digital

VAX-11
Linker Reference Manual

Order No. AA-D019A-TE

VAX11

August 1978

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

VAX-11
Linker Reference Manual

Order No. AA-D019A-TE 8

SUPERSESSION/UPDATE INFORMATION:	This is a new document for this release.
OPERATING SYSTEM AND VERSION:	VAX/VMS V01
SOFTWARE VERSION:	VAX/VMS V01

To order additional copies of this document, contact the Software Distribution
Center, Digital Equipment Corporation, Maynard, Massachusetts 01754

digital equipment corporation · maynard, massachusetts

First Printing, August 1978

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

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

The following are trademarks of Digital Equipment Corporation:

DIGITAL
DEC
PDP
DECUS
UNIBUS
COMPUTER LABS
COMTEX
DDT
DECCOMM
ASSIST-11
VAX
DECnet
DECsystem-10
DECtape
DIBOL
EDUSYSTEM
FLIP CHIP
FOCAL
INDAC
LAB-8
DECSYSTEM-20
RTS-8
VMS
IAS
MASSBUS
OMNIBUS
OS/8
PHA
RSTS
RSX
TYPESET-8
TYPESET-11
TMS-11
ITPS-10
SBI

CONTENTS

CONTENTS (Cont.)

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

The VAX-11 Linker Reference Manual describes how the VAX-11 Linker works and how to use it. This manual has both an educational and a reference function: it provides detailed explanations of significant topics, yet it is also designed for quick look-up of important information.

INTENDED AUDIENCE

This manual is intended for programming specialists and nonspecialists alike. In general, the entire manual is intended to be informative and useful to all readers; however, certain parts 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 (for example, if you are a trainee programmer), 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, 3 through 7, 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 2, 8, 9, and Appendix C are aimed especially at this type of reader.

STRUCTURE OF THIS DOCUMENT

Chapters 1 and 2 introduce the linker. Chapter 1 defines significant terms, presents the reasons for the linker's existence, and discusses in general terms how the linker works. Chapter 2 goes more deeply into the process by which the linker creates images. Chapter 2 also introduces new concepts and expands on concepts introduced in Chapter 1.

Chapters 3 and 4 focus on concepts that are important to understanding the linker's operation. The discussion of symbols and references in Chapter 3 derives from the linker's function of resolving symbolic references between modules. Chapter 4 explains libraries, which normally contain frequently used modules that the linker can include in user images.

Chapter 5 discusses the LINK command and its command and file qualifiers. Chapter 6 focuses on the /OPTIONS file qualifier, describing how to create and use a linker options file.

Chapter 7 explains the different 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 8 and 9 present detailed explanations of shareable images and image clusters. The complex information in these chapters 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.

ASSOCIATED DOCUMENTS

The following documents contain information pertinent to linking:

- VAX-11 Information Directory
- VAX/VMS Primer
- VAX/VMS Command Language User's Guide
- VAX-11 Symbolic Debugger Reference Manual
- VAX/VMS System Manager's Guide
- VAX-11 MACRO Language Reference Manual
- VAX-11 MACRO User's Guide
- VAX-11 FORTRAN IV-PLUS Language Reference Manual
- VAX-11 FORTRAN IV-PLUS User's Guide

CONVENTIONS USED IN THIS DOCUMENT

The following conventions are used in this manual:

- Brackets ([]) enclose optional material, as in the following examples:

  ```
  /[NO]DEBUG
  ```

  The positive form of the qualifier is /DUBUG, and the negative form is /NODEBUG.

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

  The base address, page fault cluster (pfc), and additional file specifications are optional entries. Note, however, that the commas following the base address and page fault cluster are outside the brackets; therefore, if you omit these entries, you must still enter the commas. For example:

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

- Uppercase letters in format illustrations show keywords that you must enter as shown; lowercase letters show variable data, with the letter "n" specifying numeric data. Examples:

  ```
  /EXECUTABLE=file-spec

  /BASE=n
  ```

- Horizontal ellipses (...) in a format illustration indicate that the preceding entry can appear several times, as in the following example:

  ```
  UNIVERSAL=symbol-name [,...]
  ```

  You can specify multiple symbol names.

- Vertical ellipses indicate that lines of file contents or code not pertinent to the example are not shown:

  ```
  .

  .

  .
  ```

CHAPTER 1

LINKER OVERVIEW

The VAX-11 Linker is a programming development tool that takes the output of language translators, such as the VAX-11 MACRO assembler or the VAX-11 FORTRAN IV-PLUS compiler, and binds it into a form that can be executed on the VAX-11 hardware. The primary outputs of VAX-11 language translators, and the primary inputs to the linker, are 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. Shareable images also provide a way of linking a very large application program in a number of smaller phases. Chapters 2 and 9 discuss 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. Chapter 3 discusses the different types of symbols (for example, local and global, strong and weak), and Chapter 4 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 qualifiers 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 5 explains the LINK command and all its qualifiers. Chapter 6 focuses on the /OPTIONS qualifier and the special items or options that can appear in an options file.

In addition to the image itself, the linker can produce a printable image map. You can control the level of detail provided in various parts of the map. Chapter 7 explains and illustrates the image map.

1.1 REASON FOR A LINKER

The object modules that a VAX/VMS compiler or assembler creates are nonexecutable. They must first be linked. The requirement that object modules be linked contrasts with systems in which the output of a compiler or assembler is directly executable.

The VAX-11 native translators require a linker for several reasons:

- Linking simplifies modular programming.
- The linker simplifies the job of each native 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 many source language statements generate calls to common functional routines developed by DIGITAL
- Deliberate modular design implemented by some user sites

Most programs are automatically modular, because many source language statements generate calls to routines that perform commonly needed functions, such as opening and closing files. Examples of these routines 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 programmer'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 means that the 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, 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 IV-PLUS to perform different 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.

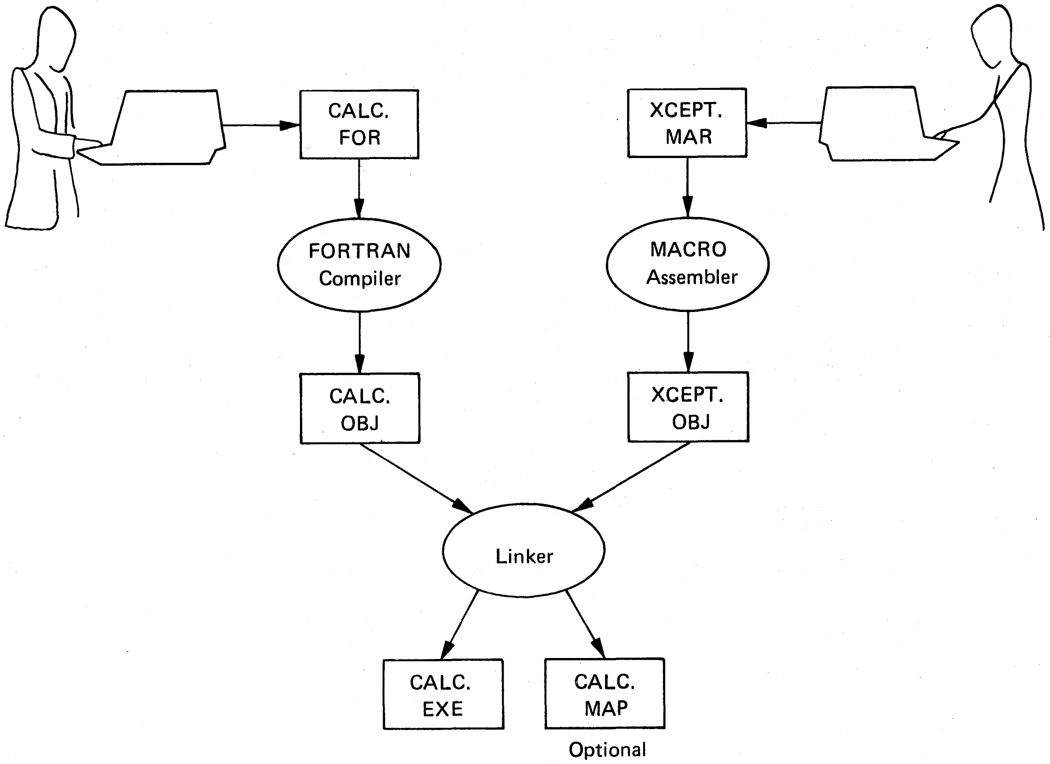


Figure 1-1 Modular Programming

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 contains the logic 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.

Each program usually interacts with the operating system. For example, a program may use the stack within its process. The linker can supply the program logic to access the stack and certain other areas, rather than require each compiler and assembler to supply this logic. The linker can also generate the proper program-to-system interfaces for program modules that call VAX/VMS system services.

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. FORTRAN users should refer to the Debugging chapter in the VAX-11 FORTRAN IV-PLUS User's Guide.

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 (read-only, read/write, etc.). For further information on how the linker maps the image, see Chapter 2.

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 the default system library. For a detailed discussion of libraries, see Chapter 4.

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 defines X as being equal to 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 7 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 3.3.1 contains further information on the symbol table file.

CHAPTER 2

IMAGE CREATION

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

2.1 PROGRAM SECTIONS

The program section is the vehicle by which a language translator describes the memory requirements of a particular object module. Program sections are areas of memory that have a name, a length, and a series of attributes describing the intended or permitted usage of that memory. Section 2.5.4 provides a detailed description of these attributes.

2.2 IMAGE SECTIONS

The image section is the means that the linker uses to describe the memory requirements of the whole image to the VAX-11 memory management software. An image section is a named collection of pages which have the same hardware protection characteristics and the same sharing nature. An image section is dealt with as a unit when page faults occur.

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 2.7).

2.3 CLUSTERS

Experience with virtual memory systems has shown that locality of reference within large application programs affects their performance. Clustering provides a way for the designer of an application to describe that locality. A cluster contains a group of highly-related object modules that are separable from some other groups of modules within the application.

For example, a compiler usually goes through a number of distinct phases during a single compilation run. These phases are often separable into groups of object modules that can be designated as clusters. The relationship between the groups or clusters is defined through internal data structures, such as the symbol table.

Chapter 9 is devoted to clustering. However, at this point it is sufficient to describe a cluster as a list of related image sections; these image sections are produced by sorting the program sections read in from a collection of related object module files. Every image consists of at least one cluster. Note, however, that the cluster is relevant only to the linker itself; it does not appear as a structure anywhere else (such as in the memory management software of the executive).

2.4 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 compile time. 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.

2.5 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

2.5.1 Program Section Name

The program section name is an ASCII character string, one to fifteen 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.

2.5.2 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.

2.5.3 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 to 9, representing a power or exponent of the number 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 entire concatenated program section will be aligned on a boundary which is a multiple of the highest power of 2 specified by any of the contributing modules.

2.5.4 Program Section Attributes

The following subsections explain the attributes that a program section can have. Section 2.7 describes how the linker considers certain significant attributes as it constructs different types of images.

2.5.4.1 **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.

2.5.4.2 **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 its own separate address space in the image. If two program sections in different modules have the same name, the sections will be placed in separate yet contiguous address spaces. For example, if PSECTA in MODULE1 and PSECTA in MODULE2 have the concatenated attribute, PSECTA from MODULE1 will be allocated first, followed by PSECTA from MODULE2. The final total size of a concatenated program section is the sum of the individual contributions, plus 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 both PSECTA in MODULE1 and PSECTA in MODULE2 have the overlaid attribute, both program section contributions will be allocated starting at the same base address in the image. The final 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. In this situation, the order in which you specified the input modules is important, because the contents of an overlaid program section are determined by the last contributing module specified.

FORTRAN common areas are the most frequent use of overlaid program sections.

2.5.4.3 **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. This subject is discussed further in the treatments of shareable images and clustering (see Chapters 8 and 9).

FORTRAN common is implemented with global program sections.

2.5.4.4 **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. Another reason for this attribute is that it 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 separate image sections from program sections that are not executable.

2.5.4.5 **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.

2.5.4.6 **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.

2.5.4.7 **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.

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

2.5.4.9 **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 zero to guarantee future compatibility.

2.6 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.

2.6.1 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 same object modules can be linked in different combinations or with different link options to form different executable images.

2.6.2 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 more manageable pieces, rather than as one monolithic link

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 which 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 3.2.3).

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.

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 Library provided with the VAX/VMS system is a shareable image.
- You cannot link the VAX-11 Symbolic Debugger with a shareable image; you must include at least one object module in the link.

- 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 5 and 6 describe LINK command qualifiers and link time options specifically intended for dealing with shareable images. See the following:

/SYSSHR /SHAREABLE	qualifiers
UNIVERSAL= GSMATCH=	options

2.6.3 System Images

A system image is a special type of image intended for stand-alone operation on the 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 upon 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 2.5.

The resulting image is a fixed-length record file, each record being a 512-byte block. A system image has no image header, 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.

2.7 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 (concatenated versus overlaid, local versus global) and the alignment, as discussed in Section 2.5. 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 writeablity (WRT and NOWRT) and executability (EXE and NOEXE) attributes.

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

Then, 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 matching program section 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. The next cluster (if there is one) is then treated in the same way.

At this point, 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 is described in Chapter 9. However, the choice depends on whether you specified an address in the CLUSTER= option, and whether the cluster contains a shareable image. It also depends upon the order in which you specified the clusters.

2.8 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 which program section is being initialized and to which of the image sections that program section has been allocated. The first attempt to initialize a part of an image section causes the linker to allocate a buffer in its own program region to contain the generated image binary contents. 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.

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 6). 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 a single image section being replaced by a potentially large number of alternating demand zero and copy on reference image sections.

The linker continues this 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 6). 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 now 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 5).

CHAPTER 3

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.

3.1 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 coding statement or a data definition. Consider the following examples (not tied to a specific programming language):

- A coding statement identified as ROUTINEA moves FIELDA to FIELDB. ROUTINEA is the symbol associated with the coding 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.

3.2 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. 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).
- Universal symbols are a special type of global symbol. You can specify universal symbols only for shareable images.

Figure 3-1 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.

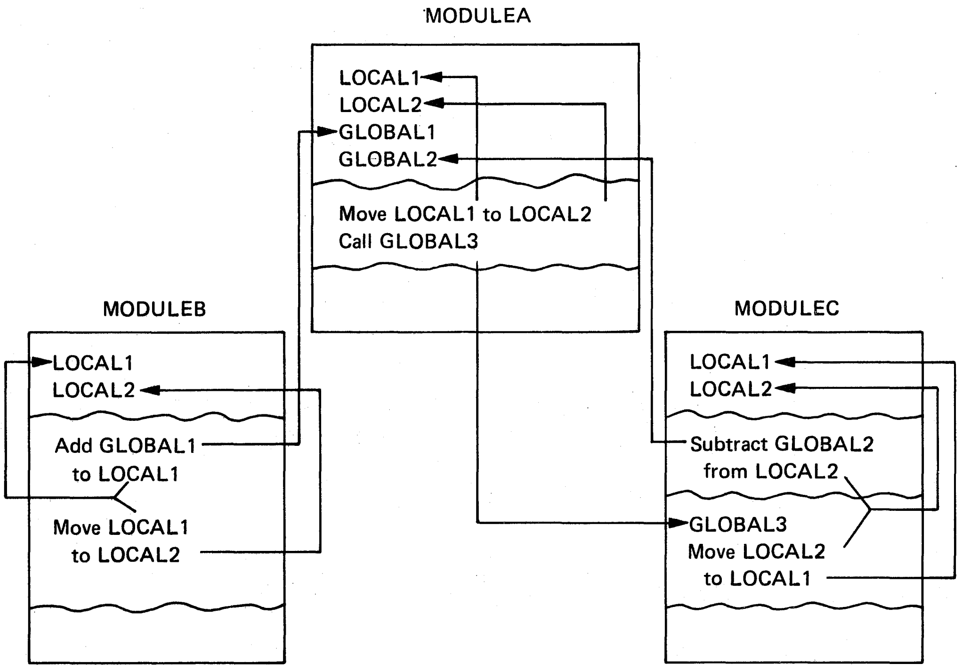


Figure 3-1 Local and Global Symbols

Local and global symbols can be designated either automatically by the language translator or 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 has a different meaning from the term in the context of a MACRO program (for example, 10$:).

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 module entry points and common areas 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 6 explains the use of the /OPTIONS qualifier with the LINK command.

3.2.1 Local Symbols

You can refer to local symbols only within the program 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.

3.2.2 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.

3.2.2.1 **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 4.

3.2.2.2 **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.

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

3.2.2.3 **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.

3.2.2.4 **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.

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.

3.2.3 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.

The VAX-11 MACRO assembler language 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 6). 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 the shareable image is linked with object files or other shareable images: for example, entry points of routines and perhaps some constants for defining possible errors.

3.3 SYMBOL TABLES

An image can have none, 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 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.

3.3.1 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. This makes the global symbols in the symbol table file and their values available to the object modules being linked, without also linking in the entire image with which the global symbols are associated. One primary use for specifying STB 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 4

LIBRARIES

The linker searches one or more libraries to resolve references to global symbols that are not defined in the object files specified previously in the LINK command. 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 also contain macros instead of object modules; however, the linker is not concerned with macro libraries.)

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 also explicitly include modules from a library in an image, thus eliminating the need for the linker to search the global symbol table of the library. In addition to any libraries that you specify, the linker automatically searches the default system library for any unresolved strong references.

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

4.1 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 address 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 4-1 shows the relationship of the module name table and the global symbol table to the rest of the library.

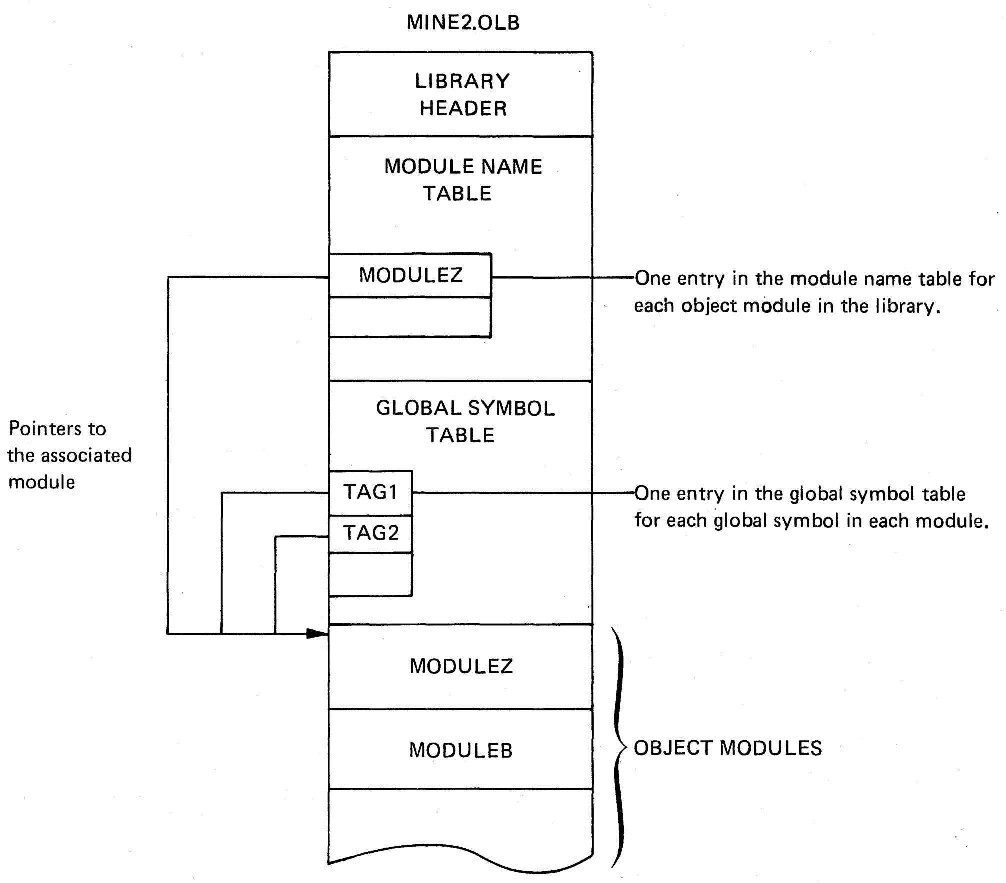

MINE2.OLB
LIBRARY HEADER
MODULE NAME TABLE
MODULEZ
One entry in the module name table for each object module in the library.
GLOBAL SYMBOL TABLE
Pointers to the associated module
TAG1
TAG2
One entry in the global symbol table for each global symbol in each module.
MODULEZ
MODULEB
OBJECT MODULES

Figure 4-1 Library Tables

4.2 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 default system library for unresolved strong references after it has processed all named input files, including user libraries. (See Section 4.3 for a discussion of the default system library.)

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 that the same symbol 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.

4.3 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.EXE and the other an object library called STARLET.OLB. Both files reside on the device and directory given by the translation of SYS$LIBRARY.

4.3.1 VMSRTL.EXE

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 Common Run-Time Procedure Library Reference Manual, including many routines required by almost all FORTRAN programs.

If the linker finds no symbols that it needs in the shareable image, it proceeds to search the object library STARLET and does not include the shareable image VMSRTL in the image being created.

You can use the /NOSYSSHR qualifier to the LINK command to suppress the linker's search of this shareable image (see Chapter 5).

4.3.2 STARLET.OLB

STARLET.OLB is an object module library in the form discussed in this chapter. It contains all of the object files that were used to create the shareable image version of the Run-Time Library, as well as many less frequently used procedures of the same class. This object library also contains modules for interfacing to VAX/VMS system services.

The linker searches SYS$LIBRARY:STARLET.OLB if any unresolved strong references remain after it has searched SYS$LIBRARY:VMSRTL.EXE.

You can use the /NOSYSLIB qualifier to the LINK command to suppress the linker's search of both STARLET.OLB and VMSRTL.EXE (see Chapter 5).

4.4 EXAMPLE OF USING LIBRARIES

The following example shows how you can specify both explicit and implicit inclusion of modules from libraries. (The file types need not be entered, but are included here for clarity.)

```
$ LINK LAUREL.OBJ,HARDY.OBJ,-
      MINE2.OLB/INCLUDE=MODULEZ,-
      MINE3.OLB/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 in 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 3.2.2.4.

S.L.

CHAPTER 5

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; however, it is not concerned with command syntax. Syntax deals with the rules for entering commands, such as how to specify a continuation line, or the number of characters you must enter before the command interpreter can recognize the entry. This chapter discusses matters of syntax only where necessary to avoid errors or misunderstanding, and uses spellings that most clearly suggest a qualifier's function. For detailed information on command syntax, see the VAX/VMS Command Language User's Guide.

5.1 COMMAND FORMAT

The LINK command has 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 normally separate the last command qualifier from the first input file specification. Commas precede the second and subsequent input file specifications.

The following examples show some acceptable formats of the LINK command (Section 5.3 explains these examples).

```
     $ LINK PROGA

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

     $ LINK/MAP/FULL/EXECUTABLE=STOOGES CURLY,-
           LARRY,MOE,TVLIB/INCLUDE=OLDIES,-
           GOODIES/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. 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.)

5.2 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 with incompatible qualifiers; depending on the specific case, it might display an error message and invalidate the entire LINK command, or it might ignore or override certain qualifiers (generally accepting only the last valid one) and allow 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 5-1 and 5-2 list the command and file qualifiers, the default value for each, and any incompatible qualifiers. A [NO] indicates that the qualifier can be negated by prefixing NO (without brackets) -- for example, /NODEBUG or /NOEXECUTABLE. Any entry after the qualifier is valid only for the positive form of the qualifier; for example, it would be nonsense to enter /NOEXECUTABLE=PAYROLL.

Table 5-1
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
/[NO]EXECUTABLE[=file-spec]	/EXECUTABLE	/SHAREABLE
/FULL	Default map	/NOMAP,/BRIEF
/[NO]MAP[=file-spec]	/NOMAP	
/[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	

Table 5-2
File Qualifiers

File Qualifier	Default	Incompatible Qualifiers
/INCLUDE=module-name[,...]	(Does not apply)	All others, except /LIBRARY
/LIBRARY	File is an object module.	All others, except /INCLUDE
/OPTIONS	File is an object module.	All others
/SELECTIVE_SEARCH	Include all module global symbols in the image's global symbol table.	All others, except /SHAREABLE
/SHAREABLE	File is an object module.	All others, except /SELECTIVE_SEARCH

Sections 5.2.1 and 5.2.2 discuss the command qualifiers and file qualifiers individually. Within each section the qualifiers are presented in alphabetical order.

5.2.1 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 specified /MAP previously in the LINK command. /BRIEF is incompatible with /FULL and /CROSS_REFERENCE.

/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 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 specified MAP previously 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.

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 4.2) 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 and the file type of EXE.

/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 Chapter 7.

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

/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. 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 Chapter 7.

If you do not specify /MAP, the default is /NOMAP; that is, the linker does not generate an image map.

/SHAREABLE[=file-spec]
/NOSHAREABLE

/SHAREABLE tells the linker to create a shareable image. (For an explanation of shareable images, see Section 2.6.2 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 specification of the first input file.

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 6.1.2.)

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

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.

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 3.3.1.

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. 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. 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.

/SYSSHR
/NOSYSSHR

/SYSSHR tells the linker to search the default system run time library shareable image (SYS$LIBRARY:VMSRTL.EXE). If any symbol within this 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 to express its negative form. /NOSYSSHR tells the linker not to try to resolve symbolic references by including the default system shareable image. Note, however, that /NOSYSSHR has no effect upon the search of the default system object library (SYS$LIBRARY:STARLET.OLB).

You might specify /NOSYSSHR, for example, when you need only one library routine for a particular program. Since the shareable image VMSRTL contains many routines, all of which would be mapped, it would be inefficient to include all the routines if you need only one. /NOSYSSHR directs the linker to use only the default object library, which includes all the routines found in VMSRTL.

/SYSTEM[=base-address]
/NOSYSTEM

/SYSTEM tells the linker to create a system image. (For an explanation of system images, see Section 2.6.3.) 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). If you specify /SYSTEM without a base address, the linker assumes %X80000000.

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 qualfier 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.

5.2.2 **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 4.

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

```
$ LINK TEAM,REDS/INCLUDE=(ROSE,MORGAN,BENCH)
```

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

```
$ LINK TEAM,REDS/LIBRARY/INCLUDE=(ROSE,MORGAN,BENCH)
```

This example also tells the linker to include ROSE, MORGAN, and BENCH in TEAM. However, the /LIBRARY qualifier tells the linker to search the rest of the library REDS and link in any other modules needed to resolve strong symbolic references in TEAM, ROSE, MORGAN, and BENCH.

/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 in the named input files and any library modules specified with the /INCLUDE qualifier. For further information on libraries, see Chapter 4.

/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 6 explains how to create an options file and what it can contain. Chapter 6 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

/SHAREABLE as an input file qualifier is valid only within a linker options file. Section 6.1.2 explains the use of the /SHAREABLE file qualifier.

5.3 EXAMPLES

1. `$ LINK PROGA`

   The linker binds the object module PROGA and creates an executable image named PROGA. The linker searches only 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 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,-
     GOODIES/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 GOODIES for any unresolved symbolic references in CURLY, LARRY, MOE, and OLDIES, before searching 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 6

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 standard command language

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

6.1 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

6.1.1 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 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 you have in an options file, the greater is the convenience of using it.

