and 75 pages of data storage (excluding image)

Total number object records read (both passes):   104
    of which 19 were in libraries and 2 were DEBUG data records containing 74 bytes
64 bytes of DEBUG data were written,starting at VBN 5 with 1 blocks allocated

Number of modules extracted explicitly            = 0
    with 1 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 MAIN1,SYS$INPUT/OPTIONS
```

The link run statistics section contains miscellaneous statistical information about the linking operation, most of which is self-explanatory.

The last item printed is the command string, that is, all qualifiers and input files specified in the LINK command, including the content of any specified options file(s). Note however that this information is printed only in a full map.

5 Linker Operations

This section discusses in detail the operations performed by the linker in creating an image.

The first section briefly discusses each of the three types of image that the linker produces, focusing in particular on the differences in the linker operations involved in their creation.

The second section discusses in detail the content and format of input processed by the linker in creating an image, focusing in particular on program sections.

The third section describes linker operations in sequential order to provide some insight into the linker's processing algorithm.

5.1 Types of Image

The linker produces three types of image: executable, shareable, and system. The /EXECUTABLE qualifier in the LINK command directs the linker to produce an executable image; the /SHAREABLE qualifier, a shareable image; and the /SYSTEM qualifier, a system image. When none of the above qualifiers is specified, the linker produces an executable image.

5.1.1 Executable Image

An executable image, the most common type of image, may be executed by the RUN command.

An executable image cannot be linked with other images, though the modules that make it up may be relinked in different combinations or with other modules to produce another executable image.

An executable image may include one or more shareable images. Shareable images are included in an executable image when they are specified as input or when they are extracted from libraries to resolve undefined symbols.

The linker's processing algorithm, described in this section, is essentially a description of how the linker creates executable images, since shareable images and system images are special cases.

5.1.2 Shareable Image

A shareable image differs from an executable image in the following ways:

- A shareable image is not intended to be directly executed by the RUN command. To be executed, a shareable image should be included as input in the creation of an executable image, which, when it executes, may cause the shareable image to execute.
- Appended to the end of each shareable image is a symbol table, which is itself an object module. The linker uses this symbol table when it resolves undefined symbols in object modules with which the shareable image is linked.

When a shareable image is created, it is the product of a linking operation. After creation, however, a shareable image serves only as input in another linking operation, namely, in the creation of an executable image.

Unless otherwise stated, discussion of shareable images in this section pertains to the use of shareable images as input in linking operations. Section 3, Shareable Images, deals extensively with shareable images as the output of linking operations.

5.1.3 System Image

A system image is intended for stand-alone operation on the VAX hardware. That is, it does not run under the control of the VAX/VMS operating system.

The content and format of a system image differs from that of shareable or executable images. Specifically, a system image does not have:

- An image header, unless the /HEADER qualifier is specified
- Debugger data
- Symbol tables

Memory allocation for a system image differs from memory allocation for shareable or executable images as described in this section. It is much simpler. For a system image, the linker allocates memory to program sections in alphabetical order by program section name, taking into account the following factors:

1. Program section size
2. Program section alignment
3. Two program section attributes: concatenated or overlaid, and relocatable or absolute.

Much of the discussion in this section does not apply to the creation of system images.

5.2 Input to the Linker

The linker deals primarily with two forms of input, object modules and shareable images. This section discusses the content and format of this input insofar as is necessary to understand the subsequent discussion of the linker's processing algorithm.

Library files can also be specified as input in a linking operation. However, since libraries contain either object modules or shareable images, the input derived from libraries must ultimately be either object modules or shareable images. Consequently, library files are not discussed separately in this section.

Symbol table files can also be specified as input in a linking operation. Symbol table files are structurally similar to object files. They contain a subset of the records contained in object files, namely, HDR, GSD, and EOM records. For this reason, symbol table files are not discussed separately in this section.

5.2.1 Object Modules

An object module consists of an ordered set of variable-length records of the following types:

- Header record (HDR)
- Global symbol directory record (GSD)
- Text information and relocation record (TIR)
- Debugger information record (DBG)
- Traceback information record (TBK)
- End of module record (EOM)

Each object module has a HDR record, which must appear first, and an EOM record, which must appear last. Some object modules also contain some or all of the other records, which may appear in any order so long as they are not first or last.

These records contain both data and commands that tell the linker how to operate on the data. The subsequent discussion of the linker's processing algorithm mentions each of these records in the context in which they are processed by the linker. Appendix A, VAX Object Language, describes the format and content of these records in great detail for the benefit of compiler writers and others who create object modules for input to the linker.

As an aid in understanding how the linker generates image sections, the following subsections present information relating to program section definition (specifically, program section name, size, alignment, and attributes). This information is specified to the linker by language processors in the program section definition (PSC) subrecord of the GSD record, though you may modify program section attributes at link time by using the PSECT_ATTR= option.

5.2.1.1 Program Section Name

The program section name is an ASCII character string, 1 through 31 characters in length. Any printable ASCII character is permissible, but the use of the dollar sign ($) is discouraged because of the danger of possible naming conflicts with software supplied by DIGITAL.

Program sections with the same name but from different modules normally must have the same attributes. Any exceptions to this rule are noted in the discussions of specific attributes.

5.2.1.2 Program Section Size

The size field of a program section definition record is a 32-bit count of the number of bytes that the particular module contributes to the program section.

5.2.1.3 Program Section Alignment

The alignment field describes the address boundary at which a module's contribution to the program section will be placed. The alignment is expressed as a number from 0 through 9, representing a power of 2. The base address of the program section is adjusted up to a multiple of that power of two.

In an overlaid program section, all contributing modules must specify the same alignment; otherwise, the linker generates a warning message.

In a concatenated program section, each contributing module can specify a different alignment. The total allocation of the concatenated program section is aligned on a boundary that is a multiple of the highest power of 2 specified by any of the contributing modules.

In addition to the keywords BYTE, WORD, LONG, QUAD, and PAGE, the linker allows specification of program section alignment using the integer values 0, 1, 2, 3, and 9 where 0 corresponds to BYTE, 1 to WORD, and so on.

5.2.1.4 Program Section Attributes

This subsection describes each program section attribute.

Relocatable (REL) and Absolute (ABS)

A relocatable program section is one that the linker can position in virtual memory according to the memory allocation strategy for the type of image being produced.

On the other hand, the linker does not allocate virtual memory for an absolute program section. An absolute program section contains no binary data or code, and appears as if it were based at a virtual address of zero. Absolute program sections are used primarily to define global symbols.

Concatenated (CON) and Overlaid (OVR)

The concatenated and overlaid attributes govern the linker's memory allocation strategy in the case where different modules define a program section of the same name. In this case, each module is said to *contribute* to the program section's definition.

If the program section is defined with the concatenated attribute, the linker places each module's contribution to the program section in contiguous memory addresses.

For example, if PSECTA in MODULE1 and PSECTA in MODULE2 have the concatenated attribute, the linker allocates memory for PSECTA from MODULE1 and then allocates additional memory for PSECTA in MODULE2 in an address space adjacent to that of PSECTA in MODULE1.

Thus, the total size of a program section with the concatenated attribute is the sum of each module's contribution plus any padding allowed for the individual alignments.

If the program section is defined with the overlaid attribute, the linker *overlays* each module's contribution to the program section, that is, it assigns each module's contribution the same base address.

For example, if PSECTA in MODULE1 and PSECTA in MODULE2 have the overlaid attribute, the linker allocates memory to PSECTA in MODULE1 and then allocates the same memory to PSECTA in MODULE2. That is, they share the same address space. Thus, the total size of an overlaid program section is the size of the largest contribution.

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

Local (LCL) and Global Scope (GBL)

The local or global attribute is significant for an image that has more than one cluster. The attribute determines whether program sections with the same name but from modules in different clusters are finally placed in separate clusters (LCL attribute) or in the same cluster (GBL attribute). The memory for a global program section is allocated in the cluster that contains the first contributing module.

VAX BASIC and VAX FORTRAN COMMON areas are implemented with global program sections.

Executability (EXE and NOEXE)

The executability attribute is reserved. The current version of the linker takes this attribute into account in only two ways:

- Error-checking of the image transfer address. The linker issues a diagnostic message if a program transfer address is defined in a nonexecutable program section.
- Sorting of program sections into image sections. Executable program sections in executable and shareable images are placed in image sections separate from program sections that are not executable.

Writeability (WRT and NOWRT)

The writeability attribute determines whether the program section contents will be protected against modification when the image is executed. If the program section has the writeable attribute, its contents may be modified during program execution. If the program section has the nonwriteable attribute, an access violation occurs if an attempt is made to modify its contents.

For executable and shareable images, writeable and nonwriteable program sections are placed in different image sections. For system images, this attribute is ignored, since by definition the VAX/VMS system is not normally in control of the memory management of a system image.

Readability (RD and NORD)

The readability attribute is reserved for possible future implementation.

Position Independence (PIC and NOPIC)

The position independence attribute identifies whether or not a program section will execute correctly anywhere in the virtual address space.

A program section with the PIC attribute will execute correctly no matter what base virtual address it has been allocated, whereas a program section with the NOPIC attribute will not.

The linker considers this attribute only when the image being produced is a shareable image, in which case it sorts program sections with this attribute (among others) into separate image sections.

See Section 3.2.2 for a complete explanation of position-independence.

Shareability (SHR and NOSHR)

The shareability attribute determines whether or not a program section may be shared among more than one process. The linker considers this attribute only when it is producing a shareable image, in which case it sorts program sections with this attribute (among others) into separate image sections.

This attribute, together with the WRT and NOWRT attributes, affects whether or not an image section can be simultaneously executed by more than one process.

User (USR) and Library (LIB)

The user or library attribute is reserved for possible future implementation. It should be set to USR to guarantee future compatibility.

Protection (VEC and NOVEC)

The protection attribute has two characteristics. The VEC attribute specifies that the program section contains privileged change-mode vectors or message vectors, whereas the NOVEC attribute specifies that it does not. Program sections with the VEC attribute are automatically protected in shareable images.

5.2.2 Shareable Images

Each shareable image exists in a shareable image file. This file consists of an image header, image sections, and a symbol table.

The processing of an input shareable image is much simpler than the processing of an object module. The linker does not need to resolve symbolic references within the shareable image, to sort program sections into image sections, or to initialize the image section contents.

Instead, the major work in processing a shareable image is that of resolving symbolic references between it and the other input object modules. To do this, the linker searches its global symbol table for undefined global symbols and, for each one it finds, looks in the shareable image's symbol table for a match. When it finds a match, it inserts the definition of that symbol as found in the shareable image symbol table into its global symbol table. Then, after Pass 1, when the linker allocates virtual memory, it replaces each occurrence of a symbol with the equivalent location or value as defined in its global symbol table.

The linker creates a new cluster for each input shareable image it encounters. It places each new shareable image cluster onto the cluster list at the time of cluster creation. Therefore, the linker's cluster list may contain object file clusters, followed by shareable image clusters, followed by other object file clusters, followed by other shareable image clusters, and so on.

Virtual memory allocation for a shareable image cluster takes place either at link time or at run time. If the shareable image cluster contains a based shareable image (a shareable image created using the BASE= option), the linker allocates virtual memory for the shareable image after Pass 1. A based shareable image is positioned at the virtual address specified in the BASE= option.

On the other hand, if the shareable image cluster contains a nonbased (or position-independent) shareable image, the image activator allocates virtual memory for the shareable image at run time. As a group, therefore, position-independent shareable image clusters will appear in the highest-addressed virtual address space of the final image. Further, since the linker processes clusters in order of their appearance on the cluster list, position-independent shareable image clusters processed first will be given lower virtual addresses than position-independent shareable image clusters processed subsequently. Thus, a shareable image explicitly specified for inclusion in a linking operation will always have a lower virtual address assignment than a shareable image included by default from SYS$LIBRARY:IMAGELIB.OLB, the system shareable image library.

In allocating virtual memory, the linker needs only to read the image header of the shareable image to determine its memory requirements since the image header contains a list of image section descriptors (ISD) that describe each image section.

5.3 The Linker's Processing Algorithm

This section describes in chronological order each major step in the linking operation.

The linker reads (passes) through its input two times. However, processing of input also occurs before its first pass, in between its first and second pass, and after its second pass.

Each major step in the total processing algorithm is described in the following sections. These major steps are named, in chronological order:

1. Command processing
2. After command processing
3. Pass 1
4. After Pass 1

5. Pass 2
6. After Pass 2

5.3.1 Command Processing

This section describes how the linker sets up its file and cluster data structures using the input specified in the LINK command. Since the linker associates each input file with a cluster, this section is in effect a description of the linker's clustering algorithm. Note however that, depending on the command input, the linker may create additional clusters during Pass 1.

By understanding the linker's clustering algorithm, a programmer will be able to specify input to the linker so as to cause the linker to put particular sections of a program adjacent to (or close to) other sections in virtual memory. If these sections frequently reference one another, putting them close together in virtual memory will improve the performance of the program in the VAX/VMS operating environment.

When the LINK command is first entered, the command interpreter calls the linker, which in turn calls back to the command interpreter to obtain descriptors for the image, symbol table, map, and input files. The linker then stores the LINK command string for future printing in the image map.

As the linker opens each input file, it allocates a file descriptor block (FDB) for the file, and links the FDB onto a cluster descriptor. This is how the linker puts a file into a cluster.

The linker keeps a record of each cluster descriptor in a cluster descriptor list so that it knows how many clusters it creates, the order in which it creates them, and the particular cluster that it is currently processing. During Pass 1 and Pass 2, the linker refers to this list to determine the order in which to process clusters (and therefore files).

The following points summarize the linker's clustering algorithm during command processing:

- The linker creates a new cluster for each shareable image file or CLUSTER= option specified in an options file.
- When the command input does not include an options file containing either a shareable image file or a CLUSTER= option, the linker puts all input files in the default cluster.
- The linker puts the default cluster onto the end of the cluster list.

In Pass 1 and Pass 2, the linker processes clusters in the order of their appearance on the cluster list. Consequently, files are processed by cluster, not by the order of their appearance in the command string. This means that files put in the default cluster will be processed after files put in other clusters even though they were specified first in the command string.

In processing the command string, the linker processes the first file specified, then the next file specified, and so on, until it has processed all files. If it encounters an options file, it reads the file and processes its contents, and then proceeds to the next file in the command string.

Input files are specified either in the command string or in an options file. The following subsections discuss how the linker processes input files specified in these two ways.

5.3.1.1 Processing Nonoptions Files

Nonoptions files are library or object files that are specified in the command string.

If the input file is a library file specified with the /LIBRARY qualifier, the linker puts the entire file into the default cluster.

If the input file is a library file specified with the /INCLUDE qualifier, the linker puts the entire file in the default cluster and makes note of the modules specified for extraction from that library.

If the file is an object file, the linker puts the file into the default cluster.

In sum, the linker puts files specified in the command string in the default cluster.

5.3.1.2 Processing Options Files

If the file is an options file, the linker reads and processes each line in the file until it reaches the end of the file. How it processes each line depends on what the line contains. There are three possibilities, each of which is further discussed later in this section:

1. If the line contains a file specification, the linker puts the file into a cluster: which cluster depends on the kind of file it is.
2. If the line contains a legal link option, the linker performs the action specified. The CLUSTER= option and the COLLECT= option, however, require some additional processing.
3. If the line does not contain either a legal link option or a file specification, the linker issues an error message and aborts the linking operation.

If the line contains a file specified with the /SHARE qualifier, that is, a shareable image file, the linker puts the file in a new cluster.

If the line contains a file specified with the /LIBRARY and/or the /INCLUDE qualifier, that is, a library file, the linker puts the file in the default cluster.

If the line contains an object file specification, the linker puts the file in the default cluster.

If the line contains a CLUSTER= option, the linker creates a new cluster and puts the specified files in that cluster. It will, however, create more than one cluster in the following cases:

- If more than one shareable image file is specified in a single CLUSTER= option, the linker creates a new cluster for each specified shareable image.
- If a shareable image file appears in the CLUSTER= option together with any other file that is not a shareable image file (hereafter referred to as a user file), the linker puts each shareable image file in a new cluster and puts any user files in another new cluster.

The CLUSTER= option should only specify either (1) a single shareable image file or (2) one or more user (object or object library) files.

To illustrate, the following line in an options file violates this rule and will result in a warning message:

