Prime™

SEG and LOAD Reference Guide

Revision 19.2

DOC3524-192P

SEG AND LOAD REFERENCE GUIDE

DOC3524-192

Second Edition

by

Anne P. Ladd

This guide documents the software operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 19.2 (Rev. 19.2).

Prime Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

HOW TO ORDER TECHNICAL DOCUMENTS

U.S. Customers

Software Distribution
Prime Computer, Inc.
1 New York Ave.
Framingham, MA 01701
(617) 879-2960 X2053

Prime Employees

Communications Services
MS 15-13, Prime Park
Natick, MA 01760
(617) 655-8000 X4837

Customers Outside U.S.

Contact your local Prime subsidiary or distributor.

PRINTING HISTORY – SEG AND LOAD REFERENCE GUIDE

Edition	Date	Number	Software Release
First Edition	May 1980	PDR3524	17.2
Update 1	December 1980	PTU77	18
Update 2	July 1982	PTU83	19
Second Edition	June 1983	DOC3524-192	18, 19

Because the revisions are extensive, no change bars are included in the present edition.

SUGGESTION BOX

All correspondence on suggested changes to this document should be directed to:

Anne P Ladd
Technical Publications Department
Prime Computer, Inc
500 Old Connecticut Path
Framingham, Massachusetts 01701

CONTENTS

APPENDIXES

ABOUT THIS BOOK

PURPOSE AND AUDIENCE

This is a detailed reference guide for Prime's linking loader utilities, SEG and LOAD. It is intended for application programmers and system programmers who need information beyond the scope of the summary information in each of Prime's language reference guides.

SEG is Prime's segmented loading utility, for generating segmented runfiles to execute in the V or I addressing modes. LOAD is Prime's R-mode loading utility, for generating runfiles to operate in the R addressing mode. Most programmers will use SEG, in order to take advantage of virtual memory and the efficiency of the V-mode instruction set.

This book documents SEG and LOAD at software Revision 19.2. However, users who are still on a lower revision of software should find that the present book is sufficient to explain all features of Revisions 18 and 19. The current edition of this book is intended to replace every previous edition for users of Rev. 18 and 19.

OTHER USEFUL BOOKS

Readers should be familiar with at least Chapters 1 through 7 of the **Prime User's Guide** (DOC4130-190), and with the reference guide for one Prime programming language or for PMA, Prime's assembly language. In addition, this book assumes that the reader has access to the **Subroutines Reference Guide** (DOC3621-190).

PRIME DOCUMENTATION CONVENTIONS

The following conventions are used in command formats, statement formats, and in examples throughout this document. Command and statement formats show the syntax of commands, program language statements, and callable routines. Examples illustrate the uses of these commands, statements, and routines in typical applications. Terminal input may be entered in either uppercase or lowercase.

Convention	Explanation	Example
UPPERCASE	In command formats, words in uppercase indicate the actual names of commands, statements, and keywords. They can be entered in either uppercase or lowercase.	SLIST
lowercase	In command formats, words in lowercase indicate items for which the user must substitute a suitable value.	SEG pathname
Abbreviations	If a command or statement has an abbreviation, it is indicated by rust color in list of commands.	MAP [option]
Rust color in examples	In examples, user input is rust color but system prompts and output are not.	OK, SEG -LOAD
Brackets [ ]	Brackets enclose a list of one or more optional items. Choose none, one, or more of these items (0 to **n**).	SPOOL $\begin{bmatrix}\text{-LIST}\\\text{-CANCEL}\end{bmatrix}$
Default Indicator •	In a list of options, the default, if one exists, is indicated by a bullet (•).	MIX $\begin{bmatrix}\text{ON •}\\\text{OFF}\end{bmatrix}$

Convention	Explanation	Example
Braces { }	Braces enclose a vertical list of items. Choose one and only one of these items.	CLOSE $\begin{Bmatrix} \text{filename} \\ \text{ALL} \end{Bmatrix}$
Ellipsis ...	An ellipsis indicates that the preceding item may be repeated.	item-x[,item-y]...
Parentheses ( )	In command or statement formats, parentheses must be entered exactly as shown.	data-name (index)
Hyphen -	Wherever a hyphen appears in a command line option, it is a required part of that option	PASCAL name -LIST
Angle brackets < >	Angle brackets must be used as shown to separate the elements of a pathname.	<FOREST>BEECH>LEAF4

Filename Conventions

Convention	Explanation	Example
filename.language or filename	Source file	MYPROG.PASCAL
filename.**BIN** or B_filename	Binary (object) file	MYPROG.BIN
filename.**LIST** or L_filename	Listing file	MYPROG.LIST
filename.**SEG**	Saved executable runfile (V-mode)	MYPROG.SEG
filename.**SAVE**	Saved executable runfile (R-mode)	MYPROG.SAVE

Filenames may be comprised of 1 to 32 characters inclusive, the first character of which must be nonnumeric. Names should not begin with a hyphen (-) or underscore (_). File names may be composed only of the following characters: A-Z, 0-9, _ # $ & - * . and /.

See Chapter 2 for an explanation of how the various names for source, object, listing, and runtime files relate to each other.

Note

On some devices, the underscore (_) may print as back arrow (←).

PART I

SEG

1
OVERVIEW OF SEG

WHAT IS IN PART I OF THIS BOOK

Part I presents SEG, Prime's virtual-mode loader and execution utility. The following topics are covered in Part I:

- Chapter 1 presents an overview of SEG's many functions and a discussion of some features of Prime's virtual memory, architecture, and software that affect SEG.
- Chapter 2 discusses what happens on a default load.
- Chapter 3 shows how to read load maps produced by SEG.
- Chapter 4 presents, with examples, a series of advanced loading techniques by which programmers can take advantage of special features of SEG for space saving, sharing, and management of stacks, common blocks, and base areas.
- Chapters 5 through 7 list all commands and subcommands available through SEG.
- Chapter 8 discusses various error messages: not only those produced by SEG but also those from PRIMOS when the cause can be corrected through SEG.

In addition, the appendixes present more aids in using some features of SEG. Appendix G is a glossary of terms.

OVERVIEW OF SEG

SEG is so named because it takes a program that has been divided into segments and assigns an address within a **virtual segment** (described below) to each program segment. The default load procedure described in Chapter 2 allows the user to ignore segmentation, as this procedure handles it automatically. However, knowledge of SEG's procedures often allows a user to create more efficient runfiles.

The Loader Function

SEG works on **object files**, executable binary files that are produced by compilation or assembly from any Prime-supported language. An object file may contain one or more binary modules.

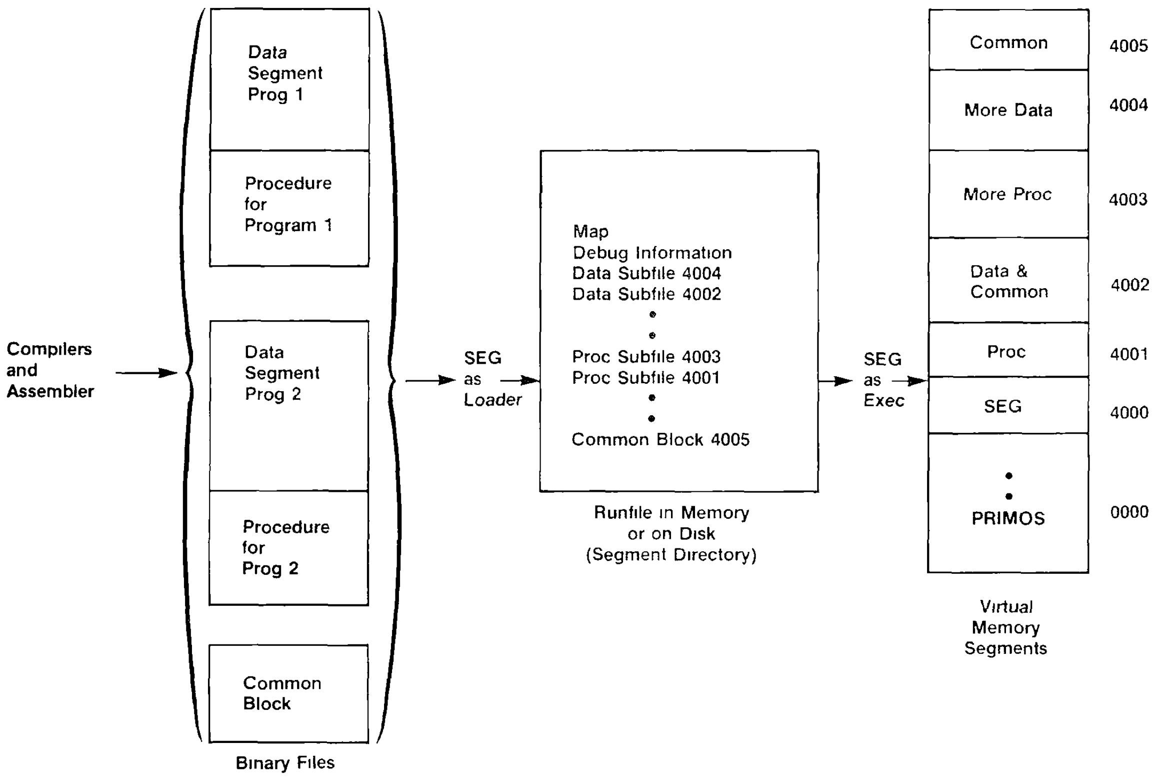
SEG includes a loader that marks modules for specific virtual segments and produces a **runfile** or executable file in the form of a **segment directory**. The organization of the runfile is explained below.

SEG has a **relocating loader** — it does not load the program into a predetermined address, but can start a program module almost anywhere. The addresses within the binary module are not absolute addresses but rather are displacements from the beginning of that module. Thus two or more separately compiled programs can be loaded without overlapping and without the programmer having thought beforehand of giving them different starting addresses.

SEG's loader is also a **linking loader**. As it loads parts of a runfile, it can check to see that all **references** or names within one module are defined by a data name, common block, or procedure name in the same module or elsewhere within the runfile. If the reference is not defined within the same module, it is called an **external reference**.

SEG maps a runfile into Prime's virtual addressing scheme, rather than loading it immediately into the segments where it will run. When SEG stores the runfile, it stores the notation of where in memory each part of it should be reloaded. The runfile may be new or may be a previously used runfile, and may be in any directory. Chapter 2 discusses how to name the runfile.

SEG is also used to execute these runfiles. Figure 1-1 represents the effects of SEG on a group of binary object files, as it first loads them into a runfile and then moves that runfile into virtual memory and executes it. The compiler or assembler produces object modules, each consisting of procedure separated from data. The compiler may also produce common blocks. SEG creates a runfile containing, in arbitrary order, subfiles that may be imagined as tagged with the number of the virtual segment that each will occupy at runtime. At runtime, the code is moved into the proper virtual segment for execution.



