digital

VAX–11
Linker Reference Manual

Order No. AA-D019B-TE

VAX11

March 1980

This document describes how the VAX-11 Linker works and how to use it.

VAX-11
Linker Reference Manual

Order No. AA-D019B-TE

SUPERSESSION/UPDATE INFORMATION:	This revised document supersedes the VAX-11 Linker Reference Manual (Order No. AA-D019A-TE)
OPERATING SYSTEM AND VERSION:	VAX/VMS V02
SOFTWARE VERSION:	VAX/VMS V02

digital equipment corporation · maynard, massachusetts

First Printing, August 1978
Revised, March 1980

CONTENTS

CONTENTS

CONTENTS

CONTENTS

PREFACE

MANUAL OBJECTIVES

The VAX-11 Linker Reference Manual describes how the VAX-11 Linker works and tells you how to use it. This manual is designed to educate as well as to provide easy reference.

INTENDED AUDIENCE

This manual is intended for programming specialists and nonspecialists alike. Certain parts of the manual are designed specifically to meet the needs of certain types of readers.

- If you are not yet proficient in programming under the VAX/VMS system or if you do not need to become an expert, this manual is designed to teach you the main concepts and techniques of linking as clearly as possible. Chapters 1 through 6 and Appendixes A and B are aimed especially at this type of reader.

- If you are already proficient in programming under the VAX/VMS system, this manual provides detailed information about some of the more complex aspects of linking. Chapters 7 and 8 and Appendixes C and D are aimed especially at this type of reader.

STRUCTURE OF THIS DOCUMENT

Chapter 1 introduces the linker. It defines significant terms, presents the reasons for the linker's existence, and discusses in general terms how the linker works.

Chapters 2 and 3 focus on concepts important to understanding the linker's operation. The discussion of symbols and references in Chapter 2 derives from the linker's function of resolving symbolic references between modules. Chapter 3 explains libraries, which normally contain frequently used modules that the linker can include in user images.

Chapter 4 discusses the LINK command and its command and file qualifiers. Chapter 5 focuses on the /OPTIONS file qualifier, describing how to create and use a linker options file.

Chapter 6 explains the various forms of the image map that the linker produces on request. This map provides information about the image that was created and about the linking process itself.

Chapter 7 goes more deeply into the process by which the linker creates images. Chapter 7 expands on concepts introduced in Chapter 1 and introduces new concepts.

Chapter 8 presents detailed explanations of shareable images. The complex information in this chapter is intended mainly for more sophisticated programmers and application designers.

The appendixes provide supplementary information. Appendix A lists the error messages that the linker can generate. Appendix B illustrates complete brief, default, and full maps of the same image. Appendix C is a specification of the object language accepted by the linker; this information is useful to anyone designing a compiler or assembler whose output must be acceptable to the VAX-11 Linker. Appendix D explains the object module analysis (ANALYZE) program.

ASSOCIATED DOCUMENTS

The following documents contain information pertinent to linking:

- VAX-11 Information Directory and Index
- VAX/VMS Primer
- VAX/VMS Command Language User's Guide
- VAX-11 Symbolic Debugger Reference Manual
- VAX/VMS System Manager's Guide
- The user's guide for your programming language(s)

This document uses the following conventions:

Convention	**Meaning**
Uppercase words and letters	Uppercase words and letters, used in examples, indicate that you should type the word or letter exactly as shown.
Lowercase words and letters	Lowercase words and letters, used in format examples, indicate that you are to substitute a word or value of your choice.
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.
[ ]	Square brackets indicate that the enclosed item is optional (except when used in file specifications where square brackets delimit directory names).
{ }	Braces are used to enclose lists from which one element is to be chosen.
...	A horizontal ellipsis indicates that the preceding item(s) can be repeated one or more times.
. . .	A vertical ellipsis indicates that not all of the statements in an example or figure are shown.

SUMMARY OF TECHNICAL CHANGES

This manual describes the VAX-11 Linker, Version 2. The following are technical changes from the previous version.

User-defined default object libraries are allowed. These libraries do not have to be explicitly specified in the LINK command and are searched before the system default libraries. The LINK command qualifier /USERLIBRARY controls user-defined default object libraries.

When creating a shareable image, the linker defers the relocation of virtual addresses. Deferred relocation allows the linker to create position-independent shareable images that contain relocatable address data. Each user is given a private copy of image sections that contain relocatable address data.

The linker supports 31-character symbol names.

The linker can create three new kinds of images: 1) system images with image headers, 2) executable images that are only stored in P0 address space, and 3) protected shareable images. The LINK command qualifiers /HEADER, /P0IMAGE, and /PROTECT are used to create these images. The PROTECT= linker option and the VEC program section attribute can also be used to create protected shareable images.

The order of image sections within a cluster and the order of clusters within an image section has been changed.

The PSECT_ATTR= linker option changes program section attributes when an image is being linked. The COLLECT= linker option collects program sections into specified clusters.

The default values for the GSMATCH= option have changed. An executable image that is linked with a shareable image created with the default GSMATCH= option must be relinked if the shareable image is changed.

The ANALYZE utility is provided to analyze the contents of object modules and to ensure that they conform to the object language specification.

CHAPTER 1

LINKER OVERVIEW

The VAX-11 Linker is a programming development tool that takes the output of a native mode language translator, such as an assembler or compiler, and binds it into a form that can be executed on the VAX-11 hardware. The primary input to the VAX-11 Linker is the primary output from the VAX-11 language compilers and assembler: files that contain object modules. The primary output of the linker is a file called an image.

The linker can produce three types of images. The most common type, called executable, is activated in response to a command that you enter (such as RUN). Another type of image, called system, is intended for stand-alone execution on the VAX-11 hardware. The third type, called shareable, provides a means for sharing procedures and data among multiple processes within the system. You use a shareable image by running an executable image that has been linked with it. Shareable images also provide a way of linking a very large application program in a number of smaller phases. Chapter 7 discusses image creation in detail. Chapter 8 focuses on shareable images.

The linker assigns values and virtual addresses not only to symbols defined within each module, but also to symbols defined outside the module that refers to them. If a symbol is not defined in a module named in the LINK command, the linker searches one or more libraries to find the definition of the symbol. Chapter 2 discusses the various types of symbols. Chapter 3 discusses the use of libraries.

The linker is activated by the LINK command, which you can enter interactively or within a command procedure. The LINK command permits many command and file qualifiers, most of which have default values that are suitable for most cases. One input file qualifier is /OPTIONS, which allows you to convey additional input file specifications and special instructions for the linker. Chapter 4 explains the LINK command and all its qualifiers. Chapter 5 focuses on the /OPTIONS qualifier and the special items or options that can appear in an options file.

The linker can produce, in addition to the image itself, a printable image map. You can control the level of detail provided in various parts of the map. Chapter 6 explains and illustrates the image map.

1.1 REASON FOR A LINKER

The object modules that a VAX-11 language compiler or assembler creates are nonexecutable. You must link the object modules before you can run them.

The VAX-11 language compilers and assembler require a linker for several reasons:

- Linking simplifies modular programming.
- The linker simplifies the job of each language compiler or assembler.
- The VAX-11 Symbolic Debugger and other features can be accessed easily.

1.1.1 Modular Programming

Modular programming is the process of combining separately compiled or assembled modules into an executable program or image. Modular programming has two aspects:

- Automatic modularity because different source language statements generate calls to the same routine developed by DIGITAL (such as a routine to open or close a file)
- Modular programming that you design into your program

Some routines developed by DIGITAL that perform commonly needed functions are the procedures in the VAX-11 Common Run-Time Procedure Library, which is installed in the system as a shareable image. These routines can be linked into different images regardless of the program's original source language. At run time each routine can be shared by a number of different processes because each routine is relocatable and reentrant. (Reentrant code does not modify itself and consequently can be reused by different processes.)

Users can also make their programs deliberately modular. Under this practice, a single complex program is written as a number of smaller program modules. The modules are compiled or assembled separately and later linked to create an executable image. Figure 1-1 illustrates program development in this environment. In this example, two programmers write two program modules: a main section in VAX-11 FORTRAN to perform various calculations, and a second section in VAX-11 MACRO to handle specific exception conditions.

Modular programming offers several advantages over the traditional practice of having one programmer write an entire complex program as a single source module:

- Smaller modules are usually more manageable and easier to write.
- Different modules of the same program can be written in different languages. You can select the language that best suits the nature of the module's function or your own personal preference.
- Errors are easier to analyze and correct in smaller modules.

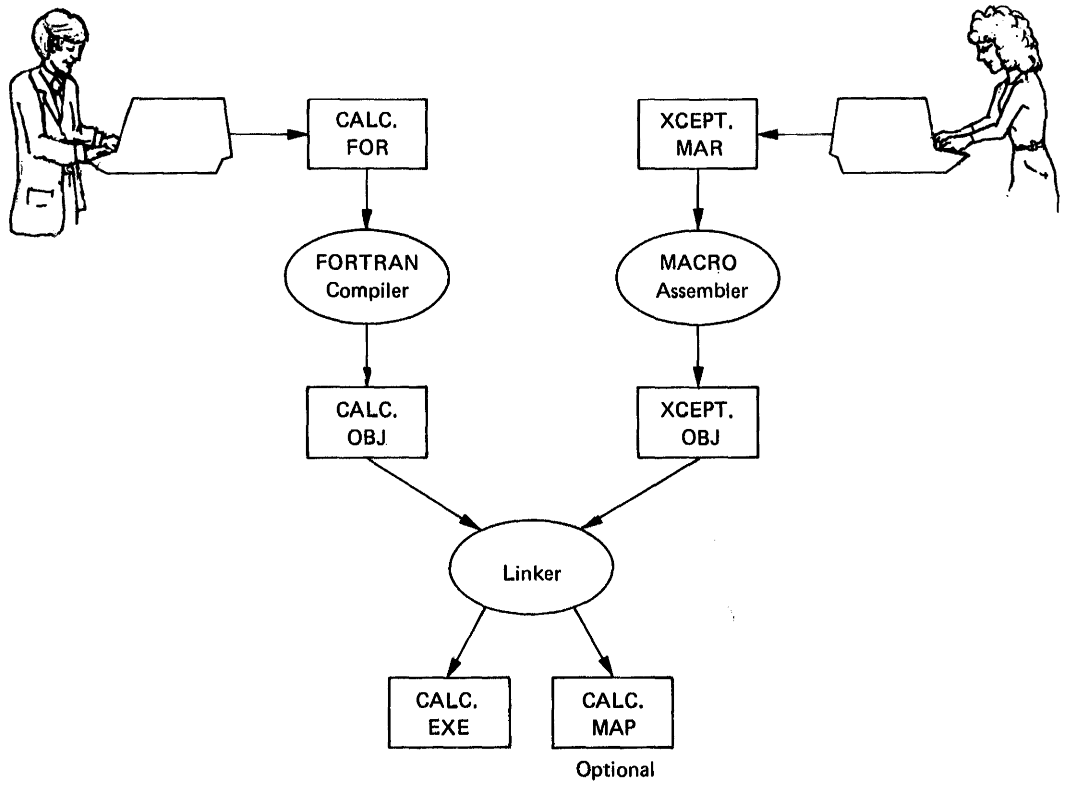

Figure 1-1 Modular Programming

1.1.2 **Simplifying Compilation And Assembly**

Having a linker perform certain essential functions eliminates the need for every native compiler and assembler to handle these functions. For example, the linker is able to allocate virtual memory and to provide the memory management interface between the program and the operating system.

A program's virtual memory can be allocated efficiently only after all its constituent modules are known. The linker contains the logic necessary to group parts of programs according to specific attributes, with the goal of conserving memory and reducing the amount of paging activity at run time.

Most programs interact with the operating system. For example, a program may use VAX/VMS system services. The linker can generate the proper program-to-system interfaces that are required.

1.1.3 **Debug Capability**

Use of the VAX-11 Linker allows you to access the VAX-11 Symbolic Debugger from the executable image. If you request the debugger, you can choose whether to activate it at run time. The VAX-11 Symbolic Debugger Reference Manual explains the capabilities and use of the debugger. Programmers should also refer to the information on debugging in their language user's guides.

1.2 LINKER OPERATION AND FUNCTIONS

The linker performs the following operations when it creates an image:

- Allocates virtual memory for the image
- Resolves symbolic references among modules
- Initializes the image contents
- Generates the image map, if requested
- Generates a symbol table file, if requested

1.2.1 Virtual Memory Allocation

The language translators that produce object modules do not allocate addresses for two reasons:

- They do not know how the modules and sections of modules will be grouped in the final executable image.
- They do not know how much address space is required for many of the external modules that are called by the module being assembled or compiled.

The linker, then, must assume the task of allocating virtual memory for the image. Each object file input to the linker consists of one or more program sections. The linker groups program sections from different object files according to various section attributes--for example, whether the program section is concatenated or overlaid and what its memory protection requirements are. For further information on how the linker maps the image, see Chapter 7.

1.2.2 Resolution Of Symbolic References

When a module makes references to symbols outside itself, the linker searches for these references in other modules explicitly named in the LINK command. If you specify any libraries, the linker searches them to resolve references made by preceding files named in the LINK command. If any references still remain unresolved, the linker searches any user-defined default libraries and the default system library. For a detailed discussion of libraries, see Chapter 3.

1.2.3 Image Initialization

After it maps virtual memory and resolves references, 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.

1.2.4 **Image Map**

If you so request, the linker generates an image map. The actual contents of the map depend on the map-related command qualifiers that you enter with the LINK command; however, entering just the /MAP qualifier generates a default map with the following sections:

- An object module synopsis
- A program section synopsis
- A list of symbols, with the name and value of each
- An image synopsis
- Statistics of the link run

Chapter 6 discusses the command qualifiers that affect the image map. It also illustrates the map sections and explains significant items.

1.2.5 **Symbol Table File**

If you so request, the linker produces a file that records the values of symbols defined within the image. Section 2.3.1 contains further information on the symbol table file.

CHAPTER 2

SYMBOLS AND REFERENCES

One of the linker's functions is to resolve symbolic references between modules. The linker recognizes different types of symbols and follows guidelines for each type when it tries to supply addresses or values to statements that refer to these symbols.

2.1 DEFINITIONS: "SYMBOL" AND "REFERENCE"

A symbol is a name associated with a coding statement or with a data area or field. A reference is the use of a symbol in a program statement or a data definition. Consider the following examples (not tied to a specific programming language):

- A program statement identified as ROUTINEA moves FIELDA to FIELDB. ROUTINEA is the symbol associated with the program statement. FIELDA and FIELDB are references made by the statement.
- A data definition statement defines FIELDA as being equal to (A+B)/2. FIELDA is the symbol associated with the computed value of (A+B)/2. A and B are references.

2.2 TYPES OF SYMBOLS AND REFERENCES

Each symbol is local, global, or universal:

- Local symbols are available for reference only within the program module that defines them.
- Global symbols can be referred to by modules outside the module that defines them. A global symbol has a strong or a weak definition (see Section 2.2.2). Another module can make a strong or a weak reference to a global symbol (regardless of whether the symbol's definition is weak or strong). The linker resolves the global references with the global definition in the other module.
- Universal symbols are a special type of global symbol that can be specified only for shareable images.

Figure 2-1 illustrates references to local and global symbols in three modules. (The statements do not reflect a specific programming language.) An arrow is drawn between each reference and the symbol to which it refers.

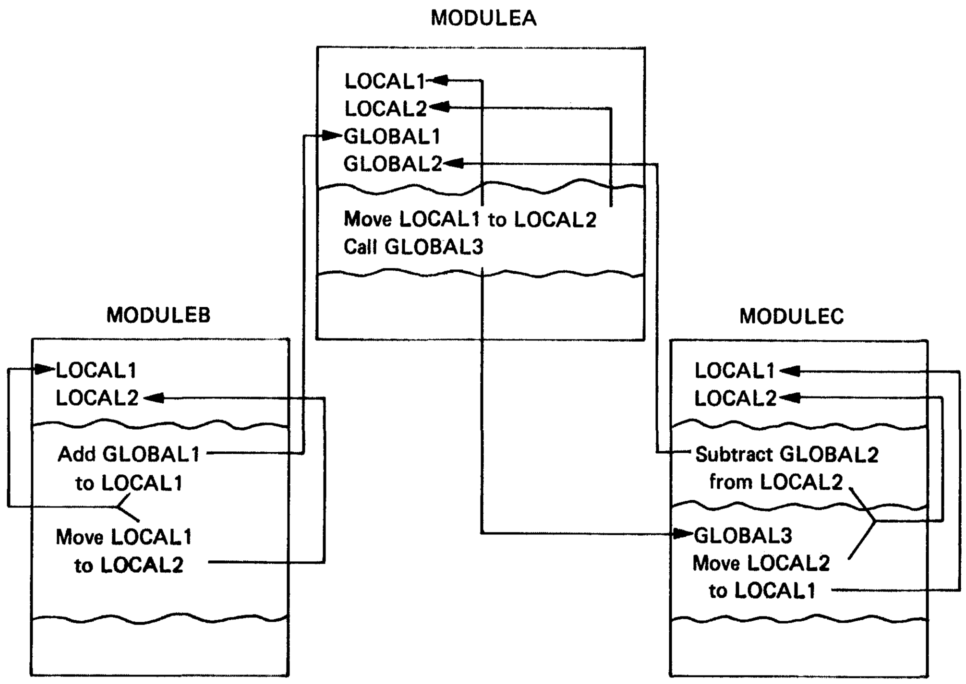

Figure 2-1 Local and Global Symbols

Local and global symbols can be designated either automatically by the language compiler or explicitly by qualifiers in program statements. You can specify the local or global symbol type only in certain languages. In VAX-11 MACRO, for example, you can define a symbol as local or global by using one or two equal signs or colons, as the following statements show. Note that the term "local symbol" in this context is different from the term "local label" (for example, 10$:) in the context of a MACRO program.

Statement	Effect
CRFC_MAXREC=292	Assigns a value of 292 to the local symbol CRFC_MAXREC
CRFC_MAXREC==292	Assigns a value of 292 to the global symbol CRFC_MAXREC
ERR_BRANCH:	Makes the coding statement label ERR_BRANCH a local symbol
ERR_BRANCH::	Makes the coding statement label ERR_BRANCH a global symbol

In certain other languages, the compiler determines whether a symbol is local or global. For example, the FORTRAN compiler makes statement numbers local symbols, and it makes module entry points and common area names global symbols. For information about designating symbol type in a specific programming language, see the appropriate language reference manual.

Universal symbols must be specified by the UNIVERSAL= option in the linker options file. Chapter 5 explains the use of the /OPTIONS qualifier with the LINK command.

2.2.1 Local Symbols

You can refer to local symbols only within the object module that defines them. Most symbols in a typical program are local.

The compiler or assembler resolves references to local symbols, and therefore they are not passed on to the linker.

2.2.2 Global Symbols

Global symbols can be referred to by object modules other than the module that defines them.

Each global symbol has either a strong or a weak definition. An external module can make a strong reference or a weak reference to any global symbol.

2.2.2.1 **Strong Definition** - A global symbol with a strong definition is available for reference if the module that defines it is either explicitly named in the LINK command or contained in a library that is searched by the linker. Global symbols usually have a strong definition, and strong is the default if neither weak nor strong is specified.

The librarian utility makes an entry for each global symbol with a strong definition in the global symbol table of a library. Libraries are discussed in Chapter 3.

2.2.2.2 **Weak Definition** - A global symbol with a weak definition is available for reference only if the module that defines it is explicitly included in the linking operation; that is, the module is listed as an input file, specified with the INCLUDE qualifier, or included from a library because another (strong) symbol in the module is needed. Weak symbols can be defined in VAX-11 MACRO and VAX-11 BLISS. See the VAX-11 MACRO Language Reference Manual for more information.

The librarian utility routine does not make entries for global symbols with weak definitions in the global symbol table of a library.

2.2.2.3 **Strong Reference** - A strong reference is one whose resolution is critical to the linking operation. If the linker cannot resolve all strong references by searching named input modules and libraries and the default system library, it reports errors and assumes that the symbol referred to has a value of zero.

Most references to global symbols are strong, and strong is the default.

2.2.2.4 **Weak Reference** - A weak reference is one whose resolution is not critical to the linking operation. For a weak reference, the linker searches only named input modules, but not user libraries or the default system library. The linker does not treat an unresolved weak reference as an error, but it does assume that the symbol referred to has a value of zero. Weak references can be made in VAX-11 MACRO.

An example of the use of weak references might occur in a program that you want to link now, but that you want to add to and relink later. In a particular subroutine you might make a weak reference to a symbol in an external module that will not be written until later. You can link the image and run it, as long as it does not try to use the nonexistent symbol during the run.

2.2.3 **Universal Symbols**

A universal symbol is a special type of global symbol in a shareable image. A universal symbol is accessible by other modules when they link with the shareable image. Universal symbols in a shareable image contrast with ordinary global symbols in the modules that make up the shareable image; the ordinary global symbols are available only when the modules are being linked to create the shareable image.

VAX-11 MACRO provides the .TRANSFER directive to identify an important class of universal symbols, namely transfer vectors. Otherwise, you must identify universal symbols with the UNIVERSAL= option in a linker options file (see Chapter 5). For example, the following LINK command shows how to designate A and B as universal symbols in the shareable image ABBOTT. COSTELLO is an options file that includes the record UNIVERSAL=A,B.

```
     $ LINK/SHAREABLE ABBOTT,COSTELLO/OPTIONS
```

COSTELLO.OPT

```
UNIVERSAL=A,B
      .
      .
      .
```

An example of the need for universal symbols might occur if you write an error-handling routine with several modules to be linked as a shareable image. You define global symbols for references between the modules. However, you must designate as universal any global symbols that are to be available when an executable image is linked with the shareable image. For example, the executable image needs access to the error-handling shareable image's entry points and perhaps some constants for defining possible errors.

2.3 **SYMBOL TABLES**

An image can have neither, one, or both of the following symbol tables:

- A debug symbol table
- A global symbol table

The debug symbol table is included only if you specify /DEBUG at link time. This debug symbol table normally contains the following types of information:

- Module names
- Routine names and/or program section names
- All local symbols

However, the local symbols are included only if you request debug at both compilation time and link time.

The global symbol table is included in an executable image whenever you include debug in the link. The global symbol table is always included in a shareable image, regardless of the qualifiers you specify at link time. The global symbol table contains an entry for each global symbol in an executable image and for each universal symbol in a shareable image. These symbols are listed in the Symbols by Name section of the image map.

2.3.1 Global Symbol Table As Separate Output

You can output a copy of the image's global symbol table as a separate file by using the /SYMBOL_TABLE qualifier at link time. The symbol table file is a sequential file containing variable-length records. Its format is identical to that of object modules (Appendix C explains this format in detail).

You can specify a symbol table file as input to a linking operation. The linker makes the global symbols in the symbol table file and their values available to the object modules being linked, without also linking the entire image with which the global symbols are associated. One primary use for specifying symbol table files at link time is to make global symbols in a system image available to a number of other images without binding the system image into each of the other images.

CHAPTER 3

LIBRARIES

A library contains object modules and related information, including a list of the names of the modules and a list of the global symbols contained in the modules. (A library can contain modules other than object modules; however, the linker is only concerned with object libraries.) The linker searches one or more libraries to resolve references to global symbols that are not defined in the object files previously specified in the LINK command.

When the linker matches a global symbol having an unresolved strong reference with an entry in a library's table of global symbols, it binds the module that defines the symbol into the image. You can eliminate the need for the linker to search the global symbol table of the library by explicitly including modules from a library in an image. In addition to any libraries that you specify, the linker automatically searches the following for any unresolved strong references:

- User-defined default libraries, if there are any (See Section 3.3)

- The default system library

To create a library, you must use the LIBRARY command, which is explained in the VAX/VMS Command Language User's Guide.

3.1 LIBRARY TABLES USED BY THE LINKER

Each object module library contains two lists or tables that the linker uses to resolve symbolic references:

- A module name table, containing an entry for each object module in the library. Each entry includes the name of the module and its location within the library file.

- A global symbol table, containing an entry for each global symbol in the modules in the library. Each entry includes the name of the symbol and the location of the module that defines the symbol.

For example, in a hypothetical library named MINE2, one of the modules is MODULEZ, which contains the global symbols TAG1 and TAG2. Although it is not intended as an exact schematic illustration, Figure 3-1 shows the relationship of the module name table and the global symbol table to the rest of the library.

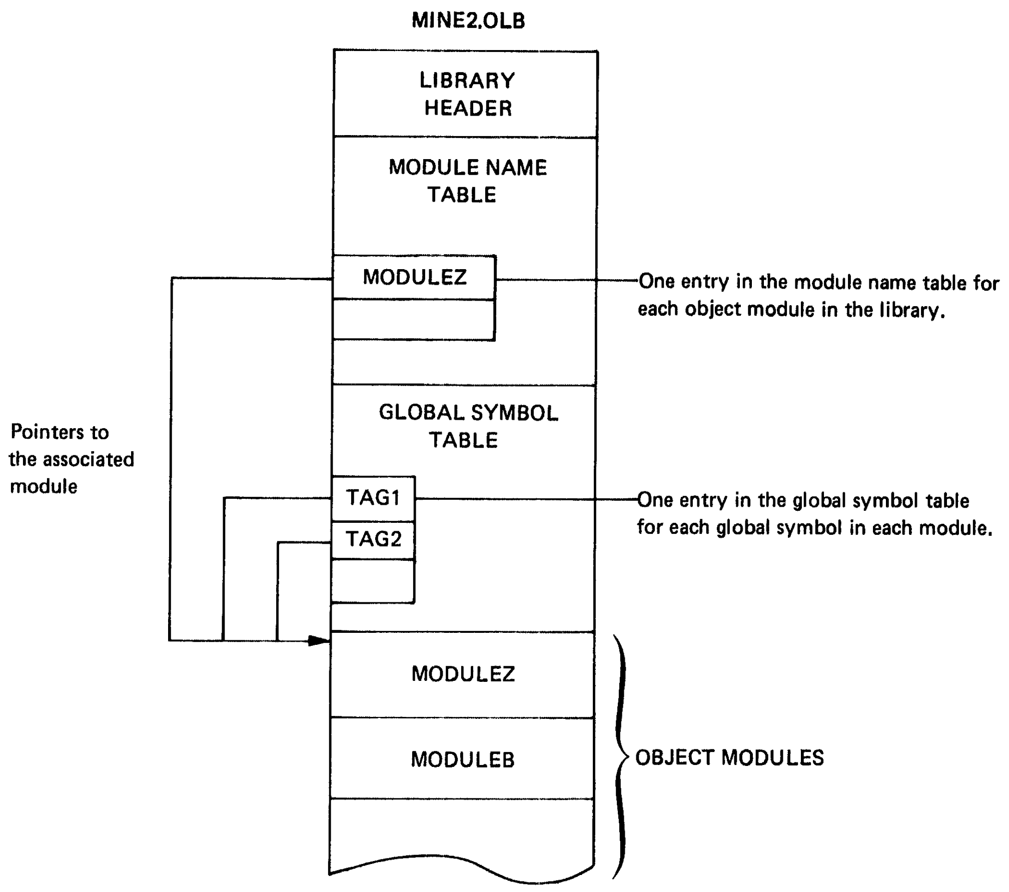
MINE2.OLB
LIBRARY HEADER
MODULE NAME TABLE
MODULEZ
One entry in the module name table for each object module in the library.
GLOBAL SYMBOL TABLE
Pointers to the associated module
TAG1
TAG2
One entry in the global symbol table for each global symbol in each module.
MODULEZ
MODULEB
OBJECT MODULES

Figure 3-1 Library Tables

3.2 LINKER'S USE OF LIBRARIES

You can include library modules in the image either implicitly or explicitly:

- Implicit inclusion occurs when a module specified in the LINK command refers to a global symbol defined in a library that the linker searches. For example, an instruction in a module named MODULE1 moves FIELDA to FIELDB, yet FIELDB is defined only in the module LIBMOD3 in the library BOBLIB.OLB. You can specify:

  ```
  $ LINK MODULE1,BOBLIB/LIBRARY
  ```

  This causes the linker to search BOBLIB for any unresolved references from MODULE1. When it discovers that FIELDB is defined in LIBMOD3, the linker includes that module in the image.

- Explicit inclusion occurs when you name a module with the /INCLUDE qualifier after the library name. To use the example in the explanation of implicit inclusion, if you know that FIELDB is defined in module LIBMOD3 in BOBLIB, you can simplify the linker's search and explicitly include LIBMOD3 in the final executable image by specifying:

  ```
  $ LINK MODULE1,BOBLIB/INCLUDE=LIBMOD3
  ```

The linker follows these conventions in using libraries:

- It processes all input files, including libraries, in the sequence in which you name them. Thus, the linker searches a library for unresolved strong references only from previously named input files. For example, assume that you enter the following command:

  ```
  $ LINK A,B,C/LIBRARY,D,E
  ```

  The linker searches library C for unresolved strong references from object modules A and B, but not D and E. The search of library C continues until no more symbols can be resolved. For example, if module X is included from library C and module X also has some unresolved strong references, the linker makes another search of library C.

- If you specify both the /LIBRARY and /INCLUDE qualifiers after a library's file specification, the linker includes the named modules first and then, if necessary, searches the library. This is true regardless of the order of the two qualifiers. For example, the following two commands cause the linker to perform identical actions:

  ```
  $ LINK A,B/INCLUDE=(MOD1,MOD2)/LIBRARY
  $ LINK A,B/LIBRARY/INCLUDE=(MOD1,MOD2)
  ```

- The linker searches the following default libraries for unresolved strong references after it has processed all named input files, including user libraries:

  - Any user-defined default libraries (see Section 3.3)

  - The default system library (see Section 3.4)

These conventions allow you considerable choice when the same global symbol name is defined differently in modules in different libraries. For example, if you know that a particular symbol is defined as you need it in a particular module but is defined differently in another module (in one of your libraries or the default system library), you can choose the desired definition by specifying the module with the /INCLUDE qualifier. If you know that your own library has global symbols that are defined differently in the default system library, you can include your own symbols by specifying your library with the /LIBRARY qualifier.