```
CLUSTER=WRONG,,,PETER/SHARE,NINA.OBJ
```

In this example, the linker puts PETER/SHARE in the new cluster WRONG and puts NINA.OBJ in another new cluster. If on the other hand, NINA.OBJ preceded PETER/SHARE on the option line, the linker would put NINA.OBJ in the new cluster WRONG and put PETER/SHARE in another new cluster.

If the line contains the COLLECT= option, the linker modifies its data base to prepare for possible cluster creation during Pass 1. The linker cannot completely process this option, which specifies that the named program sections be put in the named cluster, because the linker does not process the contents of input files (which includes program sections) until Pass 1.

5.3.1.3 Considerations in Specifying Input

Because the linker processes clusters sequentially in Pass 1 and Pass 2, care is needed when specifying input.

Consider, for example, the following command string.

```
$ LINK FRED,PEOPLE/LIB,SYS$INPUT/OPTION
```

The options file SYS$INPUT contains the following.

```
CLUSTER=MORE,,,MEN/LIB
```

If the library MEN/LIB is expected to resolve undefined symbols in FRED, the scheme will fail because the linker puts FRED in the default cluster and MEN/LIB in the cluster MORE, which precedes the default cluster on the list. Thus, the linker processes MEN/LIB before it processes FRED and therefore cannot resolve undefined symbols in FRED.

The following command string circumvents this problem:

```
$ LINK/EXE=FRED/MAP=FRED/FULL SYS$INPUT/OPTION
```

The options file SYS$INPUT contains:

```
CLUSTER=MAIN,,,FRED,PEOPLE/LIB
CLUSTER=MORE,,,MEN/LIB
```

Now the linker puts FRED in the cluster MAIN and MEN/LIB in the cluster MORE, which follows MAIN in the list. Thus, the linker processes FRED before MEN/LIB and therefore can resolve undefined symbols in FRED.

5.3.2 After Command Processing

After the linker has processed each input file as described in the preceding section, it does some further processing if a COLLECT= option was specified in an options file. The COLLECT= option directs the linker to put the specified program sections in the specified cluster.

The linker processes the COLLECT= option as follows:

1. It creates a new cluster if the cluster does not already exist
2. It gives each specified program section the global (GBL) attribute to enable a global search for the definition of that program section in Pass 1
3. It enters each program section in the specified cluster, specifically, in the program section descriptor list of that cluster

When using the COLLECT= option, care is needed to insure that the cluster named in the option is linked onto the cluster list before those clusters that contain the program section definitions.

To illustrate, consider the following options file wherein the COLLECT= option is intended to gather selected program sections from the file RACE into a new cluster GAS:

```
CLUSTER=WHEELS,,,RACE,CRUISE
COLLECT=GAS,,,RACEPSECT1,RACEPSECT2
```

In this example, the cluster WHEELS will precede the cluster GAS on the cluster list. The linker processes WHEELS first during Pass 1 and puts the program sections RACEPSECT1 and RACEPSECT2 in the cluster WHEELS, instead of in the cluster GAS.

Note that putting the second line before the first in the options file does not solve the problem because, although the linker now processes the cluster GAS before the cluster WHEELS, it will be unable to locate definitions of the specified program sections when it is processing the cluster GAS (since it only looks for definitions in previous, not subsequent, clusters). The specified program sections will ultimately be put in the cluster WHEELS.

One way to solve this problem is to specify an empty cluster GAS before the cluster WHEELS, as follows:

```
CLUSTER=GAS
CLUSTER=WHEELS,,,RACE,CRUISE
COLLECT=GAS,,,RACEPSECT1,RACEPSECT2
```

5.3.3 Pass 1

The next step in the linking operation is Pass 1. The linker now reads and processes the contents of the input files with the aim of building its global symbol table (GST) and program section table (PST), and resolving undefined symbols.

The linker builds its GST by reading various subrecords in global symbol directory (GSD) records. If it encounters a library file, the linker also refers to its GST to see if it can resolve symbols that are so far undefined. Thus, during Pass 1, the linker both stores information in and extracts information from its GST.

The linker builds its PST by reading program section definition (PSC) subrecords in GSD records. Each PSC subrecord describes a program section, which in turn describes the memory requirements of a section of an object module. Each program section represents an area of memory that has a name, a length, and a series of attributes, which describe the intended or permitted usage of that portion of memory. The linker will use the PST after Pass 1 to generate image sections for which it will allocate virtual memory.

The processing of files and clusters takes place in the following order. The linker reads and processes 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.

Once again, how the linker processes each file depends on whether the file is an object file, library, or shareable image. The following subsections discuss these in turn.

5.3.3.1 Processing Object Files

If the file is an object file, the linker reads the records in the file and processes each record as described below. Note that in the process of reading records, the linker checks to see that they are in the correct order and that all required records are present.

1 For the main header (HDR) record (there is only one per object module), the linker creates an object module descriptor into which it stores such information as the name and location of the object module and its ident. The linker then links the object module descriptor onto the object module descriptor list.

2 For each global symbol directory (GSD) record, the linker checks to see what subrecords it contains, since a GSD record may have multiple subrecords.

   - If the subrecord type is program section definition (PSC), the linker must first determine the scope of the search for a definition of the program section by examining its attributes, in particular, the GBL attribute. Since two linker options (PSECT_ATTR= and COLLECT=) change the attributes of a program section, the linker checks to see whether either of these options were specified. If so, it modifies the attributes present in the PSC subrecord in accordance with the attributes specified in the option(s). If not, it leaves the attributes in the PSC subrecord unchanged.

     If the program section does not have the GBL attribute, the linker searches for a previous definition of the program section in the program section descriptor list of only the current cluster, that is, the cluster it is currently processing. If it finds a previous definition, the linker checks for conflicting attributes and, if there are none, uses that definition. Otherwise, it defines the program section in the current cluster by extracting information about the program section out of the PSC subrecord and putting it into the program section descriptor list.

     If a program section has the GBL attribute, the linker searches for a previous definition of the program section in the program section descriptor lists of the first cluster, the second, the next, and so on, until it looks finally in that of the current cluster. If the linker finds a previous definition, it checks for conflicting attributes and, if there are none, uses that definition. Otherwise, the linker defines the program section in the current cluster.

     After it has defined the program section, the linker creates the module program section contribution block (MPC). The MPC keeps a record, for future use in the map file, of how much data this module contributes to the program section. The MPC is pointed to by the object module descriptor, which the linker created when it read the HDR record.

     The linker then checks that the number of program sections for this module does not exceed the allowable limit. Then, if the program section has the overlaid attribute OVR, it calculates the maximum length of the program section (which is the length of the longest contribution to that program section).

   - If the subrecord type is global symbol specification (SYM), entry point symbol and mask definition (EPM), or procedure and formal argument definition (PRO), the linker reads all of the information contained in the record and uses it to build its global symbol table (GST). The GST contains a list of the names of all global symbols in the image, together with other information such as its value, where it is defined, and so on. The linker also uses this information for the map file.

3 The linker ignores text information and relocation (TIR) records in Pass 1, reserving these for processing in Pass 2.

4 For debugger information (DBG) and traceback information (TBK) records, the linker calculates how big a buffer is required to store these records for future use.

5 For the end of module (EOM) record, the linker checks to see whether a transfer address is defined and if so makes a note of it.

5.3.3.2 Processing Other Files

If the file is a library file that is specified with the /INCLUDE qualifier, the linker first determines whether the library file is an object module library or a shareable image library.

If the library is an object library, the linker extracts the specified object modules and processes them like it does object files.

If the library is a shareable image library, the linker extracts the specified modules, each of which is a shareable image, and processes them like it does shareable image files, described below.

If the file is a library file that is specified with the /LIBRARY qualifier, the linker first determines whether the library is an object module library or a shareable image library. In either case, since the library will be used to resolve undefined symbols, the linker looks for a match between undefined symbols in its GST and symbols in the library symbol table. Note that the symbols in a shareable image library symbol table are called universal symbols.

If the library is an object module library, the linker extracts each module that contains a symbol definition for an undefined symbol and processes it like an object file.

If the library is a shareable image library and the linker has determined that a particular module in the library contains a symbol definition that it needs, the linker must locate that module (remember that each module in a shareable image library is itself a shareable image file). To locate the module, the linker attempts to open a file with the file name of the module in the device and directory of the library file. If this attempt fails, the linker then uses the name of the module with the device and directory name of the system default shareable image library (SYS$LIBRARY:). When the linker locates the module, it processes it like a shareable image file.

If the file is a shareable image file, that is, specified with the /SHARE qualifier in an options file, the linker reads the image header to obtain the memory requirements of the image. Then it processes the shareable image's symbol table, which is pointed to by the image header. The symbol table of a shareable image is itself an object module containing HDR, GSD, and EOM records. The linker then processes this symbol table just like an object module.

5.3.3.3 Processing Default Libraries

After it processes all files in all clusters, the linker checks its GST for undefined symbols. If there are undefined symbols, the linker processes the following default libraries in the stated order:

1. Default user libraries, if any, provided that the /NOUSERLIBRARY qualifier was not specified in the LINK command
2. The system default shareable image library SYS$LIBRARY:IMAGELIB.OLB, provided that neither the /NOSYSSHR nor the /NOSYSLIB qualifiers were specified in the LINK command
3. The system default object module library SYS$LIBRARY:STARLET.OLB, provided that the /NOSYSLIB qualifier was not specified in the LINK command

The linker processes these libraries by looking for matches between undefined symbols in its GST and symbols in the library symbol table. For each match that it finds, it extracts the object module or shareable image that contains the symbol definition and processes it as previously described.

Remember that for any user default library that is a shareable image library and for IMAGELIB.OLB, the linker locates any needed module by looking first in the device and directory of the library file and then, if that search fails, in the device and directory SYS$LIBRARY:. Note too that IMAGELIB.OLB is in SYS$LIBRARY:.

The linker puts modules extracted from any user default library that is an object library and from STARLET.OLB in the default cluster.

The linker puts modules extracted from IMAGELIB.OLB into a new cluster at the end of the cluster list (after the default cluster). Since all files explicitly specified by the user have been processed at this point, the shareable image is able to resolve undefined symbols from all files in all previous clusters.

After the linker processes default libraries (if necessary), it reports on the terminal and in the map (if a map was specified) all unresolved references to global symbols, providing that at least one of these references is a strong reference (as described in Section 1.4.1.3). Again, if at least one unresolved reference is strong, the linker reports all weak and all strong unresolved references. If all unresolved references are weak, the linker reports nothing.

5.3.4 After Pass 1

By this point, the linker has resolved undefined symbols and has built its PST. Since it has processed all input modules and all library modules that were needed to resolve undefined symbols, the linker now knows how big the final image will be. Its next job is the allocation of virtual memory.

The linker allocates virtual memory on a cluster-by-cluster basis. It proceeds by making two passes through the cluster list. On the first pass, it processes any of the following clusters as it encounters them:

- Based user clusters, that is, clusters assigned a base address by means of the CLUSTER= option (remember that user clusters are clusters that are not shareable image clusters)
- Based shareable image clusters, that is, clusters containing a based shareable image

- Default cluster, if and only if the BASE= option was specified

On the second pass, the linker processes, in order, each user cluster that was not assigned a base address. The linker ignores nonbased (or position-independent) shareable image clusters at this point because these clusters will have virtual memory allocated for them at run time.

The processing of each cluster takes place in three distinct steps, resulting in the complete allocation of virtual memory for that cluster:

1. Generation of image sections
2. Memory allocation for the cluster
3. Relocation of image sections within the cluster

The following subsections discuss each of these steps in turn.

5.3.4.1 Generation of Image Sections

An image section defines the memory requirements of an image or part of an image by means of a number of attributes, derived from the program sections that comprise that image section.

Each image section is described by an image section descriptor (ISD) that contains, among other things, the number of pages in the image section, the starting virtual address of the image section, and the descriptor of a buffer that the linker will use during Pass 2 when it executes TIR records.

To generate an image section, the linker searches the PST of the cluster for program sections that have a particular set of attributes (significant attributes). If it finds a program section with this set of attributes, it generates an image section and puts the program section in that image section. All program sections having this same set of significant attributes are put in the same image section.

The linker then repeats this procedure for all other sets of significant attributes until it has generated a complete set of image sections that contain all program sections in the cluster. Which set of attributes are significant depends on the kind of image being produced:

- For executable images, all combinations of the writeability (WRT and NOWRT), executability (EXE and NOEXE), and protected vector (VEC and NOVEC) attributes are considered.
- For shareable images, all combinations of the writeability, executability, protected vector, position-independence (PIC and NOPIC), and shareability (SHR and NOSHR) attributes are considered.

The linker places program sections in an image section in alphabetical order according to program section name. It then assigns to each program section a virtual address relative to the base address of the image section.

The linker places image sections within the cluster in order according to the particular set of significant attributes that their contained program sections have. It then assigns to each image section a virtual address relative to the base address of the cluster.

Thus, at this point, all image sections have cluster-relative addresses and all program sections have image-section-relative addresses.

Table LINK–4 shows every set of significant attributes that the linker considers in generating image sections, as well as the order in which image sections with these sets of attributes are placed in the cluster. Each line of the table specifies a set of significant attributes, and the order of the lines is the order that the image sections with those sets of attributes are placed in the cluster. The table also shows which sets of attributes are significant for each type of image. For example, the first image section in a cluster for an executable image will contain program sections with the NOWRT, NOEXE, and NOVEC attributes, if in fact program sections with these attributes appeared in the cluster.

5.3.4.2 Memory Allocation for the Cluster

After the linker has generated all image sections, it allocates virtual memory for the cluster.

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 virtual address 200 hexadecimal (for an executable image) or 0 (for a shareable image) 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.

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.)

Since 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.

5.3.4.3 Relocation of Image Sections

When virtual memory has been allocated for each cluster, the linker relocates all image sections within the cluster by adding the beginning virtual address of the cluster (derived from the free virtual memory list) to the offset of each image section into the cluster.

The linker updates the appropriate field in the image section descriptor (ISD) of each image section in the cluster with the now final starting virtual address of the image section.

Table LINK–4 Order of Image Sections in Clusters

Type of Image	Image Section PSECT Attributes				
Executable	NOWRT	NOEXE	—	—	NOVEC
	WRT	NOEXE	—	—	NOVEC
	NOWRT	EXE	—	—	NOVEC
	WRT	EXE	—	—	NOVEC
	NOWRT	NOEXE	—	—	VEC
	WRT	NOEXE	—	—	VEC
	NOWRT	EXE	—	—	VEC
	WRT	EXE	—	—	VEC
Shareable	NOWRT	NOEXE	SHR	NOPIC	NOVEC
	WRT	NOEXE	SHR	NOPIC	NOVEC
	NOWRT	EXE	SHR	NOPIC	NOVEC
	WRT	EXE	SHR	NOPIC	NOVEC
	NOWRT	NOEXE	NOSHR	NOPIC	NOVEC
	WRT	NOEXE	NOSHR	NOPIC	NOVEC
	NOWRT	EXE	NOSHR	NOPIC	NOVEC
	WRT	EXE	NOSHR	NOPIC	NOVEC
	NOWRT	NOEXE	SHR	PIC	NOVEC
	WRT	NOEXE	SHR	PIC	NOVEC
	NOWRT	EXE	SHR	PIC	NOVEC
	WRT	EXE	SHR	PIC	NOVEC
	NOWRT	NOEXE	NOSHR	PIC	NOVEC
	WRT	NOEXE	NOSHR	PIC	NOVEC
	NOWRT	EXE	NOSHR	PIC	NOVEC
	WRT	EXE	NOSHR	PIC	NOVEC
	NOWRT	NOEXE	SHR	NOPIC	VEC
	WRT	NOEXE	SHR	NOPIC	VEC
	NOWRT	EXE	SHR	NOPIC	VEC
	WRT	EXE	SHR	NOPIC	VEC
	NOWRT	NOEXE	NOSHR	NOPIC	VEC
	WRT	NOEXE	NOSHR	NOPIC	VEC
	NOWRT	EXE	NOSHR	NOPIC	VEC
	WRT	EXE	NOSHR	NOPIC	VEC
	NOWRT	NOEXE	SHR	PIC	VEC
	WRT	NOEXE	SHR	PIC	VEC
	NOWRT	EXE	SHR	PIC	VEC
	WRT	EXE	SHR	PIC	VEC
	NOWRT	NOEXE	NOSHR	PIC	VEC
	WRT	NOEXE	NOSHR	PIC	VEC
	NOWRT	EXE	NOSHR	PIC	VEC
	WRT	EXE	NOSHR	PIC	VEC
System	— (only one image section)	—	—	—	—

5.3.5 Pass 2

At this point, the linker has allocated virtual memory for each image section in each cluster in the image. The linker now opens the map file, if a map was requested in the LINK command, and allocates virtual memory for the debug symbol table (DST), unless the /NOTRACE qualifier was specified in the LINK command.

The major work of Pass 2 is the initialization of image sections, that is, the writing of the binary contents of the image sections.

To initialize the image sections, the linker makes another pass through the input to process records in each object module. The linker processes the records in each object module in the following manner:

1. Header (HDR) records are ignored unless a map was requested in the LINK command, in which case the linker copies, to the map file, information contained in these records.
2. Global symbol directory (GSD) records are ignored.
3. Text information and relocation (TIR) records are processed by the linker. These records direct the linker in the initialization of the image section by telling it what to store in the image section buffers.

   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 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 unless the linker executes a store command (with nonzero data) within that image section.
4. Debugger information (DBG) and traceback information (TBK) 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.

   These records contain stack and store object language commands like TIR records, but they store into the debugger symbol table instead of into an image section.
5. End of module (EOM) records are processed. The linker checks that its internal stack has been collapsed to its initial state.

Pass 2 is complete when the linker has processed the records, as described above, for each object module. At this point, the linker has written the binary contents of all image sections to image section buffers in its own address space.

5.3.6 After Pass 2

The linker completes the linking operation with the following three actions:

- Demand-zero compression
- Insertion of the fix-up image section
- Writing of the image file

The following subsections discuss each of these in turn.

5.3.6.1 Demand-Zero Compression

Since 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.

Demand-zero compression is the placing of several uninitialized pages of an image section into another newly created image section (a demand-zero image section). Demand-zero compression is a desirable optimization feature because it reduces the size of the image file and enhances the performance of the program. However, the linker will create demand-zero image sections only if all of the following conditions are met:

- The linker is creating an executable or a shareable image, not a system image.
- The image section contains at least as many uninitialized pages as specified in the DZRO_MIN option. If this option was not specified, the linker uses the default value of 5 pages. However, unitialized copy-on-reference image sections are made demand-zero, even if they consist of fewer pages than DZR0_MIN.
- The total number of image sections in the image is less than the allowable limit (which is either explicitly specified by the ISD_MAX option or implicitly established by the default value).

The linker proceeds by examining each image section in each user cluster, in order, from the first to the last cluster, searching for contiguous uninitialized pages. If the linker finds a collection of uninitialized pages and if all of the above conditions are met, the linker places the uninitialized pages into a new image section by creating another image section descriptor (ISD) and linking it into the ISD list at that point.

Subsequently, when the image is run and a demand-zero page is referenced, VAX/VMS will initialize an allocated page of physical memory with zeros (hence the name *demand-zero*).

5.3.6.2 Insertion of the Fix-Up Image Section

If it is creating an executable image, the linker will always insert a fix-up image section directly following the last image section in the highest-addressed user cluster, that is, the last cluster that is not a shareable image cluster.

If it is creating a nonbased shareable image, the linker will insert a fix-up image section directly following the last image section in the highest-addressed user cluster only if either or both of the following conditions are true:

- One or more nonbased (position-independent) shareable image clusters follow the highest-addressed user cluster. This occurs when additional shareable images were included in the linking operation to resolve undefined symbols.

- One or more .ADDRESS directives were encountered in TIR records.

By means of the fix-up image section, the linker performs the following two functions, each of which is described in detail in subsequent paragraphs:

- 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.
- It processes all general-addressing-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.

If it were not for the image fix-up section, an occurrence of a .ADDRESS directive in a shareable image would make that shareable image position-dependent because the .ADDRESS directive references a fixed address in virtual memory. For example, if the directive .ADDRESS 4055 appears in the instruction stream at address 4032, address 4032 now contains virtual address 4055. As a result, the shareable image can only execute correctly when it is placed in virtual memory at the same address assigned to it by the linker at the time of its creation.

By means of the fix-up image section, the linker stores information about each .ADDRESS directive it encounters and passes this information to the image activator at run time. The image activator can then substitute run-time addresses for the fixed link-time addresses, thus insuring the position-independence of the shareable image.

Consider the occurrence of the .ADDRESS 4055 directive at virtual address 4032, described above. Assume at the time the shareable image was created that its base address was 0 (the default for position-independent shareable images). Assume further that at run time, the image activator assigns that shareable image a base address of 6000. The following describes how the linker, working with the image activator, "fixes up" the .ADDRESS directive:

1. The linker calculates the offset of the directive from the base address of the shareable image (*4032 - 0 = 4032*).
2. The linker calculates the offset of the address being stored by the directive from the base address of the shareable image *(4055 - 0 = 4055)* .
3. The linker passes both offsets to the image activator.
4. The image activator adds the first offset (4032) to the run-time base address (6000) to calculate the run-time address (10032) of the occurrence of the .ADDRESS directive.
5. The image activator adds the second offset (4055) to the run-time base address (6000) to calculate the run-time address (10055) of the value being stored by the .ADDRESS directive.
6. The image activator inserts the run-time value being stored (10055) at the run-time address of the occurrence of the directive (10032).

Now the shareable image can execute correctly at the run-time virtual address assigned to it by the image activator.

The linker "fixes up" a general-addressing-mode code reference, whose target is in a position-independent shareable image, in the following manner:

1 The linker changes the addressing mode of the reference to longword-relative-deferred and changes the target of the reference to a cell within the image fix-up section.

2 The linker loads the cell of the image fix-up section with the offset of the target of the reference from the beginning of the position-independent shareable image.

3 The image activator assigns a base address to the shareable image and adds this address to the cell in the image fix-up section during image activation.

For example, consider the following general-addressing-mode code reference whose target is the square-root routine MTH$SQRT in VMSRTL:

```
CALLG     LIST,G^MTH$SQRT
```

The linker changes the reference to a longword-relative-deferred reference to a cell within the image fix-up section, as follows:

```
CALLG     LIST,@L^sqrt
```

where sqrt is a cell in the image fix-up section.

The linker then loads sqrt with the offset of MTH$SQRT from the beginning of VMSRTL.

When the image is run, the image activator adds the base address of VMSRTL to sqrt, and the linkage is complete.

5.3.6.3 Writing of the Image File

The final step in the linking operation is the writing of information from the linker buffers to the image file on disk. If a map or symbol table file was requested, the linker writes the appropriate information to these files as well.

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

- All image sections to the image file
- The image header to the image file
- The debug symbol table to the image file, unless /NOTRACE was specified in the LINK command
- 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 statistics, which it cannot write until the linking operation completes.)
- The global symbol table to the image file, and also to another separate file if requested in the LINK command
- The link statistics to the map file, if requested in the LINK command

Lastly, the linker closes the image file, and the map and symbol table files if these were requested, and exits.

6 VAX Object Language

This section describes the VAX object language according to DIGITAL software specifications. The object language described is for use by all VAX family software; no subsetting will occur.

The VAX object language describes the contents of object modules to the VAX 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 section is useful primarily to programmers writing compilers or assemblers that must generate object modules acceptable for input to the VAX Linker. These programmers may also find the ANALYZE/OBJECT command in the *VAX/VMS 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 section contains eight 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 provides a detailed description of all subrecords and fields that it contains.

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.

6.1 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. Table LINK–5 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.

Table LINK–5 Types of Module Records

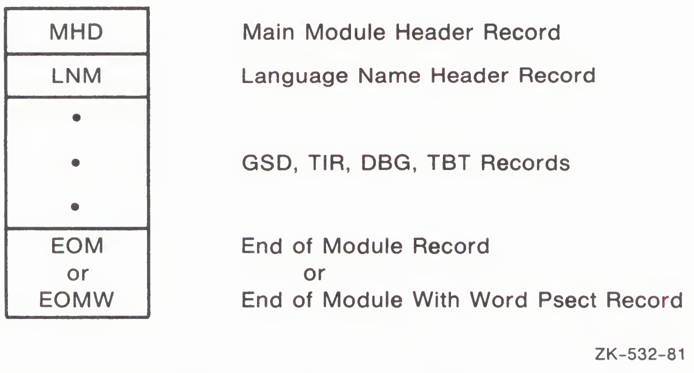
Record Type	Symbol	Code
Header (HDR)	OBJ$C_HDR	0
Global symbol directory (GSD)	OBJ$C_GSD	1
Text information and relocation (TIR)	OBJ$C_TIR	2
End of module (EOM)	OBJ$C_EOM	3
Debugger information (DBG)	OBJ$C_DBG	4
Traceback information (TBT)	OBJ$C_TBT	5
Link option specification (LNK)	OBJ$C_LNK	6
End of module with word psect (EOMW)	OBJ$C_EOMW	7
Reserved for future use by linker		8–100
Reserved always		101–200
Reserved for customer use		201–255

The term "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 of the seven legal record types need not appear in a single object module. However, each object module must contain the following:

1. One (and only one) main module header record (MHD) appearing first in the object module (see Section 6.2.1)
2. One (and only one) language name header record (LNM) appearing second in the object module (see Section 6.2.2)
3. At least one global symbol directory record (GSD)
4. 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 LINK–4 depicts the correct ordering of records within an object module.

Figure LINK–4 Order of Records in an Object Module


MHD
Main Module Header Record
LNM
Language Name Header Record
GSD, TIR, DBG, TBT Records
EOM or EOMW
End of Module Record or End of Module With Word Psect Record
ZK-532-81

If a field is currently ignored by the linker, you must nevertheless 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:

1. 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.
2. 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.

6.2 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 future use, and types 101 to 255 are ignored.

Table LINK–6 displays the types of header records. Column 1 displays the name of the header type, followed by its abbreviation. Column 2 displays its symbolic representation. Column 3 displays its corresponding numerical code.

Table LINK–6 Types of Header Records

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

[1]This record is required by the linker.
[2]This record is currently ignored by the linker.

Table LINK–6 (Cont.) Types of Header Records

Header Type	Symbol	Code
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
Reserved		7–100
Ignored		101–255

[2]This record is currently ignored by the linker.

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 following subsections. 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 *VAX/VMS DCL Dictionary*).

6.2.1 Main Module Header Record (MHD$C_MHD)

The following diagram depicts the Main module header record. The left-hand column displays the name, symbolic representation, and length of each field, where appropriate. The right-hand column displays the value (which may be a symbol) and any needed 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. Since the format of the MHD record will never change, the structure level field was provided so that changes in the format of other records could be made without requiring recompilation of 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 in the current implementation 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, 1–31 bytes for object modules; 1–39 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
Length: variable, 2–32 bytes

This field contains the module version number in standard name format.

CREATION TIME AND DATE
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 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 (this includes hyphens (-), the space, and the colon (:)).

TIME AND DATE OF LAST PATCH
Length: 17 bytes

This field is currently ignored by the linker and should be padded with 17 zeros.

6.2.2 Language Processor Name Header Record (MHD$C_LNM)

The following diagram depicts the language processor name header record:

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
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.

6.2.3 Source Files Header Record (MHD$C_SRC)

The following diagram depicts the source files header record. The contents of this record, though ignored by the linker, is displayed in an object module analysis (see the description of the ANALYZE/OBJECT command in the *VAX/VMS 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
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.

6.2.4 Title Text Header Record (MHD$C_TTL)

The following diagram depicts the title text header record. The contents of this record, though ignored by the linker, is displayed in an object module analysis.

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
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.

6.3 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 LINK–7 displays the types of GSD subrecords. Column 1 displays the name of the GSD subrecord; column 2 displays its symbolic representation; and column 3 displays its corresponding numerical code.

Table LINK–7 Types of GSD Subrecords

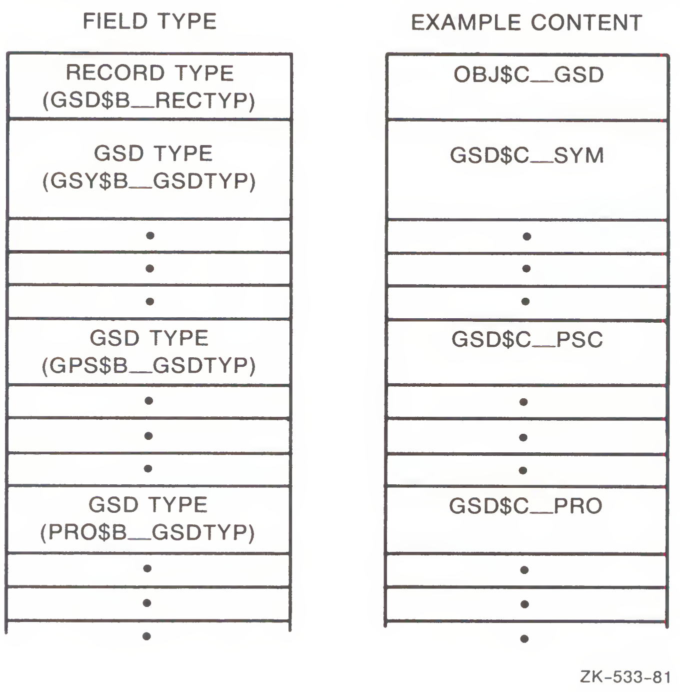
GSD Subrecord	Symbol	Code
Program section definition	GSD$C_PSC	0
Global symbol specification	GSD$C_SYM	1
Entry point symbol and mask definition	GSD$C_EPM	2
Procedure with formal argument definition	GSD$C_PRO	3
Symbol definition with word psect	GSD$C_SYMW	4
Entry point definition with word psect	GSD$C_EPMW	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 LINK–5 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 GSD subrecord therefore begins with the GSD TYPE field.

The following subsections describe the format and content of each GSD subrecord. For each subrecord, the name, length, value, and description of each field is given, where appropriate.

Note that the RECORD TYPE field is not shown in the diagrams in the subsequent subsections. Remember, therefore, that this field must always appear first in the GSD record and that it appears only once, no matter how many GSD subrecords are included in the GSD record.

Figure LINK–5 GSD Record With Multiple Subrecords



6.3.1 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 65535 ($2^{16}-1$) 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 diagram depicts the format of a program section definition subrecord, showing the fields it contains and providing a description of each. 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. Page alignment (alignment field value of 9) is the limit for program section alignment. For example:

Value	Alignment
0	1 (BYTE)
1	2 (WORD)
2	4 (LONGWORD)
3	8 (QUADWORD)
4	2^4
..	.
..	.
..	.
9	2^9 (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 5.2.1.4 for a description of program section attributes.) The following are 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.

Bit	Name	Meaning if Set
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 writeable.
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: 1–31 bytes

This field contains the name of the program section in ASCII format.

6.3.2 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 6.3.2.1 describes the format of the global symbol specification subrecord for symbol definitions; Section 6.3.2.2 does the same for symbol references. Note that the PSECT INDEX and VALUE fields are present only for symbol definitions, not for symbol references.

6.3.2.1 GSD Subrecord for a Symbol Definition

The following diagram depicts the global symbol specification subrecord for a symbol definition.

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 Appendix C of the *VAX Architecture Handbook*. 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 nature of 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$V_WEAK	When this bit is set, a strong symbol definition is indicated; when clear, a weak symbol definition.
1	GSY$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: SDF$B_PSINDX
Length: 1 byte

This field contains the program section index, described at the beginning of Section 6.3.2. 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: 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, 1–31 bytes

This field contains the symbol name in ASCII format.

6.3.2.2 GSD Subrecord for a Symbol Reference

The following diagram depicts the global symbol specification subrecord for a symbol reference:

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

FLAGS
Name: SRF$W_FLAGS
Length: 2 bytes

This field is a 2-byte bit field, whose bits describe the nature of 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$V_WEAK	When this bit is set, a weak symbol reference is indicated; when clear, a strong symbol reference.
1	GSY$V_DEF	This bit is clear for a symbol reference.
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

SYMBOL NAME
Name: SRF$T_NAME
Length: variable, 1–31 bytes

6.3.3 Entry Point Symbol and Mask Definition Subrecord (GSD$C_EPM)

The following diagram depicts the format of an entry point symbol and mask definition subrecord:

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

FLAGS
Name: EPM$W_FLAGS
Length: 2 bytes

The content of this field is described in Section 6.3.2.1. Note that bit 1 must be set, that is, GSY$V_DEF = 1, because this is a symbol definition.

PSECT INDEX
Name: EPM$B_PSINDX
Length: 1 byte

VALUE
Name: EPM$L_ADDRS
Length: 4 bytes

The value is the entry point address.

ENTRY MASK
Name: EPM$W_MASK
Length: 2 bytes

The entry mask is written at the entry point of a procedure entered via a CALLS or CALLG instruction, and in some cases is also used in transfer vectors to such procedures. A TIR command is provided for the language processor 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

SYMBOL NAME
Name: EPM$T_NAME
Length: variable, 1–31 bytes

6.3.4 Procedure With Formal Argument Definition Subrecord (GSD$C_PRO)

The following diagram depicts the format of a procedure with formal argument definition subrecord.

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

FLAGS
Name: PRO$W_FLAGS
Length: 2 bytes

The content of this field is described in Section 6.3.2.1. Note that bit 1 must be set, that is, GSY$V_DEF = 1, because this is a symbol definition.

PSECT INDEX
Name: PRO$B_PSINDX
Length: 1 byte

VALUE
Name: PRO$L_ADDRS
Length: 4 bytes

ENTRY MASK
Name: PRO$W_MASK
Length: 2 bytes

NAME LENGTH
Name: PRO$B_NAMLNG
Length: 1 byte

SYMBOL NAME
Name: PRO$T_NAME
Length: variable, 1–31 bytes

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, the linker issues a warning message if the value of MINIMUM ACTUAL ARGUMENTS is greater than the value of MAXIMUM ACTUAL ARGUMENTS.

FORMAL ARG 1 DESCRIPTOR
Length: variable, 2–256 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 ARG n DESCRIPTOR
Length: variable, 2–256 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 in the above record contains three subfields. The following diagram depicts its content and format:

ARG VAL CTL
Name: 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 BYTE 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. Since 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
Length: variable, 0–255 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$B_VALCTL are not equal to 0 and/or the value of ARG$B_BYTECNT is not equal to 0, then recompilation of the object module may be necessary in the event that argument validation is implemented in a future VAX/VMS Linker.

6.3.5 Symbol Definition With Word Psect Subrecord (GSD$C_SYMW)

This subrecord is identical in format to the global symbol definition subrecord described in Section 6.3.2.1, 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.

6.3.6 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 6.3.3, 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.

6.3.7 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 6.3.4, 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.

6.3.8 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 diagram depicts the format of an entity ident consistency check subrecord:

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 of the entity name.

ENTITY NAME

Name: IDC$T_NAME
Length: variable, 1–31 bytes

This field contains the entity name in ASCII format.

IDENT LENGTH

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

Length: variable, 1–31 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

Length: 1 byte

This field contains the length in bytes of the name of the entity in the entity name table.

OBJECT NAME
Length: variable, 1–31 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.

6.3.9 Environment Definition/Reference Subrecord (GSD$C_ENV)

The following diagram depicts the format of an environment definition and reference subrecord:

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 field is a 2-byte bit field. Bit 0, whose name is ENV$V_DEF, is a bit mask. When ENV$V_DEF = 1, the subrecord describes an environment definition; when clear, an environment reference. Bits 2–15 are ignored.

ENV$V_NESTED—The current environment is nested within another environment. The parent environment index is ENV$W_ENVINDX.

ENVIRONMENT INDEX
Name: ENV$W_ENVINDX
Length: 2 bytes

This field contains the environment index, a number from 0 through 65535. Like a program section, each environment is assigned a number (its index) that is used by TIR and GSD records in references 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, since a 0 could also be interpreted as the current environment being contained within environment 0, the ENV$V_NESTED bit may be consulted to clear up this ambiguity.

NAME LENGTH
Name: ENV$B_NAMLNG
Length: 1 byte

This field contains the length in characters of the environment name.

ENVIRONMENT NAME
Name: ENV$T_NAME
Length: variable, 1–31 bytes

This field contains the environment name.

The linker reports any undefined environments at the end of Pass 1. Note that there may be a total of 65535 environments defined or referenced in any single object module.