Sample Input to and Output from SEG
Figure 1-1

Organization of the Runfile: Segment Directories

Each runfile created by the default SEG procedure is stored as a **segment directory** The subfiles in a segment directory make up a collection of segments Each virtual segment in a segmented runfile consists of up to 32 subfiles of 4096 bytes each SEG can create a runfile as large as 256 segments or 8192 subfiles Subfile 0 of the runfile is used for startup information, the load map discussed below, and the subfile map Subfile 1 has DBG information if the program was compiled with the -DEBUG option Executable subfiles begin in subfile 2

If you give this runfile a name ending in SEG, you will remember that this is not a simple file and must be manipulated differently The SEG utility will also be able to use the naming conventions discussed in Chapter 2 If you use the command LD on your directory at Rev 19, you will see that segment directories are listed separately from simple files

There are several differences between segment directories and simple files In revisions lower than 19, a segment directory cannot be deleted by the PRIMOS command DELETE Instead, use SEG's own DELETE command, or the TREDEL command of the FUTIL utility TREDEL is slower than SEG's DELETE, but may be necessary if load pointers were destroyed by aborting

SEG. The **Subroutines Reference Guide** discusses which subroutines can be used with segment directories.

Other Functions SEG Can Perform

The rest of this chapter is devoted to describing the architecture and software features of PRIMOS with which SEG interacts. First, however, here is a brief list of functions available through the SEG utility. Appendix A describes these functions in more detail, with the commands involved in each function. Chapter 4 describes, with examples, many of the special techniques.

Produce Optimized Runfiles in Many Forms:

- If you don't want the default load of instructions and data into separate segments, you can create runfiles with no division of function. These files are usually smaller than runfiles created with the default loading method. For example, assume you have 100 programs, of which 20 might be running all the time, with many command streams changing programs frequently. It would be worthwhile to try to get each one to occupy as little virtual memory as possible, so that the system would not be allocating more virtual segments than necessary.
- An optimized runfile may have a smaller, faster execution unit than SEG. SEG includes a small execution unit called RUNIT. This unit can be loaded into a runfile, and the whole file can occupy segment '4000 instead of putting SEG in '4000 and the runfile in other segments. Execution is faster, and allows execution with RESUME as well as execution of the program as a user-defined PRIMOS command. The resulting runfiles are sometimes called **single-segment runfiles**, **sequential runfiles**, or **R-mode-like runfiles**, but none of these names accurately describes all files created by this method. This book calls them **RUNIT files**.
- Optimization can include relocating data and symbols in the runfile. Certain blocks of data, called common blocks, are put by the loader into segments separate from those used for other program data.
- An object file can be loaded into a specified segment or a relative segment.
- Data or a procedure can be loaded on a page boundary to reduce search time.
- Base area allocation can be controlled. The **base area** is that part of a procedure segment used for reference in indirect addressing instructions. It is normally at the beginning of each procedure segment, and the user need not be concerned with it unless SEG returns the message SECTOR ZERO BASE AREA FULL.

Prepare Shared Programs: If you have a program to be used concurrently by several users, you can share it with your System Administrator's authorization.

Change Stack Space and Location: If the stack runs out of space (PRIMOS error message STACK_OVF$), or if too much space is allocated for it in a small program, you can change its size, change its location, set minimum stack size, or specify an extension to the stack.

Produce Load Maps: A **load map** is a list of the segments being used by a particular runfile, with the address within each segment of the procedures and data sections that were loaded. SEG allows different map options, including one that lists only unresolved references and two that list only symbols. Users should learn to read load maps and to use the different map options, both for debugging loads and individual programs, and for studying memory allocation and how it can be optimized. Load maps can also be used for correcting errors signalled by PRIMOS error messages.

Make Templates: A template is a group of routines that you plan to use with several programs. It serves the function of a private library. This procedure can include forced loading of all routines in a binary file.

Execute Runfiles: Default runfiles may be executed with SEG.

Get Help: SEG has a HELP option that lists all commands and subcommands available.

Debug Runfiles: This can include:

- Invoke VPSD for debugging. VPSD is a symbolic debugger described in the **Assembly Language Programmer's Guide**.
- Check for unresolved references (calls to subprograms or subroutines).
- Restart a load after an error, overlaying any work already done.
- Restart a program at a certain address with the PRIMOS command START after having interrupted it with CONTROL-P (BREAK).

Change Runfiles: The following options are available:

- Load an object file (including library files) into a runfile.
- Duplicate the parameters of the previous load. The purpose is to avoid retyping.
- Modify runfiles. You can patch, save, restore, or copy a runfile, or change the starting address of a runfile, or its stack size and location. You can also add or replace modules in an existing runfile, or restore runfiles in order to modify them without execution. You can delete symbols in the runfile.
- Create a new runfile starting from one that already exists.
- Name, save, or delete a runfile.

Check attributes of a runfile: The version, date last modified, and runtime parameters can be checked. The **runtime parameters** are the starting address, stack location, keys, and contents of the A, B, and X registers.

PRIMOS ARCHITECTURE AND VIRTUAL MEMORY

This section describes system architecture and virtual memory as used by SEG. The SEG utility is designed to take full advantage of certain features of PRIMOS available in V-mode and I-mode.

V-Mode and Other Addressing Modes

Prime hardware allows several different addressing modes. They include:

32R, 64R	R-mode	Relative addressing mode, using 16-bit registers that can address 128K bytes of memory. Some PRIMOS utilities are still written in R-mode.
64V	V-mode	Virtual (segmented) addressing mode, using 32-bit registers, and allowing 512 megabytes of virtual memory to be addressed. PRIMOS itself is V-mode.
32I	I-mode	Integer mode (also called instruction or immediate mode). This has the same addressing range as V-mode, but is more efficient for decimal instructions.

The 64V and 32I modes are more efficient that R-mode and are recommended for all user-written code on the Prime 50 series. For details on these addressing modes, see the **Assembly Language Programmer's Guide**.

SEG normally handles only 64V and 32I modes. For this reason, programs that use another mode cannot call or be called by programs in these modes. If you attempt to use SEG to load a FORTRAN or PMA program compiled or assembled in 32R mode, you get the message:

```
CAN'T LOAD IN 32R MODE
```

The program can usually be recompiled with the -64V option or reassembled with minor changes in its code.

V- and I-mode programs consist of several separate **frames** or areas, named after the register that addresses each. These are:

Procedure frame	Executable code and stack segment headers.
Link frame	Static data (also called linkage area because it contains linkage information about the locations of external procedures and static external storage). It also contains the information on the executable code itself, in the form of an entry control block ECB).
Stack frame	Dynamic variables.

SEG is designed to take full advantage of V-mode and I-mode program structure. Chapter 2, DEFAULT LOADS, shows how these frames of the program are placed in segments automatically on a default load. However, you can use SEG to place them differently.

Segmentation and Paging

PRIMOS provides **virtual memory**. For the SEG user, this means that a much larger amount of memory is available for programs than the physical amount in the machine. To make use of the virtual memory capability, the program is broken into virtual segments of 128K (131072) bytes each. They are called virtual segments because they may be stored in a file as well as loaded into physical memory segments. PRIMOS automatically moves into physical memory the parts of the program that are needed at any one time. (Even if the program is smaller than one segment, SEG loads it into at least two different segments unless you change this default procedure with some of the commands discussed above in **Other Functions SEG Can Perform**.)

A segment is further divided into 64 **pages** of 2K (2048) bytes each, and it is these pages that are moved into live memory as needed. The activity of moving pages into memory (and storing them back on disk if they were changed while in live memory) is called **paging**. The pages are usually first copied out of your disk files onto a special paging disk. A constant movement of pages from the paging disk into live memory will slow execution time. There are ways in which special functions of SEG can reduce the amount of paging. These ways are listed above in **Other Functions SEG Can Perform**.

Relative and Absolute Segment Numbers

The user may either specify the virtual segment number for each part of the program, or may leave the assignment to SEG. If the segment number is specified, it is called an **absolute** segment number and the procedure is called absolute loading. If the assignment is left to SEG, the segment number is called a **relative** segment number. In Chapters 5, 6, and 7, the discussion for each command tells whether that command may be used with both absolute and relative segments, and how to make the distinction. In a default load, of course, the user leaves all assignments to SEG. In loads where every byte of memory is needed, or where shared segments are desired, every module of a load may be given a specific absolute segment number. Sometimes, however, the user may merely wish to specify, for example, that common blocks be in separate segments from other data, but may not care what the segments are. In that case, the user could specify separate relative segment numbers (such as 1 and 2), and SEG would use separate segments, with numbers in the range of '4000 to '4777. An example is in Chapter 4.

Organization of Memory

Prime's virtual memory is divided into four areas, defined by tables associated with registers called Descriptor Table Address Registers (DTARs). The segments defined within these DTARs are as follows:

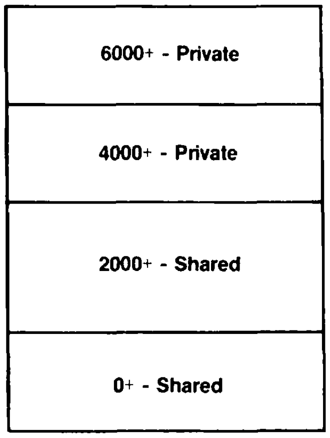
Area	**Type**	**Maximum Range of Segments***	**Usage**
0	Shared	'0 - '1777	PRIMOS code and buffers
1	Shared	'2000 - '3777	Shared libraries and programs
2	Private	'4000 - '5777	User space
3	Private	'6000 - '7776	Impure user data and stacks

*The current range varies with each software revision. In Rev. 19.2, only segments beginning with an even number are used.

Areas 0 and 1 are unique within virtual memory. This means that every user accesses the same segment when any PRIMOS command is executed or a shared library or compiler is used.

Areas 2 and 3 exist separately for every user. Note that Area 2 is variable in size, depending on the version and configuration of the system. The default allows a maximum of '40 (decimal 32) segments. Area 3 is defined by Prime. Anything in this space is used and defined by PRIMOS for the user program.

Thus the memory available to a single user might be imagined as in Figure 1-2.