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
```

6.1.2 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 command interpreter 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.

6.1.3 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. The maximum number of entries depends on 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.

6.1.4 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 6-1 lists the special functions that you can request only in an options file, giving the following information for each: its format, the default value, and a brief explanation. Section 6.3 provides detailed explanations of each special function.

Table 6-1
Special Options

Format	Default	Explanation
BASE=n	%X200 for executable and shareable %X80000000 for system	Base virtual address for the image
CHANNELS=n	At least 32	Maximum number of I/O channels the image can use during execution
CLUSTER=cluster-name,- [base-address],- [pfc],file-spec[,...]	(See explanation in Section 6.3.)	Identifies a cluster
DZRO_MIN=n	5	Minimum number of initialized pages before compression can occur
GSMATCH=keyword,- major-id,minor-id	LEQUAL,0,0	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	Maximum number of image sections
STACK=n	20	Number of pages for the user mode stack
UNIVERSAL=symbol-name [,...]	Global symbol is not universal	Identifies a global symbol as universal

6.2 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 6-1, 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 6-1 for the available special options).

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 6.1.2).

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
CHANNELS=40 !THIS IS A COMMENT.
STACK=75
IOSEG=50
```

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 enter 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.

6.3 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, the default radix), hexadecimal (%X), or octal (%O). However, the default and maximum numeric values in this manual are usually 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.

The default number of channels is determined by the operating system, but it is at least 32. You cannot specify less than 32 or more than 64. If you specify from 0 to 32, the linker uses the default; and if you specify more than 64, the linker uses 64.

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

CLUSTER= defines a cluster. (Clusters are discussed in Chapters 2, 8, and 9.) 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 Chapter 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 Chapter 9.
- If you use the CLUSTER= option but do not specify a page fault cluster, VAX/VMS memory management determines the value.

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 the area does not occupy physical space in the image on disk; but 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 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 8.2.3.

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 is the default, since it 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 only by name 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 to 255.

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

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

```
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 would also want 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 compresson 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.

STACK=n

STACK= specifies the number of pages to be allocated for the image's user mode stack area.

The default value is 20. You may need to increase the stack size if the program fails to run using the default value -- for example, if the stack is used for temporary storage of data that exceeds 20 pages.

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 3.2.3.

CHAPTER 7

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.

The map is placed on your output disk and assigned a file type of MAP. You can specify a file name with the MAP qualifier, or you can let the VAX-11 software assign a default. You can print a copy of the map with the PRINT command.

To obtain a map, you must include the /MAP qualifier in the LINK command. 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 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

7.1 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 7-1 lists all the possible section names in the order in which they can 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 7.2.

Table 7-1
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.

7.2 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 illustrates the complete brief, default, and full forms of the map whose sections appear in this chapter.

7.2.1 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 7-1 illustrates the Object Module Synopsis section.

```
                                       +-------------------------+
                                       ! OBJECT MODULE SYNOPSIS !
                                       +-------------------------+

MODULE NAME    IDENT  BYTES      FILE                         CREATION DATE        CREATOR
-----------    -----  -----      ----                         -------------        -------
AVERAGE$MAIN   01       202   DB1:[MURRAY]AVERAGE.OBJ;2     11-May-1978  09:2   VAX-11 FORTRAN IV-PLUS T0.7-92
DEBUGBOOT      01         8   DBB2:[SYSLIB]DEBUG.OBJ;1      02-JUN-1978  10:2   VAX-11 MACRO X0.3-10
OTS$LINKAGE    0-3        3   DBB2:[SYSLIB]STARLET.OLB;2    15-JUN-1978  14:3   VAX-11 MACRO X0.3-11
SYSVECTOR      02         0   DBB2:[SYSLIB]STARLET.OLB;2    25-JUN-1978  15:2   VAX-11 MACRO X0.3-11
VMSRTL         .EXE;14    0   DBB2:[SYSLIB]VMSRTL.EXE;2     10-JUL-1978 00:21   LINK-32 X01.17
```

Figure 7-1 Object Module Synopsis

```
                                                +------------------------+
                                                ! IMAGE SECTION SYNOPSIS !
                                                +------------------------+

   CLUSTER        TYPE PAGES   BASE ADDR DISK VBN PFC PROTECTION AND PAGING      GBL. 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
                    0     1     00000800        5   0 READ WRITE   COPY ON REF
                  253    20     7FFFD800        0   0 READ WRITE DEMAND ZERO

VMSRTL              3     4     00000A00        0   0 READ ONLY                  VMSRTL_001      LESS/EQUAL          0         99
                    3    48     00001200        0   0 READ ONLY                  VMSRTL_002      LESS/EQUAL          0         99
                    4     2     00007200        0   0 READ WRITE   COPY ON REF   VMSRTL_003      LESS/EQUAL          0         99
```

Figure 7-2 Image Section Synopsis

7.2.2 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
- Type code (used internally by the linker)
- Number of pages
- Base virtual address within the image
- Base virtual block number within the image file on disk
- 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," "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 7-2 illustrates the Image Section Synopsis.

7.2.3 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 2.7.

The Program Section Synopsis 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; that is, the PSECT is normally one or the other. Table 7-2 lists the attribute abbreviations (in alphabetical order), their meanings, and any contrasting attributes. Section 2.5.4 explains the attributes.

Table 7-2
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
NOWRT	Not writeable	WRT
OVR	Overlaid	CON
PIC	Position independent code	NOPIC
RD	Readable	NORD
REL	Relocatable	ABS
SHR	Shareable	NOSHR
USR	User	LIB
WRT	Writeable	NOWRT

Figure 7-3 illustrates the Program Section Synopsis.

```
                                  +--------------------------+
                                  ! PROGRAM SECTION SYNOPSIS !
                                  +--------------------------+

P-SECT NAME    MODULE(S)         BASE     END           LENGTH         ALIGN                ATTRIBUTES
-----------    ---------         ----     ---           ------         -----                ----------

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

$LOCAL                         00000400 0000040B 0000000C (      12.) LONG 2   PIC,USR,CON,REL,LCL,NOSHR,NOEXE,  RD,  WRT
               AVERAGE$MAIN    00000400 0000040B 0000000C (      12.) LONG 2

$CODE                          00000600 00000689 0000008A (     138.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT
               AVERAGE$MAIN    00000600 00000689 0000008A (     138.) LONG 2

OTS$CODE                       0000068C 0000068E 00000003 (       3.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT
               OTS$LINKAGE     0000068C 0000068E 00000003 (       3.) LONG 2

. BLANK .                      00000800 00000807 00000008 (       8.) BYTE 0 NOPIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,  WRT
               DEBUGBOOT       00000800 00000807 00000008 (       8.) BYTE 0
               OTS$LINKAGE     00000808 00000808 00000000 (       0.) BYTE 0
               SYSVECTOR       00000808 00000808 00000000 (       0.) BYTE 0
```

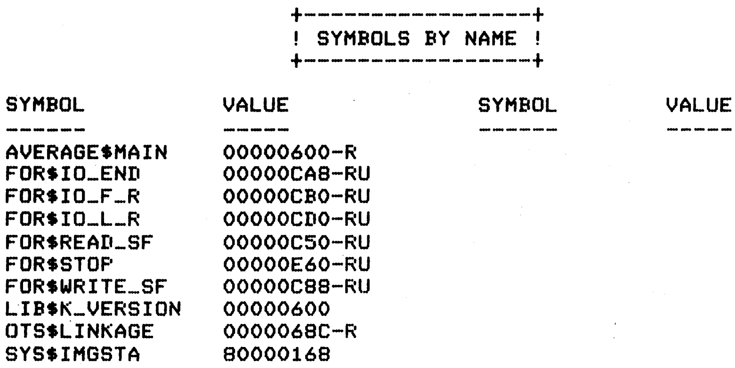
Figure 7-3 Program Section Synopsis

7.2.4 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, this section is replaced by the Symbol Cross Reference section.

Figure 7-4 illustrates the Symbols by Name section.



```
                        +-----------------+
                        ! SYMBOLS BY NAME !
                        +-----------------+

SYMBOL           VALUE                  SYMBOL          VALUE
------           -----                  ------          -----
AVERAGE$MAIN     00000600-R
FOR$IO_END       00000CA8-RU
FOR$IO_F_R       00000CB0-RU
FOR$IO_L_R       00000CD0-RU
FOR$READ_SF      00000C50-RU
FOR$STOP         00000E60-RU
FOR$WRITE_SF     00000C88-RU
LIB$K_VERSION    00000600
OTS$LINKAGE      0000068C-R
SYS$IMGSTA       80000168
```

Figure 7-4 Symbols by Name Section

7.2.5 Symbol Cross Reference

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

- Value in hexadecimal. 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.

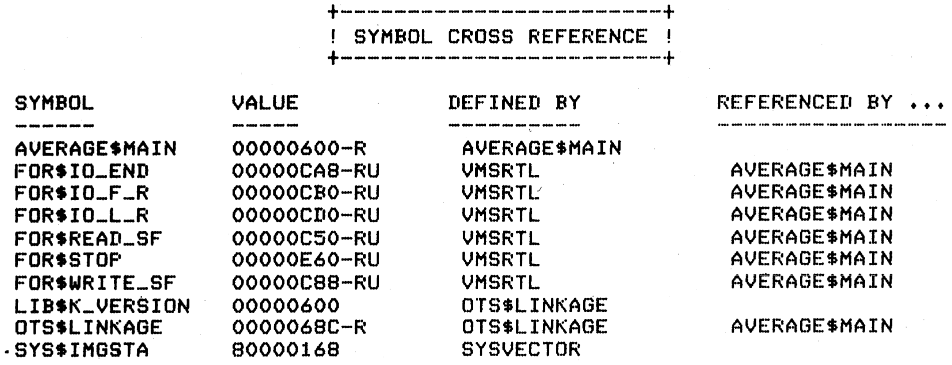
- Name of the first module that defines the symbol (blank if the symbol is undefined).

- 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 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 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 tells you where it is defined and what other modules may be affected by the change.

Figure 7-5 illustrates the Symbol Cross Reference section.



```
                         +-------------------------+
                         ! SYMBOL CROSS REFERENCE !
                         +-------------------------+

SYMBOL            VALUE            DEFINED BY           REFERENCED BY ...
------            -----            ----------           -----------------
AVERAGE$MAIN      00000600-R        AVERAGE$MAIN
FOR$IO_END        00000CA8-RU      VMSRTL                AVERAGE$MAIN
FOR$IO_F_R        00000CB0-RU      VMSRTL                AVERAGE$MAIN
FOR$IO_L_R        00000CD0-RU      VMSRTL                AVERAGE$MAIN
FOR$READ_SF       00000C50-RU      VMSRTL                AVERAGE$MAIN
FOR$STOP          00000E60-RU      VMSRTL                AVERAGE$MAIN
FOR$WRITE_SF      00000C88-RU      VMSRTL                AVERAGE$MAIN
LIB$K_VERSION     00000600          OTS$LINKAGE
OTS$LINKAGE       0000068C-R        OTS$LINKAGE          AVERAGE$MAIN
SYS$IMGSTA        80000168          SYSVECTOR
```

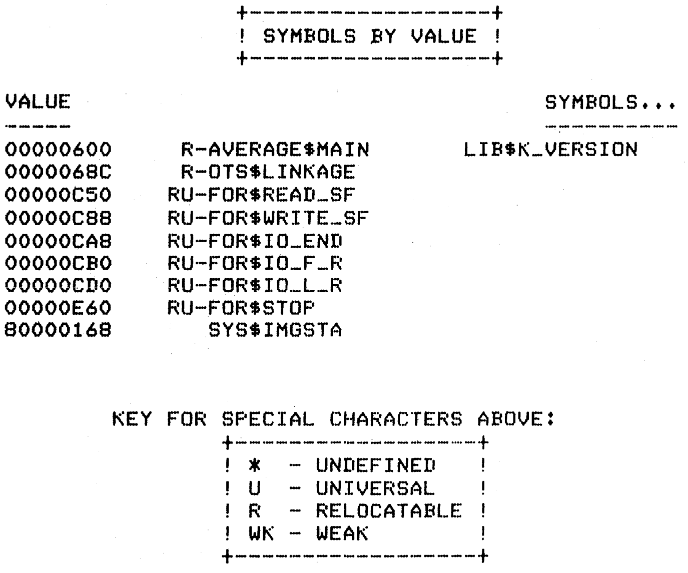
Figure 7-5 Symbol Cross Reference

7.2.6 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. An R- prefix to the symbol name indicates that the symbol is relocatable, and a U- prefix indicates that the symbol is universal.

This section appears only in a full image map.

Figure 7-6 illustrates the Symbols by Value section.



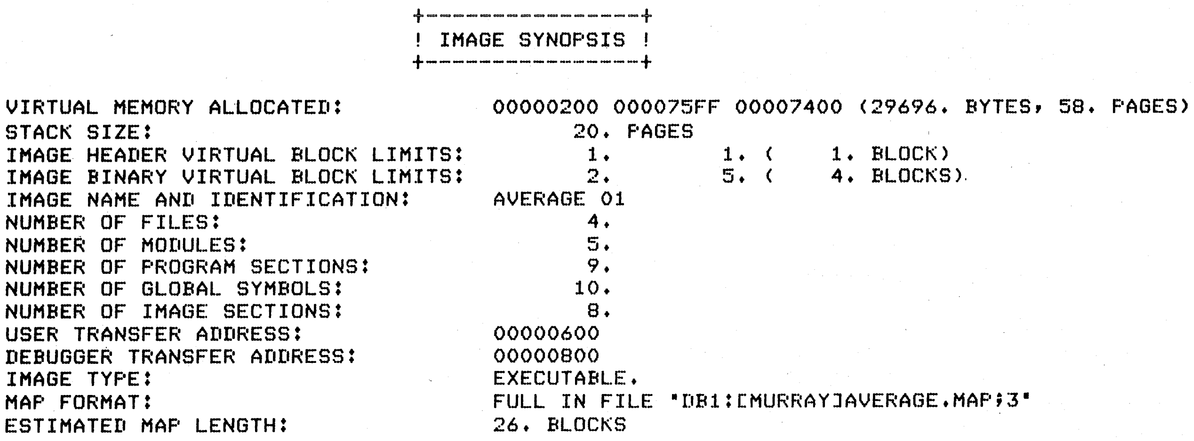
```
                  +------------------+
                  ! SYMBOLS BY VALUE !
                  +------------------+

VALUE                                       SYMBOLS...
-----                                       ----------
00000600      R-AVERAGE$MAIN          LIB$K_VERSION
0000068C      R-OTS$LINKAGE
00000C50     RU-FOR$READ_SF
00000C88     RU-FOR$WRITE_SF
00000CA8     RU-FOR$IO_END
00000CB0     RU-FOR$IO_F_R
00000CD0     RU-FOR$IO_L_R
00000E60     RU-FOR$STOP
80000168        SYS$IMGSTA


              KEY FOR SPECIAL CHARACTERS ABOVE:
                      +------------------+
                      ! *  - UNDEFINED   !
                      ! U  - UNIVERSAL   !
                      ! R  - RELOCATABLE !
                      ! WK - WEAK        !
                      +------------------+
```

Figure 7-6 Symbols By Value

7.2.7 Image Synopsis

The Image Synopsis, which appears in all maps, gives miscellaneous information about the output image. The items are self-explanatory. Numbers are decimal if they are followed by a point (.); otherwise, they are hexadecimal.

Figure 7-7 illustrates the Image Synopsis section.



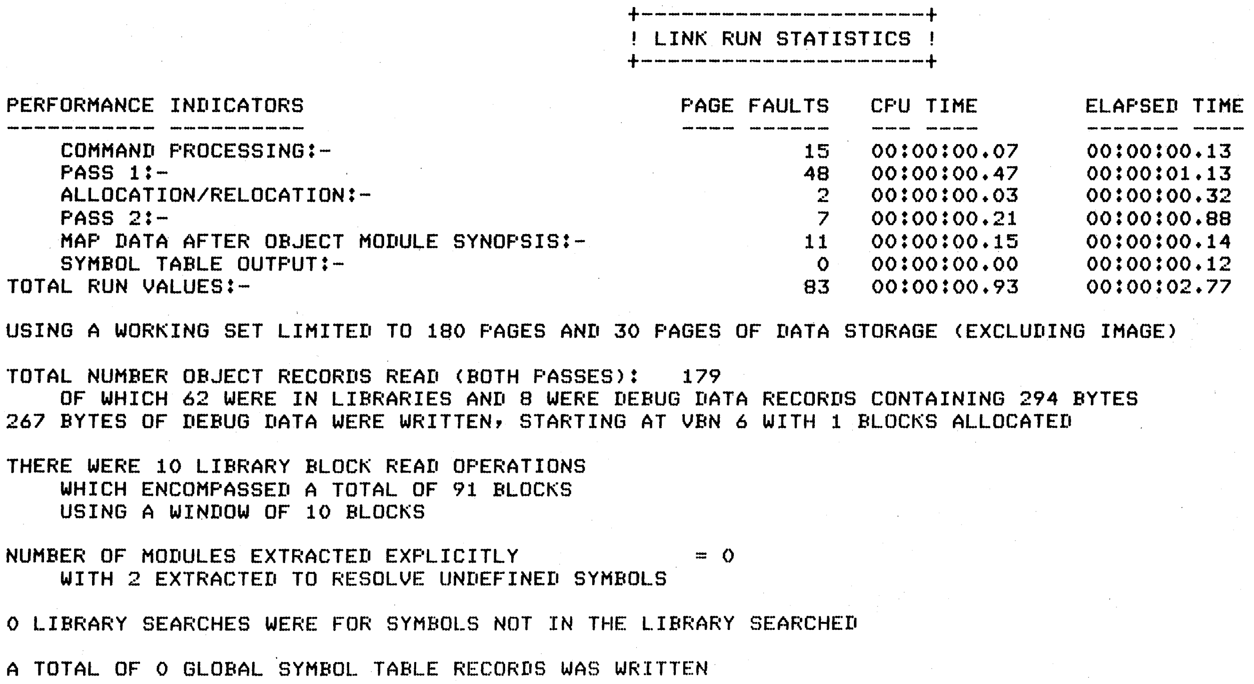
```
                                          +-----------------+
                                          ! IMAGE SYNOPSIS !
                                          +-----------------+

VIRTUAL MEMORY ALLOCATED:            00000200 000075FF 00007400 (29696. BYTES, 58. 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:       AVERAGE 01
NUMBER OF FILES:                            4.
NUMBER OF MODULES:                          5.
NUMBER OF PROGRAM SECTIONS:                 9.
NUMBER OF GLOBAL SYMBOLS:                  10.
NUMBER OF IMAGE SECTIONS:                   8.
USER TRANSFER ADDRESS:               00000600
DEBUGGER TRANSFER ADDRESS:           00000800
IMAGE TYPE:                          EXECUTABLE.
MAP FORMAT:                          FULL IN FILE "DB1:[MURRAY]AVERAGE.MAP;3"
ESTIMATED MAP LENGTH:                26. BLOCKS
```

Figure 7-7 Image Synopsis

7.2.8 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 7-8 illustrates the Link Run Statistics section.



```
                                                          +---------------------+
                                                          ! LINK RUN STATISTICS !
                                                          +---------------------+

PERFORMANCE INDICATORS                                       PAGE FAULTS   CPU TIME        ELAPSED TIME
----------- ----------                                       ---- ------   --- ----        ------- ----
    COMMAND PROCESSING:-                                              15   00:00:00.07     00:00:00.13
    PASS 1:-                                                          48   00:00:00.47     00:00:01.13
    ALLOCATION/RELOCATION:-                                            2   00:00:00.03     00:00:00.32
    PASS 2:-                                                           7   00:00:00.21     00:00:00.88
    MAP DATA AFTER OBJECT MODULE SYNOPSIS:-                           11   00:00:00.15     00:00:00.14
    SYMBOL TABLE OUTPUT:-                                              0   00:00:00.00     00:00:00.12
TOTAL RUN VALUES:-                                                    83   00:00:00.93     00:00:02.77

USING A WORKING SET LIMITED TO 180 PAGES AND 30 PAGES OF DATA STORAGE (EXCLUDING IMAGE)

TOTAL NUMBER OBJECT RECORDS READ (BOTH PASSES):   179
    OF WHICH 62 WERE IN LIBRARIES AND 8 WERE DEBUG DATA RECORDS CONTAINING 294 BYTES
267 BYTES OF DEBUG DATA WERE WRITTEN, STARTING AT VBN 6 WITH 1 BLOCKS ALLOCATED

THERE WERE 10 LIBRARY BLOCK READ OPERATIONS
    WHICH ENCOMPASSED A TOTAL OF 91 BLOCKS
    USING A WINDOW OF 10 BLOCKS

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
```

Figure 7-8 Link Run Statistics

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, and particularly with Chapter 2.

8.1 SHAREABLE IMAGES: BENEFITS AND USES

The following subsections expand on and add to the discussion in Section 2.6 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 that are 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 content of the shareable image. The installation of a shareable image causes the location of that image on disk to be recorded in the global section data base. The subsequent running of a program which 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 causes for a page to 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 central 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 for all images bound with it to be relinked.

8.2 CREATION OF SHAREABLE IMAGES

The previous section described some features of shareable images and some reasons for their development. This section deals with how to produce a shareable image.

8.2.1 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 5.

The UNIVERSAL= and GSMATCH= options are provided specifically to control characteristics of shareable images. Chapter 6 describes the syntax of these options. Sections 8.2.2 and 8.2.3 describe their purpose.

8.2.2 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. However, typically a very small set of the global symbols of the image are universal, since these are all that 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 building transfer vectors--see Section 8.2.4.) 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 8.2.8 contains an example showing the declaration of several such constants in the Cross Reference Facility as universal symbols.

8.2.3 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 uses the same algorithm when a shareable image is installed.) When the user image is activated, a search is made of the global section data base for each of the global sections described in the user image header.

Associated with the global section name, and forming a part of the name for the search, is a two-part identification field containing a major identification and a minor identification. 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 a control code which has the following possibilities:

- The minor identifications must match.

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

The GSMATCH= option is provided to set these parameters when the shareable image is being linked. See Chapter 6 for the format of the GSMATCH= option.

Another match control available with the GSMATCH= option is "NEVER". The purpose of this is to specify that the linker must always produce a private copy of the shareable image in each user image file.

8.2.4 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. This second location is the start of the instruction stream that is of actual interest. In the use of shareable images under VAX/VMS, such transfer vectors are normally displacements rather than actual 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.

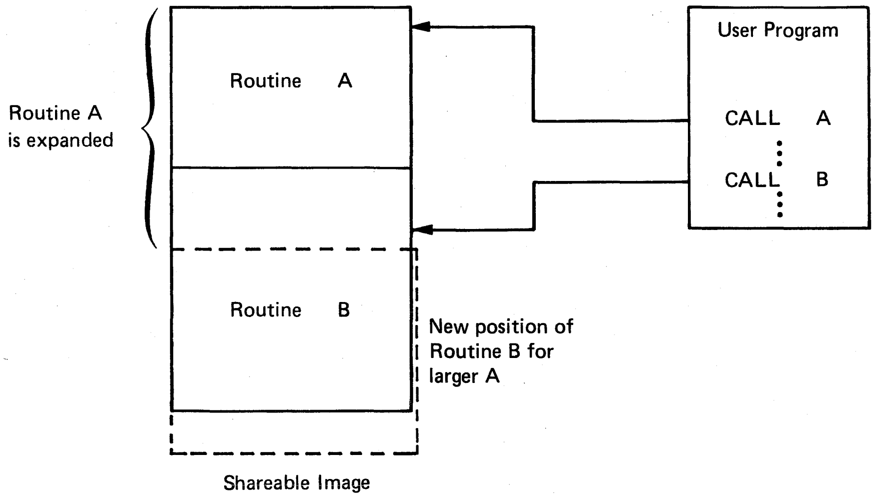


Figure 8-1 No Transfer Vectors

Using the example in Figure 8-1, assume that it becomes necessary to alter routine A, adding more code to it. When the shareable image is relinked, routine A will have the same address; but because it 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.

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

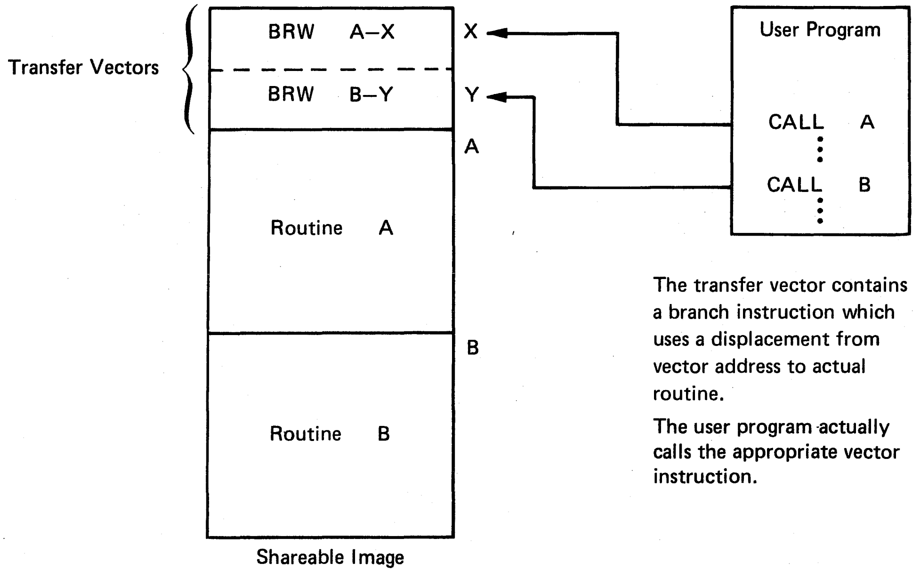

Transfer Vectors
BRW A–X
BRW B–Y
X
Y
A
B
Routine A
Routine B
Shareable Image
User Program
CALL A
CALL B
The transfer vector contains a branch instruction which uses a displacement from vector address to actual routine.
The user program actually calls the appropriate vector instruction.

Figure 8-2 Transfer Vectors

In the case of Figure 8-2, if routine A is expanded and the shareable image is relinked, the contents of the vector will change with no adverse effect 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 to be 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; for otherwise A or B may 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 build 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 the example above 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 by the call instruction. (You can achieve the same effect by 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 the 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.

8.2.5 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, say, a loop counter, each process must have a separate counter, or the routine cannot be shared simultaneously. Users familiar with this situation recognize this as part of the problem referred to as reentrancy.

It is for this situation that the shareable (SHR) attribute of program sections was introduced. As was mentioned in Chapter 2, the linker allocates program sections with the SHR attribute in separate image sections 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.)

A copy on reference image section is thus 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 shareablle 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 such applications are responsible for synchronizing access to such data. Note further that should it be necessary to revert to the initial values of such data, you must have made a separate copy before running the application the first time.

The FORTRAN example in Section 8.2.9 shows both of these 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.

8.2.6 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 that 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).
- 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 actual virtual address.
- All references to any other location inside or outside the image 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 of 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 separate image sections 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.

- 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 the first (lowest addressed) space sufficient to contain it when it is subsequently bound into a user image.

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

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 position independent code. Much of the code generated by the FORTRAN IV-PLUS compiler is position independent. However, if there are addresses in data regions (for example, precompiled argument lists), the VAX-11 FORTRAN IV-PLUS compiler indicates the existence of such NOPIC data, and the linker produces a NOPIC shareable image. The only problem area in MACRO assembler coding is the initalization of a data structure with an address; you are advised to use a self-relative technique in such cases.

8.2.7 Rules for Creating Upward-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.

8.2.8 Example of Transfer Vector and Universal Symbols

Figure 8-3 is a listing of the source for the module which 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 27 global symbols in the image, only 14 are of interest outside the image-- 3 vectored entry points and 11 constants. Note also that the transfer vector is placed in its own cluster. As you can see from the example, explicitly defined clusters are allocated first in the address space. The reason for putting the transfer vector in its own cluster is to ensure that it is allocated at the low-addressed end of the address space.

As was discussed in Section 8.2.4, 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.)

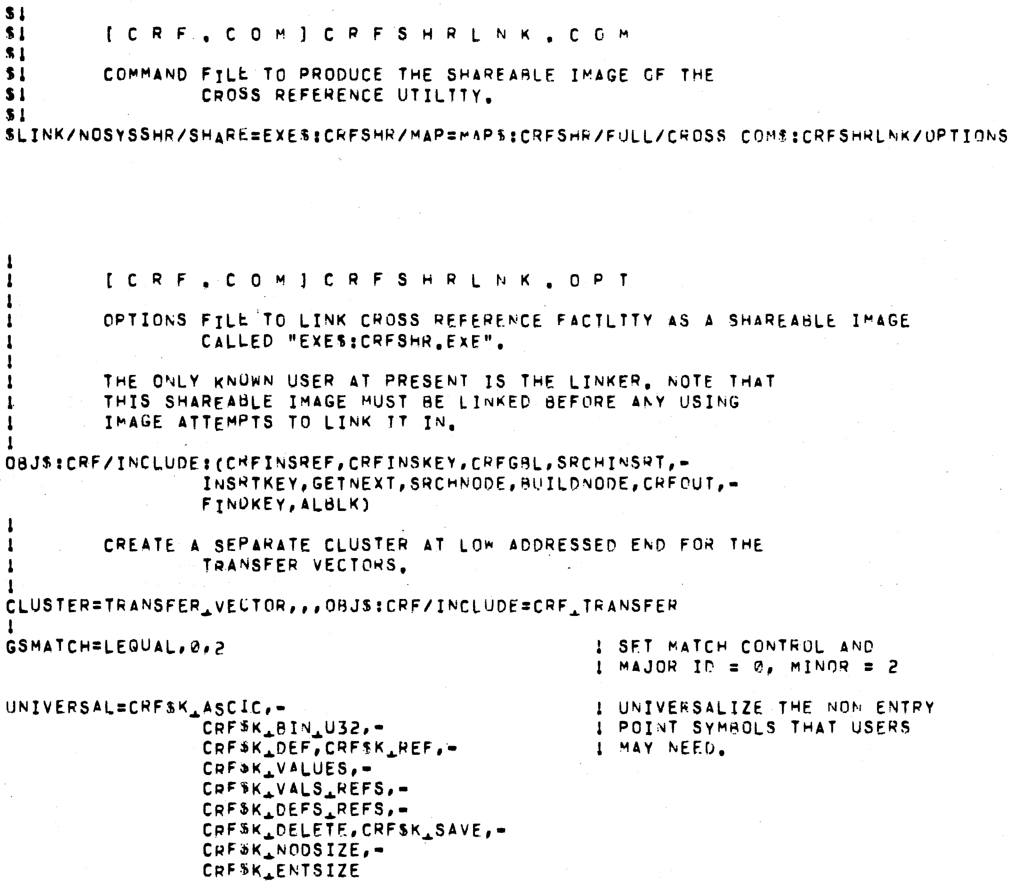
An example of copy on reference data (described in Section 8.2.5) is contained in the program section CRF$DATA.