3.3 USER-DEFINED DEFAULT LIBRARIES

You can define one or more default libraries for the linker to search before it searches the default system library. You must equate the logical names LNK$LIBRARY, LNK$LIBRARY_1, and so on (up to LNK$LIBRARY_999) to the file specifications of your default libraries. You can define these logical names in the process, group, and system logical name tables.

The linker automatically searches any user-defined default libraries for unresolved strong references, unless you specify the /NOUSERLIBRARY qualifier in the LINK command. You can specify in the /USERLIBRARY qualifier the tables, PROCESS, GROUP, or SYSTEM, that you want the linker to search. If you specify /USERLIBRARY without a table list (or do not specify it at all), the linker assumes all tables are to be searched (/USERLIBRARY=ALL). The search of user-defined 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$LIBRARY. If this entry exists, the linker translates the logical name and searches the specified library for unresolved strong references. If you exclude PROCESS from the table lsit in /USERLIBRARY or if no entry exists for LNK$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 linker searches the default system library (if any unresolved strong references remain).

An example use of user-defined default libraries is that you may want the linker to use your private libraries MINE1.OLB and MINE2.OLB as default libraries, and everyone in your group may want the linker to use [PROJX]PROJECTX.OLB as a default library. Moreover, the system manager may want SYS$LIBRARY:MYSITE.OLB as a default library for all users. The following commands make the necessary logical name definitions (the GRPNAM and SYSNAM privileges are required to use the /GROUP and /SYSTEM qualifiers, respectively):

```
$ DEFINE LNK$LIBRARY DBA1:[GOLD]MINE1
$ DEFINE LNK$LIBRARY_1 DBA1:[GOLD]MINE2
$ DEFINE/GROUP LNK$LIBRARY DBA1:[PROJX]PROJECTX
$ DEFINE/SYSTEM LNK$LIBRARY SYS$LIBRARY:MYSITE
```

Note that the logical names in each table must be used consecutively (LNK$LIBRARY, LNK$LIBRARY_1, LNK$LIBRARY_2, ...). In the preceding example, if you were to delete LNK$LIBRARY from the process logical name table, the linker would not search for LNK$LIBRARY_1 in the process logical name table, because it proceeds to the next table as soon as logical name translation fails.

3.4 DEFAULT SYSTEM LIBRARY

If any unresolved strong references remain after the linker has processed all your input, it begins a search of the default system library. This "library" is in fact two files: one a shareable image called VMSRTL (default file specification is SYS$LIBRARY:VMSRTL.EXE) and the other an object library called STARLET (default file specification is SYS$LIBRARY:STARLET.OLB).

3.4.1 VMSRTL

If the linker needs to search the default system library, it searches the VMSRTL shareable image first. This shareable image contains most of the procedures described in the VAX-11 Run-Time Procedure Library Reference Manual, including many routines required by almost all high level language programs.

If the linker finds no symbols that it needs in the shareable image, it does not include the shareable image VMSRTL in the image being created.

You can use the /NOSYSSHR qualifier with the LINK command to suppress the linker's search of VMSRTL (see Chapter 4).

3.4.2 STARLET

STARLET is an object module library. It contains all the object files used to create the shareable image version of the Run-Time Library, as well as other less frequently used procedures. This object library also contains modules for interfacing to VAX/VMS system services.

The linker searches STARLET if any unresolved strong references remain after it has searched VMSRTL.

You can use the /NOSYSLIB qualifier to the LINK command to suppress the linker's search of both STARLET and VMSRTL (see Section 4.2.1).

3.5 EXAMPLE OF USING LIBRARIES

The following example shows how you can specify both explicit and implicit inclusion of modules from libraries. (Assume that there are no user-defined default libraries.)

```
$ LINK LAUREL,HARDY-
     MINE2/INCLUDE=MODULEZ,-
     MINE3/LIBRARY
```

These statements tell the linker:

1. Link the object modules LAUREL and HARDY.

2. Extract MODULEZ from the library MINE2 and link it with the object modules LAUREL and HARDY.

3. If any unresolved strong references remain in LAUREL, HARDY, or MODULEZ, search the library MINE3, and extract and link any modules needed to resolve these references.

4. For any strong references that are still unresolved, search the default system library.

Note that the linker will not search MINE3.OLB and the default system library if the only unresolved references are weak references. For a discussion of weak references, see Section 2.2.2.4.

CHAPTER 4

THE LINK COMMAND

To invoke the VAX-11 Linker, use the DIGITAL Command Language (DCL) LINK command. You can enter the LINK command interactively, or you can include it in a command procedure.

The LINK command recognizes a number of command qualifiers and file qualifiers. A command qualifier conveys information about the linking operation and the image to be created -- for example, whether to generate an image map, or whether to include a debugger in the image. A file qualifier specifies information about a file that is input to the linker -- for example, identifying the file as a library. Some qualifiers are valid only if they are used with other qualifiers, and some qualifiers are incompatible with other qualifiers.

This chapter discusses the LINK command and its qualifiers, but discusses command syntax only where necessary to avoid errors or misunderstanding. For detailed information on command syntax, see the VAX/VMS Command Language User's Guide.

4.1 COMMAND FORMAT

The LINK command is usually entered in the following format:

```
     $ LINK/command-qualifier... file-spec/file-qualifer,...
```

You must enter at least the LINK command name and one input file name. You can enter multiple command qualifiers and file specifications, and one or more file qualifiers for each file specification.

Slashes (/) separate qualifiers from each other and from the command name or file specification with which they are associated. One or more spaces or tabs must separate the last command qualifier from the first input file specification. A comma must precede the second and subsequent input file specifications.

The following examples show some acceptable formats of the LINK command (Section 4.3 explains these examples).

```
     $ LINK PROGA

     $ LINK/MAP/DEBUG PAYROLL,FICA,PAYLIB/LIBRARY

     $ LINK/MAP/FULL/EXECUTABLE=STOOGES CURLY,-
         LARRY,MOE,TVLIB/INCLUDE=OLDIES,-
         COMEDY/LIBRARY,SLAPSTICK/OPTIONS
```

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 (see Section 4.2). In the second of the preceding examples, the image file and the map file will be named PAYROLL. In the third example, the image file will be named STOOGES because you so specified with the /EXECUTABLE qualifier, but the map file will be named CURLY. (To name the map file STOOGES, you must specify /MAP=STOOGES.)

4.2 COMMAND AND FILE QUALIFIERS

You can enter many command and file qualifiers, but normally you will not need to because most qualifiers have default values that the linker uses if you omit the qualifier.

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. 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 (say, /EXECUTABLE and /NOEXECUTABLE), the linker accepts the last one entered.

Tables 4-1 and 4-2 list the command and file qualifiers, the default value for each command qualifier, the function for each file qualifier, and incompatible qualifiers. A [NO] indicates that the qualifier can be negated by prefixing NO (without brackets) -- for example, /NODEBUG or /NOEXECUTABLE. Any entry after a negative qualifier is not valid; for example, it would be nonsense to enter /NOEXECUTABLE=PAYROLL.

Sections 4.2.1 and 4.2.2 discuss the command qualifiers and file qualifiers individually. Within each section the qualifiers are presented in alphabetical order.

Table 4-1
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)	
/P0IMAGE		
/PROTECT		/SYSTEM, /EXECUTABLE
/[NO]SHAREABLE[=file-spec]	/NOSHAREABLE	/SYSTEM, /DEBUG, /EXECUTABLE
/[NO]SYMBOL_TABLE[=file-spec]	/NOSYMBOL_TABLE	
/[NO]SYSLIB	/SYSLIB	
/[NO]SYSSHR	/SYSSHR	/NOSYSLIB
/[NO]SYSTEM[=base-address]	/NOSYSTEM	/DEBUG, /SHAREABLE
/[NO]TRACEBACK	/TRACEBACK	
/[NO]USERLIBRARY[=(table[,...])]	/USERLIBRARY=ALL	

Table 4-2
File Qualifiers

File Qualifier	Function	Incompatible Qualifiers
/INCLUDE=module-name[,...]	Includes one or more object modules from a library in the link	All others, except /LIBRARY
/LIBRARY	Identifies an object module library	All others, except /INCLUDE
/OPTIONS	Identifies a linker options file	All others
/SELECTIVE_SEARCH	Includes only global symbols referred to by previously named input files	All others, except /SHAREABLE
/SHAREABLE[=[NO]COPY]	Identifies a shareable image input file; valid only in a linker options file	All others, except /SELECTIVE_SEARCH

4.2.1 Command Qualifiers

/BRIEF

/BRIEF produces a brief form of the image map. A brief map contains only the following sections:

- Object Module Synopsis
- Image Synopsis
- Link Run Statistics

A brief map does not contain the Program Section Synopsis and the Symbols by Name sections, which are included in the default map.

/BRIEF is valid only if you also specify /MAP in the LINK command. /BRIEF is incompatible with /FULL and /CROSS_REFERENCE.

For illustrations and explanations of the image map sections, see Section 6.2.

/CONTIGUOUS
/NOCONTIGUOUS

/CONTIGUOUS forces the entire image to be placed in consecutive disk blocks. If sufficient contiguous space is not available on the output disk, the linker reports the error and terminates the link operation without generating an image.

You can use the /CONTIGUOUS qualifier to improve paging performance for all types of images because an image usually runs slower if it is not contiguous. You can also use the /CONTIGUOUS qualifier to satisfy the requirement of bootstrap programs for certain system images, since many bootstrap programs cannot handle discontiguous images.

If you do not specify /CONTIGUOUS, the linker assumes /NOCONTIGUOUS by default. That is, if sufficient contiguous space is not available, the image is divided and placed in different areas on disk. (However, the operating system still tries to make the image as close to contiguous as possible.)

/CROSS_REFERENCE
/NOCROSS_REFERENCE

/CROSS_REFERENCE causes the Symbols by Name section of the image map to be replaced by a Symbol Cross Reference section, which lists global symbols in alphabetical order and the following information about each symbol:

- Its value
- The name of the first module that defines it
- The name of each module that refers to it

The number of symbols listed in the cross reference depends on whether you specified /FULL for the map or accepted the default map. A full map contains global symbols from all modules in the image, including modules extracted from libraries. The default map generally excludes global symbols that are defined and referred to only within the default system library.

/CROSS_REFERENCE is valid only if you also specify /MAP in the LINK command. /CROSS_REFERENCE is incompatible with /BRIEF.

If you do not request a cross reference, none is provided; the map still lists global symbols in alphabetical order, but gives only the value for each one.

/DEBUG[=file-spec]
/NODEBUG

/DEBUG tells the linker to bind a debugging module into the image. When the image is run, the debugger receives control first. /DEBUG does not have any effect on the location of code within the image; the image map is the same with /DEBUG or /NODEBUG.

If you specify /DEBUG, you can also enter the file specification of a user-written debug module. If you enter a debugging module file specification without specifying the file type, the linker assumes OBJ.

If you specify /DEBUG without entering a file specification, the linker uses the VAX-11 Symbolic Debugger. This debugger includes a debug symbol table (discussed in Section 2.3) and coding logic to help in debugging the image at run time. For further information, see the VAX-11 Symbolic Debugger Reference Manual.

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

If you do not specify /DEBUG, the linker assumes /NODEBUG.

/EXECUTABLE[=file-spec]
/NOEXECTABLE

/EXECUTABLE tells the linker to create an executable image, as opposed to a shareable image or a system image. You can also enter a file specification for the image; however, if you do not enter one, the linker uses the file name of the first input file or if you specify /EXECUTABLE after an input file specification, the name of the modified file. If you do not enter a file type after the filename, the linker assumes a file type of EXE.

If you specify both /SYSTEM and /EXECUTABLE, the linker creates a system image but uses the /EXECUTABLE qualifier to determine the image's file-specification.

/NOEXECUTABLE tells the linker to perform all the actions involved in creating an executable image, but not to output it. You can use /NOEXECUTABLE to test combinations of files and qualifiers without actually creating an image.

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

/FULL

/FULL produces the most complete map of the image. The full map contains all the sections found in the default map, although several sections contain more detailed information. The full map also contains two sections not found in the default map. The following sections of a full map contain information about all modules in the image. (In the default map, these sections generally omit information about modules from the default system library.)

- Object Module Synopsis
- Program Section Synopsis
- Symbols by Name

The following sections are included in a full map, but not in the default map:

- Image Section Synopsis
- Symbols by Value

For illustrations and explanations of the image map sections, see Section 6.2.

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

/HEADER

/HEADER is used with /SYSTEM; it tells the linker to create a system image with a header. If you specify /SYSTEM without /HEADER, the linker creates a system image without a header. Executable and shareable images always have image headers; consequently, /HEADER is ignored in LINK commands creating these images.

/MAP[=file-spec]
/NOMAP

/MAP causes the linker to create an image map as a separate file. You can enter a file specification for the image map file; however, if you do not enter one, the linker uses the file name of the first input file or, if you specify /MAP after an input file specification, the name of the modified file. If you do not enter a file type after the file name, the linker assumes a file type of MAP.

If you enter /MAP, you can further specify the contents of the map with the /BRIEF, /FULL, and /CROSS_REFERENCE qualifiers. If you enter /MAP and no related qualifier, the linker produces a default map that contains the following sections:

- Object Module Synopsis
- Program Section Synopsis
- Symbols by Name
- Image Synopsis
- Link Run Statistics

For illustrations and explanations of the image map sections, see Section 6.2.

If you do not specify /MAP, the default for interactive mode is /NOMAP; that is, the linker does not generate an image map. In a batch job the default is /MAP; that is, the linker generates its standard (default) map.

/P0IMAGE

/P0IMAGE is used to create executable images that modify P1 address space. The linker places the stack and RMS buffers that usually go in P1 address space in P0 address space. See the <u>VAX-11 Architecture Handbook</u> for a description of P0 and P1 address space.

/PROTECT

/PROTECT is used with /SHAREABLE; it tells the linker to protect the shareable image, making it a privileged shareable image. A privileged shareable image can execute change mode instructions even when it is linked into a nonprivileged executable image. See the <u>VAX/VMS Real-Time User's Guide</u> for more information on privileged shareable images.

/SHAREABLE[=file-spec]
/NOSHAREABLE

/SHAREABLE tells the linker to create a shareable image. (For an explanation of shareable images, see Section 7.6.2 and Chapter 8.) You can also enter a file specification for the shareable image; however, if you do not enter one, the linker uses the file name of the first input file or, if you specify /SHAREABLE after an input file specification, the name of the modified file. If you do not enter a file type after the file name, the linker assumes a file type of EXE.

You cannot run a shareable image, but you can link it with object modules or other shareable images. (See the explanation of the /SHAREABLE file qualifier in Section 5.1.2.)

If you specify /SHAREABLE, you cannot specify /SYSTEM or /DEBUG. If you specify both /SHAREABLE and /EXECUTABLE, the linker ignores /EXECUTABLE and creates a shareable image.

If you do not specify /SHAREABLE, the linker assumes /NOSHAREABLE; that is, the image is not a shareable image. (See the explanation of the /EXECUTABLE command qualifier in this section.)

/SYMBOL_TABLE[=file-spec]
/NOSYMBOL_TABLE

/SYMBOL_TABLE tells the linker to create a separate file, with a default file type of STB, containing the image's global symbol table. This qualifier does not affect the global symbol table in the image itself; rather, it causes an additional global symbol table to be created in object module format. You can also enter a file specification for the global symbol table file; however, if you do not make this entry, the linker uses the name of the first input file or, if you specify /SYMBOL_TABLE after an input file specification, the name of the modified file.

You can include the symbol table file as input to future linking operations, just as if it were an object module. For further information, see Section 2.3.1.

If you do not specify /SYMBOL_TABLE, the linker assumes /NOSYMBOL_TABLE; that is, it does not generate a symbol table file.

/SYSLIB
/NOSYSLIB

/SYSLIB tells the linker to search the default system library for unresolved strong references to global symbols after it has searched any specified user libraries and any user-defined default libraries. (Section 3.4 explains the default system library.) You will probably want the linker to search the default system library for almost all linking operations. If you do not specify /NOSYSLIB, the linker assumes /SYSLIB by default.

/NOSYSLIB tells the linker not to search the default system library (VMSRTL.EXE and STARLET.OLB). You should specify /NOSYSLIB only if you know that other specified libraries allow the linker to resolve all symbolic references and if you have a good reason for suppressing the system library search. If you specify both /NOSYSLIB and /SYSSHR, the /SYSSHR qualifier is ignored.

/SYSSHR
/NOSYSSHR

/SYSSHR tells the linker to search the default system run time library shareable image (VMSRTL.EXE) for unresolved strong references to global symbols. If any symbol within this shareable image resolves an outstanding reference, the shareable image is included in your program as the highest-addressed part of the program region.

The primary use of this qualifier, however, is in its negative form. /NOSYSSHR tells the linker not to try to resolve symbolic references by including the default system shareable image. Instead it directs the linker to use only the default object library (STARLET.OLB), which includes all the routines in VMSRTL. To tell the linker to search neither the default system shareable image nor the default system object library, use the /NOSYSLIB qualifier.

/SYSTEM[=base-address]
/NOSYSTEM

/SYSTEM tells the linker to create a system image. (For an explanation of system images, see Section 7.6.3.) You can also specify a base address at which the system image will be loaded at run time, and you can express this address in decimal (%D), hexadecimal (%X), or octal (%O) format. If you specify /SYSTEM without a base address, the linker assumes %X80000000. The linker uses the filename of the first input file and the file type EXE. If you want a different output file-specification, you must also specify /EXECUTABLE.

If you specify /SYSTEM, you cannot specify /SHAREABLE or /DEBUG.

If you do not specify /SYSTEM, the linker assumes /NOSYSTEM; that is, the image is not a system image. (See the explanation of the /EXECUTABLE command qualifier in this section.)

/TRACEBACK
/NOTRACEBACK

/TRACEBACK tells the linker to include traceback information in the image. Traceback is a facility that automatically displays information from the call stack when a fatal program error occurs. The output shows which modules were called before the error occurred.

The linker assumes /TRACEBACK unless you exclude the facility by specifying /NOTRACEBACK. If you enter /DEBUG, the linker automatically includes traceback also; therefore, if you specify both /DEBUG and /NOTRACEBACK, you receive a warning that /NOTRACEBACK has been ignored.

/USERLIBRARY[=(table[,...])]
/NOUSERLIBRARY

/USERLIBRARY tells the linker to search any user-defined default libraries after it has searched any specified user libraries.

The linker searches the process, group, and system logical name tables to find the file specifications of the user-defined libraries. (Section 3.3 explains user-defined default libraries.) You can specify the following tables that you want the linker to search:

ALL	The linker searches the process, group, and system logical name tables for user-defined library definitions.
GROUP	The linker searches the group logical name table for user-defined library definitions.
NONE	The linker does not search any logical name table; /USERLIBRARY=NONE is equivalent to /NOUSERLIBRARY.
PROCESS	The linker searches the process logical name table for user-defined library definitions.
SYSTEM	The linker searches the system logical name table for user-defined library definitions.

If you do not specify either /NOUSERLIBRARY or /USERLIBRARY=(table), the linker assumes /USERLIBRARY=ALL by default.

/NOUSERLIBRARY tells the linker not to search any user-defined default libraries.

4.2.2 **File Qualifiers**

/INCLUDE=module-name[,...]

/INCLUDE tells the linker to include the named module or modules from the associated library in the image. To specify more than one module, enclose the list in parentheses and separate module names with commas. /INCLUDE does not cause the linker to search the rest of the associated library for unresolved references, unless you also specify /LIBRARY. For further information on libraries, see Chapter 3.

The following two examples show uses of the /INCLUDE qualifier with a library named NATIONAL that contains many modules, among them REDS, DODGERS, and PHILS.

```
     $ LINK LEAGUE,NATIONAL/INCLUDE=(REDS,DODGERS,PHILS)
```

This example tells the linker to extract modules REDS, DODGERS, and PHILS from the library NATIONAL and include them in the executable image which will be named LEAGUE (since that is the name of the first input file).

```
     $ LINK  LEAGUE,NATIONAL/LIBRARY/INCLUDE=(REDS,DODGERS,PHILS)
```

This example also tells the linker to include REDS, DODGERS, and PHILS in LEAGUE. However, the /LIBRARY qualifier tells the linker to search the rest of the library NATIONAL and link in any other modules needed to resolve strong symbolic references in LEAGUE, REDS, DODGERS, and PHILS.

/LIBRARY

/LIBRARY identifies a file as a library. The linker searches libraries that you specify if any unresolved strong symbolic references between modules remain after it links the previously named input files and any previously named library modules specified with the /INCLUDE qualifier. For further information on libraries, see Chapter 3.

/LIBRARY cannot be the only qualifier on the first input file, since there are as yet no outstanding references to be resolved from this library.

/OPTIONS

/OPTIONS identifies a file as a linker options file. This file can contain input file specifications, as well as special instructions recognized only by the linker and not by the command interpreter.

Chapter 5 explains how to create an options file and what it can contain. Chapter 5 also discusses each of the special instructions you can include in the options file.

/SELECTIVE_SEARCH

/SELECTIVE_SEARCH tells the linker to include in the image's global symbol table only those global symbols in the associated file that previously named input files refer to. If you do not specify /SELECTIVE_SEARCH for an input file, all of its global symbols are included in the global symbol table of the image.

/SHAREABLE[=[NO]COPY]

/SHAREABLE as an input file qualifier is valid only within a linker options file. Section 5.1.2 explains the use of the /SHAREABLE file qualifier.

4.3 EXAMPLES

1. `$ LINK PROGA`

   The linker binds the object module PROGA and creates an executable image named PROGA. The linker searches any user-defined default libraries and the default system library for any unresolved strong symbolic references in PROGA.OBJ. All linker defaults are used.

2. `$ LINK/MAP/DEBUG PAYROLL,FICA,PAYLIB/LIBRARY`

   The linker binds object modules PAYROLL and FICA, searching the library PAYLIB for unresolved strong references in the two object modules before searching any user-defined default libraries or the default system library. The linker also includes the VAX-11 Symbolic Debugger in the image.

   The name of the executable image is PAYROLL. The linker also generates an image map (in the default map format) with a file name of PAYROLL and a file type of MAP.

3.
```
$ LINK/MAP/FULL/EXECUTABLE=STOOGES CURLY,-
     LARRY,MOE,TVLIB/INCLUDE=OLDIES,-
     COMEDY/LIBRARY,SLAPSTICK/OPTIONS
```

The linker binds object modules CURLY, LARRY, and MOE, as well as the module OLDIES from the library TVLIB. The linker searches the library COMEDY for any unresolved symbolic references in CURLY, LARRY, MOE, and OLDIES, before searching any user-defined default libraries or the default system library. The linker uses the options file SLAPSTICK for additional input file specifications or special instructions.

The linker generates a full map, with the default file name of CURLY and the file type of MAP. The executable image is named STOOGES.

CHAPTER 5

THE /OPTIONS FILE QUALIFIER

The /OPTIONS file qualifier identifies a linker options file. You can include two types of information in this file:

- Input file specifications and associated file 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

When you specify an options file at link time, the linker reads the file before performing the linking operation.

5.1 USES FOR AN OPTIONS FILE

You can create an options file and use the /OPTIONS qualifier for a number of reasons:

- 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 to the link operation
- To enter a longer list of files and file qualifiers than the VAX/VMS command interpreter can hold in its command input buffers
- To specify information that applies only to LINK and to no other command

5.1.1 Entering Frequently Used Input Specifications

You can create an options file containing a group of file specifications and file qualifiers that you link frequently, and you can specify this options file as input to the linker. The advantages of this method are convenience and flexibility. 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 file specifications of the five modules. Then, to link the image initially or to relink it any time thereafter, you can simply enter $LINK PAYROLL/OPTIONS, instead of having to enter the /EXECUTABLE=PAYROLL qualifier and the file specifications of all the input modules each time. (Note that using the options file in this example produces an image named PAYROLL.) The more file specifications and file qualifiers that you need to link an image, the greater is the convenience of using an option file.

2. Two programmers, one writing PROGX and the other PROGY, both need to include the modules MODA, MODB, and MODC, and to search the library LIBZ. Someone 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 needs to specify only the name of his or her module and the options file-- for example:

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

5.1.2 Identifying A Shareable Image As Input

To identify a shareable image as an input file to the linker, you must use the /SHAREABLE file qualifier within an options file. (If you include /SHAREABLE in the LINK command, the linker assumes that it is a command qualifier, not an input file qualifier.)

The format for /SHAREABLE as an input file qualifier is as follows:

```
/SHAREABLE[=[NO]COPY]
```

- /SHAREABLE identifies the associated input file as a shareable image.
- You can optionally specify COPY or NOCOPY as keywords. COPY causes the linker to produce a private copy of the shareable image in the image being created. NOCOPY, which is the default, causes the linker not to produce a private copy.

5.1.3 Entering More Input Than The Command Language Can Handle

At times you may need to link a series of input files and file qualifiers that exceeds the buffer capacity of the command interpreter (255 characters). The maximum number of entries depends on the length of the specific entries themselves and how much of each line you use. However, as a general guideline, if your 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 file qualifiers in an options file.

5.1.4 Entering Non-standard Link Instructions

The linker is more complex than most VAX/VMS utilities; it can perform a number of optional functions in creating an image. Although the LINK command could have been designed to accept a very large number of command qualifiers, some of these optional functions are not frequently used and apply only to the linker -- for example, specifying the image's base address or the number of I/O channels it can use.

Therefore, to keep the size of the command interpreter's internal tables and code to a manageable level, the /OPTIONS qualifier was developed. /OPTIONS is recognizable to the command interpreter, but the special functions that the options file can specify are recognizable only to the linker. When you specify an options file, then, the command interpreter passes the file to the linker, which reads and interprets its contents.

Table 5-1 lists the special functions that you can request only in an options file, giving the following information for each: its format, the default value (if applicable), and a brief explanation. Section 5.3 provides detailed explanations of each special function.

Table 5-1
Special Options

Format	Default	Explanation
BASE=n	%X200 for executable and shareable %X80000000 for system	Base virtual address for the image
CHANNELS=n	128 (See Section 5.3)	Maximum number of I/O channels the image can use during execution; reserved for future use
CLUSTER=cluster-name,- [base-address],- [pfc],file-spec[,...]	(See explanation in Section 5.3.)	Defines a cluster
COLLECT=cluster-name,- psect-name[,...]		Moves the named program sections to the specified cluster
DZRO_MIN=n	5	Minimum number of uninitialized pages before demand zero compression can occur
GSMATCH=keyword,- major-id,minor-id	EQUAL,x,y (see Section 5.3)	Sets match control parameters of a shareable image
IOSEGMENT=n,- [[NO]P0BUFS]	32, P0BUFS	Number of pages for the image I/O segment
ISD_MAX=n	Approximately 96 (See Section 5.3)	Maximum number of image sections

(continued on next page)

Table 5-1 (Cont.)
Special Options

Format	Default	Explanation
PROTECT= $\left\{\begin{matrix}\text{YES}\\ \text{NO}\end{matrix}\right\}$	NO	Specifies protected clusters when creating a shareable image
PSECT_ATTR=psect-name,-attribute[,...]		Specifies the attributes of a program section
STACK=n	20	Initial number of pages for the user mode stack
SYMBOL=name,value		Defines the named symbol as global and assigns it a value
UNIVERSAL=symbol-name [,...]		Identifies a global symbol as universal

5.2 CREATING AND SPECIFYING AN OPTIONS FILE

To use the /OPTIONS qualifier, you must first create the options file. Use the EDIT command, specifying 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 file qualifiers, or the special link options outlined in Table 5-1, or both types of information. The following rules apply to the contents of a linker options file:

1. You must enter any input file specifications and associated file qualifiers before any special options (see Table 5-1 for the available special options). The input file specifications must be on the first line of the options file or on a continuation of the first line and must be separated by commas.

2. You cannot enter command qualifiers.

3. You cannot enter the /OPTIONS file qualifier.

4. You can enter /SHAREABLE as an input file qualifier only in an options file (see Section 5.1.2).

5. You cannot enter more than one special option on a line.

6. You can continue a file specification line or a special option line.

7. You can enter comments after an exclamation point (!).

8. You can shorten the name of a special option, as long as you enter at least the first four characters (for example, CHAN=50 instead of CHANNELS=50).

The following example shows a file named PROJECT3.OPT that contains both input file specifications and special 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 SET VERIFY command, the contents of the options file are displayed as the file is processed.

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

If you want the LINK command to be in a command procedure, you can specify SYS$INPUT: as an options file. Otherwise the LINK command will be in two files. For example, a command procedure, LINKPROC.COM could contain the following:

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

5.3 SPECIAL OPTIONS

This section lists the available special options in alphabetical order and explains each one. 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) format. The default and maximum numeric values in this manual are expressed in decimal, as are the values in any linker error or warning messages relating to these options.

BASE=n

BASE= specifies the base virtual address of the default cluster. If you do not define any clusters with the CLUSTER= option, the BASE= option value also specifies the base virtual address of the whole image. If you specify an address that is not divisible by 512, the linker automatically adjusts it upward to the next multiple of 512 (that is, the next highest page boundary).

The default base address is hexadecimal 200 (decimal 512) for executable and shareable images, and hexadecimal 80000000 for system images.

CHANNELS=n

CHANNELS= specifies the maximum number of I/O channels that the image can use while it is running. This option is currently ignored by the linker. All images have a maximum of 128 channels.

CLUSTER=cluster-name,[base-address],[pfc],file-spec[,...]