6.3.10 Module-Local Symbol Definition/Reference Subrecord (GSD$C_LSY)

This subrecord, like the global symbol specification subrecord described in Section 6.3.2, has two formats: one for a symbol definition and one for a symbol reference. The following subsections describe each of these.

6.3.10.1 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 6.3.5, 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. Bit 1 (LSY$V_DEF) must be set because this is a definition. Bit 3 (LSY$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.

6.3.10.2 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 6.3.2.2, 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$V_DEF) in the FLAGS field is used. It must be clear, that is, LSY$V_DEF = 0, because this is a reference. Bits 0, 2, and 3 are ignored.

6.3.11 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 6.3.6, 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).

6.3.12 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 6.3.7, 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).

6.3.13 Program Section Definition in a Shareable Image (GSD$C_SPSC)

This subrecord is identical in format to the program section definition subrecord described in Section 6.3.1, 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.

6.4 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$B_RECTYP) followed by one COMMAND field and one DATA field for each TIR command in the record. Since a TIR record may contain many TIR commands, it may be quite long. It may not, however, exceed the record size limit for the object module. This limit is set in the MAXIMUM RECORD SIZE field (MHD$W_RECSIZ) in the Main Module Header Record (MHD$C_MHD).

The fields in a TIR record are described below. 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$B_RECTYP
Length: 1 byte

The record type is OBJ$C_TIR.

COMMAND
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 next COMMAND field. There are 85 TIR commands (excluding the Store Immediate command), and each has a positive number ranging from 0 to 84 associated with it. Thus, this field may contain the name of a command or its corresponding number, though it is recommended that you use the name of the command rather than the number.

DATA
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 one byte.

COMMAND
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

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:

1. Stack commands place data on the stack.
2. 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.
3. Operator commands perform arithmetic operations on data currently on the stack.
4. 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 Pass 1 and executes them on Pass 2. The execution of TIR records produces the output image file. TIR records are not executed if the linking operation is aborted before Pass 2 because of a command or link-time error. Consequently, user or compiler errors potentially detectable in Pass 2 (such as truncation errors) will not be detected in this case.

  TIR commands are described in the following four subsections. The first subsection discusses the stack commands; the second, the store commands; the third, operator commands; and the fourth, 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 already been described.

6.4.1 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 both signed extension to longword and zero extension to longword formats.

The data stacked is taken from one of the following:

- The DATA field directly following the COMMAND field
- A global symbol
- A computation derived from the base address of a program section

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$C_STA_PB (Stack Psect Base Plus Byte Offset)	Data is a 1-byte program section index followed by a single signed byte offset. The psect base and byte offset are added and stacked.
5	TIR$C_STA_PW (Stack Psect Base Plus Word Offset)	Data is a 1-byte program section index followed by a single signed word offset. 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. The psect base and longword offset are added and stacked.

Note that although the offsets in the above three commands are signed, negative values are very rarely correct. Note also that the base address is that of this module's contribution to the program section.

Code	Command	Description
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 future use. 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.

Code	Command	Description
10	TIR$C_STA_WFI (Stack Word From Image)	This command is reserved for future use. It is the word equivalent of the previous command.
11	TIR$C_STA_LFI (Stack Longword From Image)	This command is reserved for future use. It is the longword equivalent of the previous two commands.
12	TIR$C_STA_EPM (Stack Entry Point Mask)	This command has the same format as the Stack Global command. However, it stacks the entry point mask (unsigned stacks word) that accompanies the symbol definition, rather than the symbol value. An error results if the symbol is not an entry point.
13	TIR$C_STA_CKARG (Compare Procedure Argument and Stack for TRUE or FALSE)	This command allows for a limited form of procedure argument validation by checking to see whether the argument passing mechanism (ARG$V_PASSMECH) in the formal argument descriptor matches that in the actual argument descriptor. The DATA field for this command consists of the ASCIC symbol name in standard name format (1 count byte followed by the symbol name (1-31 bytes)) followed by the 1-byte argument index, followed by the actual argument descriptor, whose format is indentical to that of the formal argument descriptor described in Section 6.3.4. The linker compares the values of ARG$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.

Code	Command	Description
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.

6.4.2 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.

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.

Code	Command	Description
28	TIR$C_STO_PICR (Store Position-Independent Code Reference)	The purpose of this command is to generate a position-independent code reference. The top longword on stack is the address of an item to which a position-independent instruction makes reference. If the item is absolute, the byte 9F hexadecimal (the operand specifier for absolute addressing mode) followed by the top longword on the stack are written to the image, and the location counter is incremented. If the item is relocatable, the byte EF hexadecimal (the operand specifier for longword relative addressing mode) is written to the image; the top longword on the stack is Stored Longword Displaced (see TIR$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 Stored Longword Displaced (target is in the EXIT vector); and the location counter is incremented.
29	TIR$C_STO_RSB (Store Repeated Signed Byte)	The longword on top of the stack is used as a repeat count. The low byte of the next longword on the stack is then written to the image the indicated number of times. Both longwords are removed from the stack upon completion. See note in Section 6.4 regarding the use of this command with zeros.
30	TIR$C_STO_RSW (Store Repeated Signed Word)	Same as above command except that a word rather than a byte is written.
31	TIR$C_STO_RL (Store Repeated Longword)	Same as above two commands except that a longword is written.
32	TIR$C_STO_VPS (Store Arbitrary Field)	This command writes a bitfield 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 bitfield 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.

Code	Command	Description
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$C_STO_B (Store Byte)	This command writes the low byte of the top longword on the stack to the image file. It thus permits any 8-bit (from –128 to 255) to be written to the image. If the top longword on the stack is negative, then bits 31:7 must be 1; if positive, then bits 31:8 must be zero.
38	TIR$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.
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$C_STO_RW (Store Repeated Word)	This is the word version of the above command.
41	TIR$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.

Code	Command	Description
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 Store Longword 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

6.4.3 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.

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.

Code	Command	Description
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.
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.

Code	Command	Description
65	TIR$C_OPR_REDEF (Redefine Symbol To Current Location)	This command is used only in the creation of shareable images. Data for the command consists of a symbol name in standard name format, that is, 1 count byte followed by a variable length (1-31 bytes) ASCII string. The value of the symbol, as listed in the shareable image's symbol table, is made equal to the value of the current location counter (at the time the command is processed). Values of the symbol within the image itself are not affected by the command. The value is not assigned until after all image binary has been written to the image output file. If no binary is generated (or is aborted) the value is not assigned. The symbol is also made universal.
66	TIR$C_OPR_DFLIT (Define a Literal)	Data is a 1-byte field that indicates the literal (0–255) 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

6.4.4 Control Commands

Control commands manipulate the linker's location counter.

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.

The following three commands are legal only in debugger information (DBG) and traceback information (TBT) records. For each object module, a list of debug indexes is kept. These commands operate on the list for the object module in which the DBG or TBT record occurs.

Code	Command	Description
82	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.

Code	Command	Description
83	TIR$C_CTL_STLOC (Set Location)	The value of the top longword on the stack is an index (from a previous Define Location command) that is used to locate a previously saved location counter. The value of the previously saved location counter is then set as the value of the current location counter. Upon completion of the command, the top longword is removed from the stack.
84	TIR$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

6.5 End of Module Record

The end of module 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 two 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 may 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 below.

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 shows 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$B_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. 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.

6.6 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 in length, instead of 1 byte as in the EOM record.

6.7 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 set.

The command stream in DBG records generates what is referred to as the debug 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 has been 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 were 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 was specified at link time and only if an executable image is being produced. If either of these is not true, DBG records are ignored.

6.8 Traceback Information Records

Traceback information records are the means by which language processors pass information to the facility which 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 in fact 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.

6.9 Link Option Specification Records

The link option specification records are defined for the purpose of allowing 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, since 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 table below shows the LNK subrecord types. Column 1 displays the name of the LNK subrecord; column 2 shows its symbolic representation; and column 3 displays its corresponding numerical code.

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, two flag bits are defined in LNK$W_FLAGS:

Bit	Name	Meaning if set
0	LNK$V_SELSER	Selectively search object module or symbol table. This bit is valid only for LNK$C_OBJ subrecords.
1	LNK$V_LIBSRCH	After module inclusion, search this library for resolution of currently undefined symbols. The need for this bit arises out of an ambiguity between the usage of the two record types LNK$C_OLI and LNK$C_OLB. The use of this bit is best illustrated by the file-qualifiers /LIBRARY and /INCLUDE. Note that an input file specification such as A/INCLUDE=(B,C) would correspond to a LNK$C_OLI type, and an input file spec such as A/LIB would correspond 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 one byte of zero.

COMMAND QUALIFIERS

This section discusses each command qualifier acceptable to the linker. Although you can enter one or more command qualifiers, in most cases you need not do so since the linker supplies default values for each one.

Command qualifiers direct the linker as to what kind of image to create, what to include or not include in the image, what parameters to use in creating the image, and what additional files, if any, to create.

Positional qualifiers direct the linker in its processing of a file by specifying such information as what kind of file it is and how to process it. For more information, refer to the following section, Positional Qualifiers.

Some qualifiers are valid only if they are used with other qualifiers, and some qualifiers are incompatible with other qualifiers. The linker takes one of two actions if you specify incompatible qualifiers: either it displays an error message and invalidates the entire LINK command, or it ignores certain qualifiers (generally, all except the last valid one) and allows the link to continue. For example, if you specify /FULL and /BRIEF for the map, the linker rejects the entire command; but if you specify the positive and negative forms of a qualifier (for example, /EXECUTABLE and /NOEXECUTABLE), the linker accepts the last one entered.

Table LINK–8 lists, in alphabetical order, each command qualifier, its default value, and the names of other qualifiers with which it is incompatible. A [NO] indicates that the qualifier can be negated by prefixing NO (without brackets): for example, /NODEBUG or /NOEXECUTABLE. Qualifier values are not valid with negative qualifiers: for example, /NOEXECUTABLE=PAYROLL is not valid.

Table LINK–8 Command Qualifiers

Command Qualifier	Default	Incompatible Qualifiers
/BRIEF	Default map	/NOMAP,/FULL, /CROSS_REFERENCE
/[NO]CONTIGUOUS	/NOCONTIGUOUS	/NOEXECUTABLE
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE	/NOMAP,/BRIEF
/[NO]DEBUG[=file-spec]	/NODEBUG	/NOTRACEBACK, /SHAREABLE, /SYSTEM, /NOEXECUTABLE
/[NO]EXECUTABLE-[=file-spec]	/EXECUTABLE	/SHAREABLE
/FULL	Default map	/NOMAP,/BRIEF
/HEADER		
/[NO]MAP[=file-spec]	/NOMAP (interactive) /MAP (batch)	
/POIMAGE		
/PROTECT		/SYSTEM, /EXECUTABLE

Table LINK–8 (Cont.) Command Qualifiers

Command Qualifier	Default	Incompatible Qualifiers
/[NO]SHAREABLE-[=file-spec	/NOSHAREABLE	/SYSTEM, /DEBUG, /EXECUTABLE
/[NO]SYMBOL_TABLE-[=file-spec]	/NOSYMBOL_TAB	
/[NO]SYSLIB	/SYSLIB	
/[NO]SYSSHR	/SYSSHR	/NOSYSLIB
/[NO]SYSTEM-[=base-address]	/NOSYSTEM	/DEBUG, /SHAREABLE
/[NO]TRACEBACK	/TRACEBACK	
/[NO]USERLIBRARY-[=(table[,...])]	/USERLIBRARY=ALL	

/BRIEF

Directs the linker to produce a brief form of the image map.

FORMAT

/MAP /BRIEF

qualifier values

None.

DESCRIPTION

A brief map contains only the following sections:

- Object Module Synopsis
- Image Synopsis
- Link Run Statistics

In contrast, the default image map contains the above sections, as well as the program section synopsis and symbols by name sections.

/BRIEF is incompatible with /FULL and /CROSS_REFERENCE. In general, you should request a full (rather than a brief) map because the full map contains more information.

EXAMPLE