```
                    0000              .SBTTL  TRANSFER_VECTORS
                    0000       ;++
                    0000       ; FUNCTIONAL DESCRIPTION:
                    0000       ;
                    0000       ; THIS MODULE DEFINES THE TRANSFER VECTORS FOR THE ENTRY POINTS CALLED
                    0000       ; BY A USER OF CRF.  THIS MODULE ENABLES CRF TO BE LINKED AS A SHARABLE IMAGE.
                    0000       ;
                    0000       ; CALLING SEQUENCE:
                    0000       ;
                    0000       ;      NONE
                    0000       ;
                    0000       ; INPUT PARAMETERS:
                    0000       ;
                    0000       ;      NONE
                    0000       ;
                    0000       ; IMPLICIT INPUTS:
                    0000       ;
                    0000       ;      NONE
                    0000       ;
                    0000       ; OUTPUT PARAMETERS:
                    0000       ;
                    0000       ;      NONE
                    0000       ;
                    0000       ; IMPLICIT OUTPUTS:
                    0000       ;
                    0000       ;      NONE
                    0000       ;
                    0000       ; COMPLETION CODES:
                    0000       ;
                    0000       ;      NONE
                    0000       ;
                    0000       ; SIDE EFFECTS:
                    0000       ;
                    0000       ;      NONE
                    0000       ;
                    0000       ;--
                    0000
                    0000
            00000000                  .PSECT  $$VECTOR_0_CRF,PIC,SHR,NOWRT,EXE
                    0000
                    0000              .TRANSFER       CRF$INSRTKEY     ; INSERTS A CROSS REFERENCE KEY
           0000'    0000              .MASK           CRF$INSRTKEY
    FFFD'  31       0002              BRW             CRF$INSRTKEY+2
                    0005
                    0005              .TRANSFER       CRF$INSRTREF     ; INSERTS A REFERENCE TO A KEY
           0000'    0005              .MASK           CRF$INSRTREF
    FFF8'  31       0007              BRW             CRF$INSRTREF+2
                    000A
                    000A              .TRANSFER       CRF$OUT          ; OUTPUTS CROSS REFERENCE SUMMARY
           0000'    000A              .MASK           CRF$OUT
    FFF3'  31       000C              BRW             CRF$OUT+2
                    000F
        00000200    000F              .BLKB   497                      ; ROOM FOR FUTURE ENTRY POINTS
                    0200              .END
```

Figure 8-3 Listing of CRF Transfer Vector

```
CRF$INSRTKEY       ********    X    02
CRF$INSRTREF       ********    X    02
CRF$OUT            ********    X    02


PROGRAM SECTION SYNOPSIS

.  ABS  .          00000000       00      NOPIC    USR    CON    ABS    LCL NOSHR NOEXE NORD   NOWRT BYTE
. BLANK .          00000000       01      NOPIC    USR    CON    REL    LCL NOSHR   EXE   RD     WRT BYTE
$$VECTOR_0_CRF     00000200       02        PIC    USR    CON    REL    LCL   SHR   EXE   RD   NOWRT BYTE


THERE WERE NO ERRORS OR WARNINGS.
28522. BYTES LEFT IN FREE MEMORY POOL.
OBJ$:CRFTFRVEC,LIS$:CRFTFRVEC/-SP=SRC$:CRFTFRVEC
0 MLB DIR RDS - 0 GETS TO DEFINE 0 MACROS. 1 INTER. FILE WRITES.
```

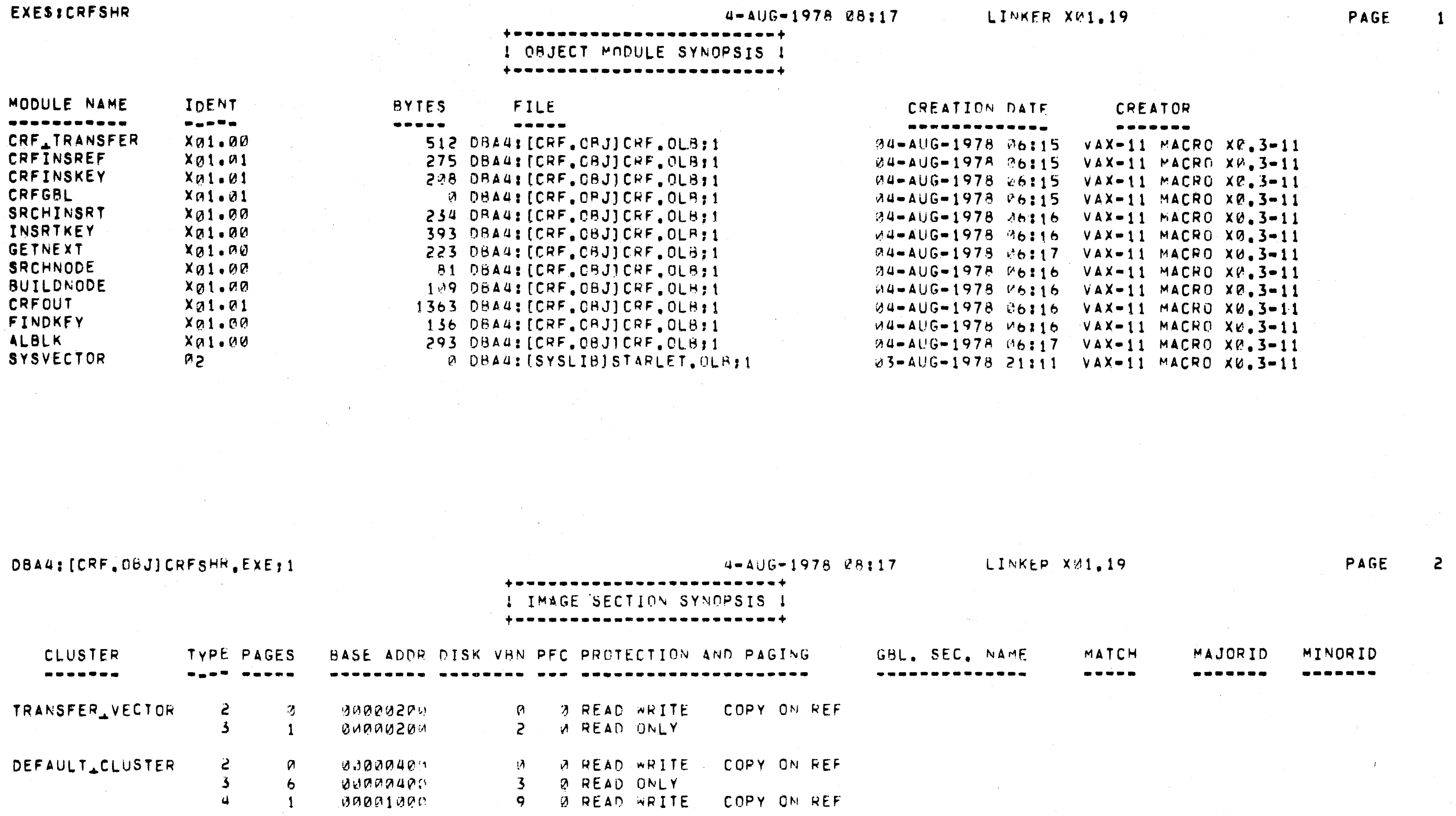
Figure 8-3 (Cont.) Listing of CRF Transfer Vector



```
$!
$!      [CRF.COM]CRFSHRLNK.COM
$!
$!      COMMAND FILE TO PRODUCE THE SHAREABLE IMAGE OF THE
$!              CROSS REFERENCE UTILTTY.
$!
$LINK/NOSYSSHR/SHARE=EXE$:CRFSHR/MAP=MAP$:CRFSHR/FULL/CROSS COM$:CRFSHRLNK/OPTIONS
```

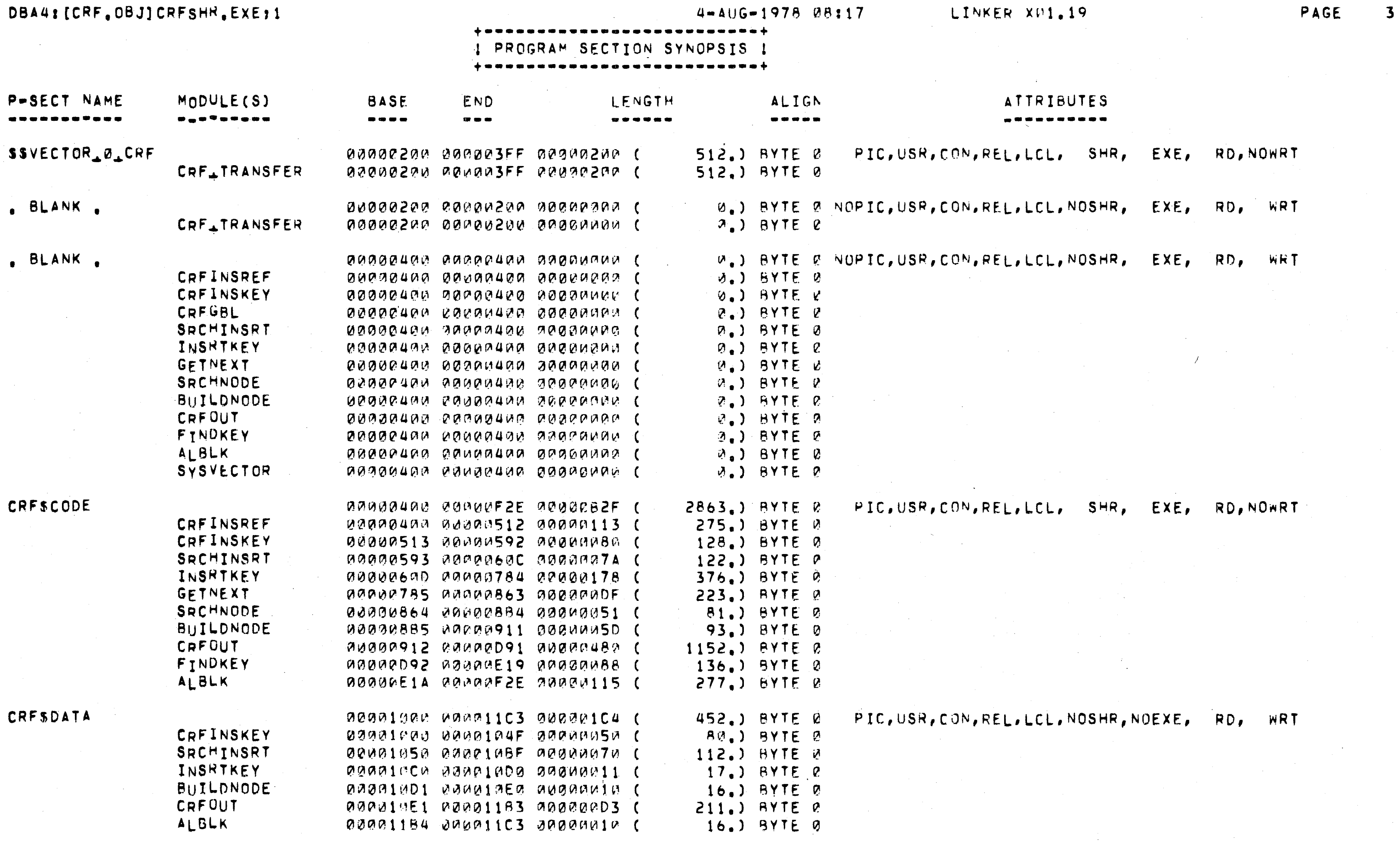
```
!
!       [CRF.COM]CRFSHRLNK.OPT
!
!       OPTIONS FILE TO LINK CROSS REFERENCE FACTLTTY AS A SHAREABLE IMAGE
!               CALLED "EXE$:CRFSHR.EXE".
!
!       THE ONLY KNOWN USER AT PRESENT IS THE LINKER. NOTE THAT
!       THIS SHAREABLE IMAGE MUST BE LINKED BEFORE ANY USING
!       IMAGE ATTEMPTS TO LINK IT IN.
!
OBJ$:CRF/INCLUDE:(CRFINSREF,CRFINSKEY,CRFGBL,SRCHINSRT,-
                INSRTKEY,GETNEXT,SRCHNODE,BUILDNODE,CRFOUT,-
                FINDKEY,ALBLK)
!
!       CREATE A SEPARATE CLUSTER AT LOW ADDRESSED END FOR THE
!               TRANSFER VECTORS.
!
CLUSTER=TRANSFER_VECTOR,,,OBJ$:CRF/INCLUDE=CRF_TRANSFER
!
GSMATCH=LEQUAL,0,2                                  ! SET MATCH CONTROL AND
                                                    ! MAJOR ID = 0, MINOR = 2

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$K_SAVE,-
                CRF$K_NODSIZE,-
                CRF$K_ENTSIZE
```

Figure 8-4 Command and Files to Create CRFSHR



```
EXES:CRFSHR                                                         4-AUG-1978 Ø8:17        LINKER XØ1.19                 PAGE    1
                                                  +-------------------------+
                                                  ! OBJECT MODULE SYNOPSIS !
                                                  +-------------------------+

MODULE NAME     IDENT              BYTES     FILE                                       CREATION DATE     CREATOR
-----------     -----              -----     -----                                      -------------     -------
CRF_TRANSFER    XØ1.ØØ               512 DBA4:[CRF.OBJ]CRF.OLB;1                   Ø4-AUG-1978 Ø6:15  VAX-11 MACRO XØ.3-11
CRFINSREF       XØ1.Ø1               275 DBA4:[CRF.OBJ]CRF.OLB;1                   Ø4-AUG-1978 Ø6:15  VAX-11 MACRO XØ.3-11
CRFINSKEY       XØ1.Ø1               2Ø8 DBA4:[CRF.OBJ]CRF.OLB;1                   Ø4-AUG-1978 Ø6:15  VAX-11 MACRO XØ.3-11
CRFGBL          XØ1.Ø1                 Ø DBA4:[CRF.OBJ]CRF.OLB;1                   Ø4-AUG-1978 Ø6:15  VAX-11 MACRO XØ.3-11
SRCHINSRT       XØ1.ØØ               234 DBA4:[CRF.OBJ]CRF.OLB;1                   Ø4-AUG-1978 Ø6:16  VAX-11 MACRO XØ.3-11
INSRTKEY        XØ1.ØØ               393 DBA4:[CRF.OBJ]CRF.OLB;1                   Ø4-AUG-1978 Ø6:16  VAX-11 MACRO XØ.3-11
GETNEXT         XØ1.ØØ               223 DBA4:[CRF.OBJ]CRF.OLB;1                   Ø4-AUG-1978 Ø6:17  VAX-11 MACRO XØ.3-11
SRCHNODE        XØ1.ØØ                81 DBA4:[CRF.OBJ]CRF.OLB;1                   Ø4-AUG-1978 Ø6:16  VAX-11 MACRO XØ.3-11
BUILDNODE       XØ1.ØØ               1Ø9 DBA4:[CRF.OBJ]CRF.OLB;1                   Ø4-AUG-1978 Ø6:16  VAX-11 MACRO XØ.3-11
CRFOUT          XØ1.Ø1              1363 DBA4:[CRF.OBJ]CRF.OLB;1                   Ø4-AUG-1978 Ø6:16  VAX-11 MACRO XØ.3-11
FINDKFY         XØ1.ØØ               136 DBA4:[CRF.OBJ]CRF.OLB;1                   Ø4-AUG-1978 Ø6:16  VAX-11 MACRO XØ.3-11
ALBLK           XØ1.ØØ               293 DBA4:[CRF.OBJ]CRF.OLB;1                   Ø4-AUG-1978 Ø6:17  VAX-11 MACRO XØ.3-11
SYSVECTOR       Ø2                     Ø DBA4:[SYSLIB]STARLET.OLB;1                Ø3-AUG-1978 21:11  VAX-11 MACRO XØ.3-11



DBA4:[CRF.OBJ]CRFSHR.EXE;1                                          4-AUG-1978 Ø8:17        LINKER XØ1.19                 PAGE    2
                                                  +-------------------------+
                                                  ! IMAGE SECTION SYNOPSIS !
                                                  +-------------------------+

  CLUSTER        TYPE PAGES  BASE ADDR DISK VBN PFC PROTECTION AND PAGING      GBL. SEC. NAME    MATCH    MAJORID  MINORID
  -------        ---- -----  --------- -------- --- ---------------------      --------------    -----    -------  -------

TRANSFER_VECTOR    2     Ø    ØØØØØ2ØØ       Ø   Ø READ WRITE   COPY ON REF
                   3     1    ØØØØØ2ØØ       2   Ø READ ONLY

DEFAULT_CLUSTER    2     Ø    ØØØØØ4ØØ       Ø   Ø READ WRITE   COPY ON REF
                   3     6    ØØØØØ4ØØ       3   Ø READ ONLY
                   4     1    ØØØØ1ØØØ       9   Ø READ WRITE   COPY ON REF
```

Figure 8-5 Map of CRFSHR

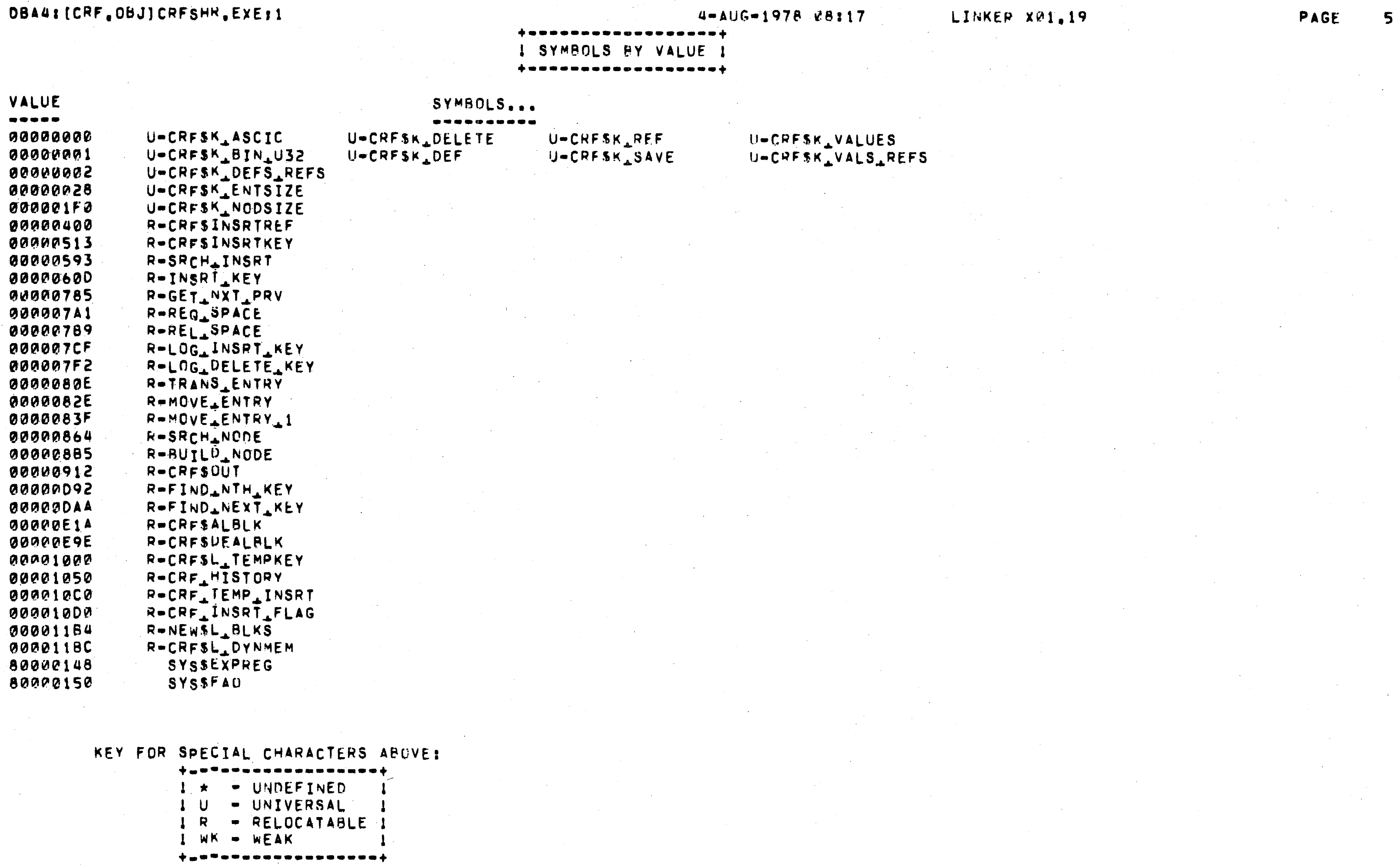


```
DBA4:[CRF.OBJ]CRFSHR.EXE;1                                 4-AUG-1978 08:17        LINKER X01.19               PAGE    3
                                                +----------------------------+
                                                ! PROGRAM SECTION SYNOPSIS !
                                                +----------------------------+

P-SECT NAME      MODULE(S)          BASE      END              LENGTH          ALIGN                ATTRIBUTES
-----------      ---------          ----      ---              ------          -----                ----------

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

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

. BLANK .                           00000400 00000400 00000000 (       0.) BYTE 0 NOPIC,USR,CON,REL,LCL,NOSHR,   EXE,   RD,   WRT
                 CRFINSREF          00000400 00000400 00000000 (       0.) BYTE 0
                 CRFINSKEY          00000400 00000400 00000000 (       0.) BYTE 0
                 CRFGBL             00000400 00000400 00000000 (       0.) BYTE 0
                 SRCHINSRT          00000400 00000400 00000000 (       0.) BYTE 0
                 INSRTKEY           00000400 00000400 00000000 (       0.) BYTE 0
                 GETNEXT            00000400 00000400 00000000 (       0.) BYTE 0
                 SRCHNODE           00000400 00000400 00000000 (       0.) BYTE 0
                 BUILDNODE          00000400 00000400 00000000 (       0.) BYTE 0
                 CRFOUT             00000400 00000400 00000000 (       0.) BYTE 0
                 FINDKEY            00000400 00000400 00000000 (       0.) BYTE 0
                 ALBLK              00000400 00000400 00000000 (       0.) BYTE 0
                 SYSVECTOR          00000400 00000400 00000000 (       0.) BYTE 0

CRF$CODE                            00000400 00000F2E 00000B2F (    2863.) BYTE 0   PIC,USR,CON,REL,LCL,  SHR,   EXE,   RD,NOWRT
                 CRFINSREF          00000400 00000512 00000113 (     275.) BYTE 0
                 CRFINSKEY          00000513 00000592 00000080 (     128.) BYTE 0
                 SRCHINSRT          00000593 0000060C 0000007A (     122.) BYTE 0
                 INSRTKEY           0000060D 00000784 00000178 (     376.) BYTE 0
                 GETNEXT            00000785 00000863 000000DF (     223.) BYTE 0
                 SRCHNODE           00000864 000008B4 00000051 (      81.) BYTE 0
                 BUILDNODE          00000885 00000911 0000005D (      93.) BYTE 0
                 CRFOUT             00000912 00000D91 00000480 (    1152.) BYTE 0
                 FINDKEY            00000D92 00000E19 00000088 (     136.) BYTE 0
                 ALBLK              00000E1A 00000F2E 00000115 (     277.) BYTE 0