CLUSTER= defines a cluster. Clusters are discussed in Chapters 7 and 8. The CLUSTER= option specifies the following information:

- The name the linker will assign to it
- Optionally, the base virtual address of the cluster
- Optionally, the page fault cluster (pfc) -- that is, the number of pages to be read into memory when a fault occurs for a page in the cluster
- Specifications for the file or files that the linker is to use in creating the cluster. Note that you should not specify in the LINK command itself any files that you specify with the CLUSTER= option (unless you want two copies of each file included in the final image).

If you omit the base address or the page fault cluster, or both, you must still enter the comma after each omitted parameter. For example:

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

The linker uses the following defaults in connection with the CLUSTER= option:

- If you do not use the CLUSTER= option, the linker creates a default cluster, as described in Section 7.9.
- If you use the CLUSTER= option but do not specify a base address, the linker allocates the cluster according to the procedure described in Section 7.9.
- If you use the CLUSTER= option but do not specify a page fault cluster, VAX/VMS memory management determines the value.

COLLECT=cluster-name,psect-name[,...]

COLLECT= causes the named program sections to be placed in the specified cluster. If the cluster name is also used in a CLUSTER= option, the named program sections are added to that defined cluster. If the cluster name is not also used in a CLUSTER= option, the COLLECT= option defines the cluster.

The COLLECT= option can only specify program sections defined with the GBL attribute.

The COLLECT= option cannot specify any program sections that are defined in shareable images.

DZRO_MIN=n

DZRO_MIN= is an option that gives you some control over the linker's compression of uninitialized pages in an executable image. Before the linker writes the binary data and code of the image, it attempts to compress certain uninitialized areas by converting them to demand zero image sections. ("Demand zero" means that although an area does not occupy physical space in the image on disk, when the area is accessed during execution, a portion of memory is allocated for it and initially filled with binary zeroes.) An uninitialized area is eligible for this compression if it can be written in by the user and if its size is equal to or greater than a threshold value: that is, the DZRO_MIN= value. The linker will not, however, continue creating demand zero sections after the total number of image sections reaches the maximum (see the ISD_MAX= option in this section).

The default value for DZRO_MIN= is 5; that is, an uninitialized, writeable area is not eligible for compression unless it occupies five or more contiguous pages. A DZRO_MIN= value less than 5 might (depending on the contents of the object modules) cause the linker to compress more sections and create a greater number of image sections, possibly reducing the image size on disk but decreasing its paging performance. A value greater than 5 might cause the linker to compress fewer sections and create a smaller number of image sections, possibly increasing the image size on disk but providing better performance during execution.

GSMATCH=keyword,major-id,minor-id

GSMATCH= sets the match control parameters for a shareable image that you are now creating. After the shareable image has been linked with an executable image, and when the executable image is being run, these parameters guide the VAX/VMS image activator in choosing global sections. For further information on this process, see Section 8.3.2.

The GSMATCH= option specifies the following information:

- A keyword expressing the match relationship between the minor identifications in the user shareable image section and in the installed global section. This keyword is one of the following:
  - EQUAL The minor identification of the user shareable image section must be identical to that of the installed shareable image section.

- LEQUAL The minor identification of the user shareable image section must be less than or equal to that of the installed shareable image section. LEQUAL permits the creator of a shareable image to update it (increasing the minor identification) and install it, and yet avoid the need for programs using that shareable image to be relinked. (The minor identification of that shareable image section in programs that are linked to it will be less than the minor identification of the updated installed shareable image section.)

- NEVER The linker is to assume that global sections will never match (perhaps because the shareable image will never be installed). Therefore, the linker will always create a private copy of this shareable image in any image that links to it. (This keyword overrides any stated or defaulted NOCOPY keyword in the /SHAREABLE file qualifier in any subsequent link operation that names this shareable image as an input file.)

- ALWAYS This keyword causes the image activator to match image sections by name only and to ignore the major and minor identifications. (However, the syntax of this option requires that you still enter major and minor identifications.)

● The major identification of the user shareable image section, expressed as a number from 0 through 255.

● The minor identification of the user shareable image section, expressed as a number from 0 through 2**24-1.

The linker uses the following defaults for the GSMATCH= option:

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

where "x" and "y" together are the middle 32 bits of the 2-longword creation time that is stored in the shareable image file header. This default value forces user images that are linked to this (installed) shareable image to be relinked each time the shareable image is updated. To ensure that other users will not have to relink images that are linked to your shareable image whenever you modify it, specify the following GSMATCH= value:

```
GSMATCH=LEQUAL,0,0
```

IOSEGMENT=n[,[NO]P0BUFS]

IOSEGMENT= specifies the number of pages for the image I/O segment, which holds the buffers and VAX-11 RMS control information for all files that the image's process uses. If the process needs more space than the IOSEGMENT value during execution, VAX-11 RMS adds space for it at the end of the program (P0) region.

You can also specify P0BUFS or NOP0BUFS as parameters. P0BUFS, which is the default, permits RMS to use the program region (P0) for any additional buffers that it needs. NOP0BUFS denies RMS the option of using P0 space for additional buffers.

The default value for IOSEGMENT= is 32,P0BUFS. The only reason to specify a number of pages greater than the default is to guarantee that the program region will be contiguous if you need to extend it and if the total size of your program's buffers and VAX-11 RMS control information exceeds 32 pages. In this case, you also need to specify NOP0BUFS.

ISD_MAX=n

ISD_MAX= is an option that gives you some control over the linker's compression of uninitialized pages in an executable image. (For an explanation of compression, see the DZRO_MIN= option in this section.) The ISD_MAX= value specifies the maximum number of image sections allowed in the image. If the linker is compressing the image by creating demand zero sections and the total number of image sections reaches the ISD_MAX= value, the compression ceases at that point.

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.

PROTECT= $\begin{Bmatrix} \text{YES} \\ \text{NO} \end{Bmatrix}$

PROTECT= controls whether clusters are protected. PROTECT=YES specifies that all clusters defined (in a CLUSTER or COLLECT option) between it and the next PROTECT= option are protected. PROTECT=NO (default condition) specifies that all clusters defined between it and the next PROTECT= option are not protected. Protected clusters are used in privileged shareable images to execute privileged instructions. See the VAX/VMS Real-Time User's Guide for more information on privileged shareable images.

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

PSECT_ATTR= specifies one or more attributes of the named program section. This option is used to change the compiled or assembled attributes of a program section.

You must use the abbreviation given in Section 6.2.3 for each attribute. If you do not list a complete set of attributes, this option does not change any attributes that are not listed. For example, the option

```
PSECT_ATTR=ALPHA,NOWRT
```

can be used to make the program section ALPHA not writeable instead of writeable; however, all other attributes of ALPHA remain the same.

STACK=n

STACK= specifies the initial number of pages to be allocated for the image's user mode stack area. If when the program is executed it requires more stack space than was allocated, the stack is automatically expanded.

The default value is 20.

SYMBOL=name,value

SYMBOL= defines "name" as an absolute global symbol with the specified value. The value must be expressed as a number.

Because the linker processes special options before input file specifications, the name and value specified by the SYMBOL= option constitute the first definition of that symbol. If an input object module also defines that symbol, one of the following occurs:

- If the object module defines the symbol as relocatable, that definition is ignored and the definition by the SYMBOL= option is used. A warning message is issued.

- If the object module defines the symbol as absolute, that definition is ignored and the definition by the SYMBOL= option is used. No warning message is issued.

UNIVERSAL=symbol-name[,...]

UNIVERSAL= identifies one or more global symbols of a shareable image as universal symbols. For a discussion of universal symbols, see Section 2.2.3.

CHAPTER 6

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.

To obtain a map, you must include the /MAP qualifier in the LINK command. You can specify a file name with the MAP qualifier, or you can let the linker assign a default. You can further specify the type of map with the /BRIEF or /FULL qualifier. If you enter either /MAP alone or /MAP with /FULL, you can also include a symbol cross reference in the map by specifying /CROSS_REFERENCE. However, if you enter /MAP and no other map-related qualifiers, the linker generates its default map.

The map is placed on your output disk and assigned a default file type of MAP. You can print a copy of the map with the PRINT command.

The following examples show the LINK command qualifiers necessary to produce different types of maps:

Command Qualifiers	**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

6.1 IMAGE MAP CONTENTS

A listing of the image map contains several sections; however, the number of sections and the contents of certain sections depend on the qualifiers that you enter.

Table 6-1 lists all the possible section names in the order in which they appear, the types of map in which each appears, and a brief explanation of each section. A section shown as appearing in "all" is included in all types of image maps; "default" and "full" identify sections appearing in default and full maps, respectively. A brief map thus contains only the map sections designated as "all." For detailed explanations and illustrations of map sections, see Section 6.2.

Table 6-1
Image Map Sections

Section Name	Appears In	Explanation
Object Module Synopsis	All	Object modules in the image
Image Section Synopsis	Full	Image sections and clusters
Program Section Synopsis	Default Full	Program sections and the modular contributions
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

The contents of the following sections vary depending on whether the map type is default or full:

- Object Module Synopsis
- Program Section Synopsis
- Symbols by Name
- Symbol Cross Reference

The difference between these sections in a default map and in a full map is in the number of items:

- A default map generally includes only information that applies to modules and shareable images that you name as input to the linker or that are extracted from libraries you name. A default map normally does not list information that applies only to modules taken from the default system library.
- A full map includes information that applies to all modules and shareable images, including those extracted from the default system library.

6.2 IMAGE MAP SECTIONS

The rest of this chapter explains and illustrates each available image map section. The sections are presented in the order in which they appear in a full map. Brief and default maps do not have all of these sections, but the sections that they do have are in the order presented here.

The illustrations reflect an image created from a simple FORTRAN program (similar to the example developed in the VAX/VMS Primer). Each illustration is from a full map. Headings and items in each illustration are explained only if they are not self-explanatory.

Appendix B contains examples of the brief, default, and full forms of the image map.

6.2.1 Object Module Synopsis

The Object Module Synopsis lists object modules in the order in which the linker processed them. This section appears in all types of maps.

The Object Module Synopsis provides the following information about each module listed:

- Module name
- Module identification as it appears in the module header
- Module length in bytes
- Complete file specification for the module
- Module creation date
- Language translator that created the module

The Object Module Synopsis also lists any errors that the linker detected when it wrote the binary data and code--for example, a warning message that a module refers to an undefined symbol. The message appears immediately below the line that indicates the module that the linker was processing when the error occurred.

Figure 6-1 illustrates the Object Module Synopsis section.

6.2.2 Image Section Synopsis

The Image Section Synopsis lists information about the image sections in the order in which they are mapped in the image. The Image Section Synopsis appears only in a full map.

The Image Section Synopsis lists the following information about each image section:

- Cluster in which the sections were allocated or found
- Code used internally by the linker
- Number of pages
- Base virtual address within the image

```
AVERAGE                                                   22-FEB-1980 14:14       LINKER V02.40                 Page     1
                                        +------------------------+
                                        ! Object Module Synopsis !
                                        +------------------------+

Module Name     Ident             Bytes      File                             Creation Date       Creator
-----------     -----             -----      -----                            -------------       -------
AVERAGE         01                  222 _DBB2:[GOLDMAN]AVERAGE.OBJ;1       22-Feb-1980 14:05  VAX-11 FORTRAN T1.95-36
OTS$LINKAGE     1-003                 3 _DRA5:[SYSLIB]STARLET.OLB;1        19-FEB-1980 21:57  VAX-11 Macro V02.41
SYSVECTOR       0219                  0 _DRA5:[SYSLIB]STARLET.OLB;1        19-FEB-1980 21:59  VAX-11 Macro V02.41
VMSRTL          .EXE;1                0 _DRA5:[SYSLIB]VMSRTL.EXE;1         20-FEB-1980 18:55  LINK-32 V02.39
```

Figure 6-1 Object Module Synopsis Section

```
_DBB2:[GOLDMAN]AVERAGE.EXE;1                              22-FEB-1980 14:14       LINKER V02.40                 Page     2
                                        +------------------------+
                                        ! Image Section Synopsis !
                                        +------------------------+

  Cluster       Type Pages   Base Addr Disk VBN PFC Protection and Paging      Global Sec. Name   Match      Majorid   Minorid
  -------       ---- -----   --------- -------- --- --------------------      ----------------   -----      -------   -------

DEFAULT_CLUSTER    0     1    00000200        2   0 READ ONLY
                   0     1    00000400        3   0 READ WRITE   COPY ON REF
                   0     1    00000600        4   0 READ ONLY
                 253    20    7FFFD800        0   0 READ WRITE DEMAND ZERO

VMSRTL             3    11    00000800        0   1 READ ONLY                  VMSRTL_001       LESS/EQUAL        1      2000
                   3   193    00001E00        0   0 READ ONLY                  VMSRTL_002       LESS/EQUAL        1      2000
                   4     4    0001A000        0   0 READ WRITE   COPY ON REF   VMSRTL_003       LESS/EQUAL        1      2000
```

Figure 6-2 Image Section Synopsis Section

- Base virtual block number within the image file on disk (zero indicates that the image section is not in the image file)
- Page Fault Cluster (PFC) (zero indicates that VAX/VMS memory management determines the value)
- Protection characteristic ("read-only" or "read/write") and paging information ("copy on reference," "copy always" (shareable images only), "demand zero," or blank for standard handling)
- Global section name if the cluster is a shareable image
- Match control of global sections
- Major and minor identification of global sections

Figure 6-2 illustrates the Image Section Synopsis section.

6.2.3 Program Section Synopsis

The Program Section Synopsis lists information about program sections (PSECTs), including relative addresses within the image and PSECT attributes. This section appears in default and full maps.

The address information enables you to translate an address from a program module listing into a virtual address in the image, and vice versa. This ability can help you isolate errors or problems in the image at run time--for example, by allowing you to relate an address in an error message to a specific location within a specific module.

The attributes of each program section are also listed. The linker considers certain attributes when it groups PSECTs into image sections (ISECTs). For further information on this process, see Section 7.7.

The Program Synopsis, illustrated in Figure 6-3, lists the following information about each program section:

- Program section name, in order of increasing base virtual addresses
- Name of the module or modules that contribute binary data or code to the program section
- Base and ending virtual addresses, in hexadecimal, of each module's contribution to the PSECT
- Alignment for the start of each module that contributes to the PSECT. The number that follows the alignment description is the power of 2 that expresses the length in bytes. (For example, 2 to the power of 2 equals 4, the number of bytes in a longword.) The alignment column can contain these entries:

BYTE	0	- Byte alignment (1 byte)
WORD	1	- Word alignment (2 bytes)
LONG	2	- Longword alignment (4 bytes)
QUAD	3	- Quadword alignment (8 bytes)
PAGE	9	- Page alignment (512 bytes)

- Attributes of the PSECT. Most attributes are parts of contrasting pairs. Table 6-2 lists the attribute abbreviations (in alphabetical order), their meanings, and any contrasting attributes. Section 7.5.4 explains the attributes.

Table 6-2
PSECT Attributes

Abbreviation	Meaning	Contrasts With
ABS	Absolute	REL
CON	Concatenated	OVR
EXE	Executable	NOEXE
GBL	Global	LCL
LCL	Local	GBL
LIB	Library (from shareable image)	USR
NOEXE	Not executable	EXE
NOPIC	Not position-independent code	PIC
NORD	Not readable	RD
NOSHR	Not shareable	SHR
NOVEC	Not vectors for privileged shareable image	VEC
NOWRT	Not writeable	WRT
OVR	Overlaid	CON
PIC	Position-independent code	NOPIC
RD	Readable	NORD
REL	Relocatable	ABS
SHR	Shareable	NOSHR
USR	User	LIB
VEC	Vectors for privileged shareable image	NOVEC
WRT	Writeable	NOWRT

```
_DBB2:[GOLDMAN]AVERAGE.EXE;1                         22-FEB-1980 14:14       LINKER V02.40                  Page    3
                                       +--------------------------+
                                       ! Program Section Synopsis !
                                       +--------------------------+

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

$PDATA                          00000200 00000233 00000034 (         52.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,NOEXE,  RD,NOWRT,NOVEC
                AVERAGE         00000200 00000233 00000034 (         52.) LONG 2

$LOCAL                          00000400 0000040F 00000010 (         16.) LONG 2   PIC,USR,CON,REL,LCL,NOSHR,NOEXE,  RD,  WRT,NOVEC
                AVERAGE         00000400 0000040F 00000010 (         16.) LONG 2

$CODE                           00000600 00000699 0000009A (        154.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT,NOVEC
                AVERAGE         00000600 00000699 0000009A (        154.) LONG 2

_OTS$CODE                       0000069C 0000069E 00000003 (          3.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT,NOVEC
                OTS$LINKAGE     0000069C 0000069E 00000003 (          3.) LONG 2

. BLANK .                       00000800 00000800 00000000 (          0.) BYTE 0 NOPIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,  WRT,NOVEC
                OTS$LINKAGE     00000800 00000800 00000000 (          0.) BYTE 0
                SYSVECTOR       00000800 00000800 00000000 (          0.) BYTE 0
```

Figure 6-3 Program Section Synopsis Section

```
_DBB2:[GOLDMAN]AVERAGE.EXE;1                        22-FEB-1980 14:14       LINKER V02.40                 Page    4
                                    +-----------------+
                                    ! Symbols By Name !
                                    +-----------------+

Symbol          Value              Symbol           Value              Symbol           Value              Symbol           Value
------          -----              ------           -----              ------           -----              ------           -----
AVERAGE         00000600-R         BAS$INPUT_LINE   000011B0-RU        BAS$SCALE_D_R1   00001078-RU        FOR$$CB_POP      00000E10-RU
   .               .                  .                .                  .                .                  .                .
   .               .                  .                .                  .                .                  .                .
   .               .                  .                .                  .                .                  .                .
FOR$IO_X_DA     00000970-RU        LIB$PUT_OUTPUT   00000D58-RU        MTH$DEXP         00000AF8-RU        OTS$$FREEN_DD    00000C28-RU
FOR$LINKAGE     0000069C-R         LIB$REVERT       00000D60-RU        MTH$DEXP_R6      00000800-RU        OTS$$FREEN_DD6   00000C30-RU
FOR$OPEN        00000978-RU        LIB$SCANC        00000D68-RU        MTH$DEXP_R7      00000800-RU        OTS$$GET1_DD     00000C08-RU
   .               .                  .                .                  .                .                  .                .
   .               .                  .                .                  .                .                  .                .
   .               .                  .                .                  .                .                  .                .
```

Figure 6-4 Symbols by Name Section

6.2.4 Symbols By Name

The Symbols by Name section lists global symbols in alphabetical order and gives the hexadecimal value of each one. The value may have one of the following suffixes: -R for a relocatable symbol, -U for a universal symbol, -RU for a relocatable universal symbol, -W for a weak definition, or -* for an undefined symbol. (The linker assigns a value of zero to undefined global symbols.)

The Symbols by Name section appears only in a default or full map that does not have a cross reference. If you include /CROSS_REFERENCE in the LINK command, the Symbols by Name section is replaced by the Symbol Cross Reference section.

Figure 6-4 illustrates the Symbols by Name section.

6.2.5 Symbol Cross Reference

The Symbol Cross Reference section lists global symbols in alphabetical order and gives the following information about each one:

- Hexadecimal value. The value can have one of the following suffixes: -R for relocatable, -W for a weak definition, -* for undefined, -U for universal, or RU for relocatable universal.
- Name of the first module that defines the symbol (blank if the symbol is undefined).
- Name of each module that refers to the symbol. The name has the prefix WK- if the module makes a weak reference to the symbol.

The Symbol Cross Reference section appears only in a default or full map for which you specify /CROSS_REFERENCE. It replaces the Symbols by Name section.

A primary value of the Symbol Cross Reference section is that it shows which modules are affected by each symbol. For example, if you want to change a symbol definition, the Symbol Cross Reference section tells you where it is defined and what other modules may be affected by the change.

Figure 6-5 illustrates the Symbol Cross Reference section.

6.2.6 Symbols By Value

The Symbols by Value section lists the hexadecimal values of global symbols in ascending numeric sequence, with the symbol or symbols that correspond to each value. The symbol name can have one of the following prefixes: R- for relocatable, U- for universal, or RU- for relocatable universal.

This section appears only in a full image map.

Figure 6-6 illustrates the Symbols by Value section.

```
_DBB2:[GOLDMAN]AVERAGE.EXE;2                                    22-FEB-1980 14:14      LINKER V02.40                 Page    4
                                                 +-------------------------+
                                                 ! Symbol Cross Reference !
                                                 +-------------------------+

Symbol           Value            Defined By       Referenced By ...
------           -----            ----------       -----------------
AVERAGE          00000600-R        AVERAGE
         .
         .
         .
FOR$IO_END       000008A8-RU       VMSRTL             AVERAGE
FOR$IO_FC_R      00000940-RU       VMSRTL
FOR$IO_FC_V      00000948-RU       VMSRTL
FOR$IO_F_R       000008B0-RU       VMSRTL             AVERAGE
FOR$IO_F_V       000008B8-RU       VMSRTL
         .
         .
         .
```

Figure 6-5 Symbol Cross Reference Section

```
_DBB2:[GOLDMAN]AVERAGE.EXE;1                                    22-FEB-1980 14:14      LINKER V02.40                 Page    7
                                                    +-------------------+
                                                    ! Symbols By Value !
                                                    +-------------------+

Value                                      Symbols...
-----                                      ----------
00000600     R-AVERAGE
0000069C     R-BAS$LINKAGE       R-FOR$LINKAGE       R-OTS$LINKAGE
00000800    RU-FOR$CLOSE
00000808    RU-FOR$DECODE_MF
00000810    RU-FOR$DECODE_MO
00000818    RU-FOR$ENCODE_MF
   .              .
   .              .
   .              .


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

Figure 6-6 Symbols by Value Section

6.2.7 **Image Synopsis**

The Image Synopsis, which appears in all maps, gives miscellaneous information about the output image. The virtual memory allocation lists (in hexadecimal radix) the image's starting address, ending address, and total size in bytes and (in decimal radix) the total size in bytes and pages. The other items are self-explanatory. Numbers are decimal if they are followed by a point (.); otherwise, they are hexadecimal.

Figure 6-7 illustrates the Image Synopsis section.

6.2.8 **Link Run Statistics**

The Link Run Statistics section, which appears in all maps, gives statistics of the link run that produced the image. The items are self-explanatory.

Figure 6-8 illustrates the Link Run Statistics section.

```
_DBB2:[GOLDMAN]AVERAGE.EXE;1                                22-FEB-1980 14:14       LINKER V02.40                    Page   15
                                                         +------------------+
                                                         ! Image Synopsis !
                                                         +------------------+

Virtual memory allocated:                    00000200 0001A7FF 0001A600 (108032. bytes, 211. 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:               AVERAGE 01
Number of files:                                    3.
Number of modules:                                  4.
Number of program sections:                         9.
Number of global symbols:                         271.
Number of image sections:                           8.
User transfer address:                       00000600
Debugger transfer address:                   80000168
Image type:                                  EXECUTABLE.
Map format:                                  FULL in file "_DBB2:[GOLDMAN]AVERAGE.MAP;1"
Estimated map length:                        58. blocks
```

Figure 6-7 Image Synopsis Section

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

Performance Indicators                                   Page Faults    CPU Time        Elapsed Time
----------------------                                   -----------    --------        ------------
    Command processing:                                          15    00:00:00.05     00:00:00.27
    Pass 1:                                                     399    00:00:01.17     00:00:02.73
    Allocation/Relocation:                                       83    00:00:00.26     00:00:00.40
    Pass 2:                                                      64    00:00:00.25     00:00:01.43
    Map data after object module synopsis:                      150    00:00:01.89     00:00:03.45
    Symbol table output:                                         12    00:00:00.00     00:00:00.18
Total run values:                                               723    00:00:03.62     00:00:08.46

Using a working set limited to 273 pages and 48 pages of data storage (excluding image)

Total number object records read (both passes):   175
    of which 67 were in libraries and 4 were DEBUG data records containing 158 bytes
143 bytes of DEBUG data were written,starting at VBN 5 with 1 blocks allocated

Number of modules extracted explicitly             = 0
    with 2 extracted to resolve undefined symbols

0 library searches were for symbols not in the library searched

A total of 0 global symbol table records was written

/MAP/FULL AVERAGE
```

Figure 6-8 Link Run Statistics Section

CHAPTER 7

IMAGE CREATION

This chapter discusses the allocation of virtual memory and the different kinds of images that the linker can produce. The concepts of program sections, image sections, and clusters are introduced, along with a description of the way in which the linker builds the final image.

7.1 PROGRAM SECTIONS

Program sections are areas of memory that have a name, a length, and a series of attributes (detailed in Section 7.5.4) that describe the intended or permitted usage of that portion of memory. The program section is the vehicle by which a language compiler describes the memory requirements of a particular object module.

7.2 IMAGE SECTIONS

Image sections are named collections of pages; each page in an image section has the same hardware protection characteristics and the same sharing nature. The image sections describe the memory requirements of the whole image to the VAX/VMS memory management software.

The linker creates image sections by collecting program sections that have similar (but not necessarily identical) attributes. The manner in which program sections are grouped into image sections depends upon both the attributes of each program section and the type of image being produced (see Section 7.7).

7.3 CLUSTERS

Clusters are collections of image sections. Clustering provides a way for the designer of an application program to ensure that an image section is near, in virtual memory, to the image sections that it references and that reference it. Having related image sections near each other improves the performance of large application programs.

An example of an application program that could use clustering is a compiler. A compiler usually goes through a number of distinct phases during a single compilation run. Each phase of the compiler could be made into a separate cluster to improve the compiler's performance.

Every image consists of at least one cluster. You can specify additional clusters in two ways: by object modules or by program sections. The CLUSTER= option (see Section 5.3) causes image sections created from the specified object modules to be in the same cluster. The COLLECT= option (see Section 5.3) causes image sections created from the specified program sections to be in the same cluster. In both cases, the image sections are put in the cluster in the order described in Section 7.7.

Note that clusters are relevant only to the linker itself; they do not appear as a structure to anything else (such as to the VAX-11 memory management software). See Section 7.9 for more information.

7.4 OBJECT MODULE CONTENTS

Each object module contains several types of records. All object modules have header records and an end-of-module record. Some also have other kinds of records, depending on the options specified at compilation. All object modules also contain the following records for each of the program sections:

- A global symbol record that includes the program section's attributes. (A global symbol record is also used to describe each global symbol defined in the module.)
- A text information and relocation record, containing the section's binary data or code and certain commands to the linker.

Appendix C contains a detailed specification of the object language accepted by the linker.

7.5 PROGRAM SECTIONS

A program section is defined to the linker by the following:

- A name
- A size
- An alignment
- A series of single-bit attributes expressing whether the program section is:
  - Relocatable or absolute
  - Concatenated or overlaid
  - Local to a cluster or global across all clusters
  - Executable or not
  - Writeable or not
  - Readable or not
  - Position-independent or not
  - Potentially shareable or not
  - Created by a user program or by the linker for internal use
  - Has protected vectors or not

7.5.1 Program Section Name

The program section name is an ASCII character string, 1 through 31 characters in length. You can use any printable ASCII character in the name, but are cautioned against using the dollar sign ($), to avoid 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.

7.5.2 Program Section Size

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

7.5.3 Program Section Alignment

The alignment field describes the address boundary at which the 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 rounded 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 diagnostic error. 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 which is a multiple of the highest power of 2 specified by any of the contributing modules.

7.5.4 Program Section Attributes

The following subsections explain the attributes that a program section can have. Section 7.7 describes how the linker considers certain significant attributes as it constructs different types of images. Section 5.3 describes the PSECT_ATTR option, which allows you to change the attributes of a program section.

7.5.4.1 **Relocatability (REL and ABS)** - A program section can be relocatable or absolute. 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.

Absolute program sections, on the other hand, are not considered in the allocation of virtual memory. They contain no binary data or code, and all appear as if they were based at a virtual address of zero. Absolute program sections are used primarily to define global symbols.

7.5.4.2 **Concatenated versus Overlaid (CON and OVR)** - This attribute determines the relationship between the memory allocations when several modules contribute program sections with the same name.

A concatenated program section contribution requires separate address space in the image. If two program sections from different modules have the same name, the sections will be placed in separate but contiguous address spaces. For example, if PSECTA in MODULE1 and PSECTA in MODULE2 have the concatenated attribute, the allocation of PSECTA from MODULE1 will be followed by the allocation of PSECTA from MODULE2. The total size of a concatenated program section is the sum of the individual contributions and any padding allowed for the individual alignments.

An overlaid program section contribution, however, can share an address space with other program sections that have the same name. For example, if PSECTA in MODULE1 and PSECTA in MODULE2 each have the overlaid attribute, both program section contributions will be allocated starting at the same base address in the image. The total size of an overlaid program section is that of the largest contribution.

Note that any module can initialize the contents of an overlaid program section. Because of this, the order in which you specify the input modules is important: the contents of an overlaid program section are determined by the last contributing module specified.

BASIC and FORTRAN common areas are the most frequently used overlaid program sections.

7.5.4.3 **Scope - Local versus Global (LCL and 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 of a global program section is allocated in the cluster that contains the first contributing module.

BASIC and FORTRAN common areas are implemented with global program sections.

7.5.4.4 **Executability (EXE and NOEXE)** - Although the current VAX-11 hardware does not implement any kind of execute protection, this attribute is reserved for possible future implementation. This attribute also permits possible future extension of link time error detection and of software security protection.

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

- Error-checking on an image start 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.

7.5.4.5 **Writeability (WRT and NOWRT)** - This attribute determines whether the program section contents will be protected against modification when the image is executed. If the program attempts to modify the contents of a non-writeable program section during execution, an access violation occurs.

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.

7.5.4.6 **Readability (RD and NORD)** - The current version of the linker ignores this attribute. It is provided merely to allow the possible future implementation of a data security system.

7.5.4.7 **Position Independence (PIC and NOPIC)** - This attribute identifies whether the content of a program section depends on where that program section or something that it refers to is allocated in the virtual address space. For example, the following types of program sections are position independent:

- A program section that contains no virtual addresses
- A program section whose references to virtual memory are in the form of a displacement from itself, if the targets of the references must always be at the same displacement from the calls which refer to them

This attribute applies only to shareable images, which are discussed in Chapter 8.