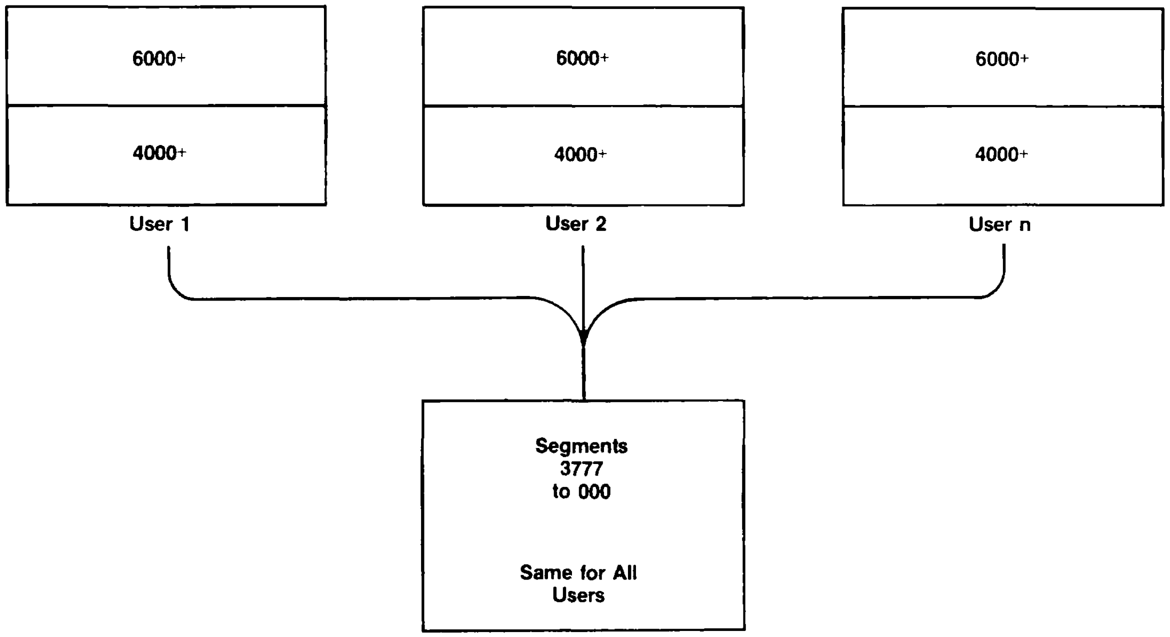
Memory Available to One User
Figure 1-2

On a default load, SEG uses only segments in the '4xxx range. In addition, PRIMOS uses some in the '6xxx range.

Sharing

Most software furnished by Prime is **shared**. This means that if multiple users call it, only one copy of the program is paged into a memory area common to all of these users, thus saving considerably on paging time. User-written programs may also be shared by the System Administrator at the supervisor terminal. These programs must be assigned to shared segments. Which segments are shared depends on the revision of PRIMOS being used. In general, as described in the preceding subsection, segments numbered between '0000 and '3777 are shared and those numbered '4000 and above are not. There is, therefore, one copy of the lower-numbered segments, while every user has a copy of the higher-numbered segments, which is switched in and out as a timeslice is allocated to each user.

A system with several users has a segment usage that may be imagined as something like Figure 1-3.



Shared Virtual Memory
Figure 1-3

SEG may be used to load parts of programs (either procedure or data) in the '2xxx range. This is illustrated for shared programs and data in Chapter 4.

Restriction on User Segment Allocation

PRIMOS has a pool of segments available to all users concurrently. When a user needs another segment, PRIMOS removes a segment from its pool and assigns it to the user, giving it the segment number requested by the user. Until the user logs out or explicitly deletes the segment, it can only be referenced by that user. PRIMOS has a limited number of segments in the available segment pool. This number varies with the configuration of each system. When a user program finishes executing, its segments remain assigned to that user until logout, and are not available to other users. It is good practice to use the PRIMOS command DELSEG after execution of a large program to release its segments. Otherwise that number of segments remains assigned to the user until logout, precluding their use by anyone else. If not enough segments are available in the common pool, the error message INSUFFICIENT SEGMENTS is displayed.

SOFTWARE FEATURES OF THE OPERATING SYSTEM

Two software features of PRIMOS should be mentioned in a discussion of what SEG can and cannot do.

Pure and Impure Code

A piece of code that is not modifiable is called **reentrant** or **pure**. Such a program or subroutine may call itself, or may call a program that calls the original caller. This is called **recursion**. The significance of this feature is that several users may share the code of one program if it is pure. In all cases, a return from a subroutine call will go back to the correct calling program. A following call will not end up with the results of previous computations. To be pure, a routine cannot have the return address of its caller modified while in memory. (This would be called **impure** code.) The most common way to assure reentrancy is to separate procedure and data into different segments and declare the procedure to be unmodifiable and reentrant, while data segments use a separate stack for each subroutine call. Then with each call the calling module gets a new copy of the local data variables.

All user binary code produced by the F77, BASIC, COBOL, Pascal, and PL1G compilers is normally reentrant or pure. The only compiler that produces impure code is the FORTRAN IV (FTN) compiler when used without the -DYNM or -64V command option. PMA also may produce impure code. Thus, a few older operating system subroutines written in FTN or PMA have impure code. Some of these are used by SEG and by the operating system. In default loads, these cause no problem. Two separate system libraries with the alluring names of the **pure FORTRAN library** (PFTNLB) and the **impure FORTRAN library** (IFTNLB) are maintained so that the proper kind of code can automatically be loaded where it is needed.

If you do not use the default load procedure of SEG, be aware that you are responsible for assuring reentrancy of code if you want it. This is usually necessary only when creating templates. To handle subroutines that have such mixed code, use the VLOAD subcommands IL and PL to load them into the correct segments of your program. This is explained in Chapter 5.

Direct Entry Links

One class of external reference is not resolved by the loader. This is a **direct entry link**. Many commonly used operating system subroutines such as EXIT and TNOUA are not actually loaded with the program calling them. They are coded as part of the PRIMOS operating system with a label that allows the program to find them at runtime. The subroutine identified by the label is a **direct entry point** into PRIMOS. This resolution at runtime is referred to as dynamic linking or **snapping a link**. Since the subroutine itself is not included in the runfile, the runfile is smaller and the program starts faster. In addition, the subroutine is shared by all users on the system. Finally, when a new version of the subroutine is installed, all programs that use it will immediately start using the new version without having to be recompiled or reloaded. SEG does not load direct entry links, but it recognizes them as such and lists them in a separate category in its load map.

WHAT IS NEXT

This chapter provided a theoretical background for users of SEG, with overviews of the features of PRIMOS architecture and software that help understand the various options of SEG. The following chapters of Part I are dedicated to practical examples of different cases. Chapter 2 shows how SEG works with virtual memory on a default load. Chapter 3 illustrates how load maps show what is going on in virtual memory. Chapter 4 discusses how to do special loads that alter SEG's interaction with PRIMOS features. Chapters 5 through 7 list all subcommands of SEG.

2 DEFAULT LOADS

The term **default load** is used for the loading sequence that uses the fewest user commands and all possible default segments and filenames that SEG can supply. This chapter first presents the normal default load sequence available for software revisions 18 and higher, and then reviews the older sequence that was the normal default load for revisions lower than 18. The intent of the first sections is to provide a list of the mechanical steps necessary for a default load.

What goes on in memory during a default load is diagrammed in Figure 2-1 later in this chapter, with segment allocation discussed in some detail.

DEFAULT LOADING AND EXECUTION

Most loads can be accomplished with the following procedure, which uses filename conventions valid for PRIMOS software revisions 18 and higher. The procedure assumes an object or binary file whose name ends with .BIN. Such files are produced by all Prime compilers and the assembler from source files whose names use the filenaming conventions shown in Table 2-1. Files with other name formats may be renamed to take advantage of this filenaming convention. The steps of this load are:

1. Invoke SEG from PRIMOS level with the -LOAD option to enter the VLOAD subprocessor.
2. Use the VLOAD subprocessor's LOAD command to load the binary file (filename.BIN) and any separately compiled subroutines or called programs. The filename entered need not be followed with .BIN, as SEG looks first for **filename.BIN**, then for **filename**. Pathnames may be used.

At this point the runfilename **filename.SEG** is automatically generated in the current UFD. If a runfile of that name already exists, it is overwritten.

3. Use the LIBRARY subcommand of VLOAD to load language libraries and any special libraries such as VSRTLI or VAPPLB. All languages except FORTRAN and PMA require that a language library be loaded, as shown in Table 2-1.

   The command LIBRARY acts like LOAD except that it seeks files in the system UFD called LIB. Thus, the command LI PASLIB seeks a file with the pathname LIB>PASLIB[.BIN].

4. Use the LIBRARY subcommand with no argument to load the system libraries IFTNLB, PFTNLB, and SPLLIB.

5. Enter QUIT to save the runfile and exit from SEG to PRIMOS level. Or, use EXECUTE to start execution of the runfile.

6. To execute the program subsequently, use the command **SEG filename**. SEG will look first for a file named **filename.SEG**, then for **filename**.

Table 2-1
File Naming Conventions and Libraries
for Each Prime Language in Rev. 18 and Higher

Language	Source Name	Library Needed	Object Name	Runfilename
COBOL	pgm.COBOL	VCOBLB	pgm.BIN	pgm.SEG
CBL (COBOL 74)	pgm.CBL	CBLLIB	pgm.BIN	pgm.SEG
FTN	pgm.FTN	none	pgm.BIN	pgm.SEG
F77	pgm.F77	none	pgm.BIN	pgm.SEG
Pascal	pgm.PASCAL	PASLIB	pgm.BIN	pgm.SEG
PL1G	pgm.PL1G	PL1GLB	pgm.BIN	pgm.SEG
PMA	pgm.PMA	none	pgm.BIN	pgm.SEG
VRPG	pgm.VRPG	VRPGLB	pgm.BIN	pgm.SEG

If loading is successful, entering the command LI(BRARY) in Step 4 will produce the message LOAD COMPLETE. If this message is not produced, enter the subcommand MAP 3 to identify the unsatisfied subroutine, program, or common block references. If necessary, use INITIALIZE to start again from Step 2, this time loading all files in the correct order.

Note

The PRIMOS filenames given by SEG may be up to 32 characters in length. However, program-names used within the programs have a maximum of eight characters that can be recognized by SEG.

Load Sequence

It is important to load modules of a runfile in correct sequence. A calling module should always be loaded before the module it calls. The normal loading sequence is:

1. Main user program
2. User subprograms in the order in which they are called
3. Compiler library or libraries, if any (PL1GLB, VCOBLB ...)
4. Any special subroutine libraries, such as VAPPLB or VSRTLI
5. The system libraries IFTNLB, PFTNLB, and SPLLIB

Error Messages

If loading is not successful, the command QUIT will cause SEG to display the error message WARNING: LOAD NOT COMPLETE. To rectify the omission, start over from Step 1. After Step 4, enter MAP 3, as discussed in the preceding paragraph.

If an error occurs during an operation, SEG prints an error message, followed by the prompt character. Few errors made during a default load would cause a return to PRIMOS level. Error messages and suggested handling techniques are discussed in Chapter 8.

Examples of Default Loading Sequences

The default loading sequence leads the user directly to the VLOAD subprocessor of SEG, so that the prompt displayed is always the dollar sign ($).