CRF$DATA                            00001000 000011C3 000001C4 (     452.) BYTE 0   PIC,USR,CON,REL,LCL,NOSHR,NOEXE,  RD,   WRT
                 CRFINSKEY          00001000 0000104F 00000050 (      80.) BYTE 0
                 SRCHINSRT          00001050 000010BF 00000070 (     112.) BYTE 0
                 INSRTKEY           000010C0 000010D0 00000011 (      17.) BYTE 0
                 BUILDNODE          000010D1 000010E0 00000010 (      16.) BYTE 0
                 CRFOUT             000010E1 000011B3 000000D3 (     211.) BYTE 0
                 ALBLK              000011B4 000011C3 00000010 (      16.) BYTE 0
```

Figure 8-5 (Cont.) Map of CRFSHR

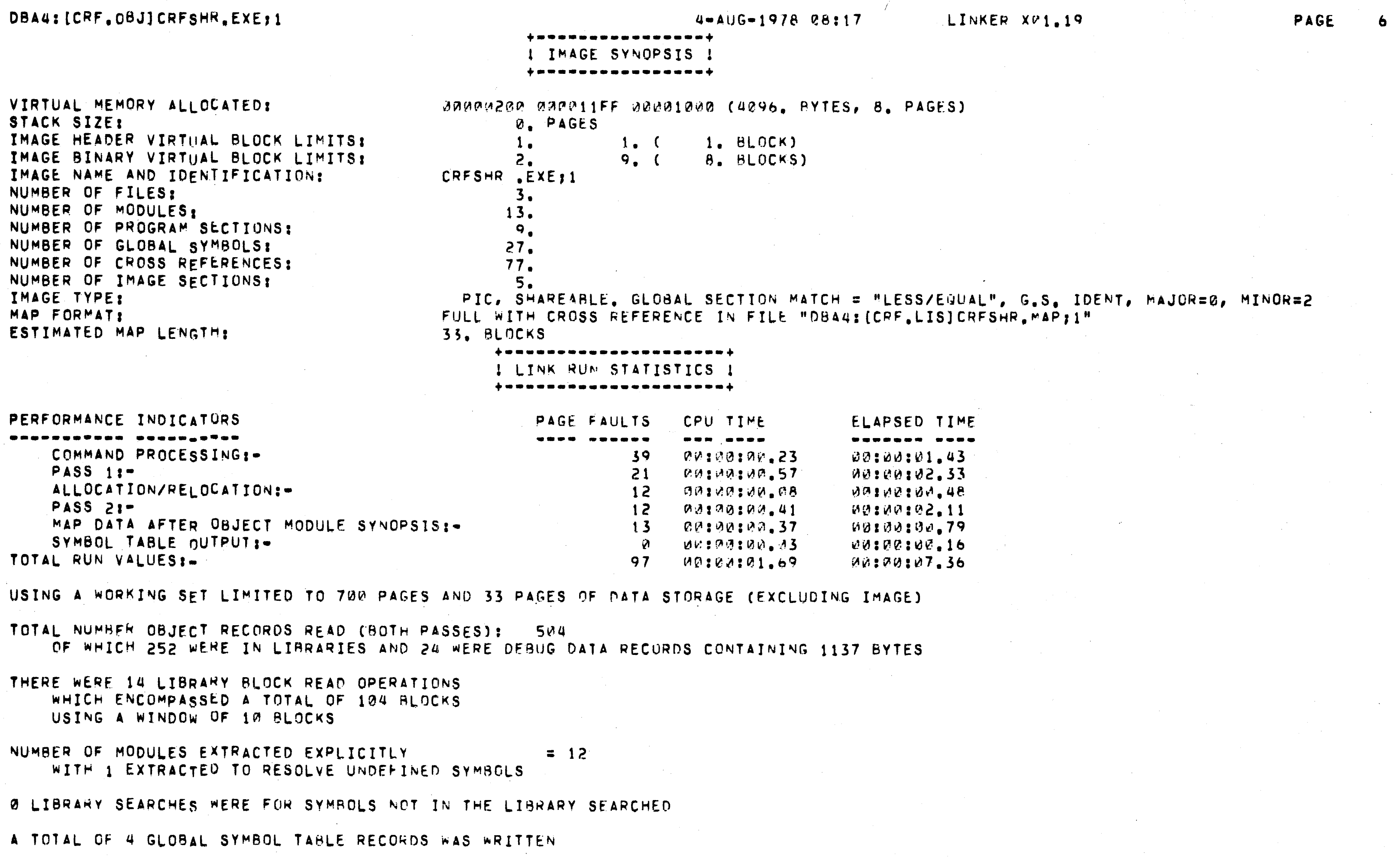
```
DBA4:[CRF.OBJ]CRFSHR.EXE;1                                       4-AUG-1978 08:17        LINKER X01.19                    PAGE    4
                                             +------------------------+
                                             ! SYMBOL CROSS REFERENCE !
                                             +------------------------+

SYMBOL          VALUE            DEFINED BY             REFERENCED BY ...
------          ----             ----------             -----------------
BUILD_NODE      00000885-R        BUILDNODE              INSRTKEY
CRF$ALBLK       00000E1A-R        ALBLK                  BUILDNODE             CRFINSREF             CRFOUT
CRF$DEALBLK     00000E9E-R        ALBLK                  CRFINSREF             CRFOUT
CRF$INSRTKEY    00000513-RU       CRFINSKEY              CRF_TRANSFER
CRF$INSRTREF    00000400-RU       CRFINSREF              CRF_TRANSFER
CRF$K_ASCIC     00000000-U        CRFGBL                 BUILDNODF             SRCHNODE
CRF$K_BIN_U32   00000001-U        CRFGBL
CRF$K_DEF       00000001-U        CRFGBL
CRF$K_DEFS_REFS 00000002-U        CRFGBL                 CRFOUT
CRF$K_DELETE    00000000-U        CRFGBL
CRF$K_ENTSIZE   00000028-U        CRFGBL
CRF$K_NODSIZE   000001F0-U        CRFGBL
CRF$K_REF       00000000-U        CRFGBL
CRF$K_SAVE      00000001-U        CRFGBL                 CRFOUT
CRF$K_VALS_REFS 00000001-U        CRFGBL
CRF$K_VALUES    00000000-U        CRFGBL                 CRFOUT
CRF$L_DYNMEM    000011BC-R        ALBLK
CRF$L_TEMPKEY   00001000-R        CRFINSKEY              CRFINSREF
CRF$OUT         00000912-RU       CRFOUT                 CRF_TRANSFER
CRF_HISTORY     00001050-R        SRCHINSRT              CRFINSKEY             CRFINSREF             CRFOUT
CRF_INSRT_FLAG  000010D0-R        INSRTKEY               CRFOUT                FINDKEY               GETNEXT
CRF_TEMP_INSRT  000010C0-R        INSRTKEY               GETNEXT
FIND_NEXT_KEY   00000DAA-R        FINDKEY                CRFOUT
FIND_NTH_KEY    00000D92-R        FINDKEY                CRFOUT
GET_NXT_PRV     00000785-R        GETNEXT                CRFOUT                FINDKEY               INSRTKEY
INSRT_KEY       0000060D-R        INSRTKEY               CRFINSKEY             CRFINSREF
LOG_DELETE_KEY  000007F2-R        GETNEXT
LOG_INSRT_KEY   000007CF-R        GETNEXT                INSRTKEY
MOVE_ENTRY      0000082E-R        GETNEXT                INSRTKEY
MOVE_ENTRY_1    0000083F-R        GETNEXT                INSRTKEY
NEW$L_BLKS      000011B4-R        ALBLK
REL_SPACE       00000789-R        GETNEXT
REQ_SPACE       000007A1-R        GETNEXT                INSRTKEY
SRCH_INSRT      00000593-R        SRCHINSRT              CRFINSKEY             CRFINSREF
SRCH_NODE       00000864-R        SRCHNODE               SRCHINSRT
SYS$EXPREG      80000148          SYSVECTOR              ALBLK
SYS$FAO         80000150          SYSVECTOR              CRFOUT
TRANS_ENTRY     0000080E-R        GETNEXT                INSRTKEY
```

Figure 8-5 (Cont.) Map of CRFSHR



```
DBA4:[CRF.OBJ]CRFSHR.EXE;1                                   4-AUG-1978 08:17        LINKER X01.19                  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$K_DEFS_REFS
00000028     U-CRF$K_ENTSIZE
000001F0     U-CRF$K_NODSIZE
00000400     R-CRF$INSRTREF
00000513     R-CRF$INSRTKEY
00000593     R-SRCH_INSRT
0000060D     R-INSRT_KEY
00000785     R-GET_NXT_PRV
000007A1     R-REQ_SPACE
000007B9     R-REL_SPACE
000007CF     R-LOG_INSRT_KEY
000007F2     R-LOG_DELETE_KEY
0000080E     R-TRANS_ENTRY
0000082E     R-MOVE_ENTRY
0000083F     R-MOVE_ENTRY_1
00000864     R-SRCH_NODE
00000885     R-BUILD_NODE
00000912     R-CRF$OUT
00000D92     R-FIND_NTH_KEY
00000DAA     R-FIND_NEXT_KEY
00000E1A     R-CRF$ALBLK
00000E9E     R-CRF$DEALBLK
00001000     R-CRF$L_TEMPKEY
00001050     R-CRF_HISTORY
000010C0     R-CRF_TEMP_INSRT
000010D0     R-CRF_INSRT_FLAG
000011B4     R-NEW$L_BLKS
000011BC     R-CRF$L_DYNMEM
80000148       SYS$EXPREG
80000150       SYS$FAO


        KEY FOR SPECIAL CHARACTERS ABOVE:
                +------------------+
                ! *  - UNDEFINED   !
                ! U  - UNIVERSAL   !
                ! R  - RELOCATABLE !
                ! WK - WEAK        !
                +------------------+
```

Figure 8-5 (Cont.) Map of CRFSHR



```
DBA4:[CRF.OBJ]CRFSHR.EXE;1                                              4-AUG-1978 08:17        LINKER X01.19                   PAGE    6
                                                     +-----------------+
                                                     ! IMAGE SYNOPSIS !
                                                     +-----------------+

VIRTUAL MEMORY ALLOCATED:                     00000200 000011FF 00001000 (4096. BYTES, 8. PAGES)
STACK SIZE:                                           0. PAGES
IMAGE HEADER VIRTUAL BLOCK LIMITS:                    1.          1. (     1. BLOCK)
IMAGE BINARY VIRTUAL BLOCK LIMITS:                    2.          9. (     8. BLOCKS)
IMAGE NAME AND IDENTIFICATION:                CRFSHR .EXE;1
NUMBER OF FILES:                                      3.
NUMBER OF MODULES:                                   13.
NUMBER OF PROGRAM SECTIONS:                           9.
NUMBER OF GLOBAL SYMBOLS:                            27.
NUMBER OF CROSS REFERENCES:                          77.
NUMBER OF IMAGE SECTIONS:                             5.
IMAGE TYPE:                                     PIC, SHAREABLE, GLOBAL SECTION MATCH = "LESS/EQUAL", G.S. IDENT, MAJOR=0, MINOR=2
MAP FORMAT:                                   FULL WITH CROSS REFERENCE IN FILE "DBA4:[CRF.LIS]CRFSHR.MAP;1"
ESTIMATED MAP LENGTH:                         33. BLOCKS
                                                   +----------------------+
                                                   ! LINK RUN STATISTICS !
                                                   +----------------------+

PERFORMANCE INDICATORS                                      PAGE FAULTS   CPU TIME         ELAPSED TIME
----------- ----------                                      ---- ------   --- ----         ------- ----
    COMMAND PROCESSING:-                                             39   00:00:00.23      00:00:01.43
    PASS 1:-                                                         21   00:00:00.57      00:00:02.33
    ALLOCATION/RELOCATION:-                                          12   00:00:00.08      00:00:00.48
    PASS 2:-                                                         12   00:00:00.41      00:00:02.11
    MAP DATA AFTER OBJECT MODULE SYNOPSIS:-                          13   00:00:00.37      00:00:00.79
    SYMBOL TABLE OUTPUT:-                                             0   00:00:00.03      00:00:00.16
TOTAL RUN VALUES:-                                                   97   00:00:01.69      00:00:07.36

USING A WORKING SET LIMITED TO 700 PAGES AND 33 PAGES OF DATA STORAGE (EXCLUDING IMAGE)

TOTAL NUMBER OBJECT RECORDS READ (BOTH PASSES):   504
    OF WHICH 252 WERE IN LIBRARIES AND 24 WERE DEBUG DATA RECORDS CONTAINING 1137 BYTES

THERE WERE 14 LIBRARY BLOCK READ OPERATIONS
    WHICH ENCOMPASSED A TOTAL OF 104 BLOCKS
    USING A WINDOW OF 10 BLOCKS

NUMBER OF MODULES EXTRACTED EXPLICITLY              = 12
    WITH 1 EXTRACTED TO RESOLVE UNDEFINED SYMBOLS

0 LIBRARY SEARCHES WERE FOR SYMBOLS NOT IN THE LIBRARY SEARCHED

A TOTAL OF 4 GLOBAL SYMBOL TABLE RECORDS WAS WRITTEN
```

Figure 8-5 (Cont.) Map of CRFSHR

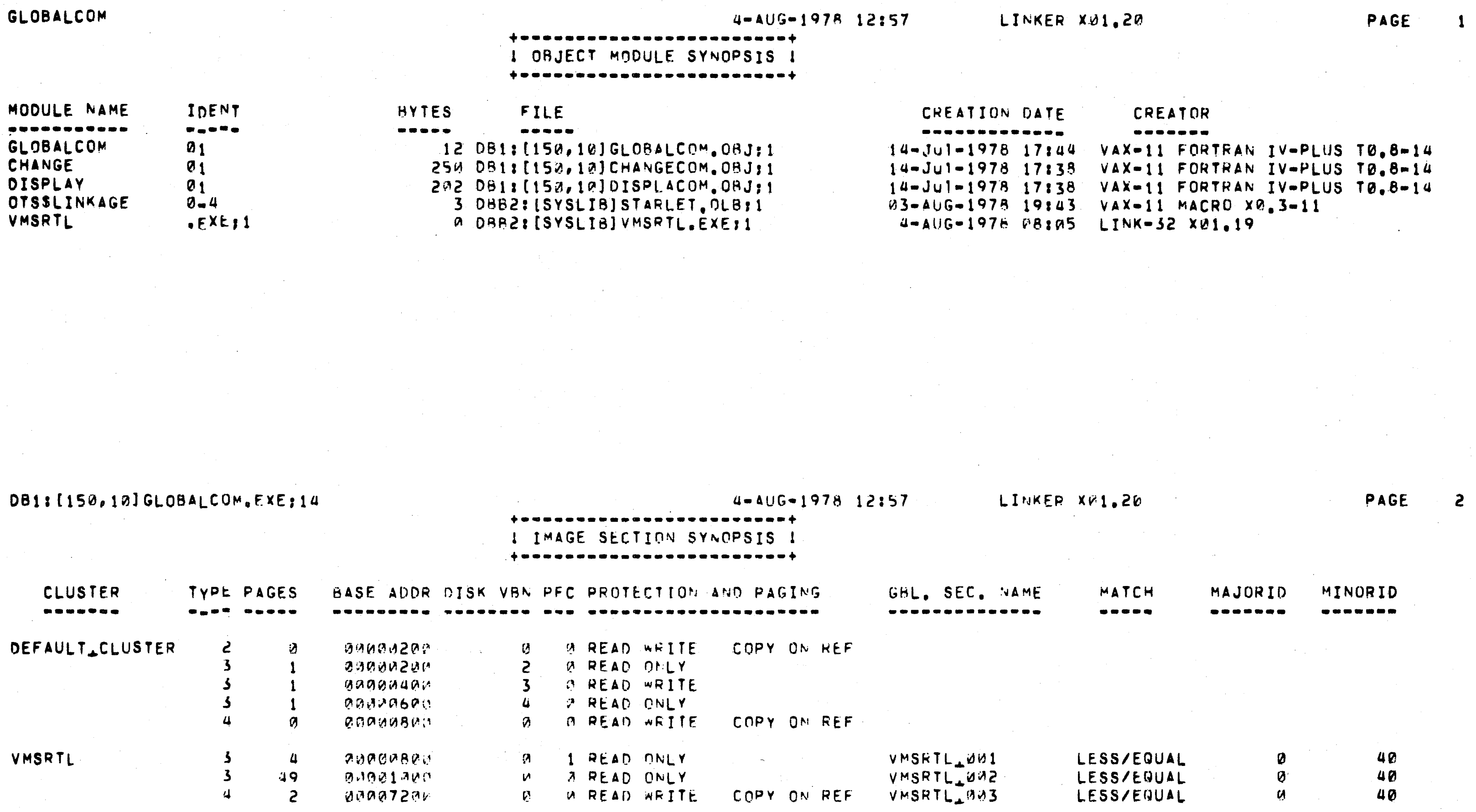
8.2.9 Example of FORTRAN Shared COMMON

Figure 8-6 shows a global common (FORTRAN BLOCKDATA subprogram) linked with a routine that modifies it (CHANGE) and one that displays its contents (DISPLAY). There are actually three common areas, shown by the program sections $BLANK, NAMEDCOMN1, and NAMEDCOMN2, which correspond to blank common of FORTRAN and two named common areas. Note the attributes of such program sections-- in particular, GBL, OVR, and SHR:

- The GBL attribute causes the program section to be recorded in the symbol table of this shareable image for later use by a subsequent program.

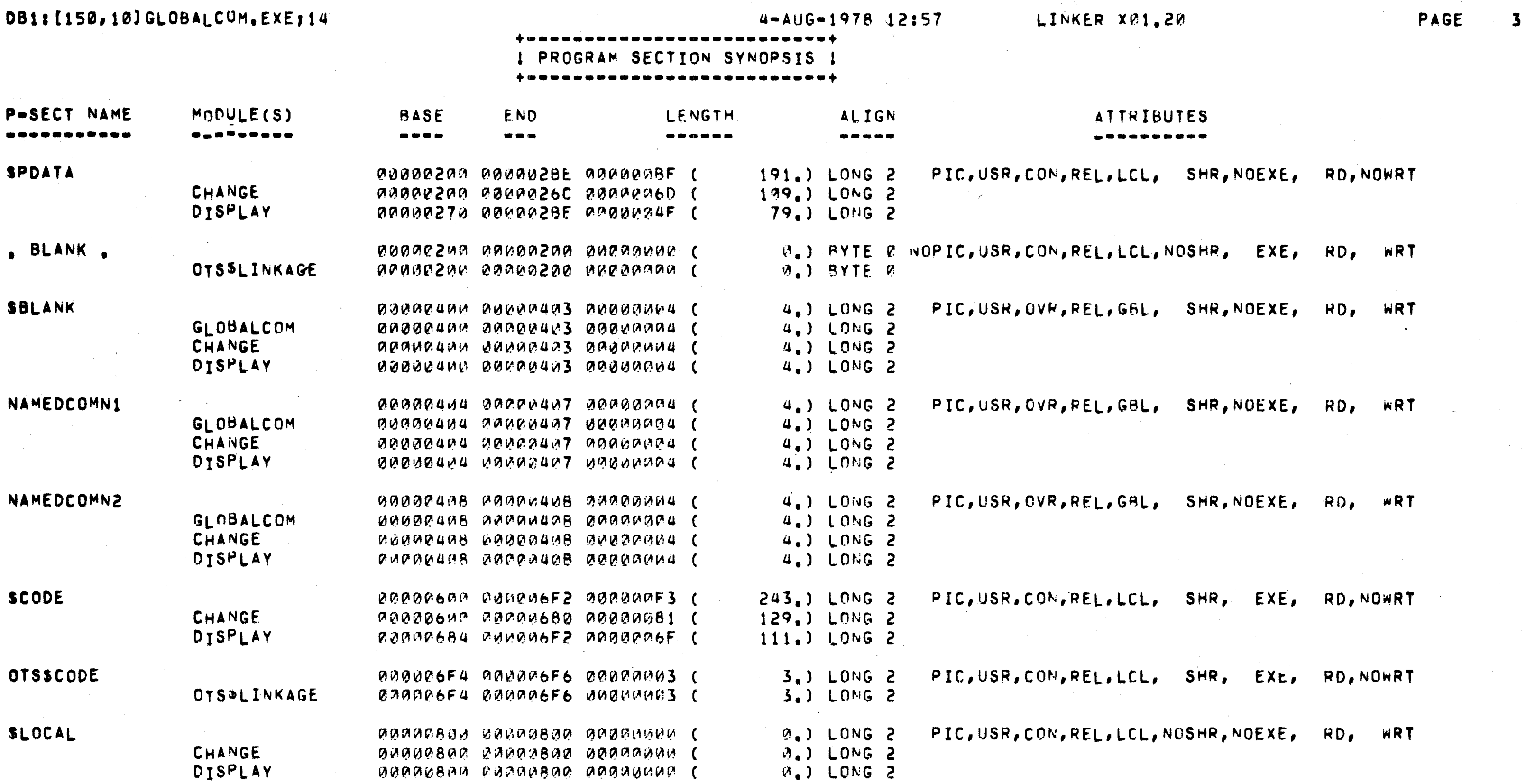
- The OVR attribute ensures that all modules contributing to the program section contribute (or in this case, map) to the same address space.

- The SHR attribute indicates that only one copy of this writeable data is to appear in memory.



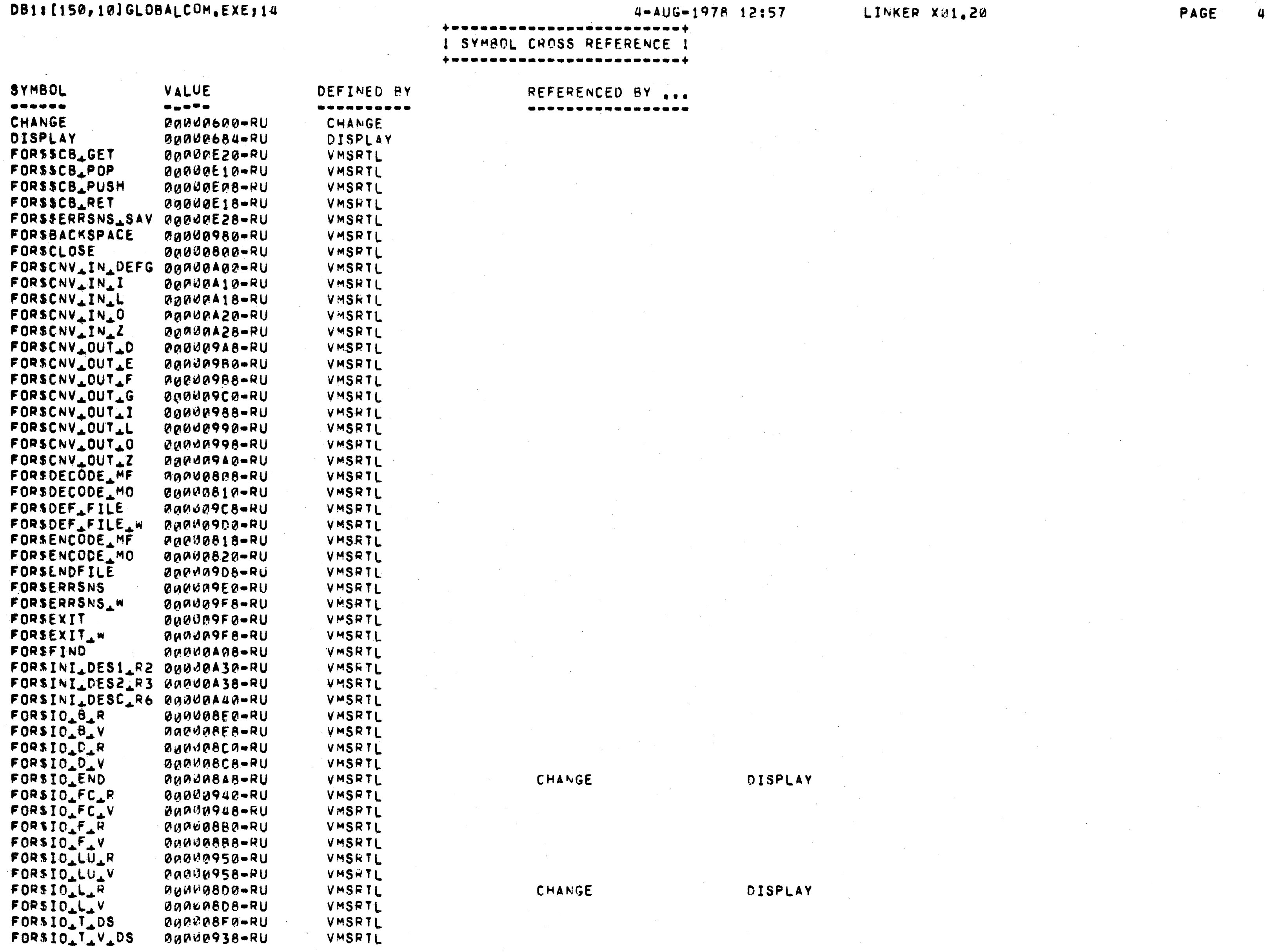
```
GLOBALCOM                                             4-AUG-1978 12:57        LINKER X01.20                  PAGE    1
                                                 +-------------------------+
                                                 ! OBJECT MODULE SYNOPSIS !
                                                 +-------------------------+

MODULE NAME      IDENT             BYTES     FILE                                CREATION DATE       CREATOR
-----------      -----             -----     -----                               -------------       -------
GLOBALCOM        01                   12 DB1:[150,10]GLOBALCOM.OBJ;1           14-Jul-1978 17:44  VAX-11 FORTRAN IV-PLUS T0.8-14
CHANGE           01                  250 DB1:[150,10]CHANGECOM.OBJ;1           14-Jul-1978 17:38  VAX-11 FORTRAN IV-PLUS T0.8-14
DISPLAY          01                  202 DB1:[150,10]DISPLACOM.OBJ;1           14-Jul-1978 17:38  VAX-11 FORTRAN IV-PLUS T0.8-14
OTS$LINKAGE      0-4                   3 DBB2:[SYSLIB]STARLET.OLB;1            03-AUG-1978 19:43  VAX-11 MACRO X0.3-11
VMSRTL           .EXE;1                0 DBB2:[SYSLIB]VMSRTL.EXE;1              4-AUG-1978 08:05  LINK-32 X01.19



DB1:[150,10]GLOBALCOM.EXE;14                          4-AUG-1978 12:57        LINKER X01.20                  PAGE    2
                                                 +-------------------------+
                                                 ! IMAGE SECTION SYNOPSIS !
                                                 +-------------------------+

   CLUSTER       TYPE PAGES   BASE ADDR DISK VBN PFC PROTECTION AND PAGING      GBL. SEC. NAME      MATCH      MAJORID   MINORID
   -------       ---- -----   --------- -------- --- ---------------------      --------------      -----      -------   -------

DEFAULT_CLUSTER    2     0    00000200        0   0 READ WRITE   COPY ON REF
                   3     1    00000200        2   0 READ ONLY
                   3     1    00000400        3   0 READ WRITE
                   3     1    00000600        4   0 READ ONLY
                   4     0    00000800        0   0 READ WRITE   COPY ON REF