7.5.4.8 **Shareability (SHR and NOSHR)** - As its name suggests, this attribute is significant only for shareable image memory allocation and memory management (see Chapter 8).

7.5.4.9 **User versus Library (USR and LIB)** - This attribute is reserved for possible future enhancements to the linker. It is ignored for the current release but should be set to USR to guarantee future compatibility.

7.5.4.10 **Protection (VEC and NOVEC)** - The VEC attribute specifies that the program section contains privileged change mode vectors. Program sections with the VEC attribute are automatically protected in shareable images. See the description of privileged shareable images in the VAX/VMS Real-Time User's Guide for more information.

7.6 TYPES OF IMAGES

The linker creates three types of images: executable, shareable, and system. Each type has specific uses. System images differ substantially in content and organization from executable images and shareable images. The following subsections define each type.

7.6.1 **Executable Images**

An executable image is a program that you can activate by the RUN command. The most common use of the linker is to create executable images.

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

7.6.2 **Shareable Images**

There are two major reasons for shareable images:

- To provide a means of sharing a single physical copy of a set of procedures and/or data between multiple application programs
- To facilitate the linking of very large applications (say, hundreds of modules) in manageable segments

As with executable images, when the link of a shareable image is complete, all symbolic references are resolved and memory is allocated to a group of image sections. A description of each image section is written to the image header. Unlike an executable image, however, a shareable image normally has a symbol table appended to it.

A shareable image is not directly runnable. It is intended for reprocessing by the linker--that is, to be included in a subsequent image. In processing a shareable image, the linker reads the image header and generates a separate image cluster from the set of image sections it finds.

After generating the cluster that is the incoming shareable image, the linker processes the symbol table appended to the image just as if it were an object module. This allows the shareable image to resolve symbols (usually routine names) referred to by the modules with which it is being linked. These symbols are called universal symbols (see Section 2.2.3).

When you run a program that has been linked with a shareable image, the VAX-11 image activator checks to see if the shareable image has been installed by the system manager. If it has been installed, the image activator sets a pointer that enables the process to use the shareable image. Thus, whenever multiple processes request an installed shareable image, the operating system makes the same physical copy of the shareable image available to each requesting process. Shareable images can therefore conserve physical memory at run time. If the shareable image has not been installed, the image activator creates a private copy of the shareable image.

Chapter 8 discusses shareable images further. At this point, however, note the following information and conventions pertaining to shareable images:

- The default common Run-Time Procedure Library provided with the VAX/VMS system is a shareable image.
- You cannot link the VAX-11 Symbolic Debugger with a shareable image.

- You can request that the linker produce a private copy of a shareable image in an executable image file. By default, however, the linker does not do so, thereby saving disk space.

- Chapters 4 and 5 describe LINK command qualifiers and link time options specifically intended for dealing with shareable images. See the following:

/SYSSHR /SHAREABLE	qualifiers
UNIVERSAL= GSMATCH=	options

7.6.3 **System Images**

A system image is intended for stand-alone operation on the VAX-11 hardware; that is, it does not run under the control of the VAX/VMS operating system.

The allocation of memory to a system image is much simpler than for the other two types of images. The linker allocates memory to the program sections based on the alphabetical order of the program section names. The only other factors that the linker considers are program section size, alignment, and the following attributes: concatenated or overlaid, and relocatable or absolute. These factors are treated as described in Section 7.5.

The resulting image is a fixed-length record file, each record being a 512-byte block. A system image has no image header (unless the /HEADER qualifier is specified), no debug data, and no symbol tables. It has no set format. That is to say, it contains binary data and code just as they would appear in memory.

7.7 GENERATION OF IMAGE SECTIONS

The linker makes two passes over the input object modules. The first pass builds the symbol table and the program section tables. The second pass writes the binary contents of the image. Memory allocation is performed between the two passes; the linker uses the program section table of each cluster and generates an image section table for each cluster.

When the first pass is complete, the linker has determined the sizes of all the relocatable program sections by considering specific attributes and the alignment, as discussed in Section 7.5. The linker has also determined relative addresses of each module's contribution to a particular program section. What remains to be done is to group the program sections into image sections, and to position the whole image cluster in the virtual address space.

Depending on the type of image being produced, the linker establishes a mask for the program section attributes that it will consider:

- For an executable image, this mask includes only the writeability (WRT and NOWRT), executability (EXE and NOEXE), and protected vector (VEC and NOVEC) attributes.

- For a shareable image, this mask includes the writeability, executability, position independence (PIC and NOPIC), shareability (SHR and NOSHR), and protected vector (VEC and NOVEC) attributes.

For each possible combination of the significant attributes, the linker searches the program section list of a cluster. If the linker finds any program section with this combination of attributes, it generates an image section. Each program section with matching attributes in the image section is assigned an address relative to the base of the image section, in alphabetical order by program section name.

All combinations of significant attributes are handled in this way, until the complete set of image sections for the particular cluster is generated. Table 7-1 lists the order that the linker stores the image sections within a cluster. Each line of the table specifies a possible combination of significant attributes. The next cluster (if there is one) is then treated in the same way.

At this point in image creation, all image sections have cluster-relative base addresses, and all program sections have image section-relative addresses. The next step consists of allocating virtual address space to the cluster and then relocating all image sections and program sections within the cluster.

The choice of address space for the cluster depends on whether you specified an address in the CLUSTER= option, and whether the cluster contains a shareable image. It also depends on the order in which you specified the clusters.

7.8 COMPRESSION OF UNINITIALIZED IMAGE SECTIONS

At the end of its first pass across the object modules, the linker sorts all the program sections into a group of distinct image sections. The sorting is determined by program section attributes and results in the complete allocation of the user virtual space.

In its second pass, the linker writes the binary contents of the image. During this image 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 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.

After completing the second pass across the object modules, the linker scans the list of image sections in an attempt to compress uninitialized pages from the image, which is about to be written. The linker attempts to perform this compression by creating demand zero image sections. The linker scans the image sections and attempts to compress uninitialized pages when it is creating executable images only.

Table 7-1
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)				

If the linker finds an image section that does not have a buffer allocated, it considers splitting the section into multiple image sections, some demand zero and others copy on reference. To be eligible for splitting, the image section must be writeable to the user and larger than the minimum compression threshold size (see the DZRO_MIN= option in Chapter 5). If the image section can be split, the linker calls a memory management system service, passing it a description of the image section buffer and the compression threshold value. By calling this service in a loop, the linker finds out which segments of the buffer are both larger than the threshold number of pages and previously unmodified by the linker. This process results in the replacement of a single image section by a potentially large number of alternating demand zero and copy on reference image sections.

The linker continues the splitting process, scanning the list of image sections until it reaches the end or until the total number of image sections reaches the limit specified or defaulted for the ISD_MAX= option (see Chapter 5). During the entire process, the linker keeps track of the size of the image header (where descriptors of the image sections will be written) and of the image binary contents. Thus, at the end of the scan the linker knows the precise size of the image header and the contents, and it can then create the image file.

When the image file is successfully created, the linker makes another scan of the image section descriptor list. During this scan it writes the contents of all existing image section buffers to the image file, assigning them virtual block numbers as it does so. Finally, the linker writes the image header, starting at virtual block number 1 of the image file.

By default, the linker creates the image with the attribute "contiguous best try," which becomes a permanent attribute of the image file. However, you can specify the /CONTIGUOUS qualifier to force the image file to be created contiguously (see Chapter 4).

7.9 MECHANICS OF CLUSTERING

Section 5.3 describes the CLUSTER= and COLLECT= options, which are used to define the position, character, and content of clusters. The cluster name is merely for convenience in reading the Image Section Synopsis of the image map.

Every image produced by the linker is automatically given a default cluster. This cluster contains any object modules not explicitly positioned in other clusters. The BASE= option serves to position the default cluster in the address space.

A shareable image is treated as a cluster. If the image is not position independent (NOPIC), it has a base address already assigned and is treated in the same manner as a user-specified cluster that has a base address.

The linker allocates virtual address space for clusters in the following order:

- Clusters that have fixed bases (including position-dependent shareable images). If any of these clusters overlap, the linker displays an error message.
- User-specified clusters without fixed bases. These are allocated in the order specified.
- Default cluster (if it contains any modules and does not have a fixed base)
- Position-independent shareable images
- Run-time library shareable image (if it is included in the image)

Clustering is not likely to have any performance advantage for applications smaller than 200K bytes. The reason is that each cluster contains a group of image sections, and thus the address space is more fragmented than it would be without clustering. Fragmentation can reduce program performance under certain circumstances.

CHAPTER 8

SHAREABLE IMAGES

This chapter describes in detail the nature and use of shareable images. The material in this chapter is more complex than much of the earlier material. Therefore, you are presumed to be familiar with the earlier chapters of this manual, particularly Chapter 7.

8.1 SHAREABLE IMAGES: BENEFITS AND USES

The following subsections expand on the discussion in Section 7.6 of the benefits you can obtain from the use of shareable images. These subsections also discuss the conceptual nature of shareable images.

8.1.1 Conserving Physical Memory

Main physical memory is one of the prime resources that any operating system has to control. The installation of shareable images produces a set of global sections of memory -- one for each image section built in the shareable image. These global sections are the mechanism by which sharing is realized, for they can be mapped into the address space of many processes. The fact that the same physical pages of a global section are mapped into many processes means that the requirements for physical memory are reduced.

8.1.2 Conserving Disk Storage Space

All programs executed under the VAX/VMS system must be disk resident. The use of shareable images, however, provides a way of reducing the amount of disk space required.

When a shareable image is linked into an executable image, it is not necessary to copy the physical contents of the shareable image. The installation of a shareable image causes the location of that image on disk to be recorded in the system's global section data base. The subsequent running of a program that uses that shareable image causes the VAX/VMS memory management software to load the copy from the separate shareable image file. Thus, many programs can reside on disk and be bound with a particular shareable image, and only one physical copy of that shareable image file need exist on disk.

8.1.3 Reducing Paging I/O

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

8.1.4 Using Shared Memory-Resident Data Bases

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 data base may be a named 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 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.

8.1.5 Making Software Updates Compatible

A major problem in maintaining a large software installation is how to incorporate a new version of a piece of software 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 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.

8.2 WRITING SOURCE PROGRAMS FOR SHAREABLE IMAGES

In order to obtain all the benefits of a shareable image, you must observe certain conventions in the source programs used to create it. This section describes these conventions.

8.2.1 Shareable and Nonshareable Data

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. However, if a piece of data used by a routine is, for instance, a loop counter, each process must have a separate counter, or the routine cannot be shared simultaneously. It is for this situation that the shareable (SHR) attribute of program sections was introduced. Users familiar with this situation recognize it as part of the problem referred to as reentrancy.

As was mentioned in Chapter 7, the linker allocates program sections with the SHR attribute in image sections separate from program sections with the NOSHR attribute. The image activator also treats image sections containing SHR program sections differently from image sections containing NOSHR program sections. The linker indicates this difference by an image section attribute called "copy on reference" in the case of writeable NOSHR program sections. (If the program section is not writeable, all processes can use the same copy regardless of SHR/NOSHR, since no form of data privacy or security is currently implemented.)

Thus, a copy on reference image section is one whose initial contents are established from the copy contained in the shareable image file, but which from then on during program execution is treated just like a user private image section. For each user, completely separate physical copies are produced for the copy on reference image sections contained in shareable images, and the system paging file is used to contain the pages of such sections 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 of its pages. In addition, when a page of such an image is removed from all user working sets, it is eventually written back into the shareable image file on disk. This last aspect makes it possible to rerun such applications as data acquisition or transaction processing with the most recent values of shareable, modifiable data.

Note that the cooperating user programs in applications like these are responsible for synchronizing access to such data. Note further that should it be necessary to revert to the initial values of the data, you must have made a separate copy before running the application the first time.

The FORTRAN example in Section 8.4.2 shows both kinds of data: variables generated by the compiler and the program are in copy on reference image sections, whereas the common areas are in shared data regions.

8.2.2 Position Independence

A position independent piece of code will execute correctly no matter where it is placed in the virtual address space after it is linked. That is, it can execute at an address different from the one at which the linker placed it. This section deals with position independence only as it concerns shareable images.

A shareable image is position independent if all of the following conditions are true:

- The only addresses that appear in the image are known to be fixed in the virtual address space (for example, the system service vectors of VAX/VMS). Note that if you specify a relocatable address in a shareable image, the linker will maintain the position independence by deferred relocation of the address.

- All instruction stream references to such addresses use absolute addressing mode (autoincrement deferred off the PC).

- All data references to such fixed addresses contain the complete virtual address.

- All references to any other location are relative to some base that is added to the address computation at execution time. For example, in the instruction stream, PC relative (or displacement from the PC) addressing mode would be used.

- There is no possibility that, after linking, the relationship between the target of a reference and the base to which it was made relative can be changed.

The current version of the linker is unable to verify that all the above conditions have been met. Therefore, the following strategy has been adopted:

- If any base address has been specified, the resultant shareable image is not position independent.

- The state of the position-independence attribute of the program sections is left to the user, and is considered only in gathering program sections into image sections. That is, the linker simply places PIC program sections in image sections separate from NOPIC program sections.

- With assistance from the compiler or assembler, the linker produces position-independent instruction stream references. (Refer to the discussion of the general addressing mode in the _VAX-11 MACRO Language Reference Manual_.) Basically, this means that the linker will choose the addressing mode (if so directed) based on the relocatability of the target of the reference.

- With assistance from the compiler or assembler, the linker produces position-independent address data by means of deferred relocation.

- A shareable image that is not position independent is placed at its link-time base address when it is subsequently bound into a user image.

- A shareable image that is position independent is allocated after user-defined clusters when it is subsequently bound into a user image.

- Shareable images that are not position independent are considered first by the linker.

Deferred relocation is the relocation of address data that is deferred until the executable image is linked from the shareable image. The linker uses deferred relocation when an image section contains a

.ADDRESS directive (VAX-11 MACRO) specifying a relocatable address. The linker marks the image section as COPY ALWAYS. When an executable image is linked from the shareable image, the linker copies the image section into the executable image and then calculates the correct address.

Note that to retain the benefits of a shareable image, .ADDRESS with a relocatable address should only occur in an image section that contains nonshareable data. These are the image sections that have the NOSHR and WRT attributes.

If shareable images are to be most useful among many processes, they should be position independent. The VAX-11 instruction set and addressing modes lend themselves to convenient generation of position-independent code. All the code generated by the VAX-11 FORTRAN and VAX-11 BASIC compilers is position independent.

8.2.3 **Transfer Vectors**

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 of interest. In the use of shareable images under VAX/VMS, transfer vectors are normally displacements rather than virtual addresses, for reasons of position independence.

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 other programs that are bound to the shareable image.

In Figure 8-1, 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.

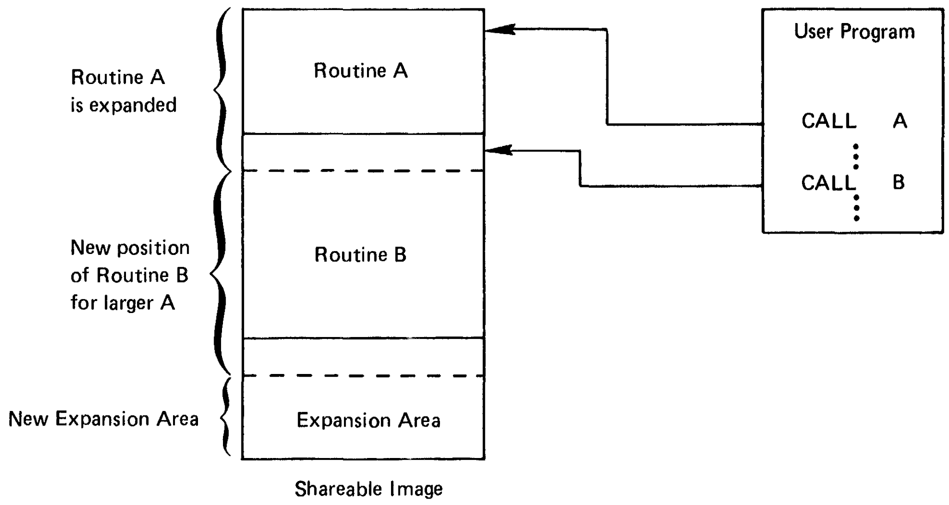

Figure 8-1 No Transfer Vectors

Using the example in Figure 8-1, 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. If the user program is not relinked, it can successfully call A, since its 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. Note that if the total size of the shareable image is larger, the user program must be relinked. (See Section 8.2.4).

In Figure 8-2, the same routines are built into a shareable image, but this time with transfer vectors at the beginning.

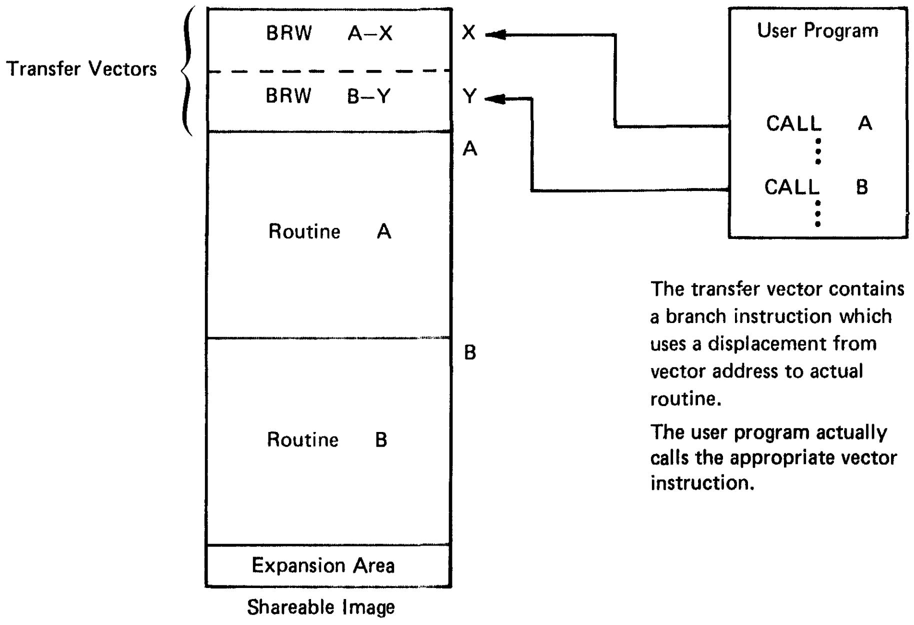
Transfer Vectors
BRW A–X
BRW B–Y
X
Y
A
B
Routine A
Routine B
Expansion Area
Shareable Image
User Program
CALL A
CALL B
The transfer vector contains a branch instruction which uses a displacement from vector address to actual routine.
The user program actually calls the appropriate vector instruction.

Figure 8-2 Transfer Vectors

In Figure 8-2, if routine A is expanded and the shareable image is relinked, the contents of the vector will change with no adverse affect on the user program. This is true so long as the user program calls the appropriate vector and the vector addresses do not change.

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. 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 a vector, you would need to link the three routines in the address sequence A,B,C; -- otherwise A or B could be in a different place and all user programs linked to the shareable image would need to be relinked. If you use a transfer vector, however, you can allocate a new vector location to C (after those for A and B). You can then link the three routines in any order.

Although you cannot create transfer vectors with FORTRAN, you can do so easily with VAX-11 MACRO. However, before you can create transfer vectors, you must define or permit the compiler to define entry points. With FORTRAN, the definition of entry points is done automatically, but with VAX-11 MACRO, you must explicitly define them. As an illustration, assume in Figure 8-2 that routines A and B are written in FORTRAN. In this case, the two global symbols A and B are defined as entry points, and the definitions given to the linker include a description of the registers to be saved with the call instruction. (You can achieve the same effect with the MACRO directive .ENTRY. See the VAX-11 MACRO Language Reference Manual.)

To create the transfer vector, you must use the VAX-11 MACRO assembler language. Consider the following fragment of MACRO code:

```
.TRANSFER    A    ;Begin transfer vector to A
.MASK        A    ;Store register save mask
BRW          A+2  ;BR to routine, beyond the
                  ;  register save mask
```

As the example suggests, register save masks (required at the target of a CALL instruction) occupy two bytes of memory. Thus, since it is the vector that you actually call, the register save mask is stored in the vector. The .MASK directive in the above example allocates the two bytes and directs the linker to (1) find the register save mask accompanying symbol A, and (2) write the word as the first two bytes of the vector. This mask is followed by a branch instruction that transfers control to routine A, at the instruction beyond the entry mask. (This example assumes that A is within 32K bytes of the vector; otherwise a JMP instruction would be required.)

The .TRANSFER directive has two purposes:

- It is an implicit universal declaration of symbol A if you are building a shareable image.
- It causes the linker to assign the universal symbol A the address of the vector, rather than the address of the routine within the image. This occurs after all uses of A within the shareable image have been given the value within the image.

Thus, all entry points of a shareable image are universal when vectored in this way. The user program outside the shareable image can call the routine A in the same way as it would an ordinary object module.

8.2.4 Rules for Creating Upwardly Compatible Shareable Images

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

- Transfer vectors must not be rearranged or removed.
- The new shareable image must have exactly the same number of image sections.
- The shareable image must not be larger than it originally was unless the shareable image is the cluster with the largest virtual address in the image (this position is usually reserved for the run-time library).

While a shareable image is being developed, it is useful to reserve expansion space to allow the image to grow. If you exceed the expansion space, you must relink all executable images that are linked with the shareable image. Because there is a substantial overhead for 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.

8.3 LINK COMMAND AND PERTINENT OPTIONS

The LINK command for creating a shareable image is similar to that for any other type of image, except that you must use the /SHAREABLE[=file-spec] qualifier, which is described in Chapter 4.

The UNIVERSAL=, GSMATCH=, and PROTECT= options and the /PROTECT qualifier are provided specifically to control characteristics of shareable images. Chapter 5 describes the syntax of these options. Sections 8.3.1, 8.3.2, and 8.3.3 describe their purpose.

8.3.1 UNIVERSAL= Option

Universal symbols are the global symbols of a shareable image which are of use to the programs that subsequently link with the shareable image. It is possible for none or all of the global symbols of a shareable image to be universal symbols. Typically, however, a very small set of the global symbols of the image are universal because few of them are of use outside the shareable image. Universal symbols are the only symbols written to the symbol table of a shareable image.

Most programming languages provide no way of characterizing a symbol as universal. (VAX-11 MACRO, however, has a declaration for creating transfer vectors. See Section 8.2.3.) Thus, to tell the linker which symbols are to be universal, the option UNIVERSAL= is provided.

Normally, all the entry points (routine names) provided in a shareable image are universal symbols. Sometimes, however, other constants are of interest to users of the facility, and these can also be declared as universal symbols. Section 8.4.1 contains an example showing the declaration of several such constants in the Cross Reference Facility as universal symbols.

8.3.2 GSMATCH= Option

When a shareable image is bound into a user executable image, its image sections are promoted to global sections. (The VAX/VMS system promotes image sections to global sections when a shareable image is installed.) When an executable image is run, the image activator checks to see if the shareable image that the executable image was linked with matches the current one. The image activator compares the values specified in the GSMATCH= option when the shareable images were linked (see Section 5.3).

The GSMATCH= option sets the matching requirements for versions of the shareable image and causes a two-part identification field to be associated with the global section name. During the search for a global section at image activation time, the global section name and the major part of the identification must match exactly. The behavior of the comparison with the minor part of the identification is determined by the keyword specified in the GSMATCH= option. The keyword can be:

- EQUAL -- the minor identifications must match.
- LEQUAL -- the minor identification of the global section in the user image must be less than or equal to that in the global data base.

- NEVER -- even if both parts of the identification match exactly, the image activator rejects the shareable image. The purpose of NEVER is to specify that the linker should always produce a private copy of this shareable image in each executable image file.

- ALWAYS -- the image activator only checks to see that the global section names are the same and ignores both parts of the identification.

The GSMATCH= option is provided to set these parameters when the shareable image is being linked. See Section 8.5 for mre information on the effects of these parameters.

8.3.3 PROTECT = Option and /PROTECT Qualifier

The PROTECT= option and the /PROTECT command qualifier are used to create privileged shareable images. A privileged shareable image can execute change mode to kernel and execute change mode to executive instructions even when it is linked with a nonprivileged executable image. A privileged shareable image allows executable images to call user-written procedures with enhanced privileges in the same way that they call system services.

The PROTECT= option protects specified image clusters, and the /PROTECT command qualifier protects the entire shareable image. Another way to protect code is by using the VEC program section attribute which protects an individual program section.

There are three effects of protecting a segment of a shareable image:

- The shareable image is made a privileged shareable image.

- Within the protected segment, the change mode instructions are allowed even when the process does not have the necessary privilege.

- Code executing in user mode cannot write in protected areas.

When creating a privileged shareable image, you should protect the clusters that require privilege, and not protect the clusters to which code executing in user mode needs write access. 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 the VAX/VMS Real-Time User's Guide for more information on privileged shareable images.

8.4 EXAMPLES OF SHAREABLE IMAGES

The following sections contain examples of shareable images.

8.4.1 Example of Transfer Vector and Universal Symbols

Figure 8-3 is a listing of the source code for the module that is the transfer vector for the Cross Reference Facility. Figure 8-4 shows the LINK command and options files used to create the shareable image CRFSHR on the logical device EXEC$:. Figure 8-5 shows the map that resulted from this link operation.

Note that of the 26 global symbols in the image, only 17 are of interest outside the image -- 8 vectored entry points and 9 constants. Note also that the transfer vector is placed in its own cluster to ensure that it is allocated at the low-addressed end of the address space. (As you can see from the example, explicitly defined clusters are allocated first in the address space.)