```
$ LINK /MAP /BRIEF GARBO
```

This example directs the linker to produce an executable image named GARBO.EXE and an image map named GARBO.MAP that contain an object module synopsis, an image synopsis, and link run statistics.

/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 linking operation.

FORMAT

/[NO]CONTIGUOUS

qualifier values

None.

DESCRIPTION

Since any type of image usually executes more slowly if its pages are not contiguous, you can use the /CONTIGUOUS qualifier to speed up the execution time of any type of image. (Note however that in most cases performance benefits do not warrant the use of /CONTIGUOUS.) You can also use the /CONTIGUOUS qualifier when linking bootstrap programs for certain system images that require contiguity.

Even when you do not specify /CONTIGUOUS, the operating system still tries to make the pages of an image contiguous whenever possible. Thus, only if sufficient contiguous space is not available, will the pages of an image be noncontiguous.

/NOCONTIGUOUS is the default.

EXAMPLE

```
$ LINK /CONTIGUOUS HARLOW
```

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

/CROSS_REFERENCE

Directs the linker to replace the symbols by name section of the image map with the symbol cross-reference section.

FORMAT

/MAP /[NO]CROSS_REFERENCE

qualifier values *None.*

DESCRIPTION

The Symbols by Name 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 is generating 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 SYS$LIBRARY:IMAGELIB.OLB and SYS$LIBRARY:STARLET.OLB.

/CROSS_REFERENCE is incompatible with /BRIEF.

/NOCROSS_REFERENCE is the default. In this case, if the linker is generating a default map or a full map, the map will contain the symbol by name section instead of the symbol cross-reference section.

EXAMPLE

```
$ LINK /MAP /CROSS_REFERENCE BARA
```

This example produces an executable image named BARA.EXE and an image map named BARA.MAP that includes a cross-reference.

/DEBUG

Directs the linker to generate a debug 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

qualifier value

file-spec
Identifies the user-written debug module.

If you specify /DEBUG without entering a file specification, the VAX/VMS Symbolic Debugger gains control at run time. Requesting the VAX/VMS Symbolic Debugger does not affect the location of code within the image, since the debugger is mapped into the process address space at run time, not at link time. See the description of the VAX/VMS Symbolic Debugger in the *VAX/VMS Utilities Reference Volume* for additional information.

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

DESCRIPTION

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

/NODEBUG is the default.

EXAMPLE

```
$ LINK /DEBUG GISH
```

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

/EXECUTABLE

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

FORMAT

/EXECUTABLE *[=file-spec]*
/NOEXECUTABLE

qualifier value

file-spec

Identifies the file name by which you want the linker to create an executable image. If you do not enter a file type after the file name, the linker assumes a file type of EXE.

If you do not give a file specification with the /EXECUTABLE qualifier, the linker creates an executable image with the file name of the first input file.

DESCRIPTION

If you specify /EXECUTABLE as a positional qualifier, the linker creates an executable image with the file name of the file to which the qualifier is attached.

If you specify both /SYSTEM and /EXECUTABLE, the linker creates a system image. In this case, the /EXECUTABLE qualifier allows you to give the file specification of the system image.

/NOEXECUTABLE directs the linker to perform the linking operation but to not create an image file.

If you do not specify /NOEXECUTABLE, /SHAREABLE, or /SYSTEM, the linker assumes /EXECUTABLE.

EXAMPLES

1
```
$ LINK /NOEXE STREEP
```

This example directs the linker to link the object module in file STREEP.OBJ without creating an image file. You can use /NOEXE with /MAP or /SYM to produce only a map or symbol table. You can also use the /NOEXE qualifier to ensure that the link operation progresses without error.

2
```
$ LINK /EXE STREEP
```

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

3
```
$ LINK /EXECUTABLE=PICKFORD STREEP
```

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

/FULL

Directs the linker to create a full image map when specified with /MAP.

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
- Program section synopsis
- Symbols by name
- Image section synopsis
- Symbols by value
- Module relocatable reference synopsis

In contrast, the default map does not contain the image section synopsis, the symbols by value, or the module relocatable reference synopsis sections.

Further, unlike the default map, the full map includes information about modules included from the system default libraries SYS$LIBRARY:STARLET.OLB and SYS$LIBRARY:IMAGELIB.OLB. Thus, the object module synopsis, program section synopsis, and symbols by name sections of a default map do not contain information about modules included from SYS$LIBRARY:STARLET.OLB and SYS$LIBRARY:IMAGELIB.OLB, whereas in a full map they do.

For these reasons, you should request a full map rather than a default or brief map.

/FULL is valid only if you also specify /MAP in the LINK command. Also, /FULL is incompatible with /BRIEF but not with /CROSS_REFERENCE.

EXAMPLE

```
$ LINK /MAP /FULL MONROE
```

This example directs the linker to produce an executable image named MONROE.EXE and an image map named MONROE.MAP that contains the image section synopsis, the symbols by value, and the module relocatable reference synopsis sections.

/HEADER

Directs the linker to include an image header in the system image when specified with /SYSTEM.

FORMAT

/HEADER

qualifier values *None.*

DESCRIPTION

If you specify /SYSTEM without /HEADER, the linker creates a system image without a header.

Executable and shareable images always have image headers. Consequently, the linker ignores /HEADER if it is creating an executable or shareable image.

EXAMPLE