To load and run a Pascal binary file named TREE.BIN, use the following sequence:

```
OK, SEG -LOAD          /* INVOKE LOAD SUBPROCESSOR OF SEG
[SEG rev. x.x]
$ LOAD TREE            /* LOAD MAIN BINARY FILE
$ LIBRARY PASLIB       /* LOAD LANGUAGE LIBRARY
$ LIBRARY              /* LOAD SYSTEM LIBRARIES
LOAD COMPLETE
$ EXECUTE              /* EXECUTE FROM WITHIN VLOAD
OK,                    /* PRIMOS-LEVEL PROMPT -- PROGRAM HAS EXECUTED
                       /* SUCCESSFULLY
```

To load and execute FORTRAN or PMA programs, no special library need be loaded. The following example shows how to load a program whose source is named EXPO.PMA. The binary file, EXPO.BIN, is not in the current UFD.

```
OK, SEG -LOAD
[SEG rev. x.x]
$ LO ANNE>OBJ>EXPO
$ LI
LOAD COMPLETE
$ EXEC
OK,
```

The next example shows loading and execution of a FORTRAN program that uses one of the application library subroutines listed in the **Subroutines Reference Guide**. It thus needs VAPPLB to run.

```
OK, SEG -LOAD
[SEG rev. x.x]
$ LO MYPROG
$ LI VAPPLB
$ LI
LOAD COMPLETE
$ EXEC
OK,
```

To load and save but not execute a COBOL 74 binary file named MYPROG.BIN, which is stored in a sub-UFD with a password, enter:

```
OK, SEG -LOAD
[SEG rev x.x]
$ LO '*>LADD SECRET>MYPROG'
$ LI CBLLIB
$ LI
LOAD COMPLETE
$ QUIT
OK,
```

To execute the program, enter:

```
SEG MYPROG
```

The following load of a PL1G program and user-written subroutine does not get a LOAD COMPLETE message, so the user does a partial map:

```
OK, SEG -LOAD
[SEG rev x.x]
$ LO MYPROG
$ LO MYSUBROUTINE
$ LI
$ MAP 3
**P$LOUT  4002  000061  **P$TER  4002  000063  **P$PUTF 4002 000067
**P$EOUTF 4002  000071  **P$EINF 4002  000074  **P$GETF 4002 000077
**P$STOP  4002  000131
```

Subroutines beginning with P$ usually are in the PL1G library, so the user restarts the load and includes that library:

```
$ INIT
$ LO MYPROG
$ LO MYSUBROUTINE
$ LI PL1GLB
$ LI
LOAD COMPLETE
$ QUIT
OK,
```

THE OLDER LOADING PROCEDURE

This section is useful only to those who must load binary files whose names do not end with .BIN, such as **B_filename** or **filename**. To load these files, an extra step is necessary to name a runfile. The extra step can be avoided by renaming the binary file with the command CNAME:

```
CNAME B_MYPROG MYPROG.BIN
```

and then using the newer procedure described above.

If the older loading procedure is still desired, use the following steps. Note that this sequence starts by entering the SEG level of processing, so that the first prompt displayed is the pound sign (#). Then the command VLOAD puts the user on the VLOAD subprocessor level, where the prompt is the dollar sign ($).

1. Enter SEG with no options.
2. Enter VLOAD and the runfilename. It is recommended that this name end with .SEG.
3. Follow Steps 3 through 5 described under **DEFAULT LOADING AND EXECUTION** at the start of this chapter.
4. For subsequent executions of the runfile, if the runfilename has the format **filename.SEG**, enter only **SEG filename**. For any other format, enter the entire filename.

As an example, the COBOL source program DISBURSE has been compiled to create the binary file B_DISBURSE. It requires the sort library VSRTLI, and calls another program whose binary filename is B_CALLED.

```
OK, SEG
[SEG rev x.x]
# VL DISBURSE.SEG
$ LO B_DISBURSE
$ LO B_CALLED
$ LI VCOBLB
$ LI VSRTLI
$ LI
LOAD COMPLETE
$ EXEC
OK,
```

For subsequent executions, enter:

```
SEG DISBURSE
```

SEGMENT ALLOCATION IN A DEFAULT LOAD

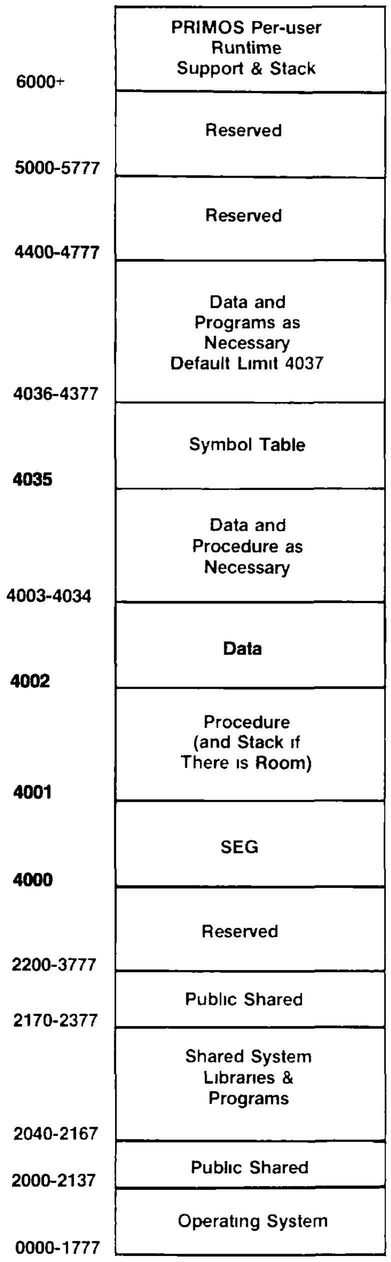
The mechanical steps for a default load are given in the first part of the chapter. The present discussion describes what goes on in memory during a default load.

On a normal or default load, SEG loads programs into virtual segments and resolves most references between modules. SEG itself resides in segment '4000. The first procedure or instruction section is loaded into segment '4001, and any subsequent procedure sections into the next available segments such as '4003, '4004, and so on. (No single procedure section can be larger than one segment. If it is, the programmer must break the program into smaller modules for separate compilation.) The static (initialized) data section is loaded into segment '4002. If it is too large for one segment it is continued in available segments such as '4003, '4004, and so on. Procedure and data are never loaded into the same segment on a default load. Then any necessary libraries are loaded following the same principles.

The dynamic data areas are allocated with a **stack**. The stack is a dynamic work area that is usually assigned as the next free location in the first procedure segment with '6000 (decimal 2048) free 16-bit locations. If no such segment exists, a new segment is assigned with the first location in the stack set to 4; locations 0 through 3 are used by the hardware. The user may change the location of the stack and may change its size. The stack is only allocated in procedure segments. It may use more than '6000 locations.

The name **stack** is also used for the concatenation of all separate stacks for all modules in the program. A **stack frame** is allocated for each module. The frame is described in Appendix G. Stacks can be traced back with Prime's Source Level Debugger (DBG) or the VPSD debugger. The load map shows the size needed for each data stack, but not its location, as location is determined at run time.

After all loading, the user's virtual memory looks something like Figure 2-1.


PRIMOS Per-user Runtime Support & Stack
6000+
Reserved
5000-5777
Reserved
4400-4777
Data and Programs as Necessary Default Limit 4037
4036-4377
Symbol Table
4035
Data and Procedure as Necessary
4003-4034
Data
4002
Procedure (and Stack if There is Room)
4001
SEG
4000
Reserved
2200-3777
Public Shared
2170-2377
Shared System Libraries & Programs
2040-2167
Public Shared
2000-2137
Operating System
0000-1777

Default Allocation of Segments at Rev. 19.2
Figure 2-1

The command LI causes SEG to load three **system libraries**: IFTNLB, PFTNLB, and SPLLIB.

Meanwhile SEG is building a **symbol table**. It contains the names of all modules, all direct entry points, and all data names identified as external references. This table is placed in segment '4035. During the load, SEG is also attempting to resolve all external references. This means that any data names or procedure calls that the compiler could not find within the same module are now sought in the symbol table. Thus, if a PL1G program calls the subroutine TNOUA or the subroutine P$AINT in the PL1G library, the compiler has recognized only that these were external calls, but now SEG can put the address of the external subroutine into the call. When all names in the symbol table have been found (except for direct entry points), SEG displays the message LOAD COMPLETE.

ADVANTAGES OF THE DEFAULT LOAD

Chapter 4 presents some ways in which loads can be changed to save space and run time. After reading that chapter, you may wonder why the default load should be used at all. There are several reasons. A runfile created by a default load is much easier to debug than most of those discussed in Chapter 4. Prime's Source Level Debugger (DBG) can only handle files created with the segment allocation described in this chapter. Shared segments in particular can only be examined with VPSD. If you want to do more sophisticated loads, first use the default load until your program seems to run without bugs, then reload to produce an optimized runfile.

The default load has several other advantages. It always finds enough room, as long as your procedure section is no larger than one segment. In the default load, SEG uses some techniques to optimize size and placement of the runfile, and may do it better than the programmer working on a special load. On a default load, SEG can also detect stack overflow, while it may not be able to do so in some special loads.

3
SEG LOAD MAPS

This chapter first describes the sections of a load map generated by SEG's MAP command. A **load map** is a listing of selected memory addresses. It shows such information as what calls are unresolved and where space can be saved. During a complicated loading session, the map should be monitored constantly. For example, the MAP 1 option can be used to check the allocated segments and the addresses used within the segment. A MAP 3 is recommended if the message LOAD COMPLETE is not returned after the command LIBRARY is entered.

The second part of the chapter presents the different options of the MAP command.

The last part of the chapter gives examples of uses of load maps to answer the following questions:

- What did I forget to load?
- Where did my program die?
- Why are there no more available segments?
- Why did the stack overflow?
- How can I save space?

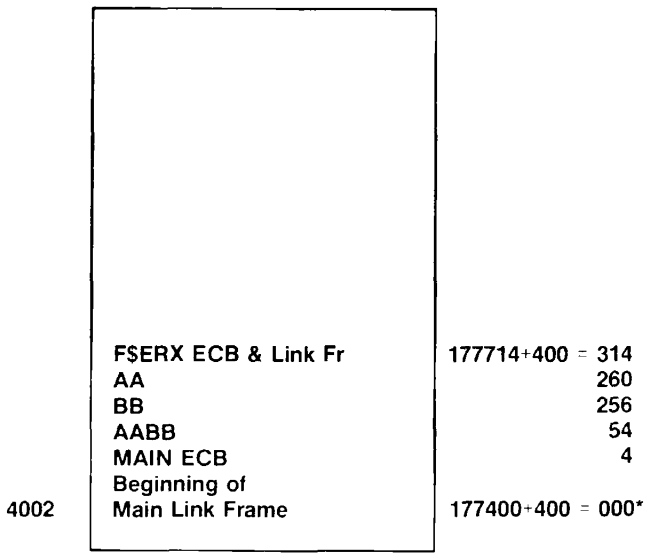
Some of these examples assume familiarity with information in Chapters 4 through 7.