VMSRTL             3     4    00000800        0   1 READ ONLY                   VMSRTL_001       LESS/EQUAL         0        40
                   3    49    00001000        0   0 READ ONLY                   VMSRTL_002       LESS/EQUAL         0        40
                   4     2    00007200        0   0 READ WRITE   COPY ON REF    VMSRTL_003       LESS/EQUAL         0        40
```

Figure 8-6 Map Showing FORTRAN Shared Common



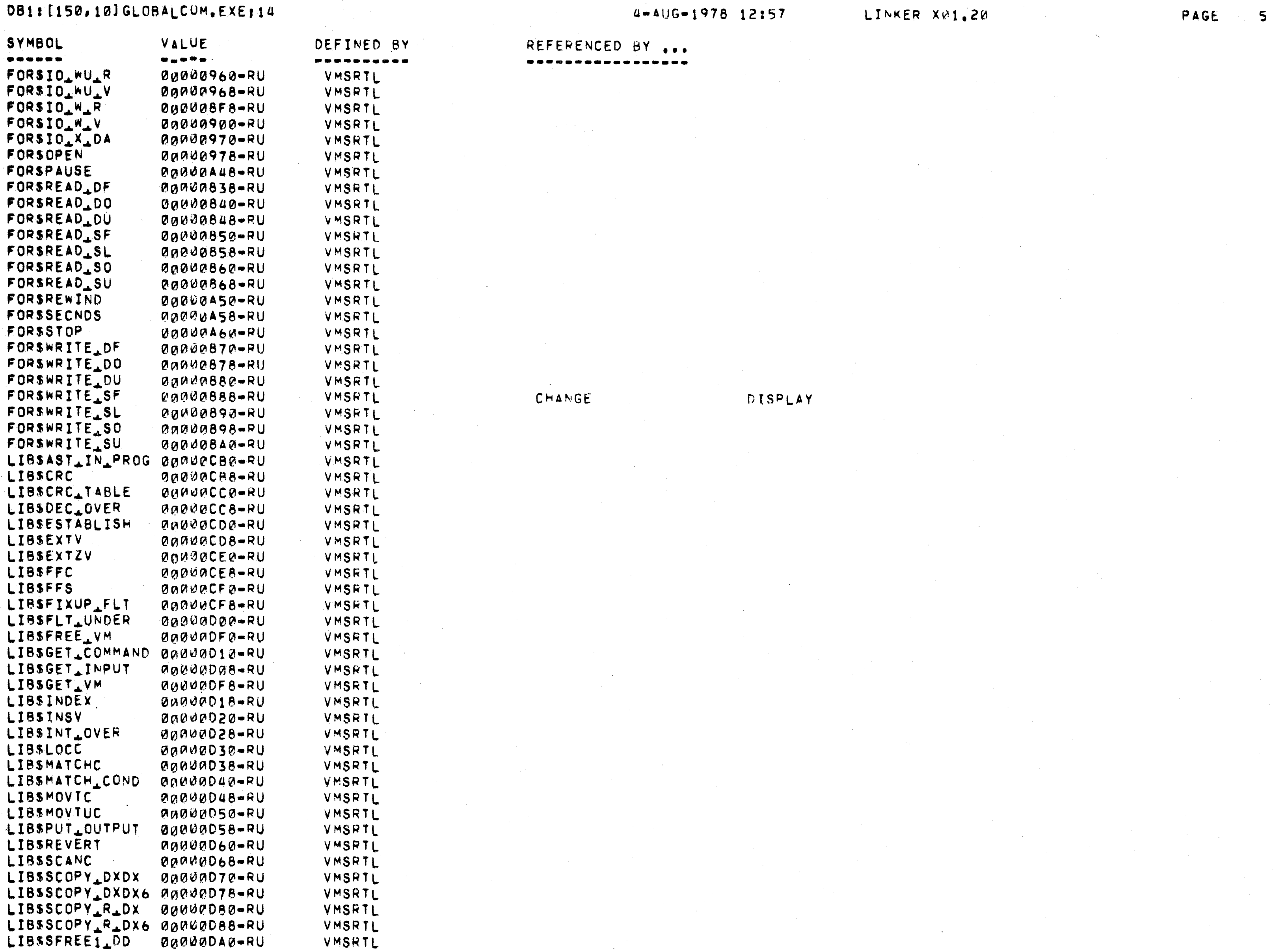
```
DB1:[150,10]GLOBALCOM.EXE;14                         4-AUG-1978 12:57      LINKER X01.20                PAGE    3
                                          +----------------------------+
                                          ! PROGRAM SECTION SYNOPSIS !
                                          +----------------------------+

P-SECT NAME     MODULE(S)      BASE     END            LENGTH          ALIGN                ATTRIBUTES
-----------     ---------      ----     ---            ------          -----                ----------

$PDATA                      00000200 000002BE 000000BF (      191.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,NOEXE,  RD,NOWRT
                CHANGE      00000200 0000026C 0000006D (      109.) LONG 2
                DISPLAY     00000270 000002BF 0000004F (       79.) LONG 2

. BLANK .                   00000200 00000200 00000000 (        0.) BYTE 0 NOPIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,  WRT
                OTS$LINKAGE 00000200 00000200 00000000 (        0.) BYTE 0

$BLANK                      00000400 00000403 00000004 (        4.) LONG 2   PIC,USR,OVR,REL,GBL,  SHR,NOEXE,  RD,  WRT
                GLOBALCOM   00000400 00000403 00000004 (        4.) LONG 2
                CHANGE      00000400 00000403 00000004 (        4.) LONG 2
                DISPLAY     00000400 00000403 00000004 (        4.) LONG 2

NAMEDCOMN1                  00000404 00000407 00000004 (        4.) LONG 2   PIC,USR,OVR,REL,GBL,  SHR,NOEXE,  RD,  WRT
                GLOBALCOM   00000404 00000407 00000004 (        4.) LONG 2
                CHANGE      00000404 00000407 00000004 (        4.) LONG 2
                DISPLAY     00000404 00000407 00000004 (        4.) LONG 2

NAMEDCOMN2                  00000408 0000040B 00000004 (        4.) LONG 2   PIC,USR,OVR,REL,GBL,  SHR,NOEXE,  RD,  WRT
                GLOBALCOM   00000408 0000040B 00000004 (        4.) LONG 2
                CHANGE      00000408 0000040B 00000004 (        4.) LONG 2
                DISPLAY     00000408 0000040B 00000004 (        4.) LONG 2

$CODE                       00000600 000006F2 000000F3 (      243.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT
                CHANGE      00000600 00000680 00000081 (      129.) LONG 2
                DISPLAY     00000684 000006F2 0000006F (      111.) LONG 2

OTS$CODE                    000006F4 000006F6 00000003 (        3.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT
                OTS$LINKAGE 000006F4 000006F6 00000003 (        3.) LONG 2

$LOCAL                      00000800 00000800 00000000 (        0.) LONG 2   PIC,USR,CON,REL,LCL,NOSHR,NOEXE,  RD,  WRT
                CHANGE      00000800 00000800 00000000 (        0.) LONG 2
                DISPLAY     00000800 00000800 00000000 (        0.) LONG 2
```

Figure 8-6 (Cont.) Map Showing FORTRAN Shared Common



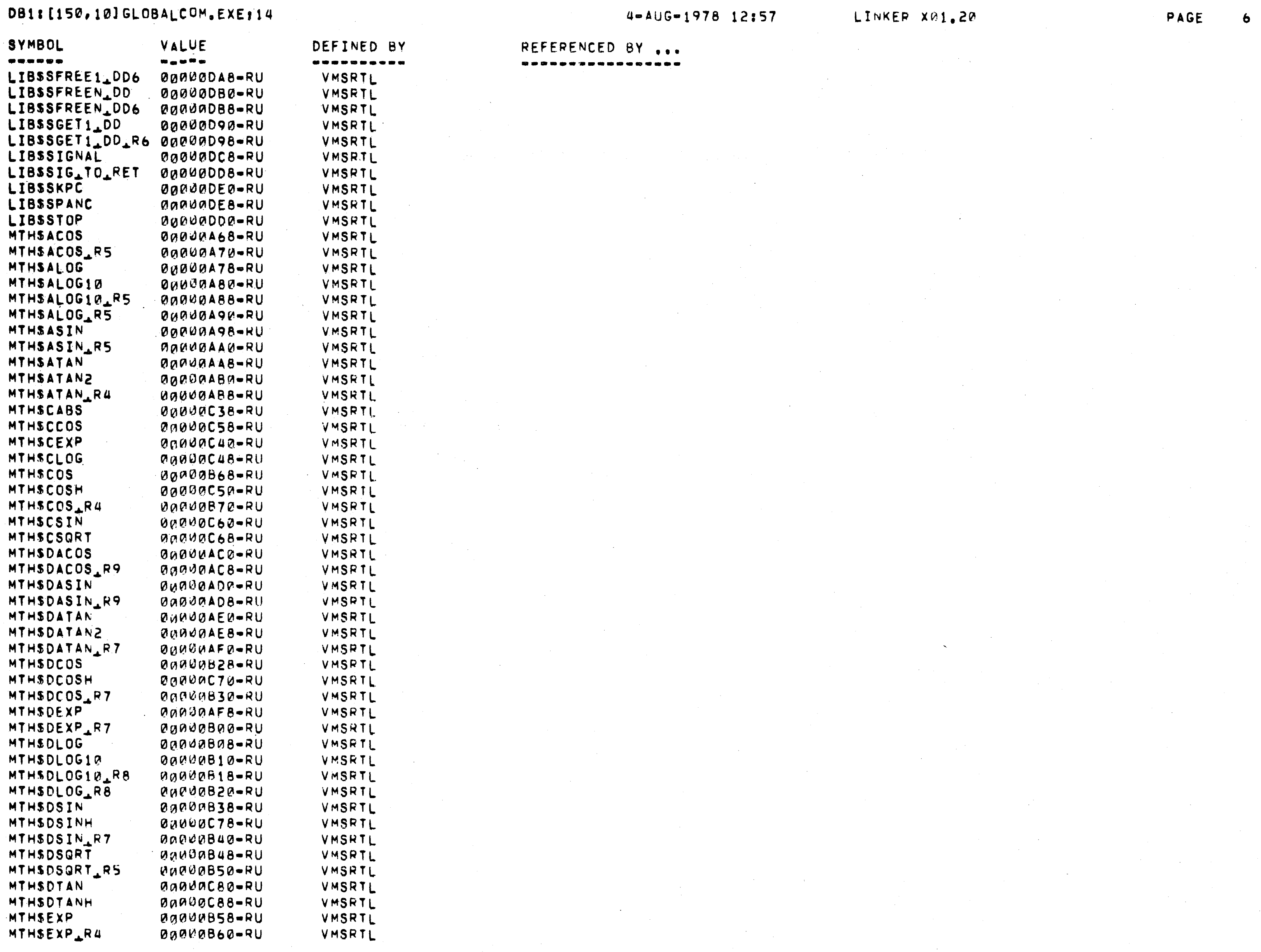
```
DB1:[150,10]GLOBALCOM.EXE;14                                                  4-AUG-1978 12:57        LINKER X01.20                   PAGE    4
                                                          +------------------------+
                                                          ! SYMBOL CROSS REFERENCE !
                                                          +------------------------+

SYMBOL            VALUE             DEFINED BY            REFERENCED BY ...
------            -----             ----------            -----------------
CHANGE            00000600-RU        CHANGE
DISPLAY           00000684-RU        DISPLAY
FOR$$CB_GET       00000E20-RU        VMSRTL
FOR$$CB_POP       00000E10-RU        VMSRTL
FOR$$CB_PUSH      00000E08-RU        VMSRTL
FOR$$CB_RET       00000E18-RU        VMSRTL
FOR$$ERRSNS_SAV   00000E28-RU        VMSRTL
FOR$BACKSPACE     00000980-RU        VMSRTL
FOR$CLOSE         00000800-RU        VMSRTL
FOR$CNV_IN_DEFG   00000A00-RU        VMSRTL
FOR$CNV_IN_I      00000A10-RU        VMSRTL
FOR$CNV_IN_L      00000A18-RU        VMSRTL
FOR$CNV_IN_O      00000A20-RU        VMSRTL
FOR$CNV_IN_Z      00000A28-RU        VMSRTL
FOR$CNV_OUT_D     000009A8-RU        VMSRTL
FOR$CNV_OUT_E     000009B0-RU        VMSRTL
FOR$CNV_OUT_F     000009B8-RU        VMSRTL
FOR$CNV_OUT_G     000009C0-RU        VMSRTL
FOR$CNV_OUT_I     00000988-RU        VMSRTL
FOR$CNV_OUT_L     00000990-RU        VMSRTL
FOR$CNV_OUT_O     00000998-RU        VMSRTL
FOR$CNV_OUT_Z     000009A0-RU        VMSRTL
FOR$DECODE_MF     00000808-RU        VMSRTL
FOR$DECODE_MO     00000810-RU        VMSRTL
FOR$DEF_FILE      000009C8-RU        VMSRTL
FOR$DEF_FILE_W    000009D0-RU        VMSRTL
FOR$ENCODE_MF     00000818-RU        VMSRTL
FOR$ENCODE_MO     00000820-RU        VMSRTL
FOR$ENDFILE       000009D8-RU        VMSRTL
FOR$ERRSNS        000009E0-RU        VMSRTL
FOR$ERRSNS_W      000009F8-RU        VMSRTL
FOR$EXIT          000009F0-RU        VMSRTL
FOR$EXIT_W        000009F8-RU        VMSRTL
FOR$FIND          00000A08-RU        VMSRTL
FOR$INI_DES1_R2   00000A30-RU        VMSRTL
FOR$INI_DES2_R3   00000A38-RU        VMSRTL
FOR$INI_DESC_R6   00000A40-RU        VMSRTL
FOR$IO_B_R        000008E0-RU        VMSRTL
FOR$IO_B_V        000008E8-RU        VMSRTL
FOR$IO_D_R        000008C0-RU        VMSRTL
FOR$IO_D_V        000008C8-RU        VMSRTL
FOR$IO_END        000008A8-RU        VMSRTL                CHANGE                 DISPLAY
FOR$IO_FC_R       00000940-RU        VMSRTL
FOR$IO_FC_V       00000948-RU        VMSRTL
FOR$IO_F_R        000008B0-RU        VMSRTL
FOR$IO_F_V        000008B8-RU        VMSRTL
FOR$IO_LU_R       00000950-RU        VMSRTL
FOR$IO_LU_V       00000958-RU        VMSRTL
FOR$IO_L_R        000008D0-RU        VMSRTL                CHANGE                 DISPLAY
FOR$IO_L_V        000008D8-RU        VMSRTL
FOR$IO_T_DS       000008F0-RU        VMSRTL
FOR$IO_T_V_DS     00000938-RU        VMSRTL
```

Figure 8-6 (Cont.) Map Showing FORTRAN Shared Common



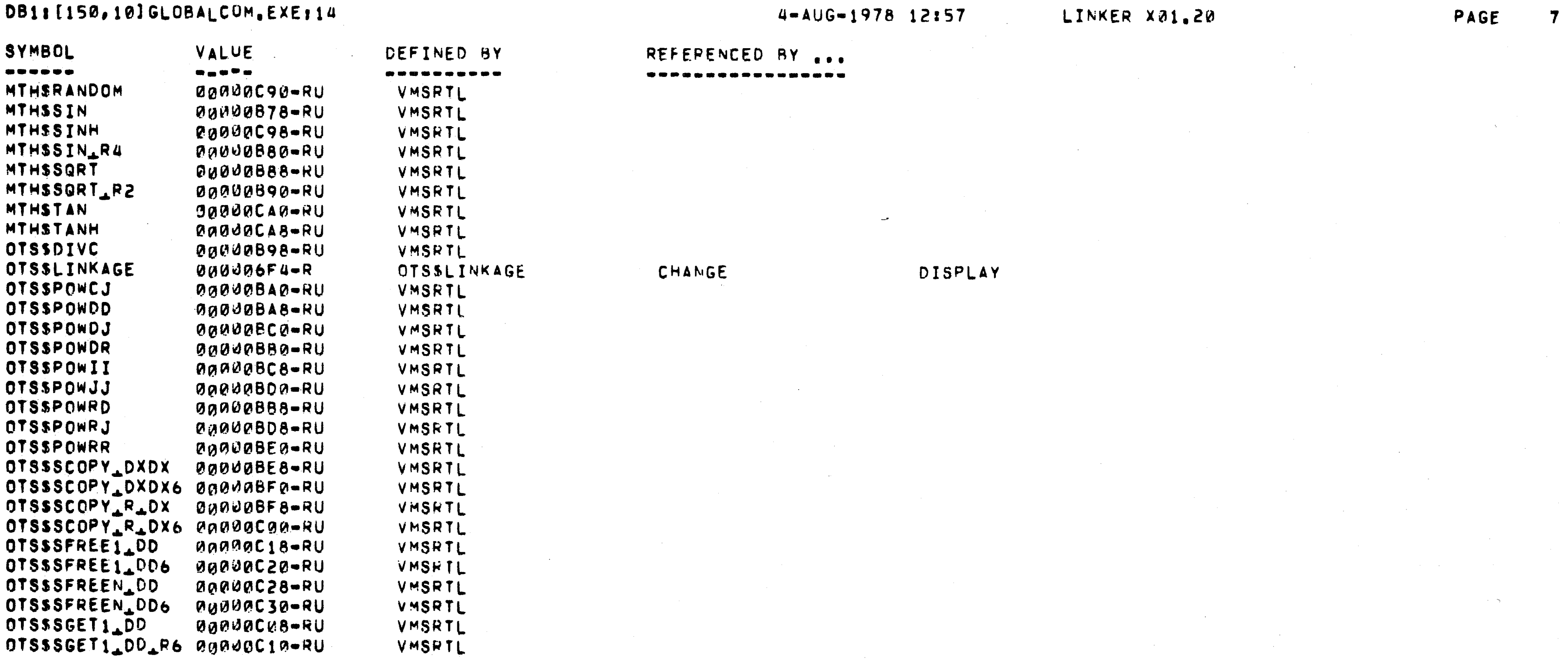
```
DB1:[150,10]GLOBALCUM.EXE;14                          4-AUG-1978 12:57        LINKER X01.20                PAGE    5

SYMBOL            VALUE             DEFINED BY          REFERENCED BY ...
------            -----             ----------          -----------------
FOR$IO_WU_R       00000960-RU        VMSRTL
FOR$IO_WU_V       00000968-RU        VMSRTL
FOR$IO_W_R        000008F8-RU        VMSRTL
FOR$IO_W_V        00000900-RU        VMSRTL
FOR$IO_X_DA       00000970-RU        VMSRTL
FOR$OPEN          00000978-RU        VMSRTL
FOR$PAUSE         00000A48-RU        VMSRTL
FOR$READ_DF       00000838-RU        VMSRTL
FOR$READ_DO       00000840-RU        VMSRTL
FOR$READ_DU       00000848-RU        VMSRTL
FOR$READ_SF       00000850-RU        VMSRTL
FOR$READ_SL       00000858-RU        VMSRTL
FOR$READ_SO       00000860-RU        VMSRTL
FOR$READ_SU       00000868-RU        VMSRTL
FOR$REWIND        00000A50-RU        VMSRTL
FOR$SECNDS        00000A58-RU        VMSRTL
FOR$STOP          00000A60-RU        VMSRTL
FOR$WRITE_DF      00000870-RU        VMSRTL
FOR$WRITE_DO      00000878-RU        VMSRTL
FOR$WRITE_DU      00000880-RU        VMSRTL
FOR$WRITE_SF      00000888-RU        VMSRTL               CHANGE                   DISPLAY
FOR$WRITE_SL      00000890-RU        VMSRTL
FOR$WRITE_SO      00000898-RU        VMSRTL
FOR$WRITE_SU      000008A0-RU        VMSRTL
LIB$AST_IN_PROG   00000CB0-RU        VMSRTL
LIB$CRC           00000CB8-RU        VMSRTL
LIB$CRC_TABLE     00000CC0-RU        VMSRTL
LIB$DEC_OVER      00000CC8-RU        VMSRTL
LIB$ESTABLISH     00000CD0-RU        VMSRTL
LIB$EXTV          00000CD8-RU        VMSRTL
LIB$EXTZV         00000CE0-RU        VMSRTL
LIB$FFC           00000CE8-RU        VMSRTL
LIB$FFS           00000CF0-RU        VMSRTL
LIB$FIXUP_FLT     00000CF8-RU        VMSRTL
LIB$FLT_UNDER     00000D00-RU        VMSRTL
LIB$FREE_VM       00000DF0-RU        VMSRTL
LIB$GET_COMMAND   00000D10-RU        VMSRTL
LIB$GET_INPUT     00000D08-RU        VMSRTL
LIB$GET_VM        00000DF8-RU        VMSRTL
LIB$INDEX         00000D18-RU        VMSRTL
LIB$INSV          00000D20-RU        VMSRTL
LIB$INT_OVER      00000D28-RU        VMSRTL
LIB$LOCC          00000D30-RU        VMSRTL
LIB$MATCHC        00000D38-RU        VMSRTL
LIB$MATCH_COND    00000D40-RU        VMSRTL
LIB$MOVTC         00000D48-RU        VMSRTL
LIB$MOVTUC        00000D50-RU        VMSRTL
LIB$PUT_OUTPUT    00000D58-RU        VMSRTL
LIB$REVERT        00000D60-RU        VMSRTL
LIB$SCANC         00000D68-RU        VMSRTL
LIB$SCOPY_DXDX    00000D70-RU        VMSRTL
LIB$SCOPY_DXDX6   00000D78-RU        VMSRTL
LIB$SCOPY_R_DX    00000D80-RU        VMSRTL
LIB$SCOPY_R_DX6   00000D88-RU        VMSRTL
LIB$SFREE1_DD     00000DA0-RU        VMSRTL
```

Figure 8-6 (Cont.) Map Showing FORTRAN Shared Common



```
DB1:[150,10]GLOBALCOM.EXE;14                          4-AUG-1978 12:57     LINKER X01.20                PAGE   6

SYMBOL            VALUE             DEFINED BY           REFERENCED BY ...
------            -----             ----------           -----------------
LIB$SFREE1_DD6    00000DA8-RU        VMSRTL
LIB$SFREEN_DD     00000DB0-RU        VMSRTL
LIB$SFREEN_DD6    00000DB8-RU        VMSRTL
LIB$SGET1_DD      00000D90-RU        VMSRTL
LIB$SGET1_DD_R6   00000D98-RU        VMSRTL
LIB$SIGNAL        00000DC8-RU        VMSRTL
LIB$SIG_TO_RET    00000DD8-RU        VMSRTL
LIB$SKPC          00000DE0-RU        VMSRTL
LIB$SPANC         00000DE8-RU        VMSRTL
LIB$STOP          00000DD0-RU        VMSRTL
MTH$ACOS          00000A68-RU        VMSRTL
MTH$ACOS_R5       00000A70-RU        VMSRTL
MTH$ALOG          00000A78-RU        VMSRTL
MTH$ALOG10        00000A80-RU        VMSRTL
MTH$ALOG10_R5     00000A88-RU        VMSRTL
MTH$ALOG_R5       00000A90-RU        VMSRTL
MTH$ASIN          00000A98-RU        VMSRTL
MTH$ASIN_R5       00000AA0-RU        VMSRTL
MTH$ATAN          00000AA8-RU        VMSRTL
MTH$ATAN2         00000AB0-RU        VMSRTL
MTH$ATAN_R4       00000AB8-RU        VMSRTL
MTH$CABS          00000C38-RU        VMSRTL
MTH$CCOS          00000C58-RU        VMSRTL
MTH$CEXP          00000C40-RU        VMSRTL
MTH$CLOG          00000C48-RU        VMSRTL
MTH$COS           00000B68-RU        VMSRTL
MTH$COSH          00000C50-RU        VMSRTL
MTH$COS_R4        00000B70-RU        VMSRTL
MTH$CSIN          00000C60-RU        VMSRTL
MTH$CSQRT         00000C68-RU        VMSRTL
MTH$DACOS         00000AC0-RU        VMSRTL
MTH$DACOS_R9      00000AC8-RU        VMSRTL
MTH$DASIN         00000AD0-RU        VMSRTL
MTH$DASIN_R9      00000AD8-RU        VMSRTL
MTH$DATAN         00000AE0-RU        VMSRTL
MTH$DATAN2        00000AE8-RU        VMSRTL
MTH$DATAN_R7      00000AF0-RU        VMSRTL
MTH$DCOS          00000B28-RU        VMSRTL
MTH$DCOSH         00000C70-RU        VMSRTL
MTH$DCOS_R7       00000B30-RU        VMSRTL
MTH$DEXP          00000AF8-RU        VMSRTL
MTH$DEXP_R7       00000B00-RU        VMSRTL
MTH$DLOG          00000B08-RU        VMSRTL
MTH$DLOG10        00000B10-RU        VMSRTL
MTH$DLOG10_R8     00000B18-RU        VMSRTL
MTH$DLOG_R8       00000B20-RU        VMSRTL
MTH$DSIN          00000B38-RU        VMSRTL
MTH$DSINH         00000C78-RU        VMSRTL
MTH$DSIN_R7       00000B40-RU        VMSRTL
MTH$DSQRT         00000B48-RU        VMSRTL
MTH$DSQRT_R5      00000B50-RU        VMSRTL
MTH$DTAN          00000C80-RU        VMSRTL
MTH$DTANH         00000C88-RU        VMSRTL
MTH$EXP           00000B58-RU        VMSRTL
MTH$EXP_R4        00000B60-RU        VMSRTL
```

Figure 8-6 (Cont.) Map Showing FORTRAN Shared Common



```
DB1:[150,10]GLOBALCOM.EXE;14                                    4-AUG-1978 12:57      LINKER X01.20                PAGE    7