The values of the transfer vector symbols retain the values of the routine addresses. (See the listing of the relocatable universal symbols in the map in Figure 8-5.)

```
CRF_TRANSFER                                                          20-FEB-1980 08:51:37  VAX-11 Macro V02.41          Page   3
V01.02                            TRANSFER_VECTORS                     6-JUL-1978 15:09:24  _DBB0:[CRF.SRC]CRFTFRVEC.MAR;9   (3)

                   0000     60          .SBTTL  TRANSFER_VECTORS
                   0000     61 ;++
                   0000     62 ; FUNCTIONAL DESCRIPTION:
                   0000     63 ;
                   0000     64 ; THIS MODULE DEFINES THE TRANSFER VECTORS FOR THE ENTRY POINTS CALLED
                   0000     65 ; BY A USER OF CRF.  THIS MODULE ENABLES CRF TO BE LINKED AS A SHARABLE IMAGE.
                   0000     66 ;
                   0000     67 ; CALLING SEQUENCE:
                   0000     68 ;
                   0000     69 ;       NONE
                   0000     70 ;
                   0000     71 ; INPUT PARAMETERS:
                   0000     72 ;
                   0000     73 ;       NONE
                   0000     74 ;
                   0000     75 ; IMPLICIT INPUTS:
                   0000     76 ;
                   0000     77 ;       NONE
                   0000     78 ;
                   0000     79 ; OUTPUT PARAMETERS:
                   0000     80 ;
                   0000     81 ;       NONE
                   0000     82 ;
                   0000     83 ; IMPLICIT OUTPUTS:
                   0000     84 ;
                   0000     85 ;       NONE
                   0000     86 ;
                   0000     87 ; COMPLETION CODES:
                   0000     88 ;
                   0000     89 ;       NONE
                   0000     90 ;
                   0000     91 ; SIDE EFFECTS:
                   0000     92 ;
                   0000     93 ;       NONE
                   0000     94 ;
                   0000     95 ;--
                   0000     96
                   0000     97
               00000000     98          .PSECT  $$VECTOR_0_CRF,PIC,SHR,NOWRT,EXE
                   0000     99
                   0000     .1 CRF_TRANSFER:
                   0000    100          .TRANSFER       CRF$INSRTKEY    ; INSERTS A CROSS REFERENCE KEY
             0000' 0000    101          .MASK           CRF$INSRTKEY
      FFFD'   31   0002    102          BRW             CRF$INSRTKEY+2
                   0005    103
                   0005    104          .TRANSFER       CRF$INSRTREF    ; INSERTS A REFERENCE TO A KEY
             0000' 0005    105          .MASK           CRF$INSRTREF
      FFF8'   31   0007    106          BRW             CRF$INSRTREF+2
                   000A    107
                   000A    108          .TRANSFER       CRF$OUT         ; OUTPUTS CROSS REFERENCE SUMMARY
             0000' 000A    109          .MASK           CRF$OUT
      FFF3'   31   000C    110          BRW             CRF$OUT+2
                   000F    111
                   000F     .1          .TRANSFER       LIB$CRF_INS_KEY         ; INSERT CROSS REFERENCE KEY
             0000' 000F     .2          .MASK           LIB$CRF_INS_KEY
      FFEE'   31   0011     .3          BRW             LIB$CRF_INS_KEY+2
                   0014     .4
```

Figure 8-3 Transfer Vector Example Listing

```
CRF_TRANSFER
V01.02                      TRANSFER_VECTORS                             20-FEB-1980 00:51:37  VAX-11 Macro V02.41           Page   4
                                                                          6-JUL-1978 15:09:24  _DBB0:[CRF.SRC]CRFTFRVEC.MAR;9    (3)

                          0014     .5          .TRANSFER      LIB$CRF_INS_REF           ; INSERT REFERENCE TO A KEY
                   0000'  0014     .6          .MASK          LIB$CRF_INS_REF
              FFE9'   31  0016     .7          BRW            LIB$CRF_INS_REF+2
                          0019     .8
                          0019     .9          .TRANSFER      LIB$CRF_OUTPUT            ; OUTPUT CROSS REFERENCE
                   0000'  0019    .10          .MASK          LIB$CRF_OUTPUT
              FFE4'   31  001B    .11          BRW            LIB$CRF_OUTPUT+2
                          001E    .12
                          001E    .13          .TRANSFER      CRF$GET_MEM               ; ALLOCATE DYNAMIC MEMORY
              FFDF'   31  001E    .14          BRW            CRF$GET_MEM               ; JSB ENTRY
                          0021    .15
                          0021    .16          .TRANSFER      CRF$FREE_MEM              ; DEALLOCATE DYNAMIC MEMORY
              FFDC'   31  0021    .17          BRW            CRF$FREE_MEM
                          0024    .18
               00000200   0024    .19          .BLKB          512-<.-CRF_TRANSFER>      ; PAD TO FULL PAGE
                          0200    113          .END
```

Figure 8-3 (Cont.) Transfer Vector Example Listing

```
CRF_TRANSFER                                                                        20-FEB-1980 00:51:37  VAX-11 Macro V02.41          Page   5
Symbol table                                                                         6-JUL-1978 15:09:24  _DBB0:[CRF.SRC]CRFTFRVEC.MAR;9    (3)

CRF$FREE_MEM      ********    X    02
CRF$GET_MEM       ********    X    02
CRF$INSRTKEY      ********    X    02
CRF$INSRTREF      ********    X    02
CRF$OUT           ********    X    02
CRF_TRANSFER      00000000 R       02
LIB$CRF_INS_KEY   ********    X    02
LIB$CRF_INS_REF   ********    X    02
LIB$CRF_OUTPUT    ********    X    02

                                                   +----------------+
                                                   ! Psect synopsis !
                                                   +----------------+

PSECT name                       Allocation          PSECT No.  Attributes
----------                       ----------          ---------  ----------
.  ABS  .                        00000000  (    0.)  00 (  0.)  NOPIC   USR   CON   ABS   LCL NOSHR NOEXE NORD  NOWRT NOVEC BYTE
. BLANK .                        00000000  (    0.)  01 (  1.)  NOPIC   USR   CON   REL   LCL NOSHR   EXE   RD    WRT NOVEC BYTE
$$VECTOR_0_CRF                   00000200  (  512.)  02 (  2.)    PIC   USR   CON   REL   LCL   SHR   EXE   RD  NOWRT NOVEC BYTE

                                                +------------------------+
                                                ! Performance indicators !
                                                +------------------------+

Phase                       Page faults    CPU Time       Elapsed Time
-----                       -----------    --------       ------------
Initialization                      23     00:00:00.05    00:00:00.40
Command processing                  32     00:00:00.24    00:00:02.42
Pass 1                             105     00:00:00.62    00:00:06.25
Symbol table sort                    0     00:00:00.01    00:00:00.01
Pass 2                              37     00:00:00.46    00:00:03.46
Symbol table output                  1     00:00:00.01    00:00:00.01
Psect synopsis output                4     00:00:00.02    00:00:00.02
Cross-reference output               0     00:00:00.00    00:00:00.00
Assembler run totals               208     00:00:01.41    00:00:12.57

The working set limit was 500 pages.
1399 bytes (3 pages) of virtual memory were used to buffer the intermediate code.
There were 10 pages of symbol table space allocated to hold 9 non-local and 0 local symbols.
135 source lines were read in Pass 1, producing 12 object records in Pass 2.
0 pages of virtual memory were used to define 0 macros.

                                               +--------------------------+
                                               ! Macro library statistics !
                                               +--------------------------+

Macro library name                             Macros defined
------------------                             --------------
_DBB4:[CRF.OBJ]CRF.MLB;1                                  0
_DBB4:[SYSLIB]STARLET.MLB;1                               0
TOTALS (all libraries)                                    0

0 GETS were required to define 0 macros.

There were no errors, warnings or information messages.

/LIS=LIS$:CRFTFRVEC/OBJ=OBJ$:CRFTFRVEC MSRC$:CRFTFRVEC+LIB$:CRF/LIB
```

Figure 8-3 (Cont.) Transfer Vector Example Listing

```
$ !
$ !     [ C R F . C O M ] C R F S H R L N K . C O M
$ !
$ !     Link cross reference facility
$ !
$ DELETE EXE$:CRFSHR.EXE;*,MAP$:.MAP;*
$ LINK /SHARE=EXE$:CRFSHR /MAP=MAP$:CRFSHR /FULL SYS$INPUT/OPTIONS
!
! Options input
!
OBJ$:CRF/INCLUDE=(CRF_CREF)/LIBRARY
!
!       CREATE A SEPARATE CLUSTER AT LOW ADDRESSED END FOR THE
!               TRANSFER VECTORS.
!
CLUSTER=TRANSFER_VECTOR,,,OBJ$:CRF/INCLUDE=CRF_TRANSFER
GSMATCH = LEQUAL, 1, 101
UNIVERSAL=CRF$K_ASCIC,-                         ! UNIVERSALIZE THE NON ENTRY
                CRF$K_BIN_U32,-                 ! POINT SYMBOLS THAT USERS
                CRF$K_DEF,CRF$K_REF,-           ! MAY NEED.
                CRF$K_VALUES,-
                CRF$K_VALS_REFS,-
                CRF$K_DEFS_REFS,-
                CRF$K_DELETE,CRF$K_SAVE
SYMBOL=CRF$C_HASHSIZE,719                       ! HASH TABLE SIZE -- SHOULD BE PRIME
```

Figure 8-4 Transfer Vector Example Link Command

```
EXE$:CRFSHR                                              20-FEB-1980 17:54       LINKER V02.39                 Page    1
                                        +------------------------+
                                        ! Object Module Synopsis !
                                        +------------------------+

Module Name      Ident               Bytes      File                                  Creation Date       Creator
-----------      -----               -----      -----                                 -------------       -------
CRF_TRANSFER     V01.02                512 _DBB4:[CRF.OBJ]CRF.OLB;1                20-FEB-1980 00:51  VAX-11 Macro V02.41
CRF_CREF         V02.01               1747 _DBB4:[CRF.OBJ]CRF.OLB;1                20-Feb-1980 00:52  VAX-11 Bliss-32 V2-622
GETMEM           V01.05                282 _DBB4:[CRF.OBJ]CRF.OLB;1                20-FEB-1980 00:50  VAX-11 Macro V02.41
CRFGBL           V02.01                  0 _DBB4:[CRF.OBJ]CRF.OLB;1                20-FEB-1980 00:49  VAX-11 Macro V02.41
CRFOR            X01.01                 76 _DBB4:[CRF.OBJ]CRF.OLB;1                20-FEB-1980 00:50  VAX-11 Macro V02.41
CRFSUB           V01.02                202 _DBB4:[CRF.OBJ]CRF.OLB;1                20-FEB-1980 00:51  VAX-11 Macro V02.41
SYSVECTOR        0219                    0 _DBB4:[SYSLIB]STARLET.OLB;1             19-FEB-1980 21:59  VAX-11 Macro V02.41
```

Figure 8-5 Transfer Vector Example Map

```
_DBB4:[CRF.OBJ]CRFSHR.EXE;1                                   20-FEB-1980 17:54        LINKER V02.39                  Page    2
                                            +-------------------------+
                                            ! Image Section Synopsis !
                                            +-------------------------+

   Cluster        Type Pages   Base Addr Disk VBN PFC Protection and Paging      Global Sec. Name   Match     Majorid    Minorid
   -------        ---- -----   --------- -------- --- --------------------      ----------------   -----     -------    -------

TRANSFER_VECTOR     3     1     00000200         2    0 READ ONLY

DEFAULT_CLUSTER     2     5     00000400         3    0 READ ONLY




_DBB4:[CRF.OBJ]CRFSHR.EXE;1                                   20-FEB-1980 17:54        LINKER V02.39                  Page    3
                                           +--------------------------+
                                           ! Program Section Synopsis !
                                           +--------------------------+

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

$$VECTOR_0_CRF                      00000200 000003FF 00000200 (         512.) BYTE 0   PIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT,NOVEC
                  CRF_TRANSFER      00000200 000003FF 00000200 (         512.) BYTE 0

. BLANK .                           00000200 00000200 00000000 (           0.) BYTE 0 NOPIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,  WRT,NOVEC
                  CRF_TRANSFER      00000200 00000200 00000000 (           0.) BYTE 0

$CODE$                              00000400 00000D02 00000903 (        2307.) LONG 2 NOPIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,NOWRT,NOVEC
                  CRF_CREF          00000400 00000AD2 000006D3 (        1747.) LONG 2
                  GETMEM            00000AD3 00000BEC 0000011A (         282.) BYTE 0
                  CRFOR             00000BED 00000C38 0000004C (          76.) BYTE 0
                  CRFSUB            00000C39 00000D02 000000CA (         202.) BYTE 0

. BLANK .                           00000E00 00000E00 00000000 (           0.) BYTE 0 NOPIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,  WRT,NOVEC
                  GETMEM            00000E00 00000E00 00000000 (           0.) BYTE 0
                  CRFGBL            00000E00 00000E00 00000000 (           0.) BYTE 0
                  CRFOR             00000E00 00000E00 00000000 (           0.) BYTE 0
                  CRFSUB            00000E00 00000E00 00000000 (           0.) BYTE 0
                  SYSVECTOR         00000E00 00000E00 00000000 (           0.) BYTE 0
```

Figure 8-5 (Cont.) Transfer Vector Example Map

```
_DBB4:[CRF.OBJ]CRFSHR.EXE;1                                    20-FEB-1980 17:54       LINKER V02.39             Page    4
                                           +-----------------+
                                           ! Symbols By Name !
                                           +-----------------+

Symbol           Value                Symbol          Value                Symbol          Value                Symbol          Value
------           -----                ------          -----                ------          -----                ------          -----
CRF$C_HASHSIZE   000002CF
CRF$C_MAXCOL     00000040
CRF$C_MAXLINWID  00000084
CRF$FREE_MEM     00000B6D-RU
CRF$GET_MEM      00000AEF-RU
CRF$INSRTKEY     00000558-RU
CRF$INSRTREF     00000589-RU
CRF$K_ASCIC      00000000-U
CRF$K_BIN_U32    00000001-U
CRF$K_DEF        00000001-U
CRF$K_DEFS_REFS  00000002-U
CRF$K_DELETE     00000000-U
CRF$K_REF        00000000-U
CRF$K_SAVE       00000001-U
CRF$K_VALS_REFS  00000001-U
CRF$K_VALUES     00000000-U
CRF$OUT          000007B1-RU
FREE_MEM         00000B6D-R
GET_MEM          00000AFA-R
GET_ZMEM         00000AD3-R
LIB$CRF_INS_KEY  00000BED-RU
LIB$CRF_INS_REF  00000C02-RU
LIB$CRF_OUTPUT   00000C1B-RU
SORT_HASH_TABLE  00000C61-R
SYS$EXPREG       80000148
SYS$FAO          80000150
```

Figure 8-5 (Cont.) Transfer Vector Example Map

```
_DBB4:[CRF.OBJ]CRFSHR.EXE;1                                      20-FEB-1980 17:54        LINKER V02.39                 Page    5
                                                       +-------------------+
                                                       ! Symbols By Value !
                                                       +-------------------+

Value                                       Symbols...
-----                                       ----------
00000000      U-CRF$K_ASCIC       U-CRF$K_DELETE       U-CRF$K_REF          U-CRF$K_VALUES
00000001      U-CRF$K_BIN_U32     U-CRF$K_DEF          U-CRF$K_SAVE         U-CRF$K_VALS_REFS
00000002      U-CRF$K_DEFS_REFS
00000040        CRF$C_MAXCOL
00000084        CRF$C_MAXLINWID
000002CF        CRF$C_HASHSIZE
00000558      R-CRF$INSRTKEY
00000589      R-CRF$INSRTREF
000007B1      R-CRF$OUT
00000AD3      R-GET_ZMEM
00000AEF      R-CRF$GET_MEM
00000AFA      R-GET_MEM
00000B6D      R-CRF$FREE_MEM      R-FREE_MEM
00000BED      R-LIB$CRF_INS_KEY
00000C02      R-LIB$CRF_INS_REF
00000C1B      R-LIB$CRF_OUTPUT
00000C61      R-SORT_HASH_TABLE
80000148        SYS$EXPREG
80000150        SYS$FAO


                  Key for special characters above:
                        +-------------------+
                        ! *  = Undefined    !
                        ! U  = Universal    !
                        ! R  = Relocatable  !
                        ! WK = Weak         !
                        +-------------------+
```

Figure 8-5 (Cont.) Transfer Vector Example Map

```
_DBB4:[CRF.OBJ]CRFSHR.EXE;1                                  20-FEB-1980 17:54       LINKER V02.39                   Page    6
                                                           +----------------+
                                                           ! Image Synopsis !
                                                           +----------------+

Virtual memory allocated:                       00000200 00000DFF 00000C00 (3072. bytes, 6. pages)
Stack size:                                           0. pages
Image header virtual block limits:                    1.         1. (     1. block)
Image binary virtual block limits:                    2.         7. (     6. blocks)
Image name and identification:                  CRFSHR .OLB;
Number of files:                                      4.
Number of modules:                                    8.
Number of program sections:                           9.
Number of global symbols:                            16.
Number of image sections:                             4.
Image type:                                       PIC, SHAREABLE, Global section match = "LESS/EQUAL", G.S. Ident, Major=1, Minor=101
Map format:                                     FULL in file "_DBB4:[CRF.LIS]CRFSHR.MAP;1"
Estimated map length:                           29. blocks
                                                      +---------------------+
                                                      ! Link Run Statistics !
                                                      +---------------------+

Performance Indicators                                      Page Faults   CPU Time         Elapsed Time
----------------------                                      -----------   --------         ------------
    Command processing:                                              49   00:00:00.17      00:00:00.85
    Pass 1:                                                         109   00:00:00.82      00:00:05.00
    Allocation/Relocation:                                           15   00:00:00.07      00:00:00.68
    Pass 2:                                                          23   00:00:00.29      00:00:02.26
    Map data after object module synopsis:                           20   00:00:00.21      00:00:00.87
    Symbol table output:                                              5   00:00:00.04      00:00:00.20
Total run values:                                                   221   00:00:01.60      00:00:09.86

Using a working set limited to 500 pages and 78 pages of data storage (excluding image)

Total number object records read (both passes):   369
    of which 173 were in libraries and 26 were DEBUG data records containing 907 bytes

Number of modules extracted explicitly          = 2
    with 5 extracted to resolve undefined symbols

2 library searches were for symbols not in the library searched

A total of 4 global symbol table records was written

/SHARE=EXES:CRFSHR/MAP=MAPS:CRFSHR/FULL SYS$INPUT/OPTIONS
```

Figure 8-5 (Cont.) Transfer Vector Example Map

8.4.2 Example of FORTRAN Shared COMMON

This section contains an example of FORTRAN shared COMMON. The FORTRAN subroutine SHRSUB is a shareable image that contains two COMMON areas. One is a shared or global COMMON area (GLOBAL_COMMON). This is a data area that is shared by all executable images linked with the shareable image. The other is a nonshared COMMON area (LOCAL_COMMON); each executable image has its own copy of LOCAL_COMMON. The program section attributes control whether a COMMON area is shared or not. Note the attributes of such program section -- in particular, GBL, OVR, and SHR.

- The GBL attribute causes the program section to be recorded in the symbol table of the shareable image for later use by an executable image. The FORTRAN compiler sets the GBL attribute for all COMMON areas.
- The OVR attribute ensures that all modules contributing to the program section start at the same address space. The FORTRAN compiler sets the OVR attribute for all COMMON areas.
- The SHR attribute specifies that only one copy of the COMMON area is to appear in physical memory. The FORTRAN compiler sets the SHR attribute for all COMMON areas. Note that a PSECT_ATTR option in an options file changes the program section attribute of LOCAL_COMMON to NOSHR.

Figure 8-6 shows the listing of the FORTRAN subprogram SHRSUB that defines GLOBAL_COMMON and LOCAL_COMMON; Figure 8-7 shows the LINK command to create the shareable image; and Figure 8-8 shows the resulting map. Figure 8-9 shows the listing of a FORTRAN program MAIN that calls SHRSUB; Figure 8-10 shows the LINK command for MAIN; and Figure 8-11 shows the resulting map.

```
                                                          5-Mar-1980 18:00:53    VAX-11 FORTRAN T1.95-36          Page 1
                                                          5-Mar-1980 17:59:01    _DBB2:[GOLDMAN]SHRSUB.FOR;5
0001            SUBROUTINE SHRSUB
        C
        C       LOCAL_COMMON will be private per-image common
        C       GLOBAL_COMMON will be a shareable common
        C
0002            COMMON /LOCAL_COMMON/ J,J1(511)
0003            COMMON /GLOBAL_COMMON/ K,K1(511)

0004            TYPE *,' J is in the COPY-ON-REF common,
                1 K is in a shareable common'
0005            TYPE *,' Entering a zero will leave the variable unchanged'
0006     10     TYPE *,' '
0007            TYPE *,' Previous values: J = ',J,'  K = ',K
0008            IF (M .NE. 0) J = M
0009            IF (N .NE. 0) K = N
0010            TYPE *,' Current values: J = ',J,'   K = ',K
0011            TYPE *,' Enter new values for J,K'
0012            ACCEPT *,M,N
0013            IF ((M .EQ. 0) .AND. (N .EQ. 0)) RETURN
0014            GOTO 10
0015            END


PROGRAM SECTIONS

    Name                                Bytes   Attributes

  0 $CODE                                 316   PIC CON REL LCL   SHR   EXE   RD NOWRT LONG
  1 $PDATA                                191   PIC CON REL LCL   SHR NOEXE   RD NOWRT LONG
  2 $LOCAL                                 72   PIC CON REL LCL NOSHR NOEXE   RD   WRT LONG
  3 LOCAL_COMMON                         2048   PIC OVR REL GBL   SHR NOEXE   RD   WRT LONG
  4 GLOBAL_COMMON                        2048   PIC OVR REL GBL   SHR NOEXE   RD   WRT LONG

ENTRY POINTS

    Address   Type  Name

  0-00000000        SHRSUB

VARIABLES

    Address   Type  Name                Address   Type  Name                Address   Type  Name                Address   Type  Name

  3-00000000  I*4   J                 4-00000000  I*4   K                 2-00000000  I*4   M                 2-00000004  I*4   N

ARRAYS

    Address   Type  Name                Bytes  Dimensions

  3-00000004  I*4   J1                   2044  (511)
  4-00000004  I*4   K1                   2044  (511)
```

Figure 8-6 Listing of FORTRAN Shared COMMON Subprogram

```
SHRSUB                                                           5-Mar-1980 18:00:53    VAX-11 FORTRAN T1.95-36               Page 2
                                                                 5-Mar-1980 17:59:01    _DBB2:[GOLDMAN]SHRSUB.FOR;5

LABELS

    Address   Label

  1-0000003F  10


Total Space Allocated = 4675 Bytes


COMMAND QUALIFIERS

  FORTRAN /LIST SHRSUB

  /CHECK=(NOBOUNDS,OVERFLOW)
  /DEBUG=(NOSYMBOLS,TRACEBACK)
  /F77  /NOG_FLOATING  /I4  /OPTIMIZE  /WARNINGS  /NOD_LINES  /NOMACHINE_CODE  /CONTINUATIONS=19


COMPILATION STATISTICS

  Run Time:           1.03 seconds
  Elapsed Time:       8.26 seconds
  Page Faults:        342
  Dynamic Memory:     37 pages
```

Figure 8-6 (Cont.)Listing of FORTRAN Shared COMMON Subprogram

```
$ LINK /SHARE=SHRSUB /MAP=SHRSUB /FULL SHRSUB, SYS$INPUT:/OPTIONS
!
! Options input for SHRSUB shareable image
!
PSECT=LOCAL_COMMON,NOSHR
COLLECT=LOCAL_CLUSTER,LOCAL_COMMON
UNIVERSAL=SHRSUB
```

Figure 8-7 LINK Command for FORTRAN Shared COMMON Subprogram

```
SHRSUB                                                22-FEB-1980 14:26       LINKER V02.40                 Page    1
                                        +------------------------+
                                        ! Object Module Synopsis !
                                        +------------------------+

Module Name     Ident              Bytes      File                              Creation Date       Creator
-----------     -----              -----      -----                             -------------       -------
SHRSUB          01                  4665 _DBB2:[GOLDMAN]SHRSUB.OBJ;1           22-Feb-1980 14:26  VAX-11 FORTRAN T1.95-36
OTS$LINKAGE     1-003                  3 _DRA5:[SYSLIB]STARLET.OLB;1           19-FEB-1980 21:57  VAX-11 Macro V02.41
VMSRTL          .EXE;1                 0 _DRA5:[SYSLIB]VMSRTL.EXE;1            20-FEB-1980 18:55  LINK-32 V02.39




_DBB2:[GOLDMAN]SHRSUB.EXE;1                           22-FEB-1980 14:26       LINKER V02.40                 Page    2
                                        +------------------------+
                                        ! Image Section Synopsis !
                                        +------------------------+

   Cluster       Type Pages   Base Addr Disk VBN PFC Protection and Paging      Global Sec. Name   Match      Majorid   Minorid
   -------       ---- -----   --------- -------- --- ---------------------      ----------------   -----      -------   -------

LOCAL_CLUSTER       4     4    00000200        2   0 READ WRITE    COPY ON REF

DEFAULT_CLUSTER     3     1    00000A00        6   0 READ ONLY
                    3     4    00000C00        7   0 READ WRITE
                    3     1    00001400       11   0 READ ONLY
                    4     1    00001600       12   0 READ WRITE    COPY ALWAYS

VMSRTL              3    11    00001800        0   1 READ ONLY                  VMSRTL_001       LESS/EQUAL        1       2000
                    3   193    00002E00        0   0 READ ONLY                  VMSRTL_002       LESS/EQUAL        1       2000
                    4     4    0001B000        0   0 READ WRITE    COPY ON REF  VMSRTL_003       LESS/EQUAL        1       2000
```

Figure 8-8 Map of FORTRAN Shared COMMON Subprogram

```
_DBB2:[GOLDMAN]SHRSUB.EXE;1                           22-FEB-1980 14:26     LINKER V02.40                Page   3
                                         +-------------------------+
                                         ! Program Section Synopsis !
                                         +-------------------------+

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

LOCAL_COMMON                  00000200 000009FF 00000800 (      2048.) LONG 2   PIC,USR,OVR,REL,GBL,NOSHR,NOEXE,  RD,  WRT,NOVEC
               SHRSUB         00000200 000009FF 00000800 (      2048.) LONG 2

$PDATA                        00000A00 00000ABE 000000BF (       191.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,NOEXE,  RD,NOWRT,NOVEC
               SHRSUB         00000A00 00000ABE 000000BF (       191.) LONG 2

. BLANK .                     00000A00 00000A00 00000000 (         0.) BYTE 0 NOPIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,  WRT,NOVEC
               OTS$LINKAGE    00000A00 00000A00 00000000 (         0.) BYTE 0

GLOBAL_COMMON                 00000C00 000013FF 00000800 (      2048.) LONG 2   PIC,USR,OVR,REL,GBL,  SHR,NOEXE,  RD,  WRT,NOVEC
               SHRSUB         00000C00 000013FF 00000800 (      2048.) LONG 2

$CODE                         00001400 00001531 00000132 (       306.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT,NOVEC
               SHRSUB         00001400 00001531 00000132 (       306.) LONG 2

_OTS$CODE                     00001534 00001536 00000003 (         3.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT,NOVEC
               OTS$LINKAGE    00001534 00001536 00000003 (         3.) LONG 2

$LOCAL                        00001600 00001647 00000048 (        72.) LONG 2   PIC,USR,CON,REL,LCL,NOSHR,NOEXE,  RD,  WRT,NOVEC
               SHRSUB         00001600 00001647 00000048 (        72.) LONG 2
```

Figure 8-8 (Cont.) Map of FORTRAN Shared COMMON Subprogram

```
_DBB2:[GOLDMAN]SHRSUB.EXE;1                                22-FEB-1980 14:26       LINKER V02.40                 Page    4
                                                  +------------------+
                                                  ! Symbols By Name !
                                                  +------------------+

Symbol          Value              Symbol          Value              Symbol          Value              Symbol           Value
------          -----              ------          -----              ------          -----              ------           -----
BAS$$BLNK_LINE  000023A0-RU        BAS$INSTR       000020B0-RU        BAS$SCRATCH     00002308-RU        FOR$$CB_PUSH     00001E08-RU
BAS$$CB_GET     00002380-RU        BAS$IN_D_R      000021F0-RU        BAS$STATUS      00002338-RU        FOR$$CB_RET      00001E18-RU
BAS$$CB_POP     00002370-RU        BAS$IN_F_R      000021E8-RU        BAS$STR_D       000020C0-RU        FOR$$ERRSNS_SAV  00001E28-RU
     .               .                  .               .                 .               .                   .              .
     .               .                  .               .                 .               .                   .              .
     .               .                  .               .                 .               .                   .              .


_DBB2:[GOLDMAN]SHRSUB.EXE;1                                22-FEB-1980 14:26       LINKER V02.40                 Page    7
                                                  +-------------------+
                                                  ! Symbols By Value !
                                                  +-------------------+

Value                                    Symbols...
-----                                    ----------
00001400    RU-SHRSUB
00001534     R-BAS$LINKAGE         R-FOR$LINKAGE         R-OTS$LINKAGE
00001800    RU-FOR$CLOSE
    .           .
    .           .
    .           .


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

Figure 8-8 (Cont.) Map of FORTRAN Shared COMMON Subprogram

```
_DBB2:[GOLDMAN]SHRSUB.EXE;1                              22-FEB-1980 14:26       LINKER V02.40                Page   15
                                                    +-----------------+
                                                    ! Image Synopsis !
                                                    +-----------------+

Virtual memory allocated:                    00000200 000187FF 00018600 (112128. bytes, 219. pages)
Stack size:                                          0. pages
Image header virtual block limits:                   1.         1. (     1. block)
Image binary virtual block limits:                   2.        12. (    11. blocks)
Image name and identification:               SHRSUB .OBJ;
Number of files:                                     3.
Number of modules:                                   3.
Number of program sections:                          9.
Number of global symbols:                          242.
Number of image sections:                            9.
Image type:                                    PIC, SHAREABLE. Global section match = "EQUAL", G.S. Ident, Major=244, Minor=8260915
Map format:                                  FULL in file "_DBB2:[GOLDMAN]SHRSUB.MAP;1"
Estimated map length:                        55. blocks
                                                  +----------------------+
                                                  ! Link Run Statistics !
                                                  +----------------------+

Performance Indicators                                 Page Faults    CPU Time         Elapsed Time
----------------------                                 -----------    --------         ------------
    Command processing:                                         42    00:00:00.06      00:00:00.78
    Pass 1:                                                    182    00:00:00.71      00:00:02.35
    Allocation/Relocation:                                     102    00:00:00.26      00:00:00.91
    Pass 2:                                                     70    00:00:00.23      00:00:01.41
    Map data after object module synopsis:                     159    00:00:01.84      00:00:03.64
    Symbol table output:                                        20    00:00:00.13      00:00:00.83
Total run values:                                              575    00:00:03.23      00:00:09.92

Using a working set limited to 268 pages and 51 pages of data storage (excluding image)

Total number object records read (both passes):   75
    of which 16 were in libraries and 4 were DEBUG data records containing 149 bytes

Number of modules extracted explicitly            = 0
    with 1 extracted to resolve undefined symbols

0 library searches were for symbols not in the library searched

A total of 24 global symbol table records was written

/SHARE=SHRSUB/MAP=SHRSUB/FULL SHRSUB,SYS$INPUT:/OPTIONS
```

Figure 8-8 (Cont.) Map of FORTRAN Shared COMMON Subprogram

```
                                                        22-Feb-1980 14:26:58    VAX-11 FORTRAN T1.95-36          Page 1
                                                        29-Nov-1979 15:24:20    _DBB2:[GOLDMAN]MAIN.FOR;1

0001          PROGRAM MAIN
0002          COMMON /LOCAL_COMMON/ J,J1(511)
0003          COMMON /GLOBAL_COMMON/ K,K1(512)

0004          TYPE *,' This program tests COPY-ON-REF commons'
0005          CALL SHRSUB
0006          TYPE *,' Final values: J = ',J,'   K = ',K
0007          STOP 'Common test complete'
0008          END


PROGRAM SECTIONS

    Name                                     Bytes   Attributes

  0 $CODE                                     115    PIC CON REL LCL   SHR   EXE   RD NOWRT LONG
  1 $PDATA                                     85    PIC CON REL LCL   SHR NOEXE   RD NOWRT LONG
  2 $LOCAL                                     32    PIC CON REL LCL NOSHR NOEXE   RD   WRT LONG
  3 LOCAL_COMMON                             2048    PIC OVR REL GBL   SHR NOEXE   RD   WRT LONG
  4 GLOBAL_COMMON                            2052    PIC OVR REL GBL   SHR NOEXE   RD   WRT LONG

ENTRY POINTS

    Address   Type  Name

  0-00000000        MAIN

VARIABLES

    Address   Type  Name              Address   Type  Name

  3-00000000  I*4   J               4-00000000  I*4   K

ARRAYS

    Address   Type  Name                 Bytes  Dimensions

  3-00000004  I*4   J1                    2044  (511)
  4-00000004  I*4   K1                    2048  (512)
```

Figure 8-9 Listing of FORTRAN Program Using Shared COMMON

```
FUNCTIONS AND SUBROUTINES REFERENCED

SHRSUB


Total Space Allocated = 4332 Bytes


MAIN                                                        22-Feb-1980 14:26:58    VAX-11 FORTRAN T1.95-36          Page 2
                                                            29-Nov-1979 15:24:20    _DBB2:[GOLDMAN]MAIN.FOR;1