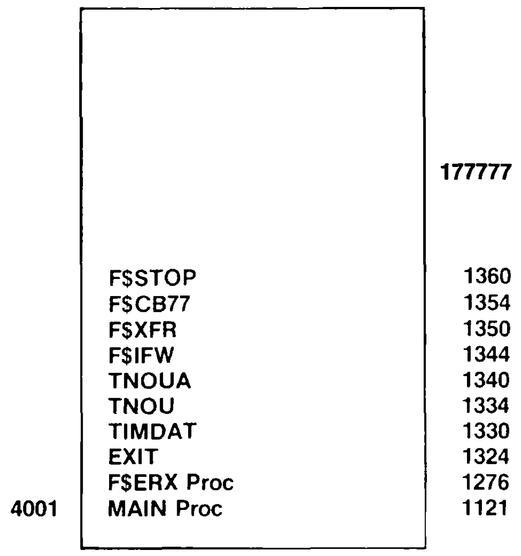
READING THE LOAD MAP

Figure 3-1 is an example of a normal map. It is used in most of the discussion in this first section. It is a FORTRAN 77 program, listed as Program 1 in Appendix D. Figure 3-2 shows the memory segments listed in the map. Each numbered section is discussed in the pages that follow.

```
     OK, SEG -LOAD
     [SEG rev x.x]
     $ LO TMDT
     $ LI
     LOAD COMPLETE
     $ MAP

1.   *START  4002  000004  *STACK  7777  000000  *SYM    000032

2.   SEG. #    TYPE        LOW      HIGH      TOP
       4001    PROC      001000    001363    001363
       4002    DATA      000000    000341    000341

4.   ROUTINE       ECB         PROCEDURE     ST. SIZE  LINK FR.
       MAIN    4002  000004  4001  001121    000056  000054  4002 177400
       F$ERX   4002  000314  4001  001276    000020  000026  4002 177714

5.   DIRECT ENTRY LINKS
       EXIT   4001 001324   TIMDAT 4001 001330   TNOU    4001 001334
       TNOUA  4001 001340   F$IFW  4001 001344   F$XFR   4001 001350
       F$CB77 4001 001354   F$STOP 4001 001360

6.   COMMON BLOCKS
       AABB      4002  000054  000201    BB        4002  000256  000001
       AA        4002  000260  000034

7.   OTHER SYMBOLS
       F184DONE  4001  001276

     $ Q
     OK,
```

A Map Obtained from a Successful Default Load
(Source is Program 1 in Appendix D.)
Figure 3-1





Segments Shown on the Map in Figure 3-1
*(The '400 offset is explained on page 3-7.)
Figure 3-2

Note

All values in load maps are given in octal notation. All addresses are 16-bit addresses.

Section 1 – Program Start

The program start area shows where the program has been loaded, the stack start location, and the current number of symbols in the symbol table.

*START	The segment number and location for the start of execution. At the beginning of a load, the start address is initialized to '7777 000000. SEG fills in *START with the address of the ECB for the first segmented procedure encountered (usually the main program). In Figure 3-1, SEG placed the ECB of the procedure at address 4 of segment '4002. The only time that multiple occurrences of a segment number are valid is for split segments, which are discussed in Chapter 4. Any other multiple occurrences indicate incorrect mixing of default and user-specified load addresses or incorrect placement of common blocks. **Note** If RUNIT has been loaded with the SPLIT command as shown in Chapter 4, the address of RUNIT is used as the start address, rather than the address for START on the load map.
*STACK	Segment number and location of the start of the stack. It is initialized to '7777 000000 at the start of a load. This value is not changed until the file is saved by a QUIT, RETURN, SAVE, or EXECUTE command within VLOAD. The default stack is in the first procedure segment above '4000 with 2048 free locations at the top of memory. In Figure 3-1, the file has not been saved, so the address shown is the one initialized by SEG. **Caution** After the file has been saved, the stack values will be meaningless if RUNIT has been loaded with the SPLIT command. SEG does not keep track of such split segments and may assign the stack to the top of the procedure portion of a split segment. This causes problems if there is not enough space between the end of the procedure portion and the start of the data portion.
*SYM	The number of symbols in the symbol table. This number includes segment numbers and ECB's, and may include empty symbols that do not appear on the load map. It allows the user to check whether the maximum number of symbols has been or is about to be exceeded. In Rev. 19.1, the maximum number of symbols is '16161. In Figure 3-1, the number of symbols is '32, or decimal 26.

Section 2 – Segment Assignments

Each segment is labeled as procedure (PROC) or data (DATA). The list is sorted in order of segment number. Figure 3-1 shows a simple load requiring only two segments, '4001 and '4002.

LOW	Lowest loaded location in the segment. (Not necessarily the lowest assigned location.) Initialized to '177777 (-1) at segment creation; if the segment is used only for uninitialized common areas, LOW is not changed.
HIGH	Highest loaded location in the segment. (Not necessarily the highest assigned location.) Initialized to '000000 at segment creation; if the segment is used only for uninitialized common areas, HIGH is not changed.
TOP	Highest assigned location in the segment. TOP should not be lower than HIGH. If it is, the user may have specified incorrect load addresses on a special load. TOP is initialized to '177777 (-1) at segment creation. When space is reserved for large common blocks, the loader will only set TOP to a maximum of '177776 even though the entire segment to '177777 is reserved. The reason for this is that a LOW, HIGH, and TOP of '177777, '000000, and '177777 indicate an empty segment. Even in the case of a split load, TOP should not be higher than the location of the split.

In Figure 3-1, the procedure part of the program covers '364 16-bit locations, and the data part requires '342 16-bit locations.

Section 3 – Base Areas

This section is not shown in Figure 3-1. See Figure 3-3 below. A **base area** is an area within a procedure segment that is used as a reference for indirect addressing. It may be established directly within PMA by the SETB pseudo-op. Programmers in high-level languages need not be concerned with base areas unless SEG returns the error message SECTOR 0 BASE AREA FULL. In that case, more base areas may be created with the SETBASE or AUTOMATIC subcommands of LOAD. Base areas are shown on a load map in the following form.

```
*BASE     AAAAAA    BBBBBB    CCCCCC    DDDDDD    EEEEEE
```

AAAAAA	Segment number.
BBBBBB	Lowest location for base area.
CCCCCC	Next available location starting up from lowest location.
DDDDDD	Next available location starting down from highest.
EEEEEE	Highest location for base area.

The lowest default location for the sector zero base area is '100. If CCCCCC is greater than DDDDDD, the base area is full.

There may be a sector zero base area in each procedure segment; there must be none in data segments.

Figure 3-3 uses a load map in which base areas are shown. This is a map of a COBOL program. Since COBOL uses many 16-bit indirect addresses, the original sector-zero base area has been allocated starting at address '100 in segment '4001. The lower '14 locations have been used, but no space has been allocated at the top of the base area (address '777 in segment '4001).

```
     OK, SEG
     [SEG rev x.x]
     # RESTORE OLDCASH
     # MAP

1.   *START  4002  000041  *STACK  4001  006654  *SYM    000034

2.   SEG. #     TYPE        LOW       HIGH      TOP
       4001     PROC##     000100    006654    006653
       4002     DATA       000000    001277    001277

3.   *BASE   004001  000100  000114  000777  000777

4.   ROUTINE       ECB        PROCEDURE     ST. SIZE  LINK FR.
       DISBUR  4002  000041  4001  001000     000070  001251  4002 177422
       F$ERX   4002  001252  4001  006606     000020  000026  4002 000652

5.   DIRECT ENTRY LINKS
       C$OS    4001  006556    C$CS    4001  006562    C$RS    4001  006566
       C$WS    4001  006572    C$IN    4001  006576    C$ADAT 4001  006602
       EXIT    4001  006634    TNOU    4001  006640    TNOUA   4001  006644
       I$AA12 4001  006650

6.   COMMON BLOCKS

7.   OTHER SYMBOLS
       CR18_4     4001  006554    F184DONE  4001  006606
```

A Map Showing Base Area Allocation
Figure 3-3

There may be a sector zero base area in each procedure segment. There must be none in data segments (except split segments). Base areas other than those in sector zero are generated by PMA modules. The appearance of such base areas in a default load is probably caused by the inclusion of an improperly coded PMA routine.

Section 4 – Symbols

A main program or subroutine compiled in 64V mode is called a procedure. It is composed of:

- ECB — entry control block.
- Procedure frame — the executable code.
- Stack frame — dynamically allocated local storage that is assigned when the routine is called, and released upon return from the routine.
- Link frame — static data, constants, and transfer vectors.

The ECB is usually part of the link frame. The procedure frame is located in a segment reserved for procedure frames and stack frames. Link frames and common blocks will normally be located in segments reserved for data. Each is displayed in the map as follows:

ROUTINE	This section of the map describes all the external procedures in the runfile. Often it will include subroutines that the user was not aware of calling, but that are called implicitly by the program.
ECB	The first pair of numbers in this section of the map give the segment and address for the ECB. (If named, the ECB has the name assigned within the program, not the PRIMOS filename. A PRIMOS file may contain several ECBs.)
PROCEDURE	The segment and address for the procedure.
ST. SIZE	The size of the stack frame (working area) created whenever the routine is called. Its segment and location therein are assigned at execution time.
LINK FR.	The first column is the size of the link frame in 16-bit locations. The last two columns are the link frame segment and offset. Note that the offset is '400 locations lower than the actual position for compatibility with the information printed by the PRIMOS command PM. This is because, at least with FTN, F77, and PL1G, the first 16 bits in the link frame that can be referenced efficiently are at offset '400 above the address in the link base register (LB% + '400). The segment number is usually that for the ECB.

Procedures with no external names (such as internal Pascal procedures) are identified by #### in the name field.

The link frame address is useful in tracing abnormal program halts. After such a halt, the PRIMOS command PM will usually be able to output a link base register value. This value shows which routine was active at the time of failure. Some conditions such as stack overflow, however, can render such information useless.

In Figure 3-1 above, the symbol section of the load map shows two routines, the FORTRAN program MAIN, and a subroutine called by the F77 compiler, identified by the initial characters F$. (This one is F$ERX, which sets the forced loading of library files.) Both routines have their link frames in segment '4002, which is always the segment used first for data and linkage on a default load. Both have their procedure frames in segment '4001. MAIN is loaded at address '1121 of segment '4001, and F$ERX is loaded right after it at address '1276. Since there is no recursion, the size of the stack required will be at most only '56 plus '20, or decimal 62. The link frame for MAIN starts at address 0 of segment '4002, and it has '54 16-bit locations allocated for it. The ECB for MAIN is in this link frame, and it starts at an offset of 4 in the link frame. Because the link frame starts at offset 0, the ECB starts at location 4 as shown in the map.