SYMBOL           VALUE            DEFINED BY           REFERENCED BY ...
------           -----            ----------           -----------------
MTH$RANDOM       00000C90-RU       VMSRTL
MTH$SIN          00000B78-RU       VMSRTL
MTH$SINH         00000C98-RU       VMSRTL
MTH$SIN_R4       00000B80-RU       VMSRTL
MTH$SQRT         00000B88-RU       VMSRTL
MTH$SQRT_R2      00000B90-RU       VMSRTL
MTH$TAN          00000CA0-RU       VMSRTL
MTH$TANH         00000CA8-RU       VMSRTL
OTS$DIVC         00000B98-RU       VMSRTL
OTS$LINKAGE      000006F4-R        OTS$LINKAGE           CHANGE                  DISPLAY
OTS$POWCJ        00000BA0-RU       VMSRTL
OTS$POWDD        00000BA8-RU       VMSRTL
OTS$POWDJ        00000BC0-RU       VMSRTL
OTS$POWDR        00000BB0-RU       VMSRTL
OTS$POWII        00000BC8-RU       VMSRTL
OTS$POWJJ        00000BD0-RU       VMSRTL
OTS$POWRD        00000BB8-RU       VMSRTL
OTS$POWRJ        00000BD8-RU       VMSRTL
OTS$POWRR        00000BE0-RU       VMSRTL
OTS$SCOPY_DXDX   00000BE8-RU       VMSRTL
OTS$SCOPY_DXDX6  00000BF0-RU       VMSRTL
OTS$SCOPY_R_DX   00000BF8-RU       VMSRTL
OTS$SCOPY_R_DX6  00000C00-RU       VMSRTL
OTS$SFREE1_DD    00000C18-RU       VMSRTL
OTS$SFREE1_DD6   00000C20-RU       VMSRTL
OTS$SFREEN_DD    00000C28-RU       VMSRTL
OTS$SFREEN_DD6   00000C30-RU       VMSRTL
OTS$SGET1_DD     00000C08-RU       VMSRTL
OTS$SGET1_DD_R6  00000C10-RU       VMSRTL
```

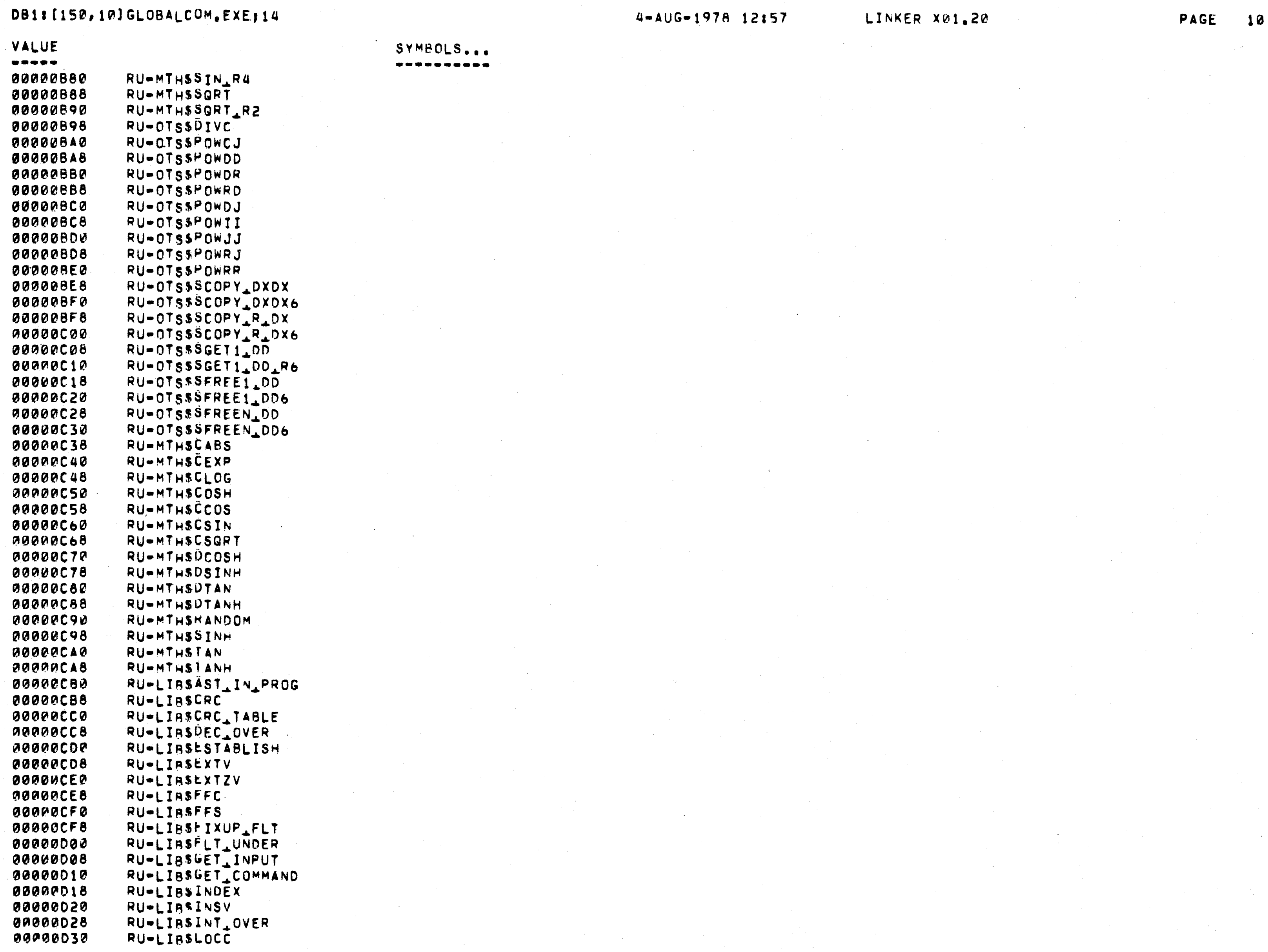
Figure 8-6 (Cont.) Map Showing FORTRAN Shared Common

```
DB1:[150,10]GLOBALCOM.EXE;14                              4-AUG-1978 12:57       LINKER X01.20              PAGE    8
                                                       +--------------------+
                                                       ! SYMBOLS BY VALUE !
                                                       +--------------------+

VALUE                                          SYMBOLS...
-----                                          ----------
00000600    RU-CHANGE
00000684    RU-DISPLAY
000006F4     R-OTS$LINKAGE
00000800    RU-FOR$CLOSE
00000808    RU-FOR$DECODE_MF
00000810    RU-FOR$DECODE_MO
00000818    RU-FOR$ENCODE_MF
00000820    RU-FOR$ENCODE_MO
00000838    RU-FOR$READ_DF
00000840    RU-FOR$READ_DO
00000848    RU-FOR$READ_DU
00000850    RU-FOR$READ_SF
00000858    RU-FOR$READ_SL
00000860    RU-FOR$READ_SO
00000868    RU-FOR$READ_SU
00000870    RU-FOR$WRITE_DF
00000878    RU-FOR$WRITE_DO
00000880    RU-FOR$WRITE_DU
00000888    RU-FOR$WRITE_SF
00000890    RU-FOR$WRITE_SL
00000898    RU-FOR$WRITE_SO
000008A0    RU-FOR$WRITE_SU
000008A8    RU-FOR$IO_END
000008B0    RU-FOR$IO_F_R
000008B8    RU-FOR$IO_F_V
000008C0    RU-FOR$IO_D_R
000008C8    RU-FOR$IO_D_V
000008D0    RU-FOR$IO_L_R
000008D8    RU-FOR$IO_L_V
000008E0    RU-FOR$IO_B_R
000008E8    RU-FOR$IO_B_V
000008F0    RU-FOR$IO_T_DS
000008F8    RU-FOR$IO_W_R
00000900    RU-FOR$IO_W_V
00000938    RU-FOR$IO_T_V_DS
00000940    RU-FOR$IO_FC_R
00000948    RU-FOR$IO_FC_V
00000950    RU-FOR$IO_LU_R
00000958    RU-FOR$IO_LU_V
00000960    RU-FOR$IO_WU_R
00000968    RU-FOR$IO_WU_V
00000970    RU-FOR$IO_X_DA
00000978    RU-FOR$OPEN
00000980    RU-FOR$BACKSPACE
00000988    RU-FOR$CNV_OUT_I
00000990    RU-FOR$CNV_OUT_L
00000998    RU-FOR$CNV_OUT_O
000009A0    RU-FOR$CNV_OUT_Z
000009A8    RU-FOR$CNV_OUT_D
000009B0    RU-FOR$CNV_OUT_E
000009B8    RU-FOR$CNV_OUT_F
000009C0    RU-FOR$CNV_OUT_G
```

Figure 8-6 (Cont.) Map Showing FORTRAN Shared Common

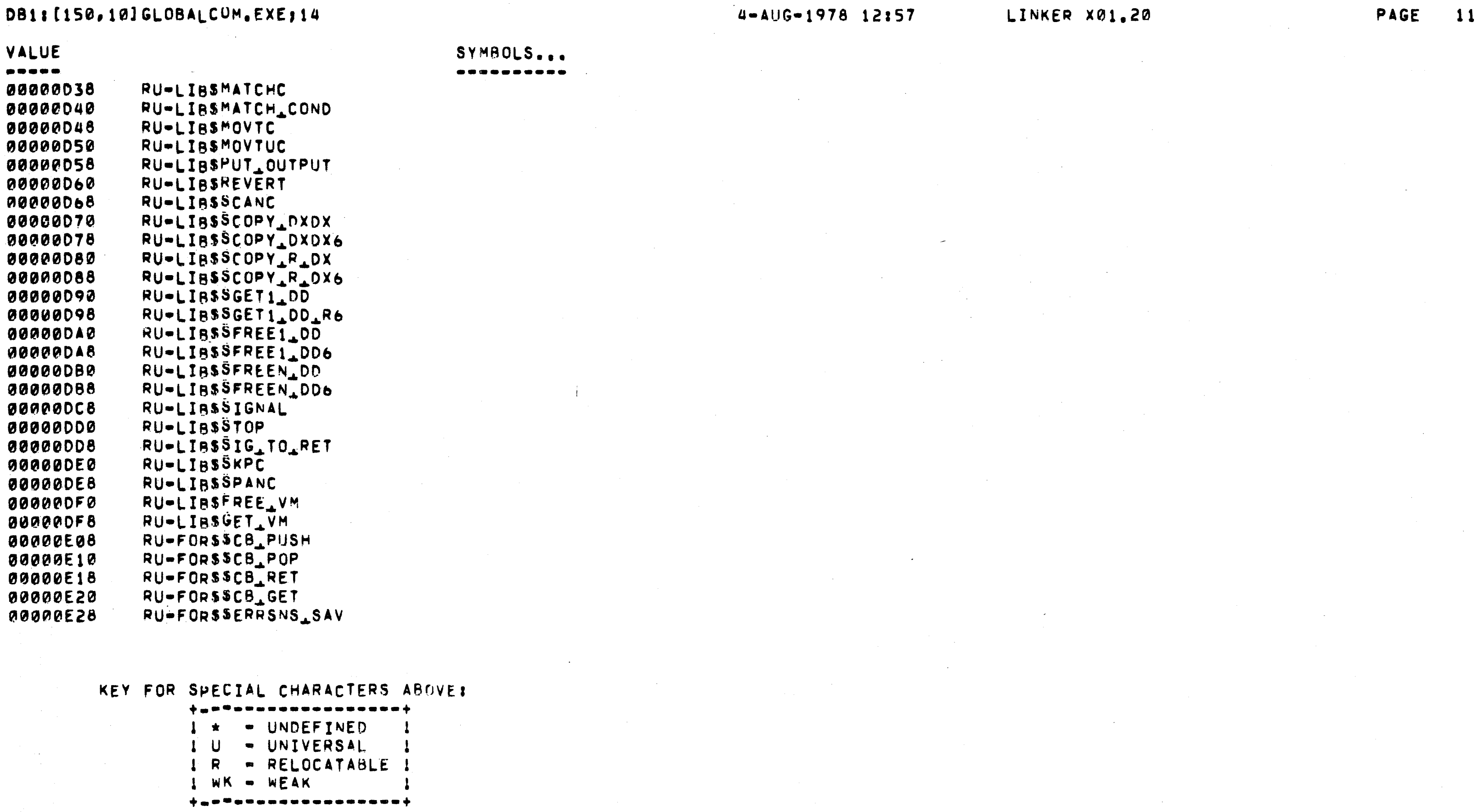
```
DB1:[150,10]GLOBALCOM.EXE;14                                    4-AUG-1978 12:57      LINKER X01.20          PAGE    9

VALUE                                   SYMBOLS...
-----                                   ----------
000009C8     RU-FOR$DEF_FILE
000009D0     RU-FOR$DEF_FILE_W
000009D8     RU-FOR$ENDFILE
000009E0     RU-FOR$ERRSNS
000009E8     RU-FOR$ERRSNS_W
000009F0     RU-FOR$EXIT
000009F8     RU-FOR$EXIT_W
00000A00     RU-FOR$CNV_IN_DEFG
00000A08     RU-FOR$FIND
00000A10     RU-FOR$CNV_IN_I
00000A18     RU-FOR$CNV_IN_L
00000A20     RU-FOR$CNV_IN_O
00000A28     RU-FOR$CNV_IN_Z
00000A30     RU-FOR$INI_DES1_R2
00000A38     RU-FOR$INI_DES2_R3
00000A40     RU-FOR$INI_DESC_R6
00000A48     RU-FOR$PAUSE
00000A50     RU-FOR$REWIND
00000A58     RU-FOR$SECNDS
00000A60     RU-FOR$STOP
00000A68     RU-MTH$ACOS
00000A70     RU-MTH$ACOS_R5
00000A78     RU-MTH$ALOG
00000A80     RU-MTH$ALOG10
00000A88     RU-MTH$ALOG10_R5
00000A90     RU-MTH$ALOG_R5
00000A98     RU-MTH$ASIN
00000AA0     RU-MTH$ASIN_R5
00000AA8     RU-MTH$ATAN
00000AB0     RU-MTH$ATAN2
00000AB8     RU-MTH$ATAN_R4
00000AC0     RU-MTH$DACOS
00000AC8     RU-MTH$DACOS_R9
00000AD0     RU-MTH$DASIN
00000AD8     RU-MTH$DASIN_R9
00000AE0     RU-MTH$DATAN
00000AE8     RU-MTH$DATAN2
00000AF0     RU-MTH$DATAN_R7
00000AF8     RU-MTH$DEXP
00000B00     RU-MTH$DEXP_R7
00000B08     RU-MTH$DLOG
00000B10     RU-MTH$DLOG10
00000B18     RU-MTH$DLOG10_R8
00000B20     RU-MTH$DLOG_R8
00000B28     RU-MTH$DCOS
00000B30     RU-MTH$DCOS_R7
00000B38     RU-MTH$DSIN
00000B40     RU-MTH$DSIN_R7
00000B48     RU-MTH$DSQRT
00000B50     RU-MTH$DSQRT_R5
00000B58     RU-MTH$EXP
00000B60     RU-MTH$EXP_R4
00000B68     RU-MTH$COS
00000B70     RU-MTH$COS_R4
00000B78     RU-MTH$SIN
```

Figure 8-6 (Cont.) Map Showing FORTRAN Shared Common



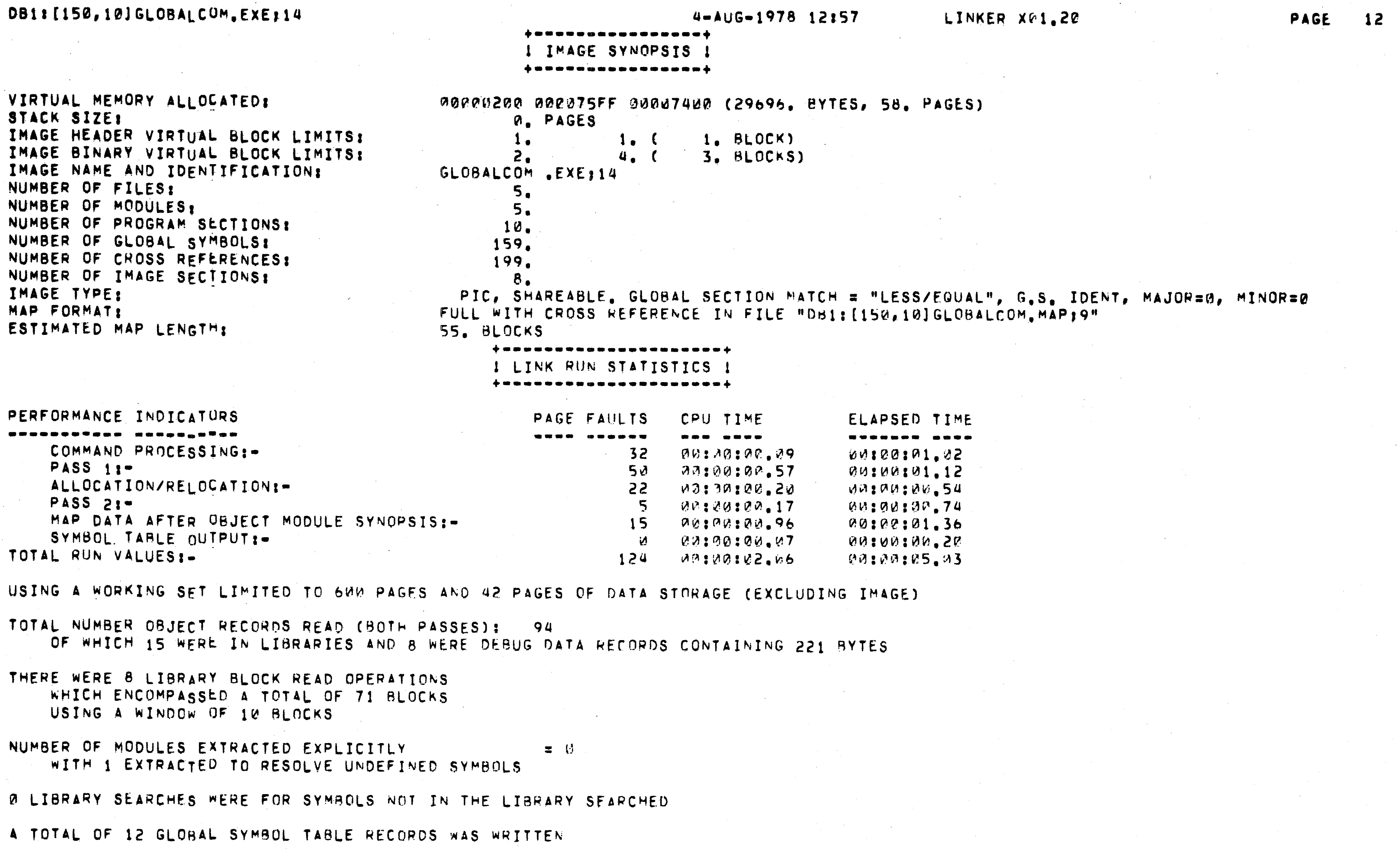
```
DB1:[150,10]GLOBALCOM.EXE;14                              4-AUG-1978 12:57      LINKER X01.20           PAGE   10

VALUE                                    SYMBOLS...
-----                                    ----------
00000B80    RU-MTH$SIN_R4
00000B88    RU-MTH$SQRT
00000B90    RU-MTH$SQRT_R2
00000B98    RU-OTS$DIVC
00000BA0    RU-OTS$POWCJ
00000BA8    RU-OTS$POWDD
00000BB0    RU-OTS$POWDR
00000BB8    RU-OTS$POWRD
00000BC0    RU-OTS$POWDJ
00000BC8    RU-OTS$POWII
00000BD0    RU-OTS$POWJJ
00000BD8    RU-OTS$POWRJ
00000BE0    RU-OTS$POWRR
00000BE8    RU-OTS$SCOPY_DXDX
00000BF0    RU-OTS$SCOPY_DXDX6
00000BF8    RU-OTS$SCOPY_R_DX
00000C00    RU-OTS$SCOPY_R_DX6
00000C08    RU-OTS$SGET1_DD
00000C10    RU-OTS$SGET1_DD_R6
00000C18    RU-OTS$SFREE1_DD
00000C20    RU-OTS$SFREE1_DD6
00000C28    RU-OTS$SFREEN_DD
00000C30    RU-OTS$SFREEN_DD6
00000C38    RU-MTH$CABS
00000C40    RU-MTH$CEXP
00000C48    RU-MTH$CLOG
00000C50    RU-MTH$COSH
00000C58    RU-MTH$CCOS
00000C60    RU-MTH$CSIN
00000C68    RU-MTH$CSQRT
00000C70    RU-MTH$DCOSH
00000C78    RU-MTH$DSINH
00000C80    RU-MTH$DTAN
00000C88    RU-MTH$DTANH
00000C90    RU-MTH$RANDOM
00000C98    RU-MTH$SINH
00000CA0    RU-MTH$TAN
00000CA8    RU-MTH$TANH
00000CB0    RU-LIB$AST_IN_PROG
00000CB8    RU-LIB$CRC
00000CC0    RU-LIB$CRC_TABLE
00000CC8    RU-LIB$DEC_OVER
00000CD0    RU-LIB$ESTABLISH
00000CD8    RU-LIB$EXTV
00000CE0    RU-LIB$EXTZV
00000CE8    RU-LIB$FFC
00000CF0    RU-LIB$FFS
00000CF8    RU-LIB$FIXUP_FLT
00000D00    RU-LIB$FLT_UNDER
00000D08    RU-LIB$GET_INPUT
00000D10    RU-LIB$GET_COMMAND
00000D18    RU-LIB$INDEX
00000D20    RU-LIB$INSV
00000D28    RU-LIB$INT_OVER
00000D30    RU-LIB$LOCC
```

Figure 8-6 (Cont.) Map Showing FORTRAN Shared Common



```
DB1:[150,10]GLOBALCUM.EXE;14                                         4-AUG-1978 12:57       LINKER X01.20                   PAGE   11

VALUE                               SYMBOLS...
-----                               ----------
00000D38    RU-LIB$MATCHC
00000D40    RU-LIB$MATCH_COND
00000D48    RU-LIB$MOVTC
00000D50    RU-LIB$MOVTUC
00000D58    RU-LIB$PUT_OUTPUT
00000D60    RU-LIB$REVERT
00000D68    RU-LIB$SCANC
00000D70    RU-LIB$SCOPY_DXDX
00000D78    RU-LIB$SCOPY_DXDX6
00000D80    RU-LIB$SCOPY_R_DX
00000D88    RU-LIB$SCOPY_R_DX6
00000D90    RU-LIB$SGET1_DD
00000D98    RU-LIB$SGET1_DD_R6
00000DA0    RU-LIB$SFREE1_DD
00000DA8    RU-LIB$SFREE1_DD6
00000DB0    RU-LIB$SFREEN_DD
00000DB8    RU-LIB$SFREEN_DD6
00000DC8    RU-LIB$SIGNAL
00000DD0    RU-LIB$STOP
00000DD8    RU-LIB$SIG_TO_RET
00000DE0    RU-LIB$SKPC
00000DE8    RU-LIB$SPANC
00000DF0    RU-LIB$FREE_VM
00000DF8    RU-LIB$GET_VM
00000E08    RU-FOR$CB_PUSH
00000E10    RU-FOR$CB_POP
00000E18    RU-FOR$CB_RET
00000E20    RU-FOR$CB_GET
00000E28    RU-FOR$ERRSNS_SAV


        KEY FOR SPECIAL CHARACTERS ABOVE:
                +-------------------+
                ! *  - UNDEFINED    !
                ! U  - UNIVERSAL    !
                ! R  - RELOCATABLE  !
                ! WK - WEAK         !
                +-------------------+
```

Figure 8-6 (Cont.) Map Showing FORTRAN Shared Common



```
DB1:[150,10]GLOBALCOM.EXE;14                                          4-AUG-1978 12:57        LINKER X01.20                     PAGE   12
                                                               +-----------------+
                                                               ! IMAGE SYNOPSIS !
                                                               +-----------------+

VIRTUAL MEMORY ALLOCATED:                               00000200 000075FF 00007400 (29696. BYTES, 58. PAGES)
STACK SIZE:                                                   0. PAGES
IMAGE HEADER VIRTUAL BLOCK LIMITS:                            1.          1. (     1. BLOCK)
IMAGE BINARY VIRTUAL BLOCK LIMITS:                            2.          4. (     3. BLOCKS)
IMAGE NAME AND IDENTIFICATION:                          GLOBALCOM .EXE;14
NUMBER OF FILES:                                              5.
NUMBER OF MODULES:                                            5.
NUMBER OF PROGRAM SECTIONS:                                  10.
NUMBER OF GLOBAL SYMBOLS:                                   159.
NUMBER OF CROSS REFERENCES:                                 199.
NUMBER OF IMAGE SECTIONS:                                     8.
IMAGE TYPE:                                               PIC, SHAREABLE, GLOBAL SECTION MATCH = "LESS/EQUAL", G.S. IDENT, MAJOR=0, MINOR=0
MAP FORMAT:                                             FULL WITH CROSS REFERENCE IN FILE "DB1:[150,10]GLOBALCOM.MAP;9"
ESTIMATED MAP LENGTH:                                   55. BLOCKS
                                                            +----------------------+
                                                            ! LINK RUN STATISTICS !
                                                            +----------------------+

PERFORMANCE INDICATORS                                           PAGE FAULTS   CPU TIME        ELAPSED TIME
----------- ----------                                           ---- ------   --- ----        ------- ----
    COMMAND PROCESSING:-                                                  32   00:00:00.09     00:00:01.02
    PASS 1:-                                                              50   00:00:00.57     00:00:01.12
    ALLOCATION/RELOCATION:-                                               22   00:00:00.20     00:00:00.54
    PASS 2:-                                                               5   00:00:00.17     00:00:00.74
    MAP DATA AFTER OBJECT MODULE SYNOPSIS:-                               15   00:00:00.96     00:00:01.36
    SYMBOL TABLE OUTPUT:-                                                  0   00:00:00.07     00:00:00.20
TOTAL RUN VALUES:-                                                       124   00:00:02.06     00:00:05.03

USING A WORKING SET LIMITED TO 600 PAGES AND 42 PAGES OF DATA STORAGE (EXCLUDING IMAGE)

TOTAL NUMBER OBJECT RECORDS READ (BOTH PASSES):   94
    OF WHICH 15 WERE IN LIBRARIES AND 8 WERE DEBUG DATA RECORDS CONTAINING 221 BYTES