COMMAND QUALIFIERS

  FORTRAN /LIST MAIN

  /CHECK=(NOBOUNDS,OVERFLOW)
  /DEBUG=(NOSYMBOLS,TRACEBACK)
  /F77  /NOG_FLOATING  /I4  /OPTIMIZE  /WARNINGS  /NOD_LINES  /NOMACHINE_CODE  /CONTINUATIONS=19


COMPILATION STATISTICS

  Run Time:           0.59 seconds
  Elapsed Time:       3.82 seconds
  Page Faults:        316
  Dynamic Memory:     36 pages
```

Figure 8-9 (Cont.) Listing of FORTRAN Program Using Shared COMMON

```
$ LINK /EXE=MAIN /MAP=MAIN /FULL MAIN, SYS$INPUT:/OPTIONS
!
! Options input to link main program
!
SHRSUB/SHARE
PSECT=LOCAL_COMMON,NOSHR
```

Figure 8-10 LINK Command for FORTRAN Program Using Shared COMMON

```
MAIN                                                          22-FEB-1980 14:27       LINKER V02.40                 Page     1
                                             +------------------------+
                                             ! Object Module Synopsis !
                                             +------------------------+

Module Name     Ident              Bytes      File                                 Creation Date      Creator
-----------     -----              -----      -----                                -------------      -------
SHRSUB          .EXE;1                 0 _DBB2:[GOLDMAN]SHRSUB.EXE;1           22-FEB-1980 14:26  LINK-32 V02.40
MAIN            01                  4332 _DBB2:[GOLDMAN]MAIN.OBJ;1             22-Feb-1980 14:26  VAX-11 FORTRAN T1.95-36
OTS$LINKAGE     1-003                  3 _DRA5:[SYSLIB]STARLET.OLB;1           19-FEB-1980 21:57  VAX-11 Macro V02.41
SYSVECTOR       0219                   0 _DRA5:[SYSLIB]STARLET.OLB;1           19-FEB-1980 21:59  VAX-11 Macro V02.41



_DBB2:[GOLDMAN]MAIN.EXE;1                                     22-FEB-1980 14:27       LINKER V02.40                 Page     2
                                             +------------------------+
                                             ! Image Section Synopsis !
                                             +------------------------+

  Cluster       Type Pages   Base Addr Disk VBN PFC Protection and Paging      Global Sec. Name   Match      Majorid    Minorid
  -------       ---- -----   --------- -------- --- --------------------      ----------------   -----      -------    -------

SHRSUB             4     4    00000800        0   0 READ WRITE   COPY ON REF   SHRSUB_001       EQUAL            244    8260915
                   3     1    00001000        0   0 READ ONLY                  SHRSUB_002       EQUAL            244    8260915
                   3     4    00001200        0   0 READ WRITE                 SHRSUB_003       EQUAL            244    8260915
                   3     1    00001A00        0   0 READ ONLY                  SHRSUB_004       EQUAL            244    8260915
                   4     1    00001C00        5   0 READ WRITE   COPY ALWAYS

VMSRTL             3    11    00001E00        0   1 READ ONLY                  VMSRTL_001       LESS/EQUAL         1       2000
                   3   193    00003400        0   0 READ ONLY                  VMSRTL_002       LESS/EQUAL         1       2000
                   4     4    00018600        0   0 READ WRITE   COPY ON REF   VMSRTL_003       LESS/EQUAL         1       2000

DEFAULT_CLUSTER    0     1    00000200        2   0 READ ONLY
                   0     1    00000400        3   0 READ WRITE   COPY ON REF
                   0     1    00000602        4   0 READ ONLY
                 253    20    7FFFD800        0   0 READ WRITE DEMAND ZERO
```

Figure 8-11 Map of FORTRAN Program Using Shared COMMON

```
_DBB2:[GOLDMAN]MAIN.EXE;1                                   22-FEB-1980 14:27      LINKER V02.40                  Page    3
                                           +------------------------+
                                           ! Program Section Synopsis !
                                           +------------------------+

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

$PDATA                           00000200 00000254 00000055 (        85.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,NOEXE,  RD,NOWRT,NOVEC
                 MAIN            00000200 00000254 00000055 (        85.) LONG 2

$LOCAL                           00000400 0000041F 00000020 (        32.) LONG 2   PIC,USR,CON,REL,LCL,NOSHR,NOEXE,  RD,  WRT,NOVEC
                 MAIN            00000400 0000041F 00000020 (        32.) LONG 2

$CODE                            00000600 00000672 00000073 (       115.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT,NOVEC
                 MAIN            00000600 00000672 00000073 (       115.) LONG 2

_OTS$CODE                        00000674 00000676 00000003 (         3.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT,NOVEC
                 OTS$LINKAGE     00000674 00000676 00000003 (         3.) LONG 2

. BLANK .                        00000800 00000800 00000000 (         0.) BYTE 0 NOPIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,  WRT,NOVEC
                 OTS$LINKAGE     00000800 00000800 00000000 (         0.) BYTE 0
                 SYSVECTOR       00000800 00000800 00000000 (         0.) BYTE 0

LOCAL_COMMON                     00000800 00000FFF 00000800 (      2048.) LONG 2   PIC,USR,OVR,REL,GBL,NOSHR,NOEXE,  RD,  WRT,NOVEC
                 SHRSUB          00000800 00000800 00000000 (         0.) LONG 2
                 MAIN            00000800 00000FFF 00000800 (      2048.) LONG 2

GLOBAL_COMMON                    00001200 00001A03 00000804 (      2052.) LONG 2   PIC,USR,OVR,REL,GBL,  SHR,NOEXE,  RD,  WRT,NOVEC
                 SHRSUB          00001200 00001200 00000000 (         0.) LONG 2
                 MAIN            00001200 00001A03 00000804 (      2052.) LONG 2
```

Figure 8-11 (Cont.) Map of FORTRAN Program Using Shared COMMON

```
_DBB2:[GOLDMAN]MAIN.EXE;1                               22-FEB-1980 14:27       LINKER V02.40                 Page    4
                                                  +------------------+
                                                  ! Symbols By Name !
                                                  +------------------+

Symbol           Value            Symbol           Value            Symbol           Value            Symbol           Value
------           -----            ------           -----            ------           -----            ------           -----
BAS$$BLNK_LINE   000029A0-RU      BAS$INSTR        00002680-RU      BAS$SCRATCH      00002908-RU      FOR$$CB_PUSH     00002408-RU
BAS$$CB_GET      00002980-RU      BAS$IN_D_R       000027F0-RU      BAS$STATUS       00002938-RU      FOR$$CB_RET      00002418-RU
BAS$$CB_POP      00002970-RU      BAS$IN_F_R       000027E8-RU      BAS$STR_D        000026C0-RU      FOR$$ERRSNS_SAV  00002428-RU
    .                .                .                .                .                .                .                .
    .                .                .                .                .                .                .                .
    .                .                .                .                .                .                .                .




_DBB2:[GOLDMAN]MAIN.EXE;1                               22-FEB-1980 14:27       LINKER V02.40                 Page    7
                                                  +-------------------+
                                                  ! Symbols By Value !
                                                  +-------------------+

Value                                       Symbols...
-----                                       ----------
00000600      R-MAIN
00000674      R-BAS$LINKAGE       R-FOR$LINKAGE       R-OTS$LINKAGE
00001A00     RU-SHRSUB
   .             .
   .             .
   .             .


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

Figure 8-11 (Cont.) Map of FORTRAN Program Using Shared COMMON

```
_DBB2:[GOLDMAN]MAIN.EXE;1                                     22-FEB-1980 14:27       LINKER V02.40                  Page  15
                                                    +-----------------+
                                                    ! Image Synopsis !
                                                    +-----------------+

Virtual memory allocated:            00000200 00018DFF 00018C00 (113664. bytes, 222. pages)
Stack size:                                 20. pages
Image header virtual block limits:           1.         1. (     1. block)
Image binary virtual block limits:           2.         5. (     4. blocks)
Image name and identification:       MAIN 01
Number of files:                             4.
Number of modules:                           5.
Number of program sections:                 13.
Number of global symbols:                  434.
Number of image sections:                   13.
User transfer address:               00000600
Debugger transfer address:           80002168
Image type:                          EXECUTABLE.
Map format:                          FULL in file "_DBB2:[GOLDMAN]MAIN.MAP;1"
Estimated map length:                88. blocks
                                                  +----------------------+
                                                  ! Link Run Statistics !
                                                  +----------------------+

Performance Indicators                            Page Faults   CPU Time      Elapsed Time
----------------------                            -----------   --------      ------------
    Command processing:                                   24    00:00:00.11   00:00:01.09
    Pass 1:                                              302    00:00:01.30   00:00:05.02
    Allocation/Relocation:                                73    00:00:00.26   00:00:00.75
    Pass 2:                                               51    00:00:00.28   00:00:02.19
    Map data after object module synopsis:               163    00:00:02.09   00:00:05.19
    Symbol table output:                                  13    00:00:00.01   00:00:00.56
Total run values:                                        626    00:00:04.05   00:00:14.80

Using a working set limited to 300 pages and 50 pages of data storage (excluding image)

Total number object records read (both passes):   219
    of which 67 were in libraries and 4 were DEBUG data records containing 139 bytes
124 bytes of DEBUG data were written,starting at VBN 6 with 1 blocks allocated

Number of modules extracted explicitly            = 0
    with 2 extracted to resolve undefined symbols

0 library searches were for symbols not in the library searched

A total of 0 global symbol table records was written

/EXE=MAIN/MAP=MAIN/FULL MAIN,SYS$INPUT:/OPTIONS
```

Figure 8-11 (Cont.) Map of FORTRAN Program Using Shared COMMON

8.5 USING SHAREABLE IMAGES

To be of use, shareable images are normally linked into another image. Usually shareable images are installed by the system manager to make them available to the cooperating users at run time. Installation of shareable images is dealt with in the VAX/VMS System Manager's Guide.

You must use an options file (see Chapter 5) to specify a shareable image as input to the linker. In an options file the /SHAREABLE qualifier becomes a legal input file qualifier, identifying the associated file as a shareable image. The /SHAREABLE qualifier optionally accepts the keywords COPY or NOCOPY, specifying whether the linker is to create a private copy of the shareable image in the user image. The default value is that no copy is produced.

When an executable image is linked with a shareable image, the shareable image is assigned virtual address space in the executable image. But the linker does not copy the shareable image binary into the executable image file unless COPY is specified.

When an executable image that is linked with a shareable image is run, the image activator opens the shareable image and checks the global section match, as described in Section 8.2.3. If the match succeeds, the image activator maps the shareable image into the assigned virtual address space. One of two things happen 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. 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 it 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 was not available.

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

NOTE

The image activator assumes that both installed and uninstalled shareable images are in the directory defined by the logical name SYS$SHARE. If you want to use a shareable image that is in another directory, you must assign the file specification of the shareable image to the name of the shareable image. Note that you should assign the complete file specification, including the device and directory. For example:

```
$ ASSIGN DBA0:[TEST]SHRSUB.EXE SHRSUB
```

APPENDIX A

LINKER MESSAGES

This appendix lists the code and text portions of messages that the linker can issue. The messages are listed in alphabetical order by code.

The messages are designed to give you all the necessary information about the error. Brief explanations are included for a few messages that are not self-explanatory.

BADCCC, Module "[name]" has bad compilation completion code = [code]

BADIMGHDR, Bad shareable image header in file "[file-spec]"

BADPSC, Module "[name]" has transfer address in unknown P-section "[number]"

BASESYM, Base address symbol "[name]" is undefined or relocatable

CLOSERR, Close failure on "[file-spec]" code = %X[error code]

CONFMEM, Conflicting virtual memory requirement at %X[address] for [number of] pages for cluster "[name]"

CRE8ERR, Failed to create file "[file-spec]"

CRFERR, Error code %X[error code] received from Cross Reference Facility

DBGTFR, Image "[file-spec]" has no Debugger transfer address

DIAGSISUED, Completed but with diagnostics

EMPTYFILE, File "[file-spec]" contains no modules

ENDPRS, Parameter parse completion error, code = %X[error code]

EOMFTL, Module "[name]" specifies Linker abort

EOMSTK, Module "[name]" leaves [number of] items on Linker internal stack

ERRORS, Module "[name]" has compilation errors - image deleted

EXCPSC, Module "[name]" defines more than 256 P-sections

EXCSPAR, Too many parameters in option: [option name] of file "[file-spec]"

FAOBUG, FAO failure

FATALERROR, Fatal error message issued

FIRSTMOD, First input being a library requires module extraction

FORMAT, File "[file-spec]" has illegal format

GSDTYP, File "[file-spec]" has an illegal GSD record (type = [type code])

ILLFMLCNT, Min. arg. count of [number] exceeds max. ([number]) in formal spec. of "[routine name]"

ILLKEY, Unrecognized keyword in parameter of option file "[file-spec]"

ILLQUALVAL, Illegal qualifier value

ILLREP, Module "[name]" has store repeated count [number] greater than [number]

ILLTIR, Module "[name]" has illegal relocation command = [number]

ILLVAL, Illegal parameter value in option file "[file-spec]"

ILLVPS, Module "[name]" contains illegal position ([number]) or size ([number]) in TIR$C_STO_VPS command

INITPRS, Parameter parse initialization error, code = %X[error code]

INSVIRMEM, Insufficient virtual memory for [number of] pages for cluster "[name]"

INTSTKOV, Linker internal stack of [number of] items overflowed by module "[name]"

INTSTKUN, Linker internal stack of [number of] items underflows in module "[name]"

IVCHAR, Invalid character in parameter - option file "[file-spec]"

LIBFIND, Failed to find valid lib. mod. or shr. image STB. at RFA %X[address] %X[address]

LIBFMT, Library "[name]" (format = [bad format]) has incorrect format (not =[correct format]) for this Linker

- Might be caused by a corrupt library or an attempt to use an RSX-11M library.

LIBNAMLNG, Library module name length ([number of characters]) is illegal

LINERR, Command line segment in error

\[error]\

MATCHID, Global section match ident ([number]) exceeds maximum ([number])

MAXCHANS, [number of] channels exceeds maximum allowed of 64

MAXIOSEG, [number of] I/O segment pages exceeds maximum allowed of 65535

MAXISDS, [number of] I-sections exceeds maximum allowed of 65535

MAXPFC, Page fault cluster factor of [number] exceeds maximum (255)

MAXSTACK, [number of] stack pages exceeds maximum allowed of 65535

MEMBUG, Memory (de)allocation bug [description] %X[address]

- Internal linker error

MEMFUL, Linker virtual address space insufficient to complete this link

MINDZRO, [number of pages] as minimum I-section size exceeds maximum allowed of 65535

- DZRO_MIN option value too high

MODNAM, Illegal module name of [number of] chars. - not 1 to [number of] chars.

MSGERR, Linker has error message bug [hex data]

MULDEF, Symbol "[name]" multiply defined by module "[name]"

- The named module defines a symbol that another module has already defined.

MULPSC, Module "[name]" has conflicting specifications for P-section "[name]"

- A previously encountered module has already defined the program section with other attributes.

MULTFR, Module "[name]" multiply defines transfer address

- The named module defines the image transfer address (starting point), but a previously processed module has already defined the transfer address.

SPNAMLNG, Illegal symbol/P-section name of [number of] chars. - not 1 to [number of] chars.

NOEOM, Module "[name]" not terminated with EOM record

NOEPM, Module "[name]" references undefined entry mask of symbol "[name]"

NONBTAB, Non blank/tab between continuation and comment or end of record in "[file-spec]"

NOMODS, No input modules specified (or found)

NOPSCTS, No P-sections defined in module "[name]"

NOSUCHMOD, Library "[name]" does not contain module "[name]"

NOTPSECT, Module "[name]" sets relocation base to other than a P-section base

NOVALU Values not allowed in qualifier - option file "[file-spec]"

NUDFSYMS, "[number]" undefined symbol(s)

NULFIL, Null parameter in option file "[file-spec]"

NULPAR, Missing required parameter in option line [erroneous line] of file "[file-spec]"

OPIDERR, Pass [number] failed to open file "[file-spec]"

OPTREDERR, Read error (code=%X[error code]) on option file "[file-spec]"

OUTSIMG, Attempted store location %X[address] is outside "[region]" (%X[base address] to %X[ending address])

- "Region" is expressed as either "image binary" or "Debug Symbol Table."

OVRALI, Module "[name]" has conflicting alignment on overlayed P-section "[name]"

PARMDEL, Invalid parameter delimiter in option file "[file-spec]"

PRIMIN, Input parameter parse error, code = %X[error code]

PRIMOUT, Image file specification error, code = %X[error code]

PSCALI, Illegal P-section alignment [number of bytes] - exceeds a page

PSCNXR, Transfer address in "[module-name]" not in EXE/REL P-section

- The transfer address is normally in a program section with the executable and relocatable attributes.

PSCOVFLO, P-section "[name]" overflows region to %X[address]

RECLNG, File "[file-spec]" contains record of illegal length ([number of] bytes)

RECTYP, File "[file-spec]" has an illegal record (type = [type code])

REDERR, Read failure in pass [number] on file "[file-spec]"

SECOUT, Map file specification error, code = %X[error code]

SEQNCE, Illegal record sequence

SHRINSYS, Shareable image(s) cannot be linked into a system image

STRLVL, LINK [version] does not implement OBJ level [structure level] - only to [structure level]

- The version of the object language is not compatible with the current version of the linker.

STKOVFLO, Stack of [number of] pages falls below control region to %X[address]

TFRSYS, Transfer address in system image "[file-spec]" ignored

TIRLNG, Module "[name]" has relocation command data ([number of] bytes) overflowing record

TIRNYI, TIR command [number or name] not yet implemented (module "[name]")

TRACIGN, Suppression of traceback overridden by DEBUG specification

- Occurs when you specify /NOTRACEBACK and /DEBUG.

TRIOUT, Symbol table file specification error, code = %X[error code]

TRUNC, Trunc. error in module "[name]", P-section "[name]", offset %X[hex value]

TRUNCDAT, Computed value = %X[hex value], value written = %X[hex value] at %X[address]

UDEFPSC, Attempt to reference P-section no. [number] undefined in "[module name]"

- Undefined program section

UDFSYM, "[symbol name]"

- Undefined symbol

UNMCOD, Initial file name was "[file-spec]", RMS error code = %X[error code]

UNRECOPT, Unrecognized option in file "[file-spec]"

UNRECQUAL, Unrecognized qualifier in option file "[file-spec]"

USEUNDEF, Module "[name]" references undefined symbol "[name]"

USRTFR, Image "[file-spec]" has no user transfer address

WRNERS, Module "[name]" has compilation warnings

WRTERR, Write failure on file "[file-spec]", code = %X[error code]

VALREQ, Value required in qualifier - option file "[file-spec]"

APPENDIX B

IMAGE MAP ILLUSTRATIONS

This appendix illustrates the brief, default, and full forms of a map of the same image.

In addition, after the full form of the map, a Symbol Cross Reference map section is illustrated.

The image map is described in Chapter 6.

BRIEF MAP

```
DEMO                                                    26-FEB-1980 10:17       LINKER V02.40                   Page  1
                                      +-------------------------+
                                      | Object Module Synopsis |
                                      +-------------------------+

Module Name     Ident               Bytes     File                                Creation Date      Creator
-----------     -----               -----     -----                               -------------      -------
DEMO            01                    388 _DBB2:[GOLDMAN]DEMO.OBJ;2           26-Feb-1980 10:16  VAX-11 FORTRAN T1.95-36
SUB1            01                    179 _DBB2:[GOLDMAN]SUB1.OBJ;2           26-Feb-1980 10:15  VAX-11 FORTRAN T1.95-36
FUN1            01                     23 _DBB2:[GOLDMAN]FUN1.OBJ;1           26-Feb-1980 10:15  VAX-11 FORTRAN T1.95-36
FUN2            01                     23 _DBB2:[GOLDMAN]FUN2.OBJ;1           26-Feb-1980 10:15  VAX-11 FORTRAN T1.95-36
```

```
_DBB2:[GOLDMAN]DEMO.EXE;1                                    26-FEB-1980 10:17       LINKER V02.40             Page     2
                                                     +-----------------+
                                                     ! Image Synopsis !
                                                     +-----------------+

Virtual memory allocated:               00000200 0001A7FF 0001A600 (108032. bytes, 211. 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:          DEMO 01
Number of files:                                   6.
Number of modules:                                 6.
Number of program sections:                        8.
Number of global symbols:                        276.
Number of image sections:                          8.
User transfer address:                  00000600
Debugger transfer address:              80000168
Image type:                             EXECUTABLE.
Map format:                             BRIEF in file "_DBB2:[GOLDMAN]DEMOBR.MAP;2"
Estimated map length:                   8. blocks
                                                  +---------------------+
                                                  ! Link Run Statistics !
                                                  +---------------------+

Performance Indicators                                 Page Faults   CPU Time        Elapsed Time
----------------------                                 -----------   --------        ------------
    Command processing:                                         17   00:00:00.18     00:00:08.69
    Pass 1:                                                    195   00:00:00.95     00:00:48.31
    Allocation/Relocation:                                      25   00:00:00.11     00:00:06.44
    Pass 2:                                                     33   00:00:00.30     00:00:12.98
    Map data after object module synopsis:                       0   00:00:00.00     00:00:00.00
    Symbol table output:                                         4   00:00:00.02     00:00:03.54
Total run values:                                              274   00:00:01.56     00:01:19.96

Using a working set limited to 242 pages and 49 pages of data storage (excluding image)

Total number object records read (both passes):   183
    of which 51 were in libraries and 8 were DEBUG data records containing 189 bytes
169 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

0 library searches were for symbols not in the library searched

A total of 0 global symbol table records was written

/MAP=DEMOBR/BRIEF DEMO,SUB1,FUN1,FUN2
```

BRIEF MAP

```
DEMO                                                   26-FEB-1980 10:18       LINKER V02.40                Page    1
                                          +-----------------------+
                                          ! Object Module Synopsis !
                                          +-----------------------+

Module Name      Ident           Bytes     File                               Creation Date      Creator
-----------      -----           -----     ----                               -------------      -------
DEMO             01                388 _DBB2:[GOLDMAN]DEMO.OBJ;2           26-Feb-1980 10:16  VAX-11 FORTRAN T1.95-36
SUB1             01                179 _DBB2:[GOLDMAN]SUB1.OBJ;2           26-Feb-1980 10:15  VAX-11 FORTRAN T1.95-36
FUN1             01                 23 _DBB2:[GOLDMAN]FUN1.OBJ;1           26-Feb-1980 10:15  VAX-11 FORTRAN T1.95-36
FUN2             01                 23 _DBB2:[GOLDMAN]FUN2.OBJ;1           26-Feb-1980 10:15  VAX-11 FORTRAN T1.95-36

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

Psect Name       Module Name       Base     End           Length            Align                 Attributes
----------       -----------       ----     ---           ------            -----                 ----------
$PDATA                           00000200 0000020F 00000010 (        16.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,NOEXE,  RD,NOWRT,NOVEC
                 DEMO            00000200 0000020F 00000010 (        16.) LONG 2
$LOCAL                           00000400 00000588 0000018C (       396.) LONG 2   PIC,USR,CON,REL,LCL,NOSHR,NOEXE,  RD,  WRT,NOVEC
                 DEMO            00000400 00000533 00000134 (       308.) LONG 2
                 SUB1            00000534 00000583 00000050 (        80.) LONG 2
                 FUN1            00000584 00000587 00000004 (         4.) LONG 2
                 FUN2            00000588 0000058B 00000004 (         4.) LONG 2
$CODE                            00000600 000006CA 000000CB (       203.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT,NOVEC
                 DEMO            00000600 0000063F 00000040 (        64.) LONG 2
                 SUB1            00000640 000006A2 00000063 (        99.) LONG 2
                 FUN1            000006A4 000006B6 00000013 (        19.) LONG 2
                 FUN2            000006B8 000006CA 00000013 (        19.) LONG 2

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

Symbol           Value           Symbol            Value           Symbol            Value           Symbol            Value
------           -----           ------            -----           ------            -----           ------            -----
DEMO             00000600-R
FUN1             000006A4-R
FUN2             000006B8-R
SUB1             00000640-R


          Key for special characters above:
                +-------------------+
                ! *   - Undefined   !
                ! U   - Universal   !
                ! R   - Relocatable !
                ! WK  - Weak        !
                +-------------------+
_DBB2:[GOLDMAN]DEMO.EXE;2                                    26-FEB-1980 10:18       LINKER V02.40                  Page    2
                                                       +-----------------+
                                                       ! Image Synopsis !
                                                       +-----------------+

Virtual memory allocated:                 00000200 0001A7FF 0001A600 (108032. bytes, 211. 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:            DEMO 01
Number of files:                                 6.
Number of modules:                               6.
Number of program sections:                      8.
Number of global symbols:                      276.
Number of image sections:                        8.
User transfer address:                    00000600
Debugger transfer address:                80000168
Image type:                               EXECUTABLE.
Map format:                               DEFAULT in file "_DBB2:[GOLDMAN]DEMODEF.MAP;2"
Estimated map length:                     33. blocks
                                                    +----------------------+
                                                    ! Link Run Statistics !
                                                    +----------------------+

Performance Indicators                                 Page Faults   CPU Time        Elapsed Time
----------------------                                 -----------   --------        ------------
    Command processing:                                         21   00:00:00.09     00:00:06.84
    Pass 1:                                                    206   00:00:01.02     00:00:50.70
    Allocation/Relocation:                                      15   00:00:00.08     00:00:04.70
    Pass 2:                                                     30   00:00:00.35     00:00:20.22
    Map data after object module synopsis:                      11   00:00:00.07     00:00:03.82
    Symbol table output:                                         7   00:00:00.02     00:00:01.20
Total run values:                                              290   00:00:01.63     00:01:27.48

Using a working set limited to 242 pages and 49 pages of data storage (excluding image)

Total number object records read (both passes):   183
    of which 51 were in libraries and 8 were DEBUG data records containing 189 bytes
169 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

0 library searches were for symbols not in the library searched

A total of 0 global symbol table records was written

/MAP=DEMODEF DEMO,SUB1,FUN1,FUN2
```

FULL MAP

```
DEMO                                          26-FEB-1980 10:20       LINKER V02.40             Page  1
                                        +------------------------+
                                        ! Object Module Synopsis !
                                        +------------------------+

Module Name     Ident           Bytes    File                                  Creation Date       Creator
-----------     -----           -----    ----                                  -------------       -------
DEMO            01                388 _DBB2:[GOLDMAN]DEMO.OBJ;2             26-Feb-1980 10:16  VAX-11 FORTRAN T1.95-36
SUB1            01                179 _DBB2:[GOLDMAN]SUB1.OBJ;2             26-Feb-1980 10:15  VAX-11 FORTRAN T1.95-36
FUN1            01                 23 _DBB2:[GOLDMAN]FUN1.OBJ;1             26-Feb-1980 10:15  VAX-11 FORTRAN T1.95-36
FUN2            01                 23 _DBB2:[GOLDMAN]FUN2.OBJ;1             26-Feb-1980 10:15  VAX-11 FORTRAN T1.95-36
SYSVECTOR       0219                0 _DRA5:[SYSLIB]STARLET.OLB;1           19-FEB-1980 21:59  VAX-11 Macro V02.41
VMSRTL          .EXE;1              0 _DRA5:[SYSLIB]VMSRTL.EXE;1            20-FEB-1980 18:55  LINK-32 V02.39
```

FULL MAP

```
_DB82:[GOLDMAN]DEMO.EXE;3                                    26-FEB-1980 10:20       LINKER V02.40                 Page    2
                                              +-----------------------+
                                              ! Image Section Synopsis !
                                              +-----------------------+

  Cluster          Type Pages    Base Addr  Disk VBN PFC Protection and Paging       Global Sec. Name    Match      Majorid    Minorid
  -------          ---- -----    ---------  -------- --- ---------------------       ----------------    -----      -------    -------

DEFAULT_CLUSTER       0     1     00000200         2   0 READ ONLY
                      0     1     00000400         3   0 READ WRITE    COPY ON REF
                      0     1     00000600         4   0 READ ONLY
                    253    20     7FFFD800         0   0 READ WRITE DEMAND ZERO

VMSRTL                3    11     00000800         0   1 READ ONLY                   VMSRTL_001       LESS/EQUAL        1       2000
                      3   193     00001E00         0   0 READ ONLY                   VMSRTL_002       LESS/EQUAL        1       2000
                      4     4     0001A000         0   0 READ WRITE    COPY ON REF   VMSRTL_003       LESS/EQUAL        1       2000
```

```
_DBB2:[GOLDMAN]DEMO.EXE;3                                  26-FEB-1980 10:20       LINKER V02.40                Page    3
                                             +--------------------------+
                                             ! Program Section Synopsis !
                                             +--------------------------+

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

$PDATA                          00000200 0000020F 00000010 (         16.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,NOEXE,   RD,NOWRT,NOVEC
                DEMO            00000200 0000020F 00000010 (         16.) LONG 2
                SUB1            00000210 00000210 00000000 (          0.) LONG 2
                FUN1            00000210 00000210 00000000 (          0.) LONG 2
                FUN2            00000210 00000210 00000000 (          0.) LONG 2

$LOCAL                          00000400 00000588 0000018C (        396.) LONG 2   PIC,USR,CON,REL,LCL,NOSHR,NOEXE,   RD,  WRT,NOVEC
                DEMO            00000400 00000533 00000134 (        308.) LONG 2
                SUB1            00000534 00000583 00000050 (         80.) LONG 2
                FUN1            00000584 00000587 00000004 (          4.) LONG 2
                FUN2            00000588 0000058B 00000004 (          4.) LONG 2

$CODE                           00000600 000006CA 000000CB (        203.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,   RD,NOWRT,NOVEC
                DEMO            00000600 0000063F 00000040 (         64.) LONG 2
                SUB1            00000640 000006A2 00000063 (         99.) LONG 2
                FUN1            000006A4 000006B6 00000013 (         19.) LONG 2
                FUN2            000006B8 000006CA 00000013 (         19.) LONG 2

. BLANK .                       00000800 00000800 00000000 (          0.) BYTE 0 NOPIC,USR,CON,REL,LCL,NOSHR,  EXE,   RD,  WRT,NOVEC
                SYSVECTOR       00000800 00000800 00000000 (          0.) BYTE 0
```

```
_DBB2:[GOLDMAN]DEMO.EXE;3                                    26-FEB-1980 10:20        LINKER V02.40                 Page    4
                                            +-----------------+
                                            ! Symbols By Name !
                                            +-----------------+

Symbol            Value              Symbol            Value              Symbol            Value              Symbol            Value
------            -----              ------            -----              ------            -----              ------            -----
BAS$$BLNK_LINE    000013A0-RU        BAS$INSTR         000010B0-RU        BAS$STATUS        00001338-RU        FOR$$CB_PUSH      00000E08-RU
BAS$$CB_GET       00001380-RU        BAS$IN_D_R        000011F0-RU        BAS$STR_D         000010C0-RU        FOR$$CB_RET       00000E18-RU
BAS$$CB_POP       00001370-RU        BAS$IN_F_R        000011E8-RU        BAS$STR_F         000010B8-RU        FOR$$ERRSNS_SAV   00000E28-RU
    .                 .                  .                 .                  .                 .                  .                 .
    .                 .                  .                 .                  .                 .                  .                 .
    .                 .                  .                 .                  .                 .                  .                 .
BAS$INIT_R8       00001118-RU        BAS$RSET_R        000010A0-RU        DEMO              00000600-R         FOR$IO_B_R        000008E0-RU
BAS$INPUT         000011A0-RU        BAS$SCALE_D_R1    00001078-RU        FOR$$CB_GET       00000E20-RU        FOR$IO_B_V        000008E8-RU
BAS$INPUT_LINE    00001180-RU        BAS$SCRATCH       00001308-RU        FOR$$CB_POP       00000E10-RU        FOR$IO_DC_R       00000928-RU
```