The link frame for F$ERX starts at address '314, and it has '26 locations allocated. The ECB for F$ERX also starts at address '314, so it resides inside the link frame at an offset of 0. The link frames have been allocated starting from the high addresses of segment '4002, and there is plenty of space left between the highest address of the ECBs and the lowest address of the link frames.

Section 5 — Direct Entry Links

As explained in Chapter 1, PRIMOS supports direct entry calls to the operating system for certain routines. These are created as faulted pointers (unsnapped links) in the SEG runfile. Where references are satisfied by these faulted pointers, they will appear in the DIRECT ENTRY LINKS section of the map.

In Figure 3-1, eight direct entry links are listed. Two of them, EXIT and TIMDAT, were called by the program TMDT.F77. The others were called implicitly by routines within MAIN. They have addresses in segment '4001, the procedure segment. These addresses contain only enough information to tell the operating system which subroutine is being referenced. The subroutine itself resides in the operating system.

Section 6 — Common Blocks

This section lists each common block, its segment number, starting address in the segment, and size, when known. The program mapped in Figure 3-1 shows three common blocks, named in the FORTRAN program as AA, BB, and AABB. All are in the first data segment, '4002, just after the link frame for program MAIN.

Section 7 — Unsatisfied References and Other Symbols

For each symbol, this section lists the symbol and its address (segment and offset number). Unsatisfied references are preceded by two asterisks (**). The numbers for unsatisfied references (segment and offset address) locate the last request for the routine that was processed by the loader.

In Figure 3-1, this section shows no unsatisfied references. The symbol F184DONE is the Rev. 18 stamp for the F77 compiler.

THE MAP OPTIONS

The MAP command of SEG and the MAP subcommand of LOAD print a load map of a runfile, either at the terminal or to a file. For details of syntax, see the discussions of these two commands in Chapters 5 and 6. Both allow the following options.

Map Options	Load Map Information
0	Full map (default).
1	Segment use map only (map sections 1 and 2).
2	Segment use map and base areas (map sections 1, 2 and 3).
3	Undefined symbols, if any, sorted by ascending address (map section 7).
4	Full map (identical to 0).
5	Reserved.
6	Undefined symbols, if any, sorted alphabetically (map section 7).
7	Full map, sorted alphabetically.
10	Symbols sorted by ascending address.
11	Symbols sorted alphabetically.

USING LOAD MAPS

The following subsections illustrate some useful readings of load maps. The first two examples illustrate default loads. The others require some familiarity with the concepts in Chapter 4.

Checking for Unsatisfied References (Load Not Complete)

In this load, the user proceeds until the command **LI** fails to evoke the response LOAD COMPLETE. Then a simple MAP 3 reveals the problem.

```
OK, SEG -LOAD
[SEG rev x.x]
$ LO CALLER
$ LI CBLLIB
$ LI
$ MAP 3


**CALLED    4002  000030
```

The missing routine is the user-written subprogram CALLED. Now the user clears memory with INITIALIZE and starts over, this time including that routine.

```
    $ INIT
    $ LO CALLER
    $ LO CALLED
    $ LI CBLLIB
    $ LI
    LOAD COMPLETE
    $ Q
    OK,
```

Locating Runtime Errors

Sometimes a runtime PRIMOS error message may be understood by looking at the load map. The following is an example. It uses Programs 4 and 5 in Appendix D. The program CALLER.F77 calls TMDT.WRONG, which calls the PRIMOS subroutine TIMDAT.

```
OK, SEG -LOAD
[SEG rev x.x]
$ LO CALLER
$ LO B_TMDT.WRONG
$ LI
LOAD COMPLETE
$ MAP
*START  4002  000004  *STACK  7777  000000  *SYM    000030

SEG. #    TYPE        LOW       HIGH      TOP
  4001    PROC       001000    001475    001475
  4002    DATA       000000    000141    000141

ROUTINE         ECB           PROCEDURE     ST. SIZE  LINK FR.
  ####     4002  000004     4001  001041     000102  000044  4002   177400
  TMDTWR   4002  000050     4001  001255     000112  000050  4002   177444
  F$ERX    4002  000114     4001  001410     000020  000026  4002   177514

DIRECT ENTRY LINKS
  EXIT     4001  001436     TIMDAT  4001  001442     TNOU    4001  001446
  TNOUA    4001  001452     F$IFW   4001  001456     F$XFR   4001  001462
  F$CB77   4001  001466     F$STOP  4001  001472

COMMON BLOCKS

OTHER SYMBOLS
  F184DONE  4001  001407

$ EXEC
FI

Error: condition "POINTER_FAULT$" raised at 6(0)/62650.
Entry to inner ring was from call at 4001(3)/1257.
ER!
```

The PRIMOS error message indicates a problem at address '1257 in segment '4001. Examination of the map shows that this address is the third 16-bit address of TMDTWR. (TMDTWR is the ECB name for the program stored in the PRIMOS file TMDT.WRONG.) If TMDT.WRONG is then compiled with the -EXPL option, the user can look at the second and third locations shown on the expanded listing. Remember that addresses in the expanded listing are relative addresses assigned before the program is given an address in the runfile. In the expanded listing below, the second and third addresses are 136 and 137.

```
Source File: <T2>MINE>DOC3524>SAMPLE.PGMS>TMDT.WRONG
Compiled on: 830105  at: 11:27  by: FORTRAN-77 Rev x.x
Options: OPTIMIZE-2 XREF EXPLIST OFFSET NOBIG INTL LOGL DYNM 64V UPCASE
ERRTTY

    1         PROGRAM TMDTWR
    2         INTEGER*2 STRING(28)
    3         INTEGER*2 NUM, DATE(3)
    4         INTEGER*2 TIME, TIME1, TIME2, NAME(16)
    5         EQUIVALENCE (STRING(1), DATE)
    6         EQUIVALENCE (STRING(4), TIME)
    7         EQUIVALENCE (STRING(5), TIME1)
    8         EQUIVALENCE (STRING(6), TIME2)
    9         EQUIVALENCE (STRING(13), NAME)
   10         NUM = 28

000135:     02.000434L    LDA    LB%+434



000141: 000700.000054S    AP     SB%+54,SL

   12         WRITE (1, 100)

000143: 061432.000440L    PCL    LB%+440,*
000145: 000100.000023     AP     PB%+23,S
000147: 001100.000436L    AP     LB%+436,S
000151: 000300.000024     AP     PB%+24,SL

   13         WRITE (1,300) DATE

000153: 061432.000440L    PCL    LB%+440,*
   .
   .
   .
```

The load map shows a problem at the third location in TMDTWR. In the expanded listing, the second and locations are 136 and 137 and surround the PMA code for the subroutine call to TIMDAT (called a PCL or procedure call). The description of TIMDAT is in the **Subroutines**

Reference Guide, and shows that TIMDAT expects two arguments, not one. If you change the call to:

```
CALL TIMDAT (STRING, 28)
```

and recompile and reload, the program will execute correctly.

Finding Why There are No More Available Segments

This load uses a Pascal program, TWO_CUBE.PASCAL, with a large external array named MIN (3674160 characters). The user does a default load and then tries to run the program with Prime's Source Level Debugger (DBG).

```
OK, PASCAL TWO_CUBE -DEBUG
0000 ERRORS [PASCAL Rev. x.x]
OK, SEG -LOAD
[SEG rev x.x]
$ LO TWO_CUBE
$ LI PASLIB
$ LI
LOAD COMPLETE
$ QUIT
OK, DBG TWO_CUBE
Fatal error:  No more segments available for permanent storage.
              (alloc_ps)
OK,
```

The user does a MAP 1 to study segment allocation:

```
OK, SEG
[SEG rev x.x]
# RESTORE TWO_CUBE
# MAP 1
*START  4002  000004  *STACK  7777  000000  *SYM    000146

SEG. #    TYPE      LOW      HIGH      TOP
  4001    PROC     001000    014475    014474
  4002    DATA     000000    150633    150633
  4003    DATA     177777    000000    177777
  4004    DATA     177777    000000    177777
  4005    DATA     177777    000000    177777
  4006    DATA     177777    000000    177777
  4007    DATA     177777    000000    177777
  4010    DATA     177777    000000    177777
  4011    DATA     177777    000000    177777
  4012    DATA     177777    000000    177777
  4013    DATA     177777    000000    177777
  4014    DATA     177777    000000    177777
  4015    DATA     177777    000000    177777
  4016    DATA     177777    000000    177777
```

```
    4017     DATA       177777     000000     177777
    4020     DATA       177777     000000     177777
    4021     DATA       177777     000000     177777
    4022     DATA       177777     000000     177777
    4023     DATA       177777     000000     177777
    4024     DATA       177777     000000     177777
    4025     DATA       177777     000000     177777
    4026     DATA       177777     000000     177777
    4027     DATA       177777     000000     177777
    4030     DATA       177777     000000     177777
    4031     DATA       177777     000000     177777
    4032     DATA       177777     000000     177777
    4033     DATA       177777     000000     177777
    4034     DATA       177777     000000     177777
    4035     DATA       177777     000000     177777
    4036     DATA       177777     000000     177777
    4037     DATA       177777     000000     004027
```

There are two reasons for not using this default load. The Pascal runtime system uses standard heap storage starting at segment '4027 and going down to '4010. In addition, Prime's Source Level Debugger needs some segments between '4001 and '4037 in order to operate properly. Since a default load occupies all of these segments, DBG gives an error message upon initialization.

Now the user repeats the load, assigning MIN to '4030.

```
 SEG -LOAD
[SEG rev x.x]
$ SYMBOL MIN 4030 0      /*SEE CHAPTER 6 FOR SYMBOL
$ LO TWO_CUBE
$ LI PASLIB
$ LI
LOAD COMPLETE
$ MAP 1
*START  4002  000004  *STACK  7777  000000  *SYM    000112

SEG. #    TYPE        LOW       HIGH       TOP
  4001    PROC       001000    014475    014474
  4002    DATA       000000    150633    150633

$ EXEC

END OF RUN
OK,
```

The segments allocated to MIN are not shown, since MIN is not initialized.

Note

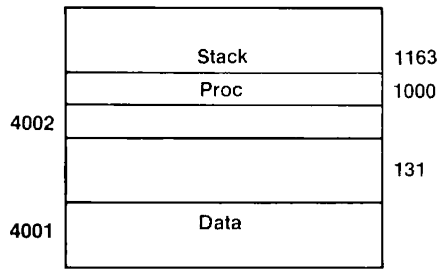
This load will cause the message ILLEGAL_SEGNO$ if your system assigns only the default number of segments ('40 or decimal 32) to each user. Check with your System Administrator.