```
$ LINK/SYSTEM/HEADER MANSFIELD
```

This example directs the linker to produce a system image named MANSFIELD.EXE with an image header. The use of /HEADER or /NOHEADER depends on the characteristics of the loader program, which loads the operating system. (For more information, see the description of the /SYSTEM qualifier.)

/MAP

Directs the linker to create an image map file.

FORMAT

/MAP *[=file-spec]*
/NOMAP

qualifier value

file-spec
Identifies the file name from which you want the linker to create an image map file. 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 and the file type MAP.

If you specify /MAP as a positional qualifier, the linker creates an image map file with the file name of the file to which the qualifier is attached.

DESCRIPTION

If you specify /MAP but do not also specify either the /BRIEF, /FULL, or /CROSS_REFERENCE qualifier, the linker produces a default map containing the following sections:

- Object module synopsis
- Program section synopsis
- Symbols by name
- Image synopsis
- Link run statistics

If you do not specify /MAP, the linker assumes, by default, /NOMAP in interactive mode and /MAP in batch mode.

See Section 4 for more information.

EXAMPLES

1
```
$ LINK /NOMAP GARLAND
```

This example directs the linker to override the default of /MAP on batch jobs.

2
```
$ LINK /MAP GARLAND
```

This example directs the linker to produce an image map (in all cases) with the default name of GARLAND.MAP.

3
```
$ LINK /MAP=RUSSELL GARLAND
```

This example directs the linker to produce an image map with the name of RUSSELL.MAP instead of the default name of GARLAND.MAP.

/P0IMAGE

Causes an executable image to be placed entirely in the P0 address space. Thus, when /P0IMAGE is specified, the user stack and VAX RMS buffers, which usually go in P1 space, are placed in P0 space.

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, since when /P0IMAGE is specified, 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.

See the *VAX Hardware Handbook* for an explanation of P0 and P1 address space.

EXAMPLE

```
$ LINK /P0IMAGE LOMBARD
```

This example directs the linker to set up an executable image named LOMBARD.EXE to be run entirely in the P0 address space.

/PROTECT

Directs the linker to protect the entire shareable image from user-mode and supervisor-mode write access when specified with /SHAREABLE.

FORMAT

/SHAREABLE /PROTECT

qualifier values

None.

DESCRIPTION

To protect one or more clusters, but not the entire image, from user-mode and supervisor-mode write access, use the PROTECT= option with the clusters that you want to protect.

The /PROTECT qualifier is incompatible with the /EXECUTABLE and /SYSTEM qualifiers.

The /PROTECT qualifier is useful in the creation of privileged shareable images. See Section 3.3.3 for more information on privileged shareable images.

EXAMPLE

```
$ LINK /SHAREABLE /PROTECT HEPBURN
```

This example directs the linker to produce a privileged, shareable image named HEPBURN.EXE.

/SHAREABLE

Is both a command qualifier and a positional qualifier. As a command qualifier, /SHAREABLE directs the linker to create a shareable image. As a file qualifier in a linker options file, /SHAREABLE identifies an input file as a shareable image file.

FORMAT

/SHAREABLE *[=file-spec]*

qualifier value

file-spec
Specifies the file name and type by which you want the linker to create a shareable image. If you omit the file type, the linker uses the default file type EXE.

If you do not specify the optional file-spec parameter with the /SHAREABLE qualifier, the linker creates a shareable image with the file name of the file to which it is appended or, if it is not appended to a file, the first input file.

DESCRIPTION

If the /SHAREABLE qualifier appears anywhere in the command string, the linker interprets it as a command qualifier and creates a shareable image. To illustrate, if /SHAREABLE is appended to a file name that is specified in the command string, the linker interprets it as a command qualifier, even though it looks like a positional qualifier. In this case, the linker creates a shareable image with the name of the file to which the /SHAREABLE qualifier is appended.

To use /SHAREABLE as a positional qualifier, you must use it in an options file, rather than in the command string.

Thus, you cannot specify a shareable image as input in the command string by means of the /SHAREABLE qualifier. However, you can specify a shareable image as input in the command string if the shareable image is in a shareable image library file, since in this case you use the /INCLUDE rather than the /SHAREABLE qualifier.

If you specify /SHAREABLE, you cannot also specify /SYSTEM, /DEBUG, or /TRACEBACK. If you specify both /SHAREABLE and /EXECUTABLE, the linker ignores /EXECUTABLE.

If you do not specify /SHAREABLE, the linker creates an executable image unless you specified /SYSTEM.

EXAMPLES

1

```
$ LINK /SHAREABLE ALLISON
```

This example directs the linker to produce a shareable image named ALLISON.EXE. (/SHAREABLE is used as an output qualifier in this example.)

2

```
$ LINK /SHAREABLE=GRABLE ALLISON
```

This example directs the linker to produce a shareable image named GRABLE.EXE. (/SHAREABLE is used as an output qualifier in this example.)

3
```
$ LINK LAMAR,SYS$INPUT/OPTION GRABLE/SHAREABLE
```

This example shows how the shareable image GRABLE.EXE is used as input to the linker with the object file LAMAR.OBJ to produce the executable image named LAMAR.EXE. (In this example, GRABLE is the name of an options file and /SHAREABLE is used as an input qualifier.)

/SYMBOL_TABLE

Directs the linker to create a symbol table file.

FORMAT

/SYMBOL_TABLE *[=file-spec]*
/NOSYMBOL_TABLE

qualifier value

file-spec

Specifies the file name and type by which you want the Linker to create a symbol table file.

If you specify /SYMBOL_TABLE as a command qualifier without the optional file-spec parameter, the linker creates a symbol table file with the file name of the first input file and the default file type STB.

If you specify /SYMBOL_TABLE as a command qualifier with the optional file-spec parameter, the linker creates a symbol table file with that file name and file type; if you do not enter a file type after the file name, the linker assumes a file type of STB.

If you specify /SYMBOL_TABLE as a positional qualifier, the linker creates a symbol table file with the file name of the file to which the qualifier is attached and the default file type STB.

DESCRIPTION

The symbol table file contains a copy of the linker's global symbol table (GST) in object module format. Note that the /SYMBOL_TABLE qualifier does not change the contents of the linker's GST.

If you do not specify /SYMBOL_TABLE, the linker assumes /NOSYMBOL_TABLE.

A symbol table file may be specified as input in a subsequent linking operation (see Section 1.3.5 for details).

EXAMPLES

1

```
$ LINK /SYMBOL_TABLE /NOEXE DEHAVILND
```

This example directs the linker to produce a symbol table file named DEHAVILND.STB. No executable image file is produced.

2

```
$ LINK /SYMBOL_TABLE=BACALL DEHAVILND
```

This example directs the linker to produce a symbol table file named BACALL.STB. An executable image file named DEHAVILND.EXE is produced.

/SYSLIB

Directs the linker to search the default system libraries SYS$LIBRARY:IMAGELIB and SYS$LIBRARY:STARLET.OLB to resolve symbolic references that remain undefined after all specified input and any user default libraries have been processed.

FORMAT

/[NO]SYSLIB

qualifier values

None.

DESCRIPTION

The linker first searches SYS$LIBRARY:IMAGELIB.OLB, the system default shareable image library, and then SYS$LIBRARY:STARLET.OLB, the system default object library.

/NOSYSLIB directs the linker not to search the default system libraries (IMAGELIB and STARLET). Since these libraries contain many routines required by almost all high-level language programs, you should specify /NOSYSLIB only if you know that your input, together with any user default libraries, allows the linker to resolve all symbolic references.

If you do not specify /NOSYSLIB, the linker assumes /SYSLIB by default. If you specify both /NOSYSLIB and /SYSSHR, the /SYSSHR qualifier is ignored.

If you want the linker to search IMAGELIB but not STARLET, specify /NOSYSLIB (to inhibit the default search of both IMAGELIB and STARLET) and then specify the search of IMAGELIB in the command string (or in an options file) as follows:

```
SYS$LIBRARY:IMAGELIB/LIBRARY
```

EXAMPLE

```
$ LINK /NOSYSLIB LAMOUR
```

This example directs the linker to produce the image LAMOUR.EXE without referencing the default system libraries SYS$LIBRARY:IMAGELIB.OLB or SYS$LIBRARY:STARLET.OLB.

/SYSSHR

Directs the linker to search SYS$LIBRARY:IMAGELIB.OLB, the system default shareable image library, to resolve symbolic references that remain undefined after all specified input files and any user default libraries have been processed. This qualifier is not often used since the linker searches IMAGELIB by default.

FORMAT

/[NO]SYSSHR

qualifier values

None.

DESCRIPTION

/NOSYSSHR directs the linker not to search IMAGELIB. By thus inhibiting the search of IMAGELIB, this qualifier allows the linker to search only the system default object library SYS$LIBRARY:STARLET.OLB providing that /NOSYSLIB was not also specified. This is the primary purpose of the /NOSYSSHR qualifier—to specify that STARLET and not IMAGELIB be searched.

See the description of the /NOSYSLIB qualifier for information on how to direct the linker to search IMAGELIB but not STARLET.

EXAMPLE

```
$ LINK /NOSHSSHR CRAWFORD
```

This example directs the linker to search only the system default object library (SYS$LIBRARY:STARLET.OLB), not the system default shareable image library (SYS$LIBRARY:IMAGELIB.OLB), to resolve symbolic references while producing an executable image named CRAWFORD.EXE.

/SYSTEM

Directs the linker to create a system image.

FORMAT

/SYSTEM *[=base-address]*
/NOSYSTEM

qualifier value

base-address
Specifies the base-address at which you want the Linker to create a system image.

DESCRIPTION

If you specify the optional base-address parameter with the /SYSTEM qualifier, the linker assigns the system image the specified base address, providing that the /HEADER qualifier is not also specified.

If, however, the /HEADER and /SYSTEM qualifiers are both specified, the linker adjusts any specified 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 both greater than the value specified in the base-address parameter and divisible by 512 decimal.

You can specify a base address in hexadecimal (%X), octal (%O), or decimal (%D) format.

If you specify /SYSTEM without a base address, the linker assumes hexadecimal (%X) 80000000 as the base address.

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 in the /EXECUTABLE qualifier.

If you specify /SYSTEM, you cannot specify /SHAREABLE or /DEBUG.

If you do not specify /SYSTEM, the linker does not create a system image. See the /EXECUTABLE command qualifier in this section for an explanation of the linker's default behavior.

EXAMPLE

```
$ LINK /SYSTEM SISSY
```

This example directs the linker to produce a system image named SISSY.EXE based at address 80000000.

/TRACEBACK

Directs the linker to include traceback information in the image.

FORMAT

/[NO]TRACEBACK

qualifier values *None.*

DESCRIPTION

Traceback is a facility that automatically displays information from the call stack when a program error occurs. The output shows which modules were called before the error occurred.

The linker assumes /TRACEBACK unless you specify /NOTRACEBACK. However, if you enter /DEBUG, the linker automatically includes traceback whether or not you also specify /NOTRACEBACK.

Images linked /NOTRACE cannot be run /DEBUG.

EXAMPLE