FULL MAP

FULL MAP

```
_DBB2:[GOLDMAN]DEMO.EXE;3                                26-FEB-1980 10:20      LINKER V02.40                 Page    5

Symbol          Value               Symbol          Value               Symbol          Value               Symbol          Value
------          -----               ------          -----               ------          -----               ------          -----
FOR$IO_DC_V     00000E30-RU         FUN1            000006A4-R          MTH$ASIN_R5     00000AA0-RU         OTS$CVT_L_TZ    00000EA0-RU
FOR$IO_D_R      000008C0-RU         FUN2            00000688-R          MTH$ATAN        00000AA8-RU         OTS$CVT_TI_L    00000A10-RU
    .               .                   .               .                   .               .                   .               .
    .               .                   .               .                   .               .                   .               .
    .               .                   .               .                   .               .                   .               .
    .               .
    .               .
    .               .
SUB1            00000640-R
SYS$IMGSTA      80000168
```

```
_DBB2:[GOLDMAN]DEMO.EXE;3                              26-FEB-1980 10:20       LINKER V02.40          Page    7
                                               +------------------+
                                               ! Symbols By Value !
                                               +------------------+

Value                                    Symbols...
-----                                    ----------
00000600     R-DEMO
00000640     R-SUB1
000006A4     R-FUN1
000006B8     R-FUN2
00000800    RU-FOR$CLOSE
00000808    RU-FOR$DECODE_MF
    .                  .
    .                  .
    .                  .
80000168       SYS$IMGSTA


          Key for special characters above:
                +------------------+
                ! *  - Undefined   !
                ! U  - Universal   !
                ! R  - Relocatable !
                ! WK - Weak        !
                +------------------+
_DBB2:[GOLDMAN]DEMO.EXE;3                                          26-FEB-1980 10:20      LINKER V02.40              Page   15
                                                    +----------------+
                                                    ! Image Synopsis !
                                                    +----------------+

Virtual memory allocated:                  00000200 0001A7FF 0001A600 (108032. bytes, 211. 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:             DEMO 01
Number of files:                                 6.
Number of modules:                               6.
Number of program sections:                      8.
Number of global symbols:                      276.
Number of image sections:                        8.
User transfer address:                     00000600
Debugger transfer address:                 80000168
Image type:                                EXECUTABLE.
Map format:                                FULL in file "_DBB2:[GOLDMAN]DEMOFULL.MAP;2"
Estimated map length:                      58. blocks
                                                 +---------------------+
                                                 ! Link Run Statistics !
                                                 +---------------------+

Performance Indicators                                   Page Faults   CPU Time        Elapsed Time
----------------------                                   -----------   --------        ------------
    Command processing:                                          17    00:00:00.15     00:00:05.07
    Pass 1:                                                     259    00:00:01.15     00:00:27.61
    Allocation/Relocation:                                       72    00:00:00.30     00:00:07.65
    Pass 2:                                                      69    00:00:00.31     00:00:13.22
    Map data after object module synopsis:                       91    00:00:01.98     00:00:33.31
    Symbol table output:                                          8    00:00:00.03     00:00:04.07
Total run values:                                               516    00:00:03.92     00:01:30.93

Using a working set limited to 237 pages and 49 pages of data storage (excluding image)

Total number object records read (both passes):   183
    of which 51 were in libraries and 8 were DEBUG data records containing 189 bytes
169 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

0 library searches were for symbols not in the library searched

A total of 0 global symbol table records was written

/MAP=DEMOFULL/FULL DEMO,SUB1,FUN1,FUN2
```

SYMBOL CROSS REFERENCE

```
_DBB2:[GOLDMAN]DEMO.EXE;4                                  26-FEB-1980 13:31      LINKER V02.40          Page    4
                                               +-------------------------+
                                               ! Symbol Cross Reference !
                                               +-------------------------+

Symbol           Value          Defined By     Referenced By ...
------           -----          ----------     -----------------
BAS$$BLNK_LINE   000013A0-RU
BAS$$CB_GET      00001380-RU
     .                .
     .                .
     .                .
DEMO             00000600-R      DEMO
FOR$$CB_GET      00000E20-RU     VMSRTL
FOR$$CB_POP      00000E10-RU     VMSRTL
     .                .
     .                .
     .                .
FUN1             000006A4-R      FUN1            SUB1
FUN2             000006B8-R      FUN2            SUB1
     .                .
     .                .
     .                .
SUB1             00000640-R      SUB1            DEMO
SYS$IMGSTA       80000168        SYSVECTOR
```

APPENDIX C

VAX-11 OBJECT LANGUAGE

The object language description in this appendix is taken from DIGITAL software specifications. This appendix is useful for anyone writing a compiler or assembler that must generate object modules acceptable for input to the VAX-11 Linker. The object module analysis program (ANALYZE), discussed in Appendix D, checks an object module to see if it conforms to the requirements in the DIGITAL software specifications.

C.1 INTRODUCTION

The VAX-11 object language is accepted by VAX-11 linkers, object module librarians, and object patch utilities.

The object language described herein is for use by all VAX-11 family software -- that is, no subsetting will occur. All language processors that produce code for execution in native mode are free to use any or all of the described object language.

C.1.1 Summary of Language

Object modules are the input to the linker and are obtained from the various language processors as individual files or as object library files. All symbol table files created by the linker are also in the format specified here.

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

OBJ$C_HDR = 0	- Header Record (HDR)
OBJ$C_GSD = 1	- Global Symbol Directory Record (GSD)
OBJ$C_TIR = 2	- Text Information and Relocation Record (TIR)
OBJ$C_EOM = 3	- End of Module Record (EOM)
OBJ$C_DBG = 4	- Debugger Information Record (DBG)
OBJ$C_TBT = 5	- Traceback Information Record (TBT)
OBJ$C_LNK = 6	- Link Option Specification Record (LNK) (Currently ignored)

Refer to Figure C-1 for an illustration of the order in which records appear in the object module.

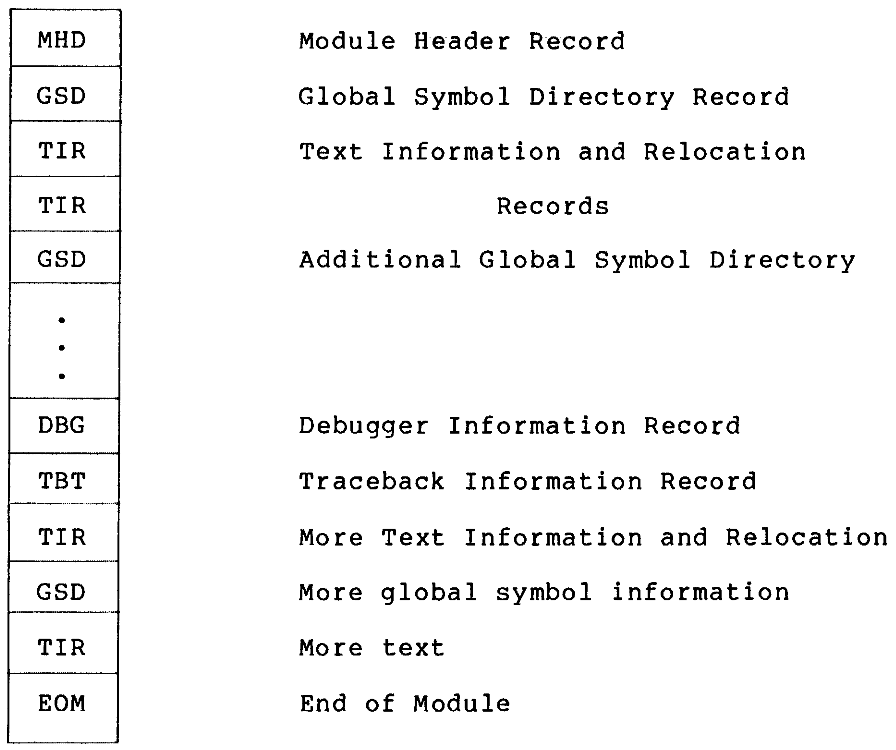
MHD
Module Header Record
GSD
Global Symbol Directory Record
TIR
TIR
Text Information and Relocation Records
GSD
Additional Global Symbol Directory
DBG
Debugger Information Record
TBT
Traceback Information Record
TIR
More Text Information and Relocation
GSD
More global symbol information
TIR
More text
EOM
End of Module

Figure C-1 Order of Records Within an Object Module

It is mandatory that there be at least two HDR records, a Module Header Record (MHD) and a Language Name Record (LNM), and exactly one EOM Record. These records must begin and end the module, respectively. Within the module, there must be at least one GSD record and there may be any number of TIR, DBG, TBT and LNK records. As is described below, some ordering is implicit within the set of GSD records.

In this document, the term "reserved" implies 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 unused and ignored fields of records must be padded to conform to the block lengths specified in this document. The content of such fields will be completely ignored by the linker and any other processors.

The remaining possible language record types are allocated as follows but not defined in this document:

Type 7-100	Reserved for future use by linker
Type 101-200	Ignored always
Type 201-255	Reserved for customer use (Ignored by current implementation)

C.2 GLOBAL AND UNIVERSAL SYMBOLS AND NAME FORMAT

The linker deals with two types of symbols, global and universal.

Global symbols are those that are accessible to more than one module of the set being linked. Universal symbols are a subset of the global symbols. They are ones that the linker retains when linking an image to which another set of object modules and/or images will subsequently be bound.

The Object Language also deals with the names of program sections and object modules and may contain the names of language processors and utilities.

All names are represented by a 1-byte character count followed by the ASCII character string.

The current implementation of the linker limits such name strings to 31 characters, except in the case of header record types 1-255 (see Section C.3.2).

C.3 HEADER RECORDS (HDR)

The object language defines a general class of header records. The Module Header Record (MHD) is described in Section C.3.1.

The Language Name Record and the remaining header types are described in Section C.3.2.

C.3.1 Module Header Records (MHD)

The module header records (MHD) collect in one place all module-wide information. The module header records are needed by the librarian and patch utilities and permit future expansion of the object language.

The MHD (Module Header Record) record contains the following information in the format shown:

RECORD TYPE 0	1 byte
HEADER TYPE 0	1 byte
STRUCTURE LEVEL	1 byte
MAXIMUM RECORD SIZE	2 bytes
MODULE NAME	Variable (2-32)
MODULE VERSION	Variable (2-32)
CREATION TIME AND DATE	17 bytes
TIME AND DATE OF LAST PATCH	17 bytes

All entries are required. Detailed descriptions of the fields follow.

C.3.1.1 **Header Type** - The MHD header type is 0 (OBJ$C_HDR_MHD).

C.3.1.2 **Structure Level OBJ$C_STRLVL** - It is intended that the format of the MHD record remain fixed from first implementation onward. The structure level is provided such that extensions to the language, which require changes to other record formats, can be dealt with without requiring recompilation of every module that conforms to the previous format. The structure level is zero.

C.3.1.3 **Maximum Record Size OBJ$C_MAXRECSIZ** - The size in bytes of the longest record that can occur within this object module. The maximum size is 2048 bytes.

C.3.1.4 **Module Name** - The module name conforms to the format of all other names, that is, length contained in a byte followed by an ASCII string. If the module is a symbol table created by the linker, the name will be the image name assigned at link time.

C.3.1.5 **Module Version** - The module version conforms to the format of all names in the object language.

C.3.1.6 **Dates and Times** - There are two date and time fields, one for module creation and one for the last modification to the module (by an object module patch utility). The format is a fixed 17-character ASCII string:

```
dd-mmm-yyyy hh:mm
```

where:

dd = day of month

mmm = standard 3-character abbreviation of month

yyyy = year; note the space that follows

hh = hour of day 00 to 23

mm = minute of hour 00 to 59

C.3.2 **Other Header Records**

The purpose of subheader records is primarily to contain optional textual information in printable form. Each record consists of a byte which is zero to indicate a header record, followed by a subtype byte. The following subtypes are defined.

OBJ$C_HDR_LNM = 1	Language Processor (LNM) Name and Version. One record is required and limited for the current implementation to 35 characters. The content of this record appears on the link map output.
OBJ$C_HDR_SRC = 2	List of file-specifications for the source files from which object module was created. Multiple records are permitted. (Currently ignored)
OBJ$C_HDR_TTL = 3	Title text (e.g., brief module description). Only one record permitted. (Currently ignored)
OBJ$C_HDR_CPR = 4	A copyright statement. Only one record permitted. (Currently ignored)
OBJ$C_HDR_MTC = 5	Maintenance Status. (MTC) Multiple records permitted. (Currently ignored)
OBJ$C_HDR_GTX = 6	General Text. Multiple records permitted. (Currently ignored)

Types 7-100 are reserved.

Types 101-255 are always ignored.

C.3.2.1 **Header Types 1 through 4 and 6** - The purpose of these records is to allow the language processors to provide printable information within the object modules for documentation purposes. The only format definition is that the record contain printing ASCII characters. Types 4 and 6 may be generated by users, whereas types 1 through 3 are restricted to the language processors.

C.3.2.2 **Maintenance Status Header Record (MTC)** - This record is of concern only to the object module patch utility and the object module analysis (ANALYZE) utility (see Appendix D). It is ignored by the librarian and the linker.

The format is as follows:

RECORD TYPE 0	1 byte
HEADER TYPE 5	1 byte
PATCH UTILITY NAME	variable 2-32 bytes
UTILITY VERSION	variable 2-32 bytes
UIC	2 bytes
INPUT FILE SPECIFICATION	variable 2-42 bytes
CORRECTION FILE SPECIFICATION	variable 2-42 bytes
DATE + TIME	17 bytes
SEQUENTIAL PATCH	1 byte

C.3.2.2.1 **Record Type** - Zero signifies a header record.

C.3.2.2.2 **Header Type** - The type is 5 signifying a maintenance status record.

C.3.2.2.3 **Patch Utility Name** - This name identifies the patch utility used to perform this patch on the module. This field begins with one byte containing the number of bytes in the field (not including the count byte itself).

C.3.2.2.4 **Utility Version** - The patch utility is further identified by its version number. This field begins with one byte containing the number of bytes in the field (not including the count byte itself).

C.3.2.2.5 **U.I.C.** - This is the user identification code under which the patch was made.

C.3.2.2.6 **Input File Specification** - This filename identifies the input file for this patch. This field begins with one byte containing the number of bytes in the field (not including the count byte itself).

C.3.2.2.7 **Correction File Specification** - This filename identifies the correction file for this patch. This field begins with one byte containing the number of bytes in the field (not including the count byte itself).

C.3.2.2.8 **Date & Time** - This 17-byte field contains the date and time that this patch was performed. Format is as described above.

C.3.2.2.9 **Sequential Patch Number** - This number is a sequential count of the patches made to this module.

C.4 GLOBAL SYMBOL DIRECTORY (GSD) RECORDS (OBJ$C_GSD)

Global symbol directory records contain all the information necessary to allocate virtual address space and to combine all the program sections into the separately protectable sections (image sections) of the image being created.

GSD records are of the following types:

OBJ$C_GSD_PSC = 0	Program section definition.
OBJ$C_GSD_SYM = 1	Global Symbol Specification.
OBJ$C_GSD_EPM = 2	Entry Point Symbol and Mask Definition.
OBJ$C_GSD_PRO = 3	Procedure and Formal Argument Definition.

Within any GSD record, there may be many entry types. In such cases, a single record appears as the concatenation of many, with the omission of the byte containing the Object Language record type (the value OBJ$C_GSD).

C.4.1 Program Section Definition (OBJ$C_GSD_PSC)

The format of a program section definition is as follows:

RECORD TYPE 1	1 byte
GSD TYPE 0	1 byte
ALIGNMENT	1 byte
FLAGS	2 bytes
ALLOCATION	4 bytes
PROGRAM SECTION NAME	Variable 2-32 bytes

C.4.1.1 **Program Section Name** - This name has the same format as all other symbol names.

C.4.1.2 **Alignment** - This field specifies the virtual address boundary at which the program section will be placed. The alignment is 2 to the power specified in the field.

Value	**Alignment**
0	1 (BYTE)
1	2 (WORD)
2	4 (LONGWORD)
3	8 (QUADWORD)
4	2**4
.	.
.	.
.	.
9	2**9 (PAGE)

Nine indicates page alignment and is the limit for program section alignment.

Each module contributing to a program section can specify its own local alignment, with the restriction that program sections whose contributions overlay each other must all have the same alignment. It should also be noted that an alignment specified within a program section (for example, the assembler .ALIGN directive) must be less than or equal to the program section alignment to be guaranteed. For example, byte alignment of the program section may or may not cause longword aligned elements within the program section.

C.4.1.3 **Flags** - The flag bits in the program section definition have the following meaning:

Bit	**Name**	**Meaning If Set**
0	GPS$V_PIC	Program section defined as position independent.
1	GPS$V_LIB	The program section was defined in the symbol table of a shareable image, to which this image is bound.
2	GPS$V_OVL	Contributions to the same program section are overlaid. (The complement is concatenation).
3	GPS$V_REL	Program section requires relocation (the complement, bit=0, means absolute and contains only symbol definitions. Thus the allocation of an absolute program section is zero).
4	GPS$V_GBL	The scope of program section is global. (The complement is local).
5	GPS$V_SHR	Program section is potentially shareable between two or more active processes.
6	GPS$V_EXE	Content of the program section is executable.

Bit	Name	Meaning If Set
7	GPS$V_RD	Content of the program section may be read.
8	GPS$V_WRT	Content of the program section may be written.
9	GPS$V_VEC	Program section contains change mode dispatch vectors.
10-15		Reserved.

Discussions of program section attributes may be found in the related documents. (See also Section 7.5.4 of this manual.)

C.4.1.4 **Allocation Field** - The allocation field contains the length contribution to the program section in bytes. It must be zero for an absolute program section.

Program sections are assigned an identifying sequence number as their respective GSD records are encountered. The program section number ranges from 0 through 255 within any single module. Note, however, that the total number of program sections in a single link operation is bounded only by the linker's virtual memory requirements. This program section number is used as an index in all references to the program section. Note that this permits any mixture of GSD records, as long as program sections are defined to the linker in the same order as the index used by symbol definitions.

C.4.2 **Global Symbol Specification OBJ$C_GSD_SYM**

Global symbol specification records may appear anywhere between the MHD and EOM records and in any order.

The format of a global symbol specification is as follows:

RECORD TYPE 1	1 byte
GSDTYPE 1	1 byte
DATA TYPE 1	1 byte
FLAGS	2 bytes
PSECT INDEX	1 byte
VALUE	4 bytes
SYMBOL NAME	Variable 2-32 bytes

The 5 bytes for PSECT INDEX and VALUE are omitted for a reference (that is, when SYM$V_DEF=0).

C.4.2.1 **Data Type** - The data type record is encoded as described in Appendix C of the VAX-11 Architecture Handbook.

NOTE

The current implementation of the linker ignores the data type field.

C.4.2.2 **Flags** - The flag bits in the global symbol specification have the following meaning:

Bit	**Name**	**Use**
0	GSY$V_WEAK	0 for strong resolution. 1 for weak resolution. Table C-1 describes the use of GSY$V_WEAK in conjunction with the definition bit (GSY$V_DEF).
1	GSY$V_DEF	0 for reference 1 for definition
2	GSY$V_UNI	0 for within facility 1 for universal symbol This bit is significant only on a definition. It indicates the symbol is to be retained if this facility is shareable.
3	GSY$V_REL	0 for absolute symbol value 1 for relative symbol and the value is augmented by the indexed program section base address (of this module's contribution)
4-15		Reserved.

Table C-1
Interpretation of GSY$V_WEAK and GSY$V_DEF

GSY$V_WEAK	GSY$V_DEF	Interpretation
0	0	Strong Reference -- symbol must be resolved
1	0	Weak Reference -- resolved only if the symbol is defined for some reason other than this reference. Does not incur any searches or module loads. Has the value zero if undefined, with no error report.
0	1	Strong Definition -- remains in all required symbol tables/maps.
1	1	Weak Definition -- is discarded from all symbol tables/maps unless there was a reference. Does not appear in the global symbol table index of an object module library.

C.4.2.3 **Program Section Index** - The program section index is a number between 0 and 255 to be used as an index into the sequence of program section definition records. This field exists only for symbol definition records (GSY$V_DEF=1) and identifies the program section in which the symbol was defined. The index is also used in TIR commands (see Section C.5.1.1) for reference to program section base addresses.

All symbols encountered must be defined within a program section, independently of the relocatability of program sections or symbols. For example, the linker does not require the base address of the "owning" program section if the symbol is absolute. However, for the purposes of generating a readable map, it is very useful to maintain the hierarchy of symbol within program section within module within file.

C.4.2.4 **Value** - This field contains the value assigned to the symbol by the language processor. This field does not exist if the record is a symbol reference (GSY$V_DEF=0).

C.4.3 Entry Point Symbol and Mask Definition (OBJ$C_GSD_EPM)

This format is an extended version of the global symbol definition format above. Following the symbol value (which will be an entry point address) is a two-byte field for the procedure's register save mask (as used by CALL instructions). The format is as shown below.

RECORD TYPE 3	1 byte
GSD TYPE 2	1 byte
DATA TYPE	1 byte
FLAGS	2 bytes
PSECT INDEX	1 byte
VALUE	4 bytes
ENTRY MASK	2 bytes
SYMBOL NAME	variable 2-32 bytes

C.4.3.1 **Entry Mask** - The entry mask is written at the entry point of a procedure entered via a CALLS or CALLG instruction, and in some cases also is used in transfer vectors to such procedures. A TIR command (see Section C.5) is provided for the language processor to direct the linker to insert the mask at the procedure entry point or at the transfer vector.

C.4.4 Procedure with Formal Argument Definiton (OBJ$C_GSD_PRO)

This GSD format is an extension of the entry point and mask definition format to define the formal arguments of the procedure. The format is as shown below.

RECORD TYPE 1	1 byte
GSD TYPE 3	1 byte
DATA TYPE	1 byte
FLAGS	2 bytes
PSECT INDEX	1 byte
VALUE	4 bytes
ENTRY MASK	2 bytes
SYMBOL NAME	variable 2-32 bytes
MINIMUM ACTUAL ARGUMENTS	1 byte
MAXIMUM ACTUAL ARGUMENTS	1 byte
FORMAL ARG 1 DESCRIPTOR	variable length (2-256 byte) descriptors of formal arguments arg n is optionally function return value.
FORMAL ARG n DESCRIPTOR	

Following is a description of the fields of a procedure definition different from the fields described in the other types of GSD records.

C.4.4.1 **Minimum and Maximum Actual Argument Counts** - Permissible values are 0 through 255 and specify, respectively, the minimum number and the maximum number of arguments required for a valid call to this procedure. The counts must include the function return value if such exists.

The current implementation does not validate procedure calls. However, for its own integrity, the current implementation validates that the minimum number of actuals is less than or equal to the maximum number of arguments. The maximum number of actuals field is then used to process the formal argument descriptors.

C.4.4.2 **Formal Argument Descriptors** - Each of the formal argument descriptors of the record shown above has the following format:

ARG. VAL. CTL.	1 byte ARG$B_VALCTL
REM. BYTE CNT.	1 byte ARG$B_BYTECNT
DETAILED ARGUMENT DESCRIPTION	variable 0-255 bytes ignored by current implementation

C.4.4.2.1 **Argument Validation Control Byte** - This (the first) byte of each formal description is used to control the validation of the argument. The only field of this control byte used by the linker is as follows:

Bits 0:1	ARG$V_PASSMECH	- Describes the mechanism by which the argument of a valid call must be passed.
Bits 2:7	Reserved	- Ignored by the current implementation.

The following argument-passing mechanisms are defined:

ARG$K_UNKNOWN	= 0	Unspecified
ARG$K_VALUE	= 1	By value
ARG$K_REF	= 2	By reference
ARG$K_DESC	= 3	By descriptor

C.4.4.2.2 **Remaining Byte Count** - This field gives the length of the remainder of this argument descriptor. For the current implementation, it is used as a count of bytes to be ignored by the linker. The content of these remaining bytes is reserved for possible future implementations.

NOTE

Any use of formal argument descriptors in which

```
    ARG$B_VALCTL    bits 2:7  NEQ 0
```

and/or

```
    ARG$B_BYTECNT   NEQ 0
```

means that, should argument validation be implemented in a future VAX-11 Linker, recompilation of all such objects may be necessary.

C.5 TEXT INFORMATION AND RELOCATION (TIR) RECORDS (OBJ$C_TIR)

Text information and relocation records contain a sequential series of commands and data for the linker to compute and record the contents of the image. The general form of a TIR record is as follows:

RECORD TYPE 2	1 byte
COMMAND 1	1 byte
DATA 1	Byte count implied by command
COMMAND 2	1 byte
DATA 2	Byte count implied by command
. . . . . .	
COMMAND N	1 byte
DATA N	Byte count implied by command

C.5.1 Commands

The linker's creation of the binary content of an image file is completely driven by the language processor via the commands contained in TIR records. To achieve this, the linker maintains an internal stack.

The commands available allow values to be placed on the stack and operations to be performed on the items on top of the stack. These commands also permit the writing of values from the stack to the output image. Other commands permit the storing of a sequence of bytes from object module to output image without alteration by the linker. They also provide for control of the relocation of the position currently being written in the image.

In commands which refer to program sections, the names are identified by the sequence numbers assigned to them as described above. The program section indexes are in the range 0 through 255.

The command byte has two formats:

```
                      7 6                0
                     ┌─┬─────────────────┐
FORMAT  1            │1│     -COUNT      │
                     └─┴─────────────────┘

                      7 6                0
                     ┌─┬─────────────────┐
FORMAT  2            │0│     COMMAND     │
                     └─┴─────────────────┘
```

The only command with FORMAT 1 is the Store Immediate (STOIM), which merely causes the copying of the following bytes (given by the negative count in the range -1 through -128) into the output image.

All other commands are described by the second format. There are four groups of commands:

Stack Group
Store Group
Operator Group
Control Group

The stack on which these commands operate is longword aligned at all times. Furthermore, it must be completely collapsed at end of module, but is retained between all other record types. The minimum stack space available will be not less than 25 longwords.

C.5.1.1 **Stack Group** - The stack group of commands provides the capability to store bytes, words, and longwords on the stack. The value placed on the stack may follow the command in the TIR record; it may be found from a global symbol; or it may be computed from the base address of a program section. Except for stacking the value of global symbols or stacking addresses (calculated from program sections), both signed extension to longword and zero extension to longword are provided for byte and word stack operations. The codes in the following list are decimal.

Code	**Command**	**Description/Interpretation**
0	Stack Global (TIR$C_STA_GBL)	Symbol specification follows. As with all other names, it consists of the symbol length in a byte followed by the ASCII string defining the symbol: LENGTH — 1 byte SYMBOL — Variable 1-31 bytes The value found from the symbol table is a 32-bit quantity.
1	Stack Signed Byte (TIR$C_STA_SB)	Single signed byte constant follows. Value is sign extended to 32 bits.
2	Stack Signed Word (TIR$C_STA_SW)	Single signed word constant follows. Value is sign extended to 32 bits.
3	Stack Longword (TIR$C_STA_LW)	Single longword constant follows.
4	Stack PSECT base plus byte offset (TIR$C_STA_PB)	1-byte program section number followed by single signed byte offset. A 32-bit quantity is computed by addition of program section base address and the byte offset.