THERE WERE 8 LIBRARY BLOCK READ OPERATIONS
    WHICH ENCOMPASSED A TOTAL OF 71 BLOCKS
    USING A WINDOW OF 10 BLOCKS

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 12 GLOBAL SYMBOL TABLE RECORDS WAS WRITTEN
```

Figure 8-6 (Cont.) Map Showing FORTRAN Shared Common

8.3 USING SHAREABLE IMAGES

To be of use, shareable images are normally linked into another image. Usually shareable images are also 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 6) 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 image containing a shareable image is activated, a search is made for the global section match, as described in Section 8.2.3. If that match fails, one of two things occurs, depending on whether the executable image has a private copy of the shareable image:

- If the executable image has a private copy, that copy is used instead of the global sections.

- If the executable image does not have a private copy, an error message is issued indicating that the required global sections are not available.

CHAPTER 9

CLUSTERING

The concept and main uses of image clustering were introduced in Chapter 2. The present chapter expands on the earlier material, describing the mechanics of clustering and some guidelines for usage.

9.1 MECHANICS OF CLUSTERING

Chapter 6 describes the CLUSTER= option, which is 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.

Clusters are allocated virtual address space in the order in which you specify them, unless you specify base addresses. In allocating virtual address space, the linker first deals with clusters to which you gave base addresses, and it considers them in the order of specification. The linker reports an error if it detects any overlap.

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.

After the linker has allocated virtual space to all user-specified clusters and shareable images, it allocates space to the default cluster, if it contains any modules. Finally, the linker allocates address space to the Run-Time Library shareable image, if it has been automatically acquired.

9.2 USAGE GUIDELINES

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. Fragmentation can reduce program performance under certain circumstances.

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]"

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 overidden 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]

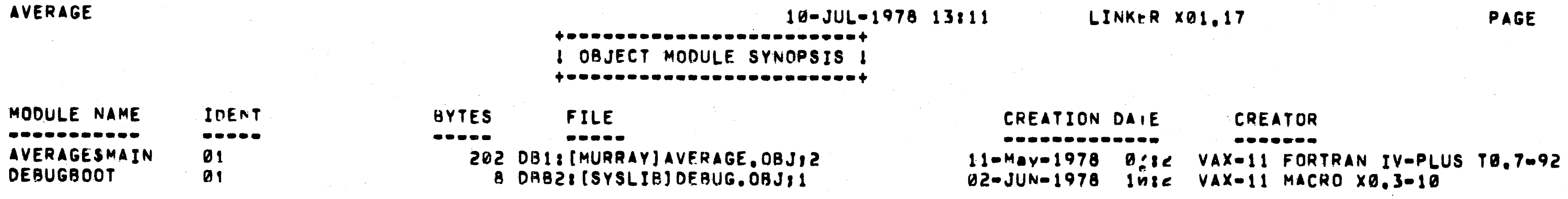
VALREQ, Value required in qualifier - option file "[file-spec]"

APPENDIX B

IMAGE MAP ILLUSTRATIONS

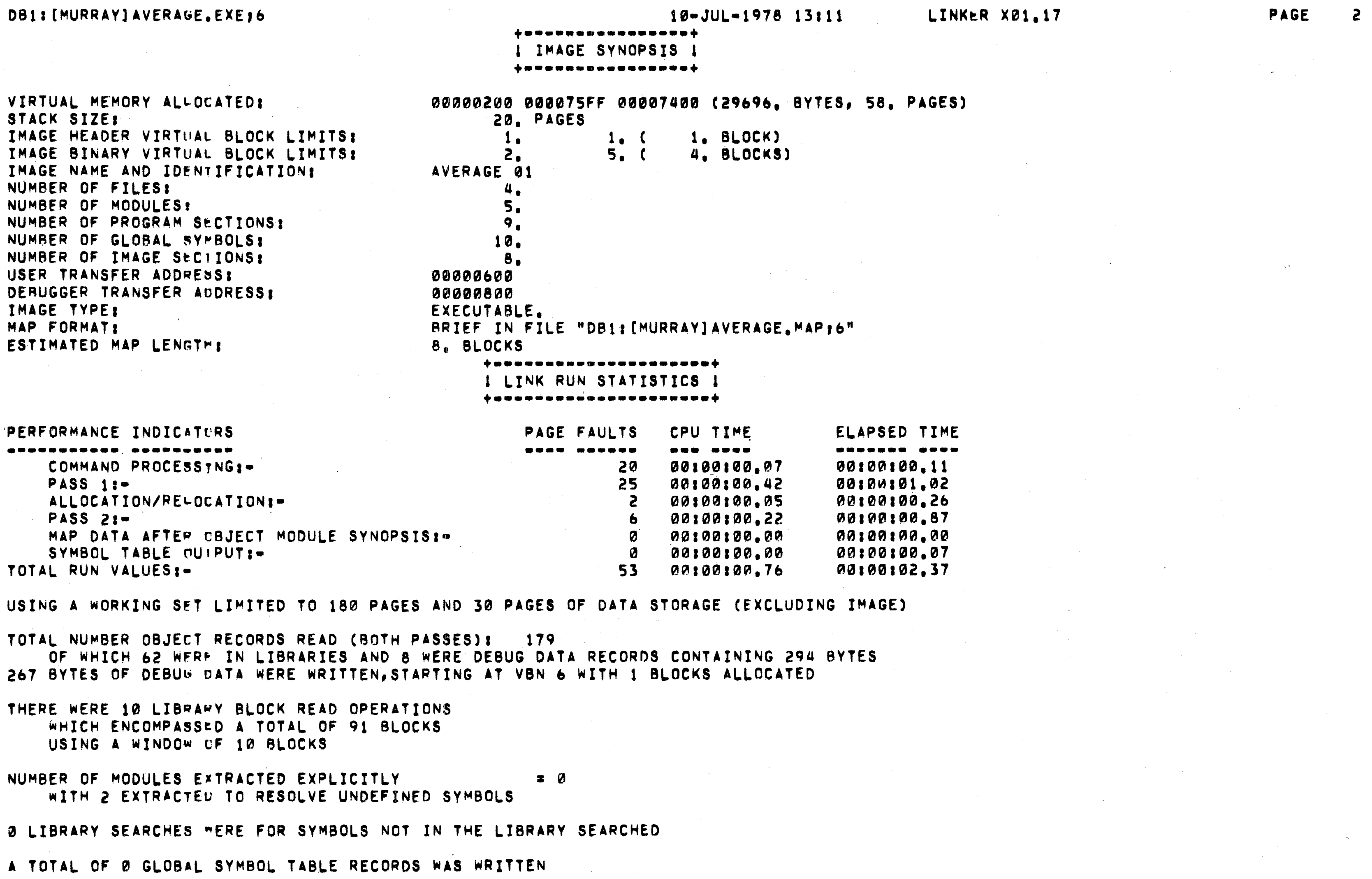
This appendix illustrates the complete brief, default, and full forms of a map of the same image. These illustrations do not include a Symbol Cross Reference map section; however, this section does appear in Chapter 7 (Figure 7-5).

The illustrations in this appendix are forms of the map used in Chapter 7.

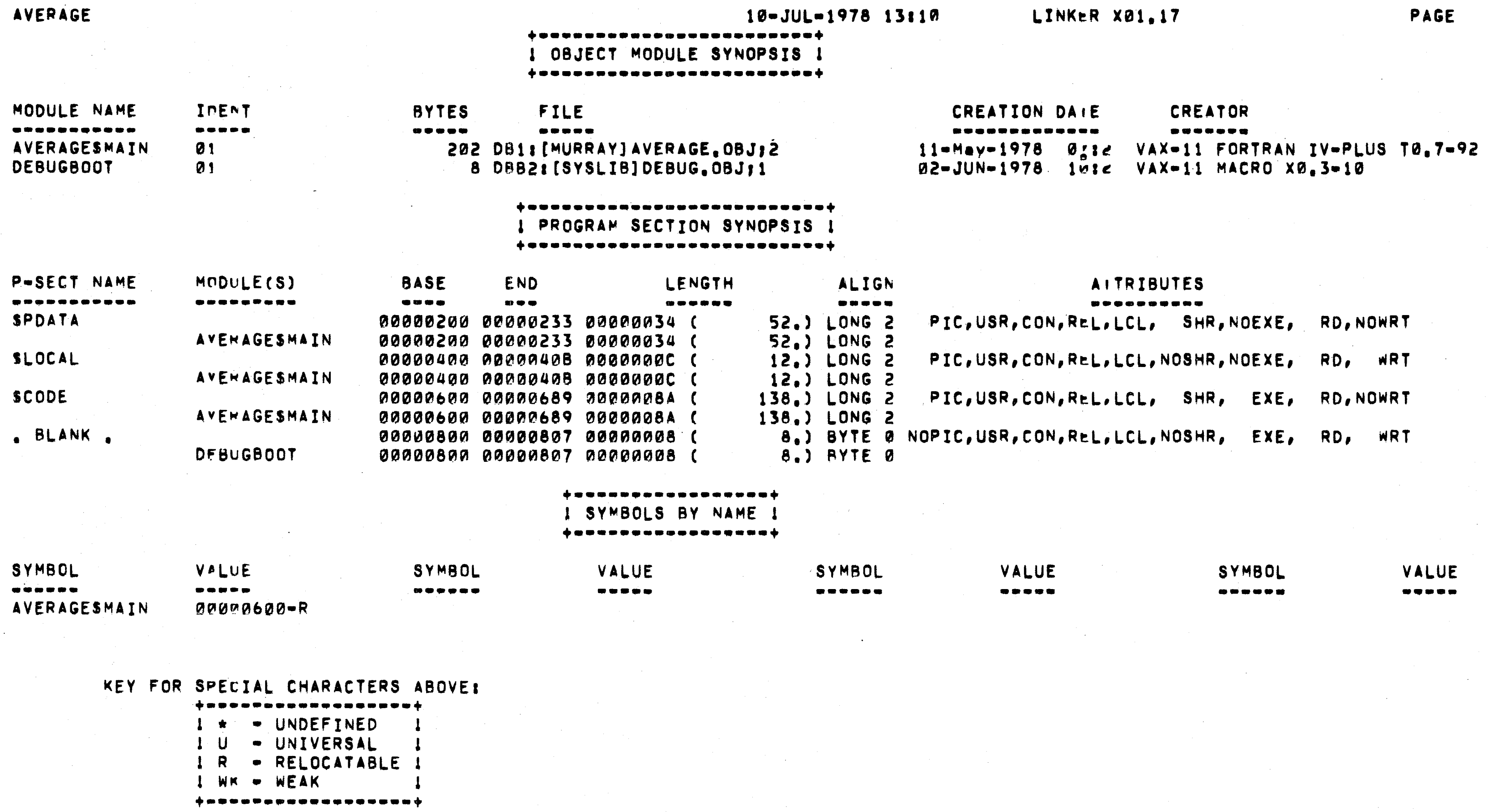


```
AVERAGE                                              1Ø-JUL-1978 13:11       LINKER XØ1.17                PAGE    1
                                          +-------------------------+
                                          ! OBJECT MODULE SYNOPSIS !
                                          +-------------------------+

MODULE NAME     IDENT            BYTES      FILE                          CREATION DAIE       CREATOR
------------    -----            -----      -----                         --------------      -------
AVERAGE$MAIN    Ø1                 2Ø2 DB1:[MURRAY]AVERAGE.OBJ;2        11-May-1978  Ø':2  VAX-11 FORTRAN IV-PLUS TØ.7-92
DEBUGBOOT       Ø1                   8 DRB2:[SYSLIB]DEBUG.OBJ;1         Ø2-JUN-1978  1n:2  VAX-11 MACRO XØ.3-1Ø
```

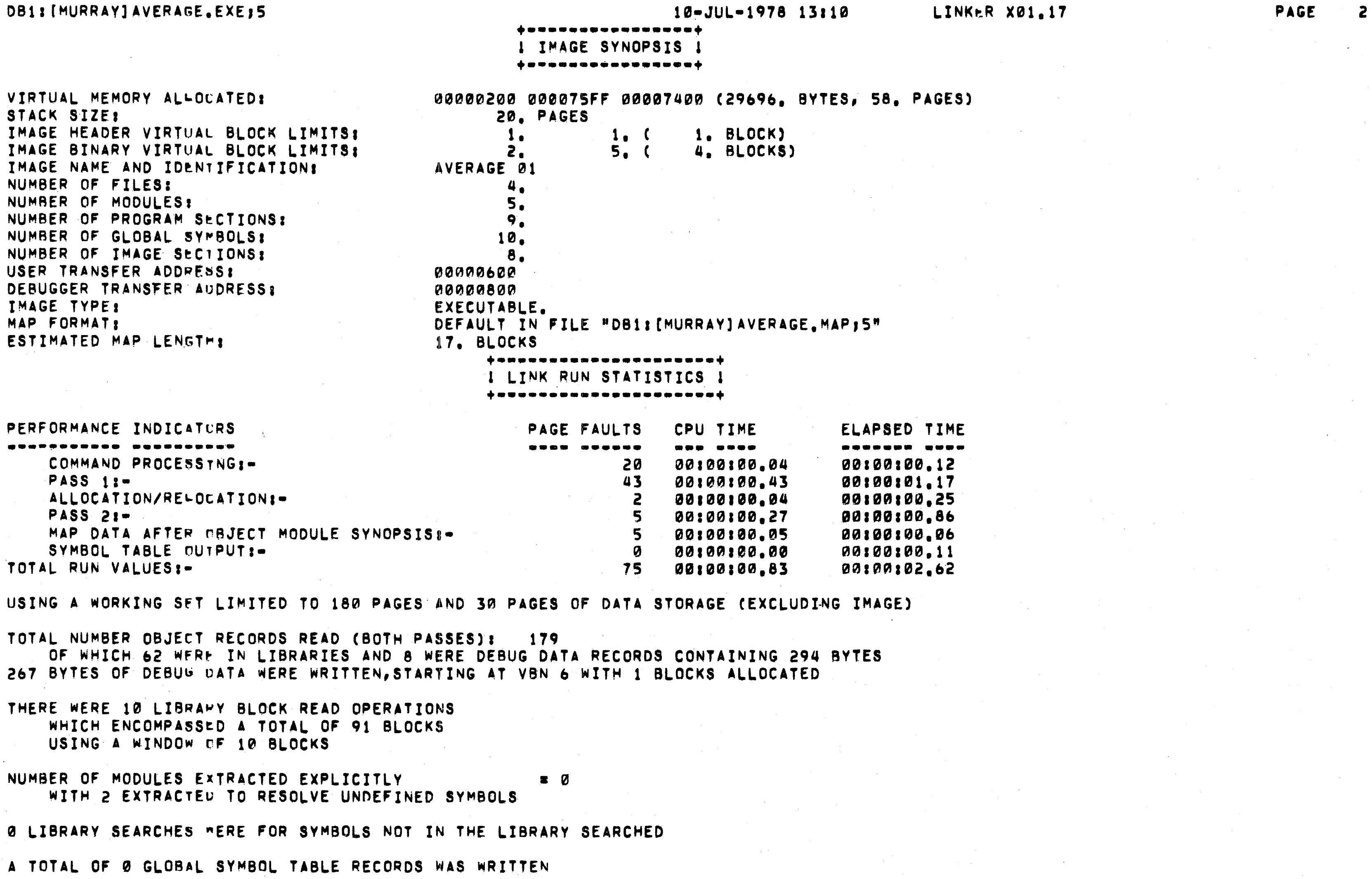
BRIEF MAP



```
DB1:[MURRAY]AVERAGE.EXE;6                                    10-JUL-1978 13:11        LINKER X01.17                        PAGE    2
                                                        +-----------------+
                                                        ! IMAGE SYNOPSIS !
                                                        +-----------------+

VIRTUAL MEMORY ALLOCATED:                  00000200 000075FF 00007400 (29696. BYTES, 58. 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:             AVERAGE 01
NUMBER OF FILES:                                  4.
NUMBER OF MODULES:                                5.
NUMBER OF PROGRAM SECTIONS:                       9.
NUMBER OF GLOBAL SYMBOLS:                        10.
NUMBER OF IMAGE SECTIONS:                         8.
USER TRANSFER ADDRESS:                     00000600
DEBUGGER TRANSFER ADDRESS:                 00000800
IMAGE TYPE:                                EXECUTABLE.
MAP FORMAT:                                BRIEF IN FILE "DB1:[MURRAY]AVERAGE.MAP;6"
ESTIMATED MAP LENGTH:                      8. BLOCKS
                                                +----------------------+
                                                ! LINK RUN STATISTICS !
                                                +----------------------+

PERFORMANCE INDICATORS                                    PAGE FAULTS   CPU TIME         ELAPSED TIME
----------- ----------                                    ---- ------   --- ----         ------- ----
    COMMAND PROCESSING:-                                           20   00:00:00.07      00:00:00.11
    PASS 1:-                                                       25   00:00:00.42      00:00:01.02
    ALLOCATION/RELOCATION:-                                         2   00:00:00.05      00:00:00.26
    PASS 2:-                                                        6   00:00:00.22      00:00:00.87
    MAP DATA AFTER OBJECT MODULE SYNOPSIS:-                         0   00:00:00.00      00:00:00.00
    SYMBOL TABLE OUTPUT:-                                           0   00:00:00.00      00:00:00.07
TOTAL RUN VALUES:-                                                 53   00:00:00.76      00:00:02.37

USING A WORKING SET LIMITED TO 180 PAGES AND 30 PAGES OF DATA STORAGE (EXCLUDING IMAGE)

TOTAL NUMBER OBJECT RECORDS READ (BOTH PASSES):   179
    OF WHICH 62 WERE IN LIBRARIES AND 8 WERE DEBUG DATA RECORDS CONTAINING 294 BYTES
267 BYTES OF DEBUG DATA WERE WRITTEN,STARTING AT VBN 6 WITH 1 BLOCKS ALLOCATED

THERE WERE 10 LIBRARY BLOCK READ OPERATIONS
    WHICH ENCOMPASSED A TOTAL OF 91 BLOCKS
    USING A WINDOW OF 10 BLOCKS

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
```



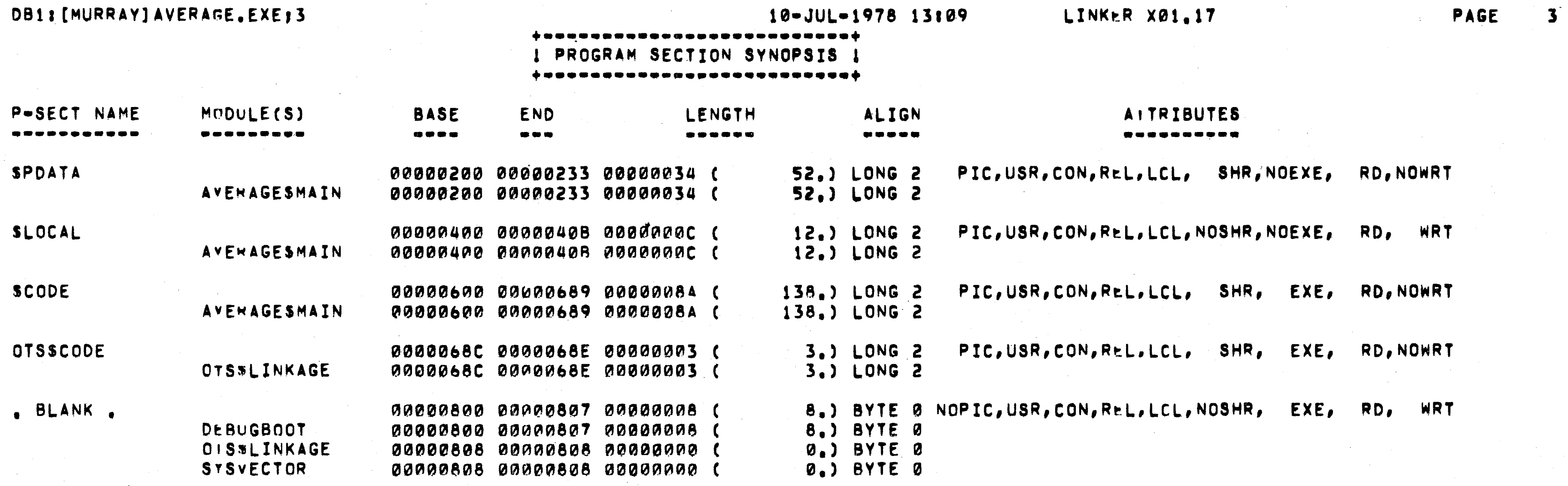
```
AVERAGE                                                         10-JUL-1978 13:10       LINKER X01.17                   PAGE    1
                                                  +-------------------------+
                                                  ! OBJECT MODULE SYNOPSIS !
                                                  +-------------------------+

MODULE NAME     IDENT               BYTES      FILE                                   CREATION DATE        CREATOR
-----------     -----               -----      ----                                   -------------        -------
AVERAGE$MAIN    01                    202 DB1:[MURRAY]AVERAGE.OBJ;2                 11-May-1978  0:12  VAX-11 FORTRAN IV-PLUS T0.7-92
DEBUGBOOT       01                      8 DBB2:[SYSLIB]DEBUG.OBJ;1                  02-JUN-1978 10:12  VAX-11 MACRO X0.3-10

                                                 +---------------------------+
                                                 ! PROGRAM SECTION SYNOPSIS !
                                                 +---------------------------+

P-SECT NAME     MODULE(S)           BASE     END              LENGTH           ALIGN                    ATTRIBUTES
-----------     ---------           ----     ---              ------           -----                    ----------
$PDATA                          00000200 00000233 00000034 (       52.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,NOEXE,  RD,NOWRT
                AVERAGE$MAIN    00000200 00000233 00000034 (       52.) LONG 2
$LOCAL                          00000400 0000040B 0000000C (       12.) LONG 2   PIC,USR,CON,REL,LCL,NOSHR,NOEXE,  RD,  WRT
                AVERAGE$MAIN    00000400 0000040B 0000000C (       12.) LONG 2
$CODE                           00000600 00000689 0000008A (      138.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,  RD,NOWRT
                AVERAGE$MAIN    00000600 00000689 0000008A (      138.) LONG 2
. BLANK .                       00000800 00000807 00000008 (        8.) BYTE 0 NOPIC,USR,CON,REL,LCL,NOSHR,  EXE,  RD,  WRT
                DEBUGBOOT       00000800 00000807 00000008 (        8.) BYTE 0

                                                   +------------------+
                                                   ! SYMBOLS BY NAME !
                                                   +------------------+

SYMBOL          VALUE               SYMBOL          VALUE               SYMBOL          VALUE               SYMBOL          VALUE
------          -----               ------          -----               ------          -----               ------          -----
AVERAGE$MAIN    00000600-R


        KEY FOR SPECIAL CHARACTERS ABOVE:
                +------------------+
                ! *  - UNDEFINED   !
                ! U  - UNIVERSAL   !
                ! R  - RELOCATABLE !
                ! WK - WEAK        !
                +------------------+
```



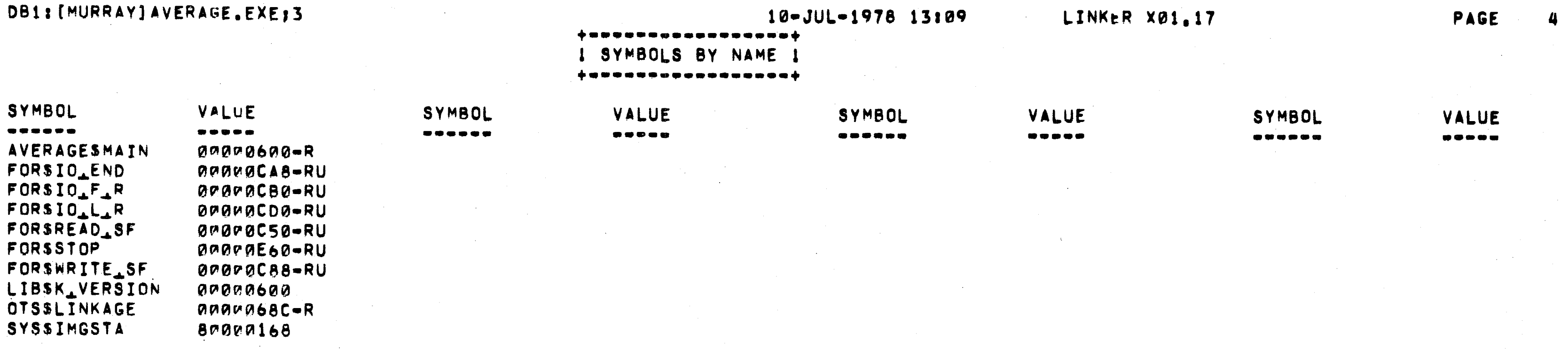
```
DB1:[MURRAY]AVERAGE.EXE;5                                   10-JUL-1978 13:10       LINKER X01.17                    PAGE    2
                                                           +-----------------+
                                                           ! IMAGE SYNOPSIS !
                                                           +-----------------+

VIRTUAL MEMORY ALLOCATED:                 00000200 000075FF 00007400 (29696. BYTES, 58. 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:            AVERAGE 01
NUMBER OF FILES:                                 4.
NUMBER OF MODULES:                               5.
NUMBER OF PROGRAM SECTIONS:                      9.
NUMBER OF GLOBAL SYMBOLS:                       10.
NUMBER OF IMAGE SECTIONS:                        8.
USER TRANSFER ADDRESS:                    00000600
DEBUGGER TRANSFER ADDRESS:                00000800
IMAGE TYPE:                               EXECUTABLE.
MAP FORMAT:                               DEFAULT IN FILE "DB1:[MURRAY]AVERAGE.MAP;5"
ESTIMATED MAP LENGTH:                     17. BLOCKS
                                                     +----------------------+
                                                     ! LINK RUN STATISTICS !
                                                     +----------------------+

PERFORMANCE INDICATORS                               PAGE FAULTS   CPU TIME       ELAPSED TIME
----------- ----------                               ---- ------   --- ----       ------- ----
    COMMAND PROCESSING:-                                      20   00:00:00.04    00:00:00.12
    PASS 1:-                                                  43   00:00:00.43    00:00:01.17
    ALLOCATION/RELOCATION:-                                    2   00:00:00.04    00:00:00.25
    PASS 2:-                                                   5   00:00:00.27    00:00:00.86
    MAP DATA AFTER OBJECT MODULE SYNOPSIS:-                    5   00:00:00.05    00:00:00.06
    SYMBOL TABLE OUTPUT:-                                      0   00:00:00.00    00:00:00.11
TOTAL RUN VALUES:-                                            75   00:00:00.83    00:00:02.62

USING A WORKING SET LIMITED TO 180 PAGES AND 30 PAGES OF DATA STORAGE (EXCLUDING IMAGE)

TOTAL NUMBER OBJECT RECORDS READ (BOTH PASSES):   179
    OF WHICH 62 WERE IN LIBRARIES AND 8 WERE DEBUG DATA RECORDS CONTAINING 294 BYTES
267 BYTES OF DEBUG DATA WERE WRITTEN,STARTING AT VBN 6 WITH 1 BLOCKS ALLOCATED

THERE WERE 10 LIBRARY BLOCK READ OPERATIONS
    WHICH ENCOMPASSED A TOTAL OF 91 BLOCKS
    USING A WINDOW OF 10 BLOCKS

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
```

```
AVERAGE                                          10-JUL-1978 13:09       LINKER X01.17                 PAGE    1
                                              +--------------------------+
                                              ! OBJECT MODULE SYNOPSIS !
                                              +--------------------------+

MODULE NAME     IDENT          BYTES     FILE                                 CREATION DATE      CREATOR
-----------     -----          -----     -----                                -------------      -------
AVERAGE$MAIN    01               202 DB1:[MURRAY]AVERAGE.OBJ;2              11-May-1978  0:12  VAX-11 FORTRAN IV-PLUS T0.7-92
DEBUGBOOT       01                 8 DBB2:[SYSLIB]DEBUG.OBJ;1              02-JUN-1978  1n:2  VAX-11 MACRO X0.3-10
OTS$LINKAGE     0-3                3 DBB2:[SYSLIB]STARLET.OLB;2            15-JUN-1978  14:5  VAX-11 MACRO X0.3-11
SYSVECTOR       02                 0 DBB2:[SYSLIB]STARLET.OLB;2            25-JUN-1978  1>:2  VAX-11 MACRO X0.3-11
VMSRTL          .FXF;14            0 DBB2:[SYSLIB]VMSRTL.EXE;2             10-JUL-1978 00:21  LINK-32 X01.17
```

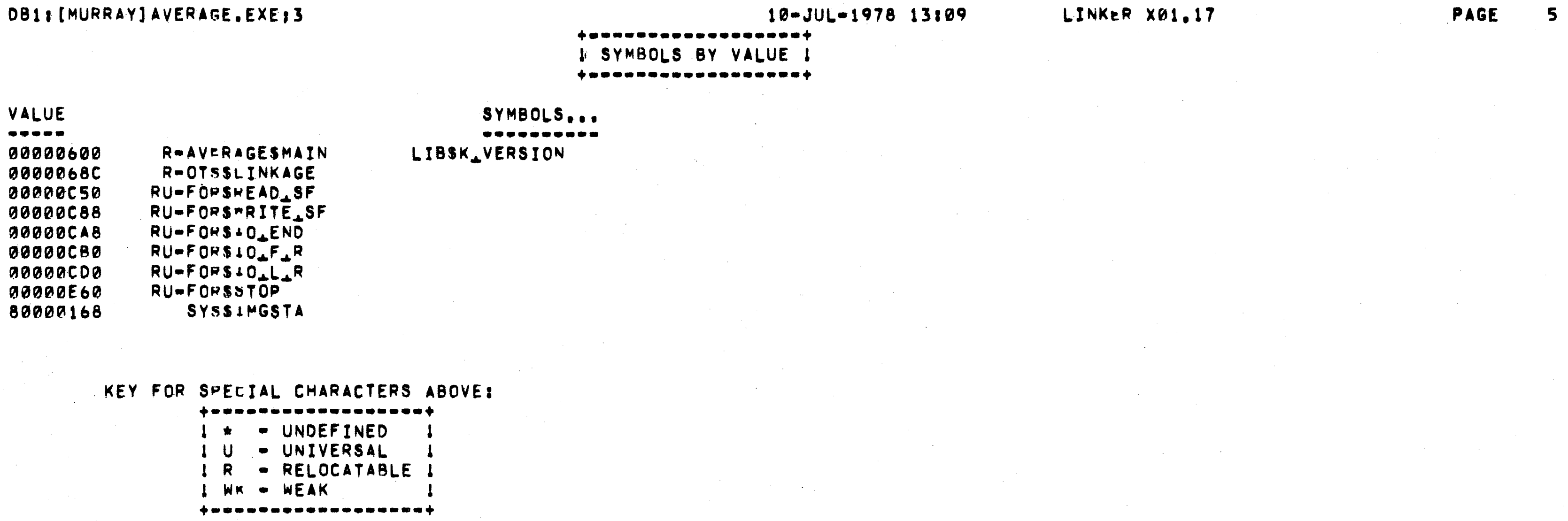
```
DB1:[MURRAY]AVERAGE.EXE;3                                  10-JUL-1978 13:09       LINKER X01.17                PAGE    2
                                                  +-------------------------+
                                                  ! IMAGE SECTION SYNOPSIS !
                                                  +-------------------------+