Looking at the Stack

The stack is allocated at runtime, so a load map is not always enough by itself to explain why the stack has overflowed or how much space is needed for a stack, particularly with multiple or recursive subroutine calls. The following discussion shows what can be determined through the load map. The program is Program 2 (MINDLESS.F77) in Appendix D. Its default space allocation is visualized in Figure 3-4. When run, it gives the error message STACK_OVF$.

```
OK, SEG -LOAD
[SEG rev x.x]
$ LO MINDLESS
$ LI
LOAD COMPLETE
$ MAP 1

*START  4002  000004  *STACK  7777  000000  *SYM    000030

SEG. #    TYPE        LOW       HIGH      TOP
  4001    PROC       001000    001163    001163
  4002    DATA       000000    000131    000131

$ EXEC

VALUE:   1
VALUE:   2
VALUE:   3
.
.
.
VALUE:  30
VALUE:  31
VALUE:  32

Error: condition "STACK_OVF$" raised at 4001(3)/1072.
```

MAP 1 above showed that space allocation looked like this:

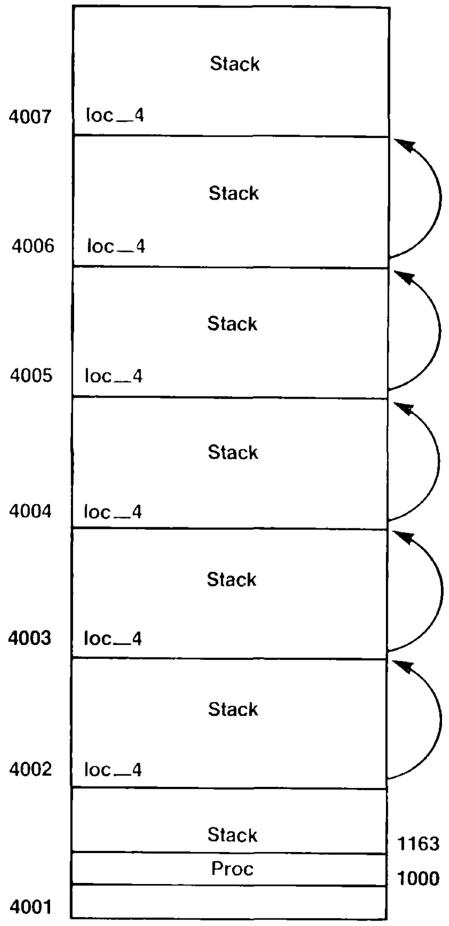


Space Allocation for Program MINDLESS
Figure 3-4

STACK_OVF$ was raised at location '1072 in segment 4001 (ring 3 or user area). On the map, the highest code location was '1163, so now reset the stack to extend farther.

```
OK, SEG
[SEG rev x.x]
# RESTORE MINDLESS
# MODIFY   /*THE FOLLOWING SK COMMAND IS DISCUSSED UNDER MODIFY
#          /*IN CHAPTER 5.  IT STARTS THE STACK AT SEGMENT '4001,
#          /*OFFSET '1163, AND EXTENDS IT INTO SEGMENT '4002.
$ SK 4001 1163 4002
$ RETURN

#                 /*NOW YOU WILL SEE THAT THE STACK IS INITIALIZED,
#   MAP 1         /* BUT NO OTHER NEW INFO
*START  4002  000004  *STACK  4001  001163  *SYM    000030

SEG. #     TYPE       LOW       HIGH      TOP
  4001     PROC##     001000    001163    001163
  4002     DATA       000000    000131    000131
# Q
OK, SEG MINDLESS

VALUE:   1
VALUE:   2
VALUE:   3
.
.
.
VALUE: 195
VALUE: 196
VALUE: 197
VALUE: 198
VALUE: 199

OK,
```

The use of VPSD together with the second load map will show that space allocation now looks like Figure 3-5.

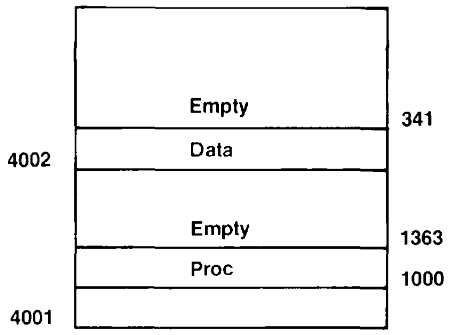


Space Allocation for Program MINDLESS after Stack Extension
Figure 3-5

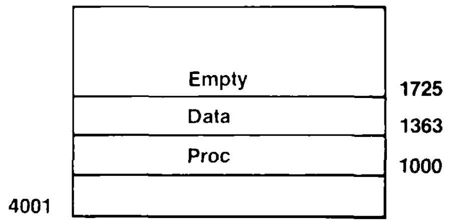
Looking for Wasted Space

This example uses Program 1 in Appendix D. The user first does a default load, then looks at the map, and sees that two segments are used for a program whose entire data plus procedure size is '724 16-bit locations. This space allocation is represented in Figure 3-6a. The user then redoes the load using the VLOAD subcommand MIX to do the mixed load described in Chapter 4. The new map shows that only one segment is used for the entire program. The new space allocation is shown in Figure 3-6b.

```
OK, SEG -LOAD
[SEG rev x.x]
$ LO TMDT
$ LI
LOAD COMPLETE
$ MAP 1

*START  4002  000004  *STACK  7777  000000  *SYM    000032

SEG. #    TYPE        LOW       HIGH       TOP
  4001    PROC       001000               001363
  4002    DATA       000000    000341     000341

$   /*TOTAL '363 AND '341 = '724 WORDS NEEDED, BUT TWO SEGMENTS
$   /*ALLOCATED -- SEE FIGURE 3-6a
$ INITIALIZE         /*REINITIALIZE SEG
$ MIX                /*COMPRESS EVERYTHING INTO ONE SEGMENT
$ LO TMDT
$ LI
LOAD COMPLETE
$ MAP 1

*START  4001  001302  *STACK  7777  000000  *SYM    000031

SEG. #    TYPE        LOW       HIGH       TOP
  4001    PROC       001000    001725

$     /*'724 WORDS USED, BUT ALL IN ONE SEGMENT NOW -- FIGURE 3-6b
```



Allocation of a Small Program on the Default Load
Figure 3-6a



Allocation of a Small Program with MIX (Compression)
Figure 3-6b

Refining Storage Allocation

Usually it is necessary to try a load more than once to get the correct, or the best, storage allocation. In the following example, a COBOL 74 program is loaded as a RUNIT file with all procedure and data in segment '4000, as explained in Chapter 4. The program is Program 3 in Appendix D. When the new file, LADD4000, is run the first time, it aborts with the error message POINTER_FAULT$.

```
OK, SEG -LOAD
[SEG rev x.x]
$ SPLIT 3000 4000 150000 4001
$ MIX
$ S/LO LADD 0 4000 4000
$ D/LI  CBLLIB
$ D/LI
LOAD COMPLETE
$ MAP
*START  0266  000000  *STACK  4000  150000  *SYM    000164

SEG. #    TYPE       LOW      HIGH      TOP
  4000    PROC##     000100   002775    002775
  4000    DATA##     177777   000000    002777
  4001    PROC       000100   047071    047070
```

```
*BASE    004000   000100   000126   000777   000777
*BASE    004001   000100   000315   000777   000777

ROUTINE          ECB            PROCEDURE       ST. SIZE   LINK FR.
  STACK_OV 4000   001212   4000   001152   000030   000024   4000   000612
  MAIN     4001   001004   4000   001241   000054   003721   4001   000400
  C$OS     4000   002544   4000   001264   000256   000064   4000   002144
  C$ER     4000   002632   4001   006334   000232   000034   4000   002232
  .
  .
  .
  P$ACKVB  4001   046620   4001   046506   000036   000032   4001   046220
  F$ERX    4001   046700   4001   046652   000020   000026   4001   046300

DIRECT ENTRY LINKS
  P$ASB  4001   046154   P$ALR1  4001   046160    P$LLR1  4001   046164
  P$LLR3 4001   046170     P$LRR1 4001   046174     P$ARL2 4001   046200
  P$LRL4 4001   046204     P$ARR1 4001   046210     ATCH$$ 4001   046726
  .
  .
  .
  O$AD07 4001   047056     TONL    4001   047062     T1OU    4001   047066

COMMON BLOCKS
  STACK$      4000   001006                FIL$CM     4001   004722   001412
  POOLBG      4001   026656   014000       ISMTMP     4001   042656   000033
  ISMPAR      4001   042712   000011       ISMAR1     4001   042724   000175
  POOLPR      4001   043122   000100       STM        4001   043222   000634
  SL$COM      4001   044056   000005       TR$COM     4001   044064   000001
  ISMARR      4001   044066   000041       CODE       4001   044130   000002
  P$ONCH      4001   046500   000001       P$ONSRC    4001   046502   000003

OTHER SYMBOLS
  RUNIT     4000   001000   RESUME 4000   001053   CR18_9    4000   002630
  F192QFP4  4000   002776   ERRCOM 4001   044132   P$ENTP    4001   046142

$ RE
# SHARE
FILE ID: LADD               /*MULTICHARACTER ID IN REV. 19.2
Creating LADD4000
# Q
OK, R LADD4000

Error: condition "POINTER_FAULT$" raised at 4000(3)/1104.
```

Inspection of the map shows two things wrong. The COBOL 74 library (CBLLIB), like the PL1G library, loads some modules as common blocks. The map shows that most of these are still in segment '4001 and need to be relocated specifically in segment '4000. In addition, the SPLIT command allowed only '3000 locations for the procedure section. (The command SIZE LIB>CBLLIB.BIN reveals that CBLLIB is over '240000 16-bit locations in size.) The following retrial corrects these errors.

```
OK, SEG -LOAD
[SEG rev x.x]
$ COMMON ABS 4000                  /*NOTE WELL FOR COBOL 74
$ SPLIT 67777 4000 150000 4001
$ MIX
$ S/LO LADD 0 4000 4000
$ D/LI CBLLIB
$ D/LI
LOAD COMPLETE
$ MAP
*START  0266  004000  *STACK  4000  150000  *SYM    000162

SEG. #    TYPE        LOW       HIGH      TOP
  4000    PROC##     000100    051067    051066
  4000    DATA##     177777    000000    067776

*BASE   004000  000100  000343  000777  000777

ROUTINE          ECB          PROCEDURE      ST. SIZE  LINK FR.
  STACK_OV 4000   001212 4000   001152  000030  000024   4000   000612
  MAIN     4000   001270 4000   001241  000054  003721   4000   000664
  C$OS     4000   006466 4000   005206  000256  000064   4000   006066
  C$ER     4000   010610 4000   010166  000232  000034   4000   010210
  .
  .
  .
  P$ACKVB  4000 050616  4000   050504  000036  000032   4000   050216
  F$ERX    4000 050676  4000   050650  000020  000026   4000   050276

DIRECT ENTRY LINKS
  P$ASB  4000  050152    P$ALR1 4000   050156   P$LLR1 4000   050162
  P$LLR3 4000  050166    P$LRR1 4000   050172   P$ARL2 4000   050176
  P$LRL4 4000  050202    P$ARR1 4000   050206   ATCH$$ 4000   050724
  O$AD07 4000  051054    TONL   4000   051060   T1OU   4000   051064

COMMON BLOCKS
  STACK$     4000   001006              FIL$CM    4000   006552   001412
  POOLBG     4000   030654   014000     ISMTMP    4000   044654   000033
  ISMPAR     4000   044710   000011     ISMAR1    4000   044722   000175
  POOLPR     4000   045120   000100     STM       4000   045220   000634
  SL$COM     4000   046054   000005     TR$COM    4000   046062   000001
  ISMARR     4000   046064   000041     CODE      4000   046126   000002
  P$ONCH     4000   050476   000001     P$ONSRC   4000   050500   000003

OTHER SYMBOLS
  RUNIT  4000   001000    RESUME 4000   001053  CR18_9 4000  010164
  ERRCOM 4000   046130    P$ENTP 4000   050140  F192QFP4 4000  050212

$ RE
# SHARE
FILE ID: LADD
Creating LADD4000
```

```
# Q
OK, R LADD4000
    SO FAR SO GOOD.
OK,
```

4 ADVANCED SEG TECHNIQUES

When an application demands greater control over the placement of modules, procedures, data areas, and stacks, SEG has groups of commands specifically designed for this purpose. The commands are listed in Chapters 5 through 7. This chapter shows the use of these commands in a series of techniques, and illustrates the concepts by case studies.