Code	Command	Description/Interpretation
5	Stack PSECT base plus word offset TIR$C_STA_PW)	1-byte program section number followed by single signed word offset. A 32-bit quantity is computed by addition of program section base address and the word offset.
6	Stack PSECT base plus long word offset (TIR$C_STA_PL)	1-byte program section number followed by signed longword offset. A 32-bit quantity is computed by addition of program section base address and the longword offset. 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.
7	Stack Unsigned Byte (TIR$C_STA_UB)	As for TIR$C_STA_SB except that the value is zero extended to 32 bits.
8	Stack Unsigned Word (TIR$C_STA_UW)	As for TIR$C_STA_SW except that the value is zero extended to 32 bits.
9	Stack Byte From Image (TIR$C_STA_BFI)	This command is reserved for future use. The longword on top of the stack is used as an address, in the image, from which to retrieve a byte. The byte is zero extended and replaces the top longword of stack.
10	Stack Word From Image (TIR$C_STA_WFI)	This command is reserved for future use. It is the word variant of the previous command.
11	Stack Longword From Image (TIR$C_STA_LFI)	This command is reserved for future use. It is analogous to the previous two commands.
12	Stack Entry Point Mask (TIR$C_STA_EPM)	This command has the same format as TIR$C_STA_GBL. However, instead of stacking the value of the symbol, the entry point mask (unsigned word) which accompanies the symbol definition is stacked. An error is produced if the symbol referenced is not an entry point.

Code	**Command**	**Description/Interpretation**
13	Compare procedure arguments and stack TRUE or FALSE. (TIR$C_STA_CKARG)	The format of the command is as follows: COMMAND CODE SYMBOL NAME ARG INDEX ACTUAL ARGUMENT DESCRIPTOR The purpose of this command is to compare an actual argument descriptor with a formal descriptor for a particular procedure, stacking an indicator based on match or mismatch of arguments. This indicator is TRUE if match is found or if there is no formal argument descriptor. The indicator is FALSE if (and only if) the specified formal is described by a procedure definition but the descriptor does not match the accompanying actual argument descriptor. The argument that is checked is given by the index, and is thus number 0 through 255. The format of the actual argument descriptor is identical to that of the procedure definition GSD record described in Section C.4.4.2 above. The linker currently compares only the fields ARG$V_PASSMECH, stacking the TRUE indicator if they agree, FALSE if they do not.
14-19	Reserved Commands	

C.5.1.2 **Store Group** - All commands of the store group pop the top longword from the stack upon completion of the command. Several of the commands provide validation of the quantity being stored, with the possibility of issuing truncation errors during the operation. Upon completion of the command, the location counter is pointing to the next byte in the output image.

Code	Command	Description/Interpretation
20	Store Signed byte (TIR$C_STO_SB)	Bits 31:7 must be identical. Low byte written to image.
21	Store Signed Word (TIR$C_STO_SW)	Bits 31:15 must be identical. Lower word written to image.
22	Store Longword (TIR$C_STO_LW)	One longword written to image.
23	Store Byte Displaced (TIR$C_STO_BD)	Location counter subtracted from top of stack. Decrement value. Bits 31:7 must be identical. Byte is then written to image.
24	Store Word Displaced (TIR$C_STO_WD)	Location counter plus 2 subtracted from top of stack. Bits 31:15 must be identical. Word written to image.
25	Store Longword Displaced (TIR$C_STO_LD)	Location counter plus 4 subtracted from top of stack. Longword written to image.
26	Store Short Literal (TIR$C_STO_LI)	One longword from stack, bits 31:6 must be zero. Single byte written to image.
27	Store Position-Independent Data Reference (TIR$C_STO_PIDR)	The longword on top of stack is assumed to be the address of a data item. It occurs in a nonexecutable program section. If the address is absolute, command behaves as store longword. If address is relocatable, command behaves as store longword displaced and in addition provides information in the image header for subsequent linker processing.

Code	Command	Description/Interpretation
28	Store Position-Independent Code Reference (TIR$C_STO_PICR)	The longword on top of the stack is assumed to be the address of an item to which a position-independent instruction makes reference. The purpose of the command is to generate a position-independent reference. If the top of stack is absolute, the byte "9F" (hex) is written (which is autoincrement deferred addressing mode on the PC and therefore absolute) followed by the top as for store longword. If, however, top of stack is relocatable, the byte "EF" (hex) is written (which is longword displacement mode off PC and therefore relative addressing). Location counter is incremented. Then the longword is written just as for store longword displaced. This and the previous command are discussed further in the references on generation of position independent images.
29	Store Repeated Signed Byte (TIR$C_STO_RSB)	The longword on top of the stack is used as the repeat count. The low order byte of next longword on the stack is written to the image the indicated number of times. Both longwords are cleaned off stack on completion.
30	Store Repeated Signed Word (TIR$C_STO_RSW)	As above except that words are written.
31	Store Repeated Longword (TIR$C_STO_RL)	Analogous to above.
32	Store Arbitrary Field (TIR$C_STO_VPS)	The bits 0 to (s-1) of the top longword are written to image starting at bit p of the current location. The command byte in the object module is followed by p and s (respectively) which are unsigned bytes. Only the specified bits of the image are altered. After the operation the location counter is the address of the byte containing bit (p+s) of the location modified. Note that the value of p+s must be greater than zero and less than or equal to either 32 or ((P+8)/8)*8-1, whichever is less.
33	Store Unsigned Byte (TIR$C_STO_USB)	Same as TIR$C_STO_SB except that bits 31:8 must be zero.

Code	Command	Description/Interpretation
34	Store Unsigned Word (TIR$C_STO_USW)	Analogous to above (Bits 31:16 must be zero)
35	Store Repeated Unsigned Byte (TIR$C_STO_RUB)	Analogous to above.
36	Store Repeated Unsigned Word (TIR$C_STO_RUW)	Analogous to above.
37	Store Byte (TIR$C_STO_B)	If top longword on stack is is negative, then bits 31:7 must be 1. Otherwise, bits 31:8 must be zero. Low order byte is written to image. This command permits any 8 bit value from -128 to 255 to be written to the image.
38	Store Word (TIR$C_STO_W)	If top longword is negative, bits 31:15 must be 1. Otherwise bits 31:16 must be zero. Low order word is written to image. This command permits any 16-bit value from -32768 to 65535 to be written to the image.
39	Store Repeated Byte (TIR$C_STO_RB)	The repeated version of store byte. See TIR$C_STO_RSB for description of repeat count.
40	Store Repeated Word (TIR$C_STO_RW)	Analogous to above.
41	Store repeated Immediate Variable Bytes (TIR$C_STO_RIVB)	One byte of byte count (N) followed by N bytes. The N bytes are written to the image and repeated the number of times specified by the top longword of the stack, which is removed from the stack on completion. (A 0 repeat count stores nothing.)
42	Store Position Independent Reference Relative (TIR$C_.STO_.PIRR)	The longword (longword 1) on the top of the stack is the address of a data item. If it is an absolute value the command is the same as Store Longword except that a second value is cleared from the linker stack. If the address is relocatable, the second longword (longword 2) is taken from the stack. If its value is -1, the current value of the location counter is substituted for longword 2. The value then stored is longword 1 minus longword 2).
43-49	Reserved Commands	

C.5.1.3 **Operator Group** - The linker evaluates expressions in Post Fix Polish form. All arithmetic operations are performed in signed 32-bit 2's complement integers. There is no provision for floating point, string, or quadword computation.

The commands of the operator group take as operands the top one or two longwords on the stack. Upon completion of the operation, the result is the top longword on the stack. Attempts to divide by zero produce a zero result, and a nonfatal diagnostic is issued.

Code	Command	Description/Interpretation
50	No-operation (TIR$C_OPR_NOP)	
51	Add (TIR$C_OPR_ADD)	Top two longwords are added.
52	Subtract (TIR$C_OPR_SUB)	Top longword is subtracted from next.
53	Multiply (TIR$C_OPR_MUL)	Top two longwords are multiplied.
54	Divide (TIR$C_OPR_DIV)	Divisor is top longword.
55	Logical AND (TIR$C_OPR_AND)	Logical AND of top two longwords.
56	Logical Inclusive OR (TIR$C_OPR_IOR)	Inclusive OR of top two longwords.
57	Logical Exclusive OR (TIR$C_OPR_EOR)	Exclusive OR of top two longwords.
58	Negate (TIR$C_OPR_NEG)	Top longword is negated.
59	Complement (TIR$C_OPR_COM)	Top longword is complemented.
60	Insert Field (TIR$C_OPR_INSV)	This command is reserved for future use. It is analogous to the store of arbitrary bit field TIR$C_STO_VPS (code 32). The only difference is that the target for bits from top of stack is the next longword on the stack, and location counter is therefore unaffected. Note that top longword is popped and that p and s are bytes following command in the TIR record.
61	Arithmetic Shift (TIR$C_OPR_ASH)	The longword on top of stack is the shift count to apply to next longword. Negative quantity causes a right shift (with replication of sign bit). Positive causes left shift with zeroes moved into low order bits.

Code	Command	Description/Interpretation
62	Unsigned Shift (TIR$C_OPR_USH)	As above except that zeroes are moved into high and low order.
63	Rotate (TIR$C_OPR_ROT)	Rotate count is top longword to apply in a rotate (left if positive, otherwise right) of next longword on stack. Rotate count must have an absolute value between 0 and 32.
64	Select (TIR$C_OPR_SEL)	Remove the top longword from the stack. If it has the value TRUE (low bit set) remove and discard the next longword on the stack. If the first longword removed has the value FALSE (low bit clear) copy the next longword on the stack to the one that follows. Thus, the command presumes there are three longwords on the stack. These are collapsed to a single longword which is the value of the second or third based on the value of the first.
65	Redefine Symbol to Current Location. (TIR$C_OPR_REDEF)	The command has the same format as the TIR$C_STA_GBL command. Causes the symbol to be re-defined on output of symbol table(s) to have the value of the location counter when this command is processed. The re-definition does not occur until after all image binary is written. If no binary is generated (or is aborted) the redefinition does not occur.
66-79	Reserved Commands	

C.5.1.4 **Control Group** - The control group of commands is provided for manipulation of the location counter.

Code	Command	Description/Interpretation
80	Set Relocation Base (TIR$C_CTL_SETRB)	The value on top of the stack is popped into the location counter.
81	Augment Relocation Base (TIR$C_CTL_AUGRB)	Signed longword which is an increment to location counter follows the command.
82	Define Location (TIR$C_CTL_DFLOC)	The top longword on the stack is removed and used as an index. The current location counter is stored away, qualified by the index and the object module containing it. This command is only legal in traceback and debug records.

Code	Command	Description/Interpretation
83	Set Location (TIR$C_CTL_STLOC)	The top longword on the stack is removed and used as an index. The saved location counter (from a corresponding TIR$C_CTL_DFLOC) is set as the current location counter, and the location counter saved for index (n) is deleted. This command is only legal in traceback and debug records.
84-127	Reserved Commands	

C.5.2 **Record Length**

TIR records may be quite long. There is an implementation limit defined by OBJ$C_MAXRECSIZ. The maximum record size of the module is recorded in the header word.

C.5.3 **Side Effects and Optimization**

In the interest of performance of the linker a few implementation decisions and their possible side effects should be noted.

1. For all store repeated commands, if the quantity being stored is zero, the linker does not write the zeroes into the bytes. The reason for this is that the pages of an image are guaranteed to be zero unless otherwise initialized by the compiler. To achieve this, demand zero pages are used within the linker and were the linker to attempt to write zeroes, the fact that these are still empty pages of the image is lost. Thus, it becomes very difficult to compress from the image all empty pages.

   There is, however, a side effect to this behavior, in that if a cell of an image has been previously initialized, it will not be zeroed by any repeated store commands. This can occur in multiple modules contributing to and attempting to initialize the content of overlaid program sections. Notice, however, that the results of such multiple initialization are then dependent on the order of processing of object modules. This side effect is therefore considered to be acceptable.

2. The linker is a two-pass processor of object modules. The content of TIR records is completely ignored on the first pass but verified and acted upon on the second pass. However, if no image is being produced because of a command or link time error, all TIR records (as well as DBG and TBT records) are ignored. A side effect, considered quite acceptable, is that errors (user or compiler) potentially detectable on pass two will not be detected. Truncation errors are the most likely example of such undetected situations.

C.6 END OF MODULE (EOM) RECORD (OBJ$C_EOM)

This record declares the end of a module. It declares the severity of errors encountered by language processor, and, optionally, it declares a transfer address within a program section in this module. The format is as follows:

RECORD TYPE 3	1 byte
ERROR SEVERITY	1 byte
P-SECT INDEX	1 byte
TRANSFER ADDRESS	4 bytes
TRANSFER FLAGS	1 byte

This record will be 2, 7, or 8 bytes, depending on the existence of a transfer address. Note that the program section specification is by its index within the module, as used above. The transfer address is an offset from the base of this module's contribution to the specified program section. The transfer flags byte may only be present if the transfer address is present.

C.6.1 **Error Severity**

The error severity byte specifies to the linker whether errors were encountered in the source code. It also indicates the severity of any errors encountered.

Value	**Interpretation by linker**
0	No errors
1	Warnings were generated by language processor. Proceed with link but issue warning message.
2	Errors were severe, proceed with link, but do not produce an executable image.
3	Abort the link.
4-10	Reserved.
11-255	Ignored.

C.6.2 **Transfer Address Flags**

The transfer address flags byte directs the linker in the processing of transfer addresses.

Bit	Interpretation by linker
0	Weak transfer address. If a transfer address is already defined, no error is produced.
1:7	Reserved, must be 0.

C.7 DEBUGGER INFORMATION (DBG) RECORDS (OBJ$C_DBG)

The purpose of debugger information records is to allow the language processors to pass information, such as descriptions of local variables, of the compilation 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 DGB records generates what is referred to as the debug symbol table (DST). The DST follows immediately 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 in fact mapped into the user image.

The linker produces a DST only if the debugging 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. See the above discussion of the side effects in TIR record processing.

C.7.1 Traceback Information (TBT) Records (OBJ$C_TBT)

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 this information from that 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. See the above section on side effects in processing TIR records.

C.8 LINK OPTION SPECIFICATION (LNK) RECORDS (OBJ$C_LNK)

The link option specification records are defined for the purpose of allowing the compiler to provide the linker with default parameters that are used if none were given by the user at link time. Options of interest are libraries to be searched to resolve undefined symbols, modules to be included in the link, adjustment of stack and buffer region sizes.

The exact set of commands allowable will be supplied later, along with the interaction of conflicting object module LNK records and user commands. The general philosophy is to use the most recently specified parameters unless there are good reasons to the contrary. These records are currently ignored by the linker.

APPENDIX D

THE ANALYZE PROGRAM

The object module analysis (ANALYZE) program checks an object module to see if it is in the correct format for input to the VAX-11 Linker. (The program can also analyze a concatenated file containing several object modules.) This program is a diagnostic tool for writers of compilers or assemblers that must generate native-mode code. To use this program intelligently, you must be familiar with the VAX-11 object language specification in Appendix C.

To invoke the program, use the DCL command ANALYZE. The program can analyze the entire module or only specified types of records. It checks the record type, contents, and sequence of each object module record it examines. The program creates an output file containing a record-by-record analysis of the object module, including identification of any errors in the module.

The program, however, has a more limited set of operations than the linker. The ANALYZE program:

- Does not verify that all data arguments to commands are in the correct format
- Does not check whether "store data" commands are storing within legal address limits

These restrictions exist because the program does not actually perform the object language commands in the module. Therefore, if you have run ANALYZE against an object module and received no errors, you should perform the additional check of linking the module, requesting a full map.

D.1 THE ANALYZE COMMAND

The ANALYZE command requires no user privileges and follows the standard DCL syntax conventions. For a complete discussion of this command, see the VAX/VMS Command Language User's Guide. The general format of the command is as follows:

```
$ANALYZE  input-file-spec[/MHD][/GSD][/TIR][/DBG][/TBT][/EOM]
```

Command Qualifiers:	Default:
/OUTPUT=file-spec	Output file name = input file name; output file type = ANL
/INTERACTIVE	/NOINTERACTIVE -- you are not prompted after each record is displayed.

D.2 THE OUTPUT FILE

The analysis of each record contains information pertinent to that record type, and begins with a line in the following format:

```
>>>>>RECORD [number] IS [record type] [number of] BYTES LONG
```

For example:

```
>>>>>RECORD 4 IS A TEXT/RELOCATION 58 BYTES LONG
```

Errors in the object module are identified in lines beginning with asterisks. For example:

```
***** RECORD 3 IS RESERVED TYPE 49 ******************
```

The program also prints the hexadecimal representation of the erroneous record's contents. A complete list of the error messages appears later in this appendix.

The following subsections (D.2.1 to D.2.7) discuss the information given for the following record types:

Debugger information record
End of module record
Global symbol directory (GSD) record
Module header record
Subheader record
Text information and relocation (TEXT/RELOCATION) record
Traceback information record

The record types are explained in alphabetical order for ease of reference. In the actual output file, the order reflects the order of the records in the module. The only requirement for sequence is that a module header and any subheader records come first, and an end of module record come last.

D.2.1 Debugger Information Record

Debugger information records identify one or more commands to the linker, including the data or symbol associated with each command. (The object language commands are discussed in Appendix C.) For example, a Store Immediate (STOIM) command is followed by the decimal number of bytes to be stored and the hexadecimal representation of this data. A Stack Global (STA_GBL) command is followed by the name of the symbol whose value is to be placed on the linker stack.

To the right of each command is the notation "STACK = n," where "n" is the decimal number of bytes that would be on the linker's internal stack if the linker were actually processing the object module. The ANALYZE program reports an error if the count is not equal to zero at the end of each object module analysis (that is, if bytes would be left on the linker stack or if more bytes would be removed than were placed on the stack).

D.2.2 **End of Module Record**

The end of module record contains the following information:

- Number of compiler errors and warnings, including any link abort specification by the language processor due to fatal compilation errors
- Number of the program section that contains the transfer address
- Transfer address expressed as an offset from the program section base
- Number of program sections defined in the module

D.2.3 **Global Symbol Directory (GSD) Record**

A GSD record can contain one or more program section definitions, one or more global symbol specifications, or a combination or both.

For each program section definition, the following information is given:

- Program section alignment
- Flag bits that are set
- Maximum length of the program section

For each global symbol specification, the following information is given:

- Specification function - that is, whether the specification is a reference to a global symbol or a definition of a global symbol
- Data type (For example: "procedure entry mask," data type 23; or "unknown," data type 0)
- Flag bits that are set
- Number of the program section in which the symbol is defined (for global symbol definitions only)
- Value of the symbol (for global symbol definitions only)
- Name of the global symbol

D.2.4 **Module Header Record**

The module header record lists the following information about the module:

- Structure level of the object language
- Maximum length that a record in the module can have
- Name

- Identification
- Creation date and time
- Date and time of the last patch

D.2.5 **Subheader Records**

Most module subheader records consist of ASCII data and are printed as such. These subheader types include identification of the language processor used, the compilation options specified, and the title specified for the compilation listing.

Another type of subheader identifies the maintenance status if the module has been patched. The maintenance status subheader record lists the following information:

- Patch utility name and version number
- UIC under which the patch was executed
- Input file specification for the patch
- Correction file specification for the patch
- Date and time of the patch
- Sequential patch number

D.2.6 **Text Information and Relocation (TEXT/RELOCATION) Record**

Text information and relocation records contain the same type of information as debugger information records (see Section D.2.1).

D.2.7 **Traceback Information Records**

Traceback information records contain the same type of information as debugger information records (see Section D.2.1).

D.3 **ANALYZE PROGRAM ERROR MESSAGES**

Errors in the object module are identified by lines beginning with asterisks (*****). Most of these error messages appear in the output file, but some are displayed on the terminal. The messages are presented in alphabetical order, with explanations for those that are not self-explanatory.

ABSOLUTE PSECT HAS NON-ZERO LENGTH

All absolute program sections must have a length of zero.

ARGUMENT DESCRIPTOR MISSING FOR FORMAL ARGUMENT #[number]

The specified argument requires a character string descriptor.

ARGUMENT INDEX IS MISSING

BYTE COUNT GOES BEYOND END OF RECORD [number of] BYTES

A byte count contained in the record indicates that the data that follows does not fit within the record.

[number of] BYTES WERE LEFT ON THE LINKER'S STACK

The object language commands in the module place more bytes on the linker's internal stack than they remove. The linker's stack should be empty when it finishes processing each object module.

[number of] BYTES WERE NOT PLACED ON THE LINKER'S STACK BUT WERE REMOVED FROM IT

The object language commands in the module remove more bytes from the linker's stack than they place on it.

COMMAND [number] IS ILLEGAL TYPE [number]

COMMAND [number] IS RESERVED [code]

(See Appendix C for a list of the assigned and reserved codes.)

CORRECTION FILE SPECIFICATION WAS NOT IN COMPRESSED FORM

The correction file specification in the maintenance status subheader record contains nulls, tabs, or blanks.

DATE/TIME FIELD CONTAINS ILLEGAL CHARACTER CHARACTER [position] IS [ASCII representation] ([number] HEX)

ENTRY POINT GSD HAS ILLEGAL LENGTH [number of] BYTES - NOT BETWEEN [minimum number] AND [maximum number]

FILE [file name] DOES NOT END WITH EOM

The end of the module record is missing or out of sequence.

FIRST CHARACTER OF GLOBAL SYMBOL IS NUMERIC OR BLANK

FIRST CHARACTER OF CORRECTION FILE SPECIFICATION IS NUMERIC OR BLANK

FIRST CHARACTER OF INPUT FILE SPECIFICATION IN NUMERIC OR BLANK

FIRST CHARACTER OF MODULE NAME IS NUMERIC OR BLANK

FIRST CHARACTER OF PATCH UTILITY NAME IS NUMERIC OR BLANK

FIRST CHARACTER OF P-SECT NAME IS NUMERIC OR BLANK

FORMAL ARGUMENT DESCRIPTOR IS MISSING REMAINING BYTE COUNT

The record is not long enough to hold the remaining byte count.

GLOBAL SYMBOL CONTAINS ILLEGAL CHARACTERS CHARACTER [position] IS [ASCII representation] ([number] HEX)

Valid characters are . (period), $ (dollar sign), _(underscore), 0 through 9, and A through Z.

GLOBAL SYMBOL DEFINITION RECORD HAS ILLEGAL LENGTH [number of] BYTES - NOT BETWEEN [minimum length] AND [maximum length]

GLOBAL SYMBOL FIELD LENGTH [length] ILLEGAL NOT 1 TO [number of] CHARACTERS

GLOBAL SYMBOL LENGTH ([number of] BYTES) ILLEGAL - SHOULD BE 1 TO [number of] CHARACTERS

GLOBAL SYMBOL REFERENCE RECORD HAS ILLEGAL LENGTH [number of] BYTES - NOT BETWEEN [minimum number] AND [maximum number]

GSD TYPE [number] DOES NOT EXIST

(Appendix C lists the valid GSD types.)

IDENT CONTAINS ILLEGAL CHARACTER CHARACTER [position] IS [ASCII representation] ([number] HEX)

IDENT LENGTH [length] ILLEGAL SHOULD BE 1 TO [number of] CHARACTERS

ILLEGAL ALIGNMENT - GREATER THAN [maximum permitted]

ILLEGAL CORRECTION FILE SPECIFICATION LENGTH OF ZERO, SHOULD BE 1 TO [number of] CHARACTERS

The correction file specification in the maintenance status subheader record is zero.

ILLEGAL INPUT FILE SPECIFICATION LENGTH OF ZERO, SHOULD BE 1 TO [number of] CHARACTERS

The input file specification in the maintenance status subheader record has a length of zero.

ILLEGAL MAXIMUM RECORD LENGTH - MUST BE BETWEEN [minimum length] AND [maximum length]

The maximum record length for the module, as specified in the module header record, is outside the acceptable range.

ILLEGAL PSECT ALLOCATION - EXCEEDS [number of] BYTES

ILLEGAL RECORD LENGTH [length] NOT BETWEEN [minimum number] AND [maximum number] BYTES

ILLEGAL SEVERITY CODE [code]

The severity code specified in an end of module record is illegal (see Section C.6.1).

ILLEGAL STARTING BIT POSITION- NOT 0 TO 31

A variable bit field command specifies an illegal starting bit position.

ILLEGAL STRUCTURE LEVEL - ONLY [highest level currently supported] IS SUPPORTED

The structure level (format) of the object language in the module is not yet supported by the ANALYZE program. The program analyzes only structure levels 0 through the level specified in the message.

ILLEGAL SUBHEADER TYPE OF [type]

The specified subheader type is not recognized by the ANALYZE program.

ILLEGAL SYNTAX FOR DATE/TIME

The date and time must be in a fixed-length ASCII string with the following format:

```
dd-mon-yy hh:mm:ss
```

ILLEGAL TRANSFER ADDRESS (NOT BETWEEN 0 AND [highest available virtual address])

The transfer address is outside the virtual address space.

[flag bit number] ILLEGALLY

The specified flag bit is set, but is reserved (invalid). This message follows the message "THE FOLLOWING FLAG BITS ARE SET:" and a listing of the flag bits legally set.

INCOMPLETE RECORD - NEXT FIELD AT BYTE [location within record] BEYOND RECORD

A byte count contained in a module header record is greater than the number of bytes remaining in the record.

INPUT FILE SPECIFICATION WAS NOT IN COMPRESSED FORM

The input file specification of the maintenance status subheader record contains blanks, nulls, or tabs.

INVALID ARGUMENT INDEX OF [number] NOT BETWEEN [minimum] AND [maximum]

INVALID MAXIMUM ACTUAL ARGUMENT COUNT OF [number] NOT BETWEEN [number] AND [number]

INVALID MINIMUM ACTUAL ARGUMENT COUNT OF [number] NOT BETWEEN [number] AND [number]

INVALID SEQUENCE - MHD SHOULD NOT FOLLOW TYPE [type] RECORD

The only record type that can precede a module header is type 3 (end of module record for the previous module in a concatenated file).

INVALID SEQUENCE - SHOULD NOT FOLLOW EOM OR BEGIN A MODULE

Nothing can follow an end of module record except an end-of-file or a module header record in a concatenated file.

INVALID SEQUENCE - SUB-HEADER RECORD SHOULD NOT FOLLOW TYPE [type] RECORD

A subheader record should only follow a module header record or another subheader record.

INVALID SYNTAX OF CORRECTION FILE SPECIFICATION IN: [error part]

The correction file specification in the maintenance status subheader record has a syntax error in the specified part.

LANGUAGE PROCESSOR RECORD FOLLOWS [number of] TIR, GSD, OR TBT RECORDS

The language processor subheader record should follow the module header record.

LANGUAGE PROCESSOR RECORD LARGER THAN [number of] CHARACTERS

MINIMUM IS GREATER THAN MAXIMUM

The minimum actual argument count specified is greater than the maximum actual argument count specified.

MODULE NAME CONTAINS ILLEGAL CHARACTERS CHARACTER [position] IS [ASCII representation] ([number] HEX)

MODULE NAME LENGTH ILLEGAL SHOULD BE 1 TO [number of] CHARACTERS

NO P-SECTIONS DEFINED IN MODULE

An object module must contain at least one program section.

PATCH UTILITY NAME CONTAINS ILLEGAL CHARACTER CHARACTER [position] IS [ASCII representation] ([number] HEX)

The patch utility name in the maintenance status subheader record contains the specified illegal character.

PATCH UTILITY NAME LENGTH ILLEGAL SHOULD BE 1 TO [number of] CHARACTERS

The length of the patch utility name in the maintenance status subheader record is illegal.

PATCH UTILITY VERSION CONTAINS ILLEGAL CHARACTERS

The patch utility version in the maintenance status subheader record contains an illegal character.

PATCH UTILITY VERSION LENGTH [length] ILLEGAL SHOULD BE 1 TO [number of] CHARACTERS

The length of the patch utility version in the maintenance status subheader record is outside the indicated range.

PROCEDURE WITH FORMAL ARGUMENT DEFINITION HAS ILLEGAL LENGTH ([number of] BYTES) - NOT BETWEEN [minimum number] AND [maximum number]

PROCEDURE WITH FORMAL ARGUMENT DEFINITION IS MISSING MAXIMUM ACTUAL ARGUMENT COUNT

PROCEDURE WITH FORMAL ARGUMENT DEFINITION IS MISSING MINIMUM ACTUAL ARGUMENT COUNT

P-SECT DEFINITION RECORD HAS ILLEGAL LENGTH [number of] BYTES - NOT BETWEEN [minimum length] AND [maximum length]

P-SECT NAME CONTAINS ILLEGAL CHARACTER CHARACTER [position] IS [ASCII representation] ([number] HEX)

PSECT NAME LENGTH IS ILLEGAL [length]

P-SECTION NUMBER EXCEEDS COUNT OF THOSE DEFINED IN MODULE [number of program sections defined]

The program section number supplied in the program section index field of an end of module record is greater than the number assigned to any program section in the module. In other words, the transfer address is specified as being in a nonexistent program section.

[number of] PSECT(S) DEFINED - TIR REFERENCES PSECT NUMBER (number)

A text information and relocation record contains a reference to a program section that is not defined in the module. Program sections are assigned a number in the range 0 through 255, in the order in which they are defined.

RECORD [number] IS ILLEGAL - ZERO LENGTH

The record length byte in the specified record contains zero.

RECORD [number] IS RESERVED TYPE [type]

The record type of the specified record in invalid.

RECORD [number] LENGTH ([length]) GREATER THAN MAX. PERMITTED LENGTH ([maximum length])

The length of the specified record is greater than the maximum permitted length specified in the module header record. The last number in the message is the actual length of the specified record.

REQUIRED DATA NOT CONTAINED WITHIN RECORD

Inconsistencies exist within the record that make it impossible for the ANALYZE program to interpret it. For example, a byte count of 3 might be followed by a symbol with more than 3 characters.

This message is followed by the contents of the record in hexadecimal representation.

RESERVED BIT #[number] SET IN ARGUMENT VALIDATION CONTROL BYTE

RESERVED DATA TYPE

(For a list of legal data types, see the "Procedure Calling and Condition Handling" appendix in the <u>VAX-11 Architecture Handbook</u> or in the <u>VAX-11 Run-Time Library Reference Manual</u>.)

VAX-11
Linker Reference Manual
AA-D019B-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 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 ______

Please cut along this line.

— — 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

BSSG PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

— Do Not Tear - Fold Here —

Cut Along Dotted Line