   CLUSTER       TYPE PAGES   BASE ADDR DISK VBN PFC PROTECTION AND PAGING      GBL. 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
                    0     1    00000800        5   0 READ WRITE   COPY ON REF
                  253    20    7FFFD800        0   0 READ WRITE DEMAND ZERO

VMSRTL              3     4    00000A00        0   0 READ ONLY                  VMSRTL_001       LESS/EQUAL         0        99
                    3    48    00001200        0   0 READ ONLY                  VMSRTL_002       LESS/EQUAL         0        99
                    4     2    00007200        0   0 READ WRITE   COPY ON REF   VMSRTL_003       LESS/EQUAL         0        99
```



```
DB1:[MURRAY]AVERAGE.EXE;3                                       10-JUL-1978 13:09        LINKER X01.17                    PAGE     3
                                            +--------------------------+
                                            ! PROGRAM SECTION SYNOPSIS !
                                            +--------------------------+

P-SECT NAME     MODULE(S)          BASE      END            LENGTH         ALIGN                 ATTRIBUTES
-----------     ---------          ----      ---            ------         -----                 ----------

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

$LOCAL                          00000400 0000040B 0000000C (      12.) LONG 2   PIC,USR,CON,REL,LCL,NOSHR,NOEXE,   RD,  WRT
                AVERAGE$MAIN    00000400 0000040B 0000000C (      12.) LONG 2

$CODE                           00000600 00000689 0000008A (     138.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,   RD,NOWRT
                AVERAGE$MAIN    00000600 00000689 0000008A (     138.) LONG 2

OTS$CODE                        0000068C 0000068E 00000003 (       3.) LONG 2   PIC,USR,CON,REL,LCL,  SHR,  EXE,   RD,NOWRT
                OTS$LINKAGE     0000068C 0000068E 00000003 (       3.) LONG 2

. BLANK .                       00000800 00000807 00000008 (       8.) BYTE 0 NOPIC,USR,CON,REL,LCL,NOSHR,  EXE,   RD,  WRT
                DEBUGBOOT       00000800 00000807 00000008 (       8.) BYTE 0
                OTS$LINKAGE     00000808 00000808 00000000 (       0.) BYTE 0
                SYSVECTOR       00000808 00000808 00000000 (       0.) BYTE 0
```



```
DB1:[MURRAY]AVERAGE.EXE;3                                           10-JUL-1978 13:09        LINKER X01.17                       PAGE     4
                                                           +------------------+
                                                           ! SYMBOLS BY NAME !
                                                           +------------------+

SYMBOL          VALUE                SYMBOL          VALUE                SYMBOL          VALUE                SYMBOL          VALUE
------          -----                ------          -----                ------          -----                ------          -----
AVERAGE$MAIN    00000600-R
FOR$IO_END      00000CA8-RU
FOR$IO_F_R      00000CB0-RU
FOR$IO_L_R      00000CD0-RU
FOR$READ_SF     00000C50-RU
FOR$STOP        00000E60-RU
FOR$WRITE_SF    00000C88-RU
LIB$K_VERSION   00000600
OTS$LINKAGE     0000068C-R
SYS$IMGSTA      80000168
```



```
DB1:[MURRAY]AVERAGE.EXE;3                                  10-JUL-1978 13:09       LINKER X01.17                  PAGE    5
                                                        +-------------------+
                                                        ! SYMBOLS BY VALUE !
                                                        +-------------------+

VALUE                                        SYMBOLS...
-----                                        ----------
00000600       R-AVERAGE$MAIN         LIB$K_VERSION
0000068C       R-OTS$LINKAGE
00000C50      RU-FOR$READ_SF
00000C88      RU-FOR$WRITE_SF
00000CA8      RU-FOR$IO_END
00000CB0      RU-FOR$IO_F_R
00000CD0      RU-FOR$IO_L_R
00000E60      RU-FOR$STOP
80000168          SYS$IMGSTA


        KEY FOR SPECIAL CHARACTERS ABOVE:
                +------------------+
                ! *   - UNDEFINED   !
                ! U   - UNIVERSAL   !
                ! R   - RELOCATABLE !
                ! WK  - WEAK        !
                +------------------+
```

FULL MAP

```
DB1:[MURRAY]AVERAGE.EXE;3                                    10-JUL-1978 13:09       LINKER XØ1.17                  PAGE    6
                                                     +-----------------+
                                                     ! IMAGE SYNOPSIS !
                                                     +-----------------+

VIRTUAL MEMORY ALLOCATED:                  ØØØØØ2ØØ ØØØØ75FF ØØØØ74ØØ (29696. BYTES, 58. PAGES)
STACK SIZE:                                      2Ø. PAGES
IMAGE HEADER VIRTUAL BLOCK LIMITS:                1.         1. (     1. BLOCK)
IMAGE BINARY VIRTUAL BLOCK LIMITS:                2.         5. (     4. BLOCKS)
IMAGE NAME AND IDENTIFICATION:             AVERAGE Ø1
NUMBER OF FILES:                                  4.
NUMBER OF MODULES:                                5.
NUMBER OF PROGRAM SECTIONS:                       9.
NUMBER OF GLOBAL SYMBOLS:                        1Ø.
NUMBER OF IMAGE SECTIONS:                         8.
USER TRANSFER ADDRESS:                     ØØØØØ6ØØ
DEBUGGER TRANSFER ADDRESS:                 ØØØØØ8ØØ
IMAGE TYPE:                                EXECUTABLE.
MAP FORMAT:                                FULL IN FILE "DB1:[MURRAY]AVERAGE.MAP;3"
ESTIMATED MAP LENGTH:                      26. BLOCKS
                                                  +----------------------+
                                                  ! LINK RUN STATISTICS !
                                                  +----------------------+

PERFORMANCE INDICATORS                                    PAGE FAULTS   CPU TIME         ELAPSED TIME
----------- ----------                                    ---- ------   --- ----         ------- ----
    COMMAND PROCESSING:-                                           15   ØØ:ØØ:ØØ.Ø7      ØØ:ØØ:ØØ.13
    PASS 1:-                                                       48   ØØ:ØØ:ØØ.47      ØØ:ØØ:Ø1.13
    ALLOCATION/RELOCATION:-                                         2   ØØ:ØØ:ØØ.Ø3      ØØ:ØØ:ØØ.32
    PASS 2:-                                                        7   ØØ:ØØ:ØØ.21      ØØ:ØØ:ØØ.88
    MAP DATA AFTER OBJECT MODULE SYNOPSIS:-                        11   ØØ:ØØ:ØØ.15      ØØ:ØØ:ØØ.14
    SYMBOL TABLE OUTPUT:-                                           Ø   ØØ:ØØ:ØØ.ØØ      ØØ:ØØ:ØØ.12
TOTAL RUN VALUES:-                                                 83   ØØ:ØØ:ØØ.93      ØØ:ØØ:Ø2.77

USING A WORKING SET LIMITED TO 18Ø PAGES AND 3Ø PAGES OF DATA STORAGE (EXCLUDING IMAGE)

TOTAL NUMBER OBJECT RECORDS READ (BOTH PASSES):   179
    OF WHICH 62 WERE IN LIBRARIES AND 8 WERE DEBUG DATA RECORDS CONTAINING 294 BYTES
267 BYTES OF DEBUG DATA WERE WRITTEN,STARTING AT VBN 6 WITH 1 BLOCKS ALLOCATED

THERE WERE 1Ø LIBRARY BLOCK READ OPERATIONS
    WHICH ENCOMPASSED A TOTAL OF 91 BLOCKS
    USING A WINDOW OF 1Ø BLOCKS

NUMBER OF MODULES EXTRACTED EXPLICITLY               = Ø
    WITH 2 EXTRACTED TO RESOLVE UNDEFINED SYMBOLS

Ø LIBRARY SEARCHES WERE FOR SYMBOLS NOT IN THE LIBRARY SEARCHED

A TOTAL OF Ø GLOBAL SYMBOL TABLE RECORDS WAS WRITTEN
```

APPENDIX C

VAX-11 OBJECT LANGUAGE

The object language description in this appendix is taken from DIGITAL software specifications.

C.1 INTRODUCTION

This document is a specification of the Object Language accepted by VAX-11 Linkers, Object Module Librarians, and Object Patch Utilities.

The Object Language specified herein is for use by all VAX-11 family software - i.e., no subsetting will occur. All language processors which produce code for execution in native mode are free to use any or all of the described functionality.

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) (Ignored by Release 1 of VMS Linker)

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

It is mandatory that there be at least two HDR records 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, TBK 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, as it is reserved for possible future use by the Linker and DEC. If the particular implementation of the Linker does not have a specification of use of such items, an error will be produced if such an item is encountered.

All unused and ignored fields of records must be padded to conform to the block lengths specified herein. 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 specification:

Type 7-100	Reserved for future use by Linker
Type 101-200	Ignored always and completely
Type 201-255	Reserved for CSS and customer use (Ignored by initial implementation)

MHD	Module Header Record
GSDi	Global Symbol Directory Record
TIR	Text Information and Relocation
TIR	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
EM	End of Module

Figure C-1
General Structure of an Object Module

This language is a development from RSX-11 systems. The reader who is not familiar with the RSX-11 Task Builders is referred to the documents listed.

C.2 GLOBAL AND UNIVERSAL SYMBOLS AND NAME FORMAT

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

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

As well as the names of symbols, the Object Language deals with the names of p-sections and object modules and may contain the names of language processors and utilities. All such names are represented by a 1-byte character count followed by the ASCII character string.

The first customer ship (FCS) implementation of the Linker limits such name strings to 15 characters, except in the case of header record types 1-255 (see below). The size of symbols and names, etc., is given by the parameter OBJ$C_SYMSIZ.

C.3 MODULE HEADER RECORDS (HDR)

This is a new type of record that is additional to the language used in RSX-11. Its purposes are to collect in one place all module-wide information, to include information never included by RSX-11, to permit more functionality in the Librarian and Patch utilities, and to permit extensibility of the language.

The MHD (Main 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
MAX RECORD SIZE	2 bytes
MODULE NAME	Variable (2-16 for FCS)
MODULE VERSION	Variable (2-16 for FCS)
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 Header Type

The language defines a general class of header records. Type 0 (OBJ$C_HDR_MHD) is the record depicted above and is required in every object module. Other types are described below.

C.3.2 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 which conforms to the previous format. The structure level is zero FCS.

C.3.3 Maximum Record Size OBJ$C_MAXRECSIZ

The size in bytes of the longest record that can occur within this object module. Limited by file system only. The FCS implementation sets a practical limit of 512 bytes.

C.3.4 Module Name

The module name conforms to the format of all other names, i.e., 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.3.5 Module Version

The module version conforms to the format of all names in the object language.

C.3.6 Dates And Times

There are two date and time fields - that for module creation and that of the last modification to the module (e.g., 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.3.7 Other Header Records

The purpose of sub-header 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 sub-type byte. The following sub-types are defined.

OBJ$C_HDR_LNM = 1 - Language Processor (LNM) Name and Version. One record is required and limited for FCS 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. (Ignored by FCS)

OBJ$C_HDR_TTL = 3 - Title text (e.g., brief module description). Only one record permitted. (Ignored by FCS)

OBJ$C_HDR_CPR = 4 - A copyright statement. Only one record permitted. (Ignored by FCS)

OBJ$C_HDR_MTC = 5 - Maintenance Status. (MTC) Multiple records permitted. (Ignored by FCS)

OBJ$C_HDR_GTX = 6 - General Text. Multiple records permitted. (Ignored by FCS)

Types 7-100 are reserved.

Types 101-255 always ignored.

C.3.8 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.3.9 Maintenance Status Header Record (MTC)

This record is of concern only to the object module patch utility. 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 2-16 bytes
UTILITY VERSION	variable 2-16 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.3.9.1 **Record Type** - Zero signifies a header record.

C.3.9.2 **Header Type** - The type is 5 signifying a maintenance status record.

C.3.9.3 **Patch Utility Name** - This name identifies the patch utility used to perform this patch on the module.

C.3.9.4 **Utility Version** - The patch utility is further identified by its version number.

C.3.9.5 **U.I.C.** - This is the user identification code under which the patch was made.

C.3.9.6 **Inut File Specification** - This filename identifies the input file for this patch.

C.3.9.7 **Correction File Specification** - This filename identifies the correction file for this patch.

C.3.9.8 **Date & Time** - This 17-byte field contains the date and time that this patch was performed. Format is as described above.

C.3.9.9 **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$C_GSD_PSC = 0 - P-section definition.
     OBJ$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 P-Section Definition (OBJ$C_GSD_PSC)

The format of a p-section definition is as follows:

RECORD TYPE 1	1 byte
GSD TYPE 0	1 byte
ALIGNMENT	1 byte
FLAGS	2 bytes
ALLOCATION	4 bytes
P-SECTION NAME	Variable 2-16 bytes

C.4.1.1 **P-Section Name** - This name has same format as all other symbol names.

C.4.1.2 **Alignment** - This field specifies the virtual address boundary at which the p-section will be placed.

0 BYTE
1 WORD
2 LONGWORD
3 QUADWORD
i.e., n 2**N BYTES

Where n=0 to 9

Nine indicates page alignment and is the limit for p-section alignment.

Each module contributing to a p-section can specify its own local alignment with the restriction that p-sections whose contributions overlay each other must all have the same alignment. It should also be noted that an alignment specified within a p-section (e.g., assembler .ALIGN directive) must be less than or equal to the p-section alignment to be guaranteed. For example, byte alignment of the p-section may or may not cause longword aligned elements within the p-section.

C.4.1.3 **Flags** -

Bit	Name	Use (meaning if set)
0	PSC$V_PIC	P-section defined as position independent.
1	PSC$V_LIB	The p-section was defined in the symbol table of a shareable image, to which this image is bound.
2	PSC$V_OVL	Contributions to the same p-section are overlaid. (The complement is concatenation).
3	PSC$V_REL	P-section requires relocation (complement, i.e., bit=0, means absolute and contains only symbol definitions, thus the allocation of an absolute p-section is zero).
4	PSC$V_GBL	Scope of p-section is global. (Complement is local).
5	PSC$V_SHR	P-section is potentially shareable between two or more active processes.
6	PSC$V_EXE	The content of p-section is executable.
7	PSC$V_RD	The content of the p-section may be read.
8	PCS$V_WRT	The content of the p-section may be written.
9-15		Reserved.

Discussions of p-section attributes may be found in the related documents. [See also Section 2.5.4 of this manual.]

C.4.1.4 **Allocation Field** - The allocation field contains the length contribution to the p-section in bytes. It must be zero for an absolute p-section.

P-sections are assigned an identifying sequence number as their respective GSD records are encountered. The p-section number ranges from 0 through 255 within any single module. Note, however, that the total number of p-sections in a single link operation is bounded only by the Linker's virtual memory requirements. This p-section number is used as an index in all references to the p-section. Note that this permits any mixture of GSD records, as long as p-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	
GSD TYPE 1	1 byte	
DATA TYPE	1 byte	
FLAGS	2 bytes	
PSECT INDEX	1 byte	5 bytes omitted for a reference (i.e. when SYM$V_DEF=0)
VALUE	4 bytes	
SYMBOL NAME	Variable 2-16 bytes	

C.4.2.1 **Data Type** - The data type record is encoded as described in Appendix C of the VAX-11/780 Architecture Handbook.

NOTE

The first implementation of the Linker ignores the data type field.

C.4.2.2 **Flags** -

Bit	Name	Use
0	SYM$V_WEAK	0 for strong resolution. 1 for weak resolution. Table C-1 describes the usage of SYM$V_WEAK in conjunction with the definition bit (SYM$V_DEF).
1	SYM$V_DEF	0 for reference 1 for definition
2	SYM$V_UNI	0 for within facility 1 for universal symbol This bit is only of significance on a definition. It indicates the symbol is to be retained if this facility is shareable.
3	SYM$V_REL	0 for absolute symbol value 1 for relative symbol and the value is augmented by the indexed p-section base address (of this module's contribution)
4-15		Reserved.

Table C-1
Interpretation of SYM$V_WK and SYM$V_DEF

SYM$V_WEAK	SYM$V_DEF	Interpretation
0	0	Strong Reference - symbol must be resolved
1	0	Weak Reference - only resolved 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 - will remain in all required symbol tables/maps.
1	1	Weak definition - will be discarded from all symbol tables/maps unless there was a reference. Will also not appear in the global symbol table index of an object module library.

C.4.2.3 **P-Section index** - The p-section index is a number between 0 and 255 to be used as an index into the sequence of p-section definition records. This field exists only for symbol definition records (SYM$V_DEF=1) and identifies the p-section in which the symbol was defined. The index is also used in TIR commands (see Section 5.1.1) for reference to p-section base addresses.

All symbols encountered must be defined within a p-section, independently of the relocatability of p-sections or symbols. For example, the Linker does not require the base address of the "owning" p-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 p-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 (SYM$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 2-16 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 5 of this appendix) 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 2-16 bytes
MIN ACTUAL ARGS.	1 byte
MAX ACTUAL ARGS.	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 which are in addition to other GSD records.

C.4.4.1 Minimum and Maximum Actual Argument Counts

C.4.4.1 Minimum and Maximum Actual Argument Counts - Permissible values are 0 to 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 function return value if such exists.

The FCS implementation does not validate procedure calls. However, for its own integrity it validates that 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$BVALCTL
REM. BYTE CNT.	1 byte ARG$BBYTECNT
DETAILED ARGUMENT DESCRIPTION	variable 0-255 bytes ignored by FCS 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$VPASSMECH - Describes the mechanism by which the argument of a valid call must be passed.

Bits 2:7 Reserved - Ignored by the FCS 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 FCS implementation, it is used as a count of bytes to be ignored by the linker. The content of these remaining bytes is of a format not specified here and reserved for possible future implementations.

NOTE: Any usage of formal argument descriptors in which

ARG$B_VALCTL bits 2:7 NEQ 0

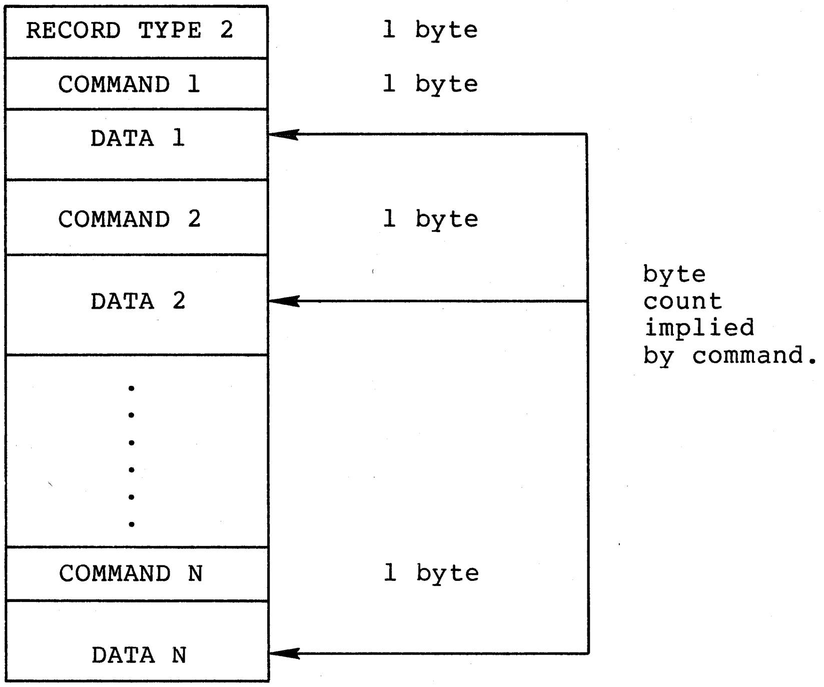
and/or

ARG$B_BYTECNT NEQ 0

means that, should argument validation be implemented in a future VAX-11 linker, re-compilation 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:



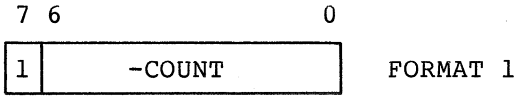
C.5.1 Commands

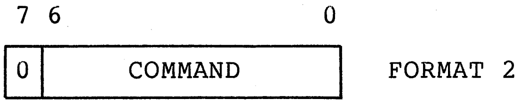
The Linker's creation of the binary content of an image file is completely driven by the language processor via the commands contained in TIR records. To achieve this, the Linker maintains an internal stack.

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 p-sections, the names are identified by the sequence numbers assigned to them as described above. The p-section indices are in the range 0 through 255.

The command byte has two formats:





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 to -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 upon 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 p-section. Except for stacking the value of global symbols or stacking addresses (calculated from p-sections), both signed extension to longword and zero extension to longword are provided for byte and word stack operations.

Code	Command	Description/Interpretation
0.	Stack Global (TIR$C_STA_GBL)	Symbol specification follows. As with all other names, it consists of the symbol length in a byte followed by the ASCII string defining the symbol: LENGTH 1 byte SYMBOL Variable 1-15 bytes The value found from the symbol table is a 32-bit quantity.
1.	Stack Signed Byte (TIR$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 p-section number followed by single signed byte offset. A 32-bit quantity is computed by addition of p-section base address and the byte offset.

Code	Command	Description/Interpretation
5.	Stack PSECT base plus word offset TIR$C_STA_PW)	1-byte p-section number followed by single signed word offset. A 32-bit quantity is computed by addition of p-section base address and the word offset.
6.	Stack PSECT base plus long word offset (TIR$C_STA_PL)	1-byte p-section number followed by signed longword offset. A offset. A 32-bit quantity is computed by addition of p-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 p-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$C_STA_UW)	As for TIR$CSTASW except that the value is zero extended to 32 bits.
9.	Stack Byte From Image (TIR$C_STA_BFI)	The longword on top of the stack is used as an address, in the image, from which to retrieve a byte. The byte is zero extended and replaces top longword of stack.
10.	Stack Word From Image (TIR$C_STA_WFI)	The word variant of previous command.
11.	Stack Longword From Image (TIR$C_STA_LFI)	Analogous to above.
12.	Stack Entry Point Mask (TIR$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$C_STA_CKARG)	The format of the command is as follows: COMMAND CODE SYMBOL NAME ARG INDEX ACTUAL ARGUMENT DESCRIPTOR The purpose of this command is to compare an actual argument descriptor with a formal descriptor for a particular procedure, stacking an indicator based upon match or mismatch of arguments. This indicator is TRUE if match is found or if there is no formal argument description. The indicator is FALSE if (and only if) the specified formal is described by a procedure definition but the description does not match the accompanying actual argument description. The argument that is checked is given by the index, and is thus number 0 through 255. The format of the actual argument descriptor is identical to that of the procedure definition GSD record described in section 4.4.2 above. The FCS linker compares only the fields ARG$VPASSMECH, 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$C_STO_LI)	One longword from stack, bits 31:6 MBZ. Single byte written to image.
27.	Store Position Independent Data Reference (TIR$C_STO_PIDR)	The longword on top of stack is assumed to be the address of a data item. It occurs in a non-executable p-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.
28.	Store Position Independent Code Reference (TIR$C_STO_PICR)	The longword on top of the stack is assumed to be the address of address of an item to which a 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.

Code	Command	Description/Interpretation
28. (Cont.)	Store Position Independent Code Reference (TIR$C_STO_PICR)	This and the previous command are discussed further in the references on generation of position independent images.
29.	Store Repeated Signed Byte (TIR$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$C_STO_RSW)	As above except that words are written.
31.	Store Repeated Longword (TIR$CSTORL)	Analogous to above.
32.	Store Arbitrary Field (TIR$CSTOVPS)	The bits 0 to (s-1) of the top longword are written to image starting at bit p of the current location. The command byte in the object module is followed by p and s (respectively) which are unsigned bytes such that 0 LEQ p+s LEQ 32. Only the specified bits of the image are altered. After the operation the location counter is the address of the byte containing bit (p+s) of the location modified.
33.	Store Unsigned Byte (TIR$C_STO_USB)	Same as TIR$C_STO_SB except that bits 31:8 must be zero.
34.	Store Unsigned Word (TIR$C_STO_USW)	Analogous to above (Bits 31:16 (Bits 31:16 MBZ).
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. Else, 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.

Code	Command	Description/Interpretation
38.	Store Word (TIR$C_STO_W)	If top longword is negative, bits bits 31:15 must be 1. Else bits 31:16 MBZ. One word is longword is popped from stack. This command permits any 16 bit value from -32768 to 65535 to be written to the image.
39.	Store Repeated Byte (TIR$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.
41-49.	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 two'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.

Code	Command	Description/Interpretation
60.	Insert field (TIR$C_OPR_INSV)	This command is analogous to the store of arbitrary bit field above. The only difference is that the target for bits from top of stack is the next longword on the stack, and location counter is therefore unaffected. Note that top longword is popped and that p,s are bytes following command in the TIR record.
61.	Arithmetic Shift (TIR$C_OPR_ASH)	The longword on top of stack is 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.
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, else right) of next long word on stack. Rotate count must have an absolute value between 0 and 32.
64.	Select (TIR$C_OPR_SEL)	Remove the top longword from the stack. If it has the value TRUE (low bit set) remove and discard the next longword on the stack. If the first longword removed has the value FALSE (low bit clear) copy the next longword on the stack to the one that follows. Thus, the command presumes there are three longwords on the stack. These are collapsed to a single longword which is the value of the second or third based on the value of the first.
65.	Re-define Symbol to current location. (TIR$C_OPR_REDEF)	The command has the same format as the TIR$CSTAGBL 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 re-definition 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.
82-127.	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 Differences From RSX-11

Note that TIR Records combine the information and capabilities of two types (TXT and RLD) of record used by the RSX-11 Task Builder. The result is a sequential writing of the output image and a more efficient object language. Note also the omission of the End GSD Record, the addition of Module Header Record, and the placement of Transfer Address at the end of the module.

In this specification there is also no mechanism for handling the RSX-11 assembler directive to obtain program limits. The usefulness of the LIMIT directive in VAX systems is questionable, and no proposal is made to deal with it in the Linker.

C.5.4 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 had 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 overlayed p-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, either due to the command or some Link time error, no image is being produced, 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 p-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

This record will be two or seven bytes, depending on existence of a transfer address. Note that the p-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 p-section.

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.7 DEBUGGER INFORMATION (DBG) RECORDS (OBJ$C_DBG)

The purpose of debugger information records is to allow the language processors to pass information concerning local variables, etc., 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 desabled 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 which are used if none were given by the user at link time. Such 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 ignored by the FCS Linker.

INDEX

VAX-11
Linker Reference Manual
AA-D019A-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.

Fold Here

Do Not Tear - Fold Here and Staple

digital

Software Documentation
146 Main Street ML5-5/E39
Maynard, Massachusetts 01754

digital

PRINTED IN USA