```
$ LINK /NOTRACEBACK HAYS
```

This example directs the linker not to include traceback information in the executable image named HAYS.EXE.

/USERLIBRARY

Directs the linker to search one or more user default libraries to resolve symbolic references that remain undefined even after all specified input has been processed.

FORMAT

/USERLIBRARY *[=(table[,...])]*
/NOUSER_LIBRARY

qualifier value

table
Specifies the logical name table(s) that the linker will look through when it searches for user default library definitions. The following are acceptable values of the table parameter:

ALL	Causes the linker to search the process, group, and system logical name tables for user default library definitions
GROUP	Causes the linker to search the group logical name table for user default library definitions
NONE	Causes the linker not to search any logical name table; /USERLIBRARY=NONE is equivalent to /NOUSERLIBRARY.
PROCESS	Causes the linker to search the process logical name table for user default library definitions.
SYSTEM	Causes the linker to search the system logical name table for user default library definitions.

DESCRIPTION

A user default library may be an object module library or a shareable image library.

If you do not specify either /NOUSERLIBRARY or /USERLIBRARY=(table), the linker assumes /USERLIBRARY=ALL by default.

/NOUSERLIBRARY directs the linker not to search any user default libraries.

To define a user default library, you must use the DCL commands DEFINE or ASSIGN to equate the logical name LNK$LIBRARY to the file specification of the library, since the linker looks for this logical name to determine if a user default library exists.

Further, to control access to the library, you can define LNK$LIBRARY in the process, group, or system logical name table by using the /PROCESS, /GROUP, and /SYSTEM qualifiers, respectively, in the DEFINE command.

For example, if you want the library MINE to be your default user library, the library THEIRS to be the default user library of everyone else in your group, and the library ANY to be the default user library of everyone else on the system, you would issue the following commands:

```
DEFINE LNK$LIBRARY DB2:[MARK]MINE
DEFINE/GROUP LNK$LIBRARY DB2:[PROJECT]THEIRS
DEFINE/SYSTEM LNK$LIBRARY SYS$LIBRARY:ANY
```

Note that the GRPNAM and SYSNAM privileges are required to use the /GROUP and /SYSTEM qualifiers, 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$LIBRARY_2 for the third, and so on, up to the last possible logical name LNK$LIBRARY_999. However, you must specify these logical names in numerical order without skipping any, for when the linker fails to find the next sequential logical name it ceases its search in that logical name table.

The search of user default libraries proceeds as follows:

1. If you specify /USERLIBRARY=PROCESS or /USERLIBRARY, the linker searches the process logical name table for the name LNK$LIBRARY. If this entry exists, the linker translates the logical name and searches the specified library for unresolved strong references. If you exclude PROCESS from the table list in /USERLIBRARY or if no entry exists for LNK$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 /USERLIBRARY=GROUP or /USERLIBRARY, the linker follows the logic in steps 1–3 using the group logical name table. If you exclude GROUP from the table list in /USERLIBRARY or when any logical name translation fails, the linker proceeds to step 5.
5. If you specify /USERLIBRARY=SYSTEM or /USERLIBRARY, the linker follows the logic in steps 1–3 using the system logical name table. If you exclude SYSTEM from the table list in /USERLIBRARY or when any logical name translation fails, the search of user default libraries is complete. By default, the linker proceeds to search the default system libraries if any unresolved references remain.

EXAMPLE

```
$ LINK /USERLIBRARY=(GROUP) ROGERS
```

This example directs the linker to reference only the group logical name table to translate the logical names LNK$LIBRARY, LNK$LIBRARY_1, LNK$LIBRARY_2, and so on to LNK$LIBRARY_999.

POSITIONAL QUALIFIERS

This section discusses each positional qualifier acceptable to the linker. Positional qualifiers direct the linker in its processing of a file by specifying such information as what kind of file it is and how to process it. Although you can enter one or more positional qualifiers, in most cases you need not do so since the linker supplies default values for each one.

Some qualifiers are incompatible with certain other qualifiers. The linker takes one of two actions if you specify incompatible qualifiers: either it invalidates the entire LINK command and displays an error message, or it ignores certain qualifiers (generally, all except the last valid one) and allows the link to continue.

Table LINK–9 lists, in alphabetical order, each positional qualifier, its function, and the names of other qualifiers with which it is incompatible.

Table LINK–9 Positional Qualifiers

Positional Qualifier	Function	Incompatible Qualifiers
/INCLUDE=module-name[,...]	Includes one or more modules from a library in the link	All others, except /LIBRARY
/LIBRARY	Identifies a library	All others, except /INCLUDE
/OPTIONS	Identifies an options file	All others
/SELECTIVE_SEARCH	Includes only global symbols referred to by previously named input files	All others, except /SHAREABLE
/SHAREABLE	Identifies a shareable image file; valid only in an options file	All others, except /SELECTIVE_SEARCH

/INCLUDE

Identifies the file to which it is appended as a library file and directs the linker to include the named module or modules from that library in the linking operation.

FORMAT

library-name **/INCLUDE=module-name[,...]**

qualifier value

module-name
Indicates the module or modules that you want included from the specified library in the linking operation.

DESCRIPTION

Note that the /INCLUDE qualifier does not also cause the linker to search that library for unresolved references. It only directs the linker to extract the specified modules. If you want the library searched as well, you must also specify the /LIBRARY positional qualifier.

To specify more than one module, enclose the list in parentheses and separate module names with a comma.

EXAMPLES

1

```
$ LINK LEAGUE,NATIONAL/INCLUDE=(REDS,DODGERS,PHILS)
```

This example directs the linker to combine modules REDS, DODGERS, and PHILS to the input module LEAGUE.

2

```
$ LINK LEAGUE,NATIONAL/LIBRARY/INCLUDE=(REDS,DODGERS,PHILS)
```

This example directs the linker to combine modules REDS, DODGERS, and PHILS to the input module LEAGUE and then to search the library NATIONAL for symbol definitions that are unresolved in all four modules.

/LIBRARY

Identifies the file to which it is appended as a library file, specifies that the linker search the symbol table of this library for symbols that are undefined in previously processed modules, and if it finds such a symbol, specifies that the linker include the library module containing the symbol definition in the linking operation.

FORMAT

library-name **/LIBRARY**

qualifier values

None.

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 only in previously processed, not subsequently processed, modules.

When specifying library files in lengthy and complex command input, be sure to read Section 5.3 to learn how the linker places input files in clusters, since which symbols are resolved by the library file depends on which cluster the library file is placed in.

EXAMPLES

1

```
$ LINK ROSE,FLOWERS/LIBRARY,DAISY,PANSY
```

In this example, the linker uses the library FLOWERS to resolve undefined symbols in ROSE, but not in DAISY or PANSY.

2

```
$ LINK ROSE,DAISY,PANSY,FLOWERS/LIBRARY
```

In this example, the linker uses the library FLOWERS to resolve undefined symbols in ROSE, DAISY, and PANSY.

/OPTIONS

Identifies a file as a linker options file. This file can contain input file specifications, as well as special instructions to the linker called link options.

FORMAT *options-file* **/OPTIONS**

qualifier values *None.*

DESCRIPTION Section 2 discusses the purpose, contents, and specification of options files. Section 5.3 discusses how the linker processes options files.

EXAMPLE

```
$ LINK GARDNER,LOREN/OPTIONS
```

This example directs the linker to use an options file named LOREN.OPT to produce an executable image named GARDNER.EXE. The options file named LOREN.OPT contains the line GRABLE/SHAREABLE.

/SELECTIVE_SEARCH

Directs the linker to copy from the specified file into its global symbol table (GST) only those global symbols that are both defined in the specified file and referenced by previously processed input files.

FORMAT

object-file **/SELECTIVE_SEARCH**

qualifier values *None.*

DESCRIPTION

If you do not specify /SELECTIVE_SEARCH with an input file, the linker includes all global symbols from that file in its GST.

The /SELECTIVE_SEARCH positional qualifier is useful when you want to control the size of the GST in your image by eliminating the inclusion of irrelevant global symbols. For example, if you were including the system symbol table (SYS$SYSTEM:SYS.STB) in the linking operation to resolve a few references, you would want to specify this file with the /SELECTIVE_SEARCH qualifier to avoid the inclusion of numerous, irrelevant global symbols.

EXAMPLE

```
$ LINK WEAVER,SYS$SYSTEM:SYS.STB/SELECTIVE
```

This example directs the linker to produce an executable image named WEAVER.EXE. The linker will not include global symbols in its global symbol table (GST) defined in SYS.STB that were not referenced by WEAVER.

/SHAREABLE

/SHAREABLE is both a command qualifier and a positional qualifier. As a command qualifier, /SHAREABLE directs the linker to create a shareable image. As a positional qualifier, /SHAREABLE identifies an input file as a shareable image file.

FORMAT

shareable-image-file **/SHAREABLE**

qualifier values

None.

DESCRIPTION

If the /SHAREABLE qualifier appears anywhere in the command string, the linker interprets it as a command qualifier and creates a shareable image. To illustrate, if /SHAREABLE is appended to a file name that is specified in the command string, the linker interprets it as a command qualifier, even though it looks like a positional qualifier. In this case, the linker creates a shareable image with the name of the file to which the /SHAREABLE qualifier is appended.

See the Command Qualifier Section for an explanation of the use of /SHAREABLE as a command qualifier.

EXAMPLES

1
```
$ LINK /SHAREABLE=GRABLE ALLISON
```

This example directs the linker to produce a shareable image named GRABLE.EXE. (/SHAREABLE is used as an output qualifier in this example.)

2
```
$ LINK LAMAR,SYS$INPUT/OPTION GRABLE/SHAREABLE
```

This example shows how the shareable image GRABLE.EXE is used as input to the linker with the object file LAMAR.OBJ to produce the executable image named LAMAR.EXE. (In this example, GRABLE is the name of an options file and /SHAREABLE is used as an input qualifier.)

EXAMPLES

1
```
$ LINK PROGA
```

This command directs the linker to create an executable image using the object module PROGA.OBJ and to name it PROGA.EXE. If there are unresolved references in PROGA, the linker searches any user default libraries to resolve them. If there are still unresolved references in PROGA or in any module included from a user default library, the linker searches the system default libraries SYS$LIBRARY:IMAGELIB.OLB and SYS$LIBRARY:STARLET.OLB to resolve them.

2
```
$ LINK/DEBUG  LOVE,HATE
```

This command directs the linker to combine the modules LOVE and HATE, and the debugger, into an executable image with the file name LOVE.EXE. The linker searches user default and system default libraries as in the first example.

3
```
$ LINK/EXECUTABLE=SPIRIT  LOVE,HATE,FEELINGS/INCLUDE=PEACE
```

This command directs the linker to combine the modules LOVE and HATE and the library module PEACE (to be extracted from the library FEELINGS) into an executable image with the file name SPIRIT.EXE. The linker searches user default and system default libraries as in the first example.

4
```
$ LINK/MAP=TEST/FULL/CROSS_REFERENCE   PAYROLL,FICA,PAYLIB/LIBRARY
```

This command directs the linker to combine the modules PAYROLL and FICA, to search the library PAYLIB for unresolved references in PAYROLL and FICA and include any needed modules, into an executable image with the file name PAYROLL.EXE. The linker also creates an image map file with the file name TEST.MAP, which contains all sections provided in the full map, as well as a Symbol Cross Reference section. The linker searches user default and system default libraries as in the first example.

5
```
$ LINK/SYMBOL_TABLE/NOUSERLIBRARY   CURLY,LARRY,MOE,TVLIB/INCLUDE=OLDIES,-
COMEDY/LIBRARY,SLAPSTICK/OPTIONS
```

This command directs the linker to combine object modules CURLY, LARRY, and MOE, as well as the module OLDIES from the library TVLIB, into an executable image with the file name CURLY.EXE. The linker searches the library COMEDY for any unresolved symbolic references in CURLY, LARRY, MOE, and OLDIES, and includes any modules in COMEDY that resolve those references. After the linker processes the options file SLAPSTICK (see Section 2), it does not search any user default library but it does search the system default libraries. Finally, the linker creates a symbol table file with the file name CURLY.STB.

Index

VAX/VMS
Linker Reference
Manual
AA-Z420A-TE

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ______

Name ______ Date ______

Organization ______

Street ______

City ______ State ______ Zip Code or Country ______

Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line