The following techniques are presented in this chapter:

- Optimizing runfile size
- Managing common areas
- Preparing procedures for sharing
- Creating external commands
- Creating shared data
- Managing the stack
- Replacing program modules
- Creating and using templates
- Allocating base areas
- Using relative segment numbers

To follow the explanations of loads in this chapter, you should be familiar with maps produced by SEG. How to read these maps is explained in Chapter 3.

OPTIMIZING RUNFILE SIZE

A runfile can be made smaller with the following techniques.

- Instead of using both segments '4001 and '4002 for procedure and data, place both procedure and data in one segment with the MIX command of VLOAD.
- Instead of using segment '4000 for SEG and one or more other segments for the program, use RUNIT, SEG's small execution module, and fit all of a small program into the same segment with it.
- If any symbols or common blocks would be loaded into segment '4001 or higher by default, use the SYMBOL or COMMON commands of VLOAD to put them into '4000 or another segment that you choose.

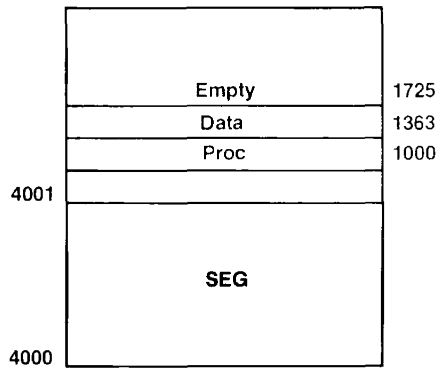
The following three sections illustrate these techniques.

PERFORMING A MIXED LOAD

The MIX command of VLOAD allows you to put procedure and data in the same segment, thus using one segment instead of two for a small program (less than 128K bytes). Once MIX is used, the following commands in the load must use specific or absolute segment numbers, so the commands are prefixed by **S/** (specify segment) or **D/** (duplicate the previous command's arguments).

The following example is the one used in the section **Looking for Wasted Space** in Chapter 3. The program is Example 1 in Appendix D. After verifying that the program is small enough to fit into one segment, the user does the following load. The map shows segment allocation as represented in Figure 4-1.

```
SEG -LOAD
[SEG rev x.x]
$ MIX               /*COMPRESS EVERYTHING INTO ONE SEGMENT
$ LO TMDT
$ LI
LOAD COMPLETE
$ MAP 1

*START  4001  001302  *STACK  7777  000000  *SYM    000031

SEG. #    TYPE         LOW       HIGH       TOP
                                          001725

$     /*'724 WORDS USED, BUT ALL IN ONE SEGMENT
```



Allocation of a Small Program with MIX (Compression)
Figure 4-1

The map shows that all of the program is in segment '4001, which is labeled as a procedure segment.

MAKING A RUNIT FILE

The runfile produced in the previous example still uses two segments to run a tiny program, because segment '4000 is required by SEG. The next example uses not only MIX but also SPLIT, RETURN, and SHARE. SPLIT defines where the program stack will be placed, with procedure loaded below that address and data loaded above that address. The example uses SPLIT to place everything in segment '4000. SPLIT can also load a small execution program, RUNIT, which replaces the large SEG utility. RETURN takes the user back to SEG-level commands. SHARE converts the segmented runfile into a single-segment or SAM (sequential access method) file. The SAM file may be executed with the PRIMOS command RESUME, which executes more quickly than SEG. In addition, the prefixes S/ and D/ are used to specify which segments are to be used for procedure and data. These prefixes are listed with commands in Chapter 6.

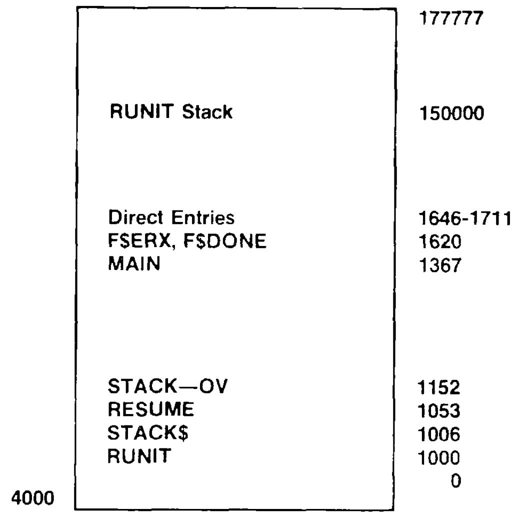
The purpose behind this series of steps is to make a single-segment or RUNIT file, similar to an R-mode file but using V-mode instructions, and located in segment '4000. A small program can thus be compressed from three segments (procedure, data, and SEG) to one. Further, it can be run with RESUME, the PRIMOS command that expects to find an executable file and load it into segment '4000.

Before a detailed discussion about these steps, here is an example of a simple standard load that reduces the size of a small runfile from three segments to one. The program is Program 6 in Appendix D. Its ECB is defined as MAIN with the statement PROGRAM MAIN.

```
OK, SEG -LOAD
[SEG rev x.x]
$ SPLIT 167777 4000 150000/*SPLIT SEGMENT '4000,
                          /*PROCEDURE BELOW '167777, STACK STARTS
                          /*AT '150000, AND LOAD RUNIT IN '4000 AT '1000
$ MIX                     /*ALLOW BOTH PROCEDURE AND DATA IN SAME SEGMENT
$ S/LO TMDT 0 4000 4000   /*LOAD PROCEDURE IN '4000, DATA IN '4000
$ D/LI                    /*SAME FOR SYSTEM LIBRARIES
LOAD COMPLETE
$ MAP TMDT.MAP            /*SAVE A MAP
$ RETURN                  /*BACK TO SEG LEVEL
# SHARE                   /*CREATE A SAM (UNSEGMENTED) FILE
TWO CHARACTER FILE ID: AL /*TWO CHARACTERS ONLY BEFORE REV. 19.2
CREATING AL4000
# DELETE                  /*SEGMENTED FILE IS NO LONGER NEEDED
# Q
OK, R AL4000              /*SAM FILES MUST BE EXECUTED WITH RESUME

DATE IS 012282
TIME SINCE MIDNIGHT IN MINUTES+SECONDS+TICKS:
   825    41    55
USER IS ANNE
OK,
```

The steps in this load are discussed below. Here is the map that was saved in TMDT.MAP. You can see that only one segment, '4000, is used. It is listed twice in section 2 of the map because SPLIT has established it as both a procedure and a data segment. RUNIT is listed in OTHER SYMBOLS, so a separate segment is not required for SEG to reside in. The diagram of the new, optimized runfile is Figure 4-2.

```
*START  0266  001206  *STACK  7777  000000  *SYM    000035

SEG. #    TYPE        LOW      HIGH      TOP
  4000    PROC       001000   [illegible]  [illegible]
  4000    DATA       177777   000006    003777

ROUTINE        ECB          PROCEDURE     ST. SIZE  LINK FR.
  STACK_OV  4000  001212    4000  001152    000030  000024  4000  000612
  MAIN      4000  001530    4000  001367    000112  000046  4000  001124
  F$ERX     4000  001620    4000  001572    000020  000026  4000  001220

DIRECT ENTRY LINKS
  EXIT      4000  001646    MKONU$    4000  001652    TIMDAT    4000  001656
  TNOU      4000  001662    TNOUA     4000  001666    F$IFW     4000  001672
  F$XFR     4000  001676    F$CB77    4000  001702    F$STOP    4000  001706


COMMON BLOCKS
  STACK$    4000  001006

OTHER SYMBOLS
  [illegible]  [illegible]  001000    RESUME    4000  001053    F184DONE  4000  001572
```



Single-segment or RUNIT Load
Figure 4-2

More About RUNIT

RUNIT is the part of SEG that executes programs. It is small enough that it can usually occupy the same segment with a small user program. Thus it is a useful substitute for the entire code of SEG. SEG can be used to invoke a runfile only if segment '4000 contains nothing below a certain address ('150000 in Rev. 19.2). It is possible to load small programs into the same segment with SEG, but in a later software revision, SEG may be larger, and your program will overwrite it. Therefore, if you want to put a program in this segment, it is better to load RUNIT and plan on executing the program with RESUME. (This manual gives no examples of programs loaded into segment '4000 with the entire SEG program, except when NEW will be required for making templates.)

Be careful of the following points when using RUNIT.

- RUNIT expects the user program to be named MAIN. This means that the entry control block or ECB set up by the compiler or assembler must be so named. See the Glossary for how to name the ECB in each Prime language.

- Use format 2 or 3 of SPLIT (Chapter 6), which loads RUNIT, only if all loaded code and data are in segment '4000 or below (shared segments).

- The program must fit into one segment (128K bytes), except for its common blocks.

The start address of the runfile is now RUNIT, so at execution time RUNIT initializes the stack at its current default and transfers control to MAIN. There is a way, however, to keep the stack from being initialized if you want to maintain the current stack value during a program interrupt (such as during debugging). RESUME is an entry point that prevents RUNIT from initializing the stack. To set the start address to RESUME instead of RUNIT, use the PRIMOS commands RESTORE and SAVE with special parameters to change the starting address. (See the **PRIMOS Commands Reference Guide**.)

Usage: Three steps are required to incorporate RUNIT into segment '4000.

1. Assemble or compile the program as a subroutine whose ECB is labeled MAIN (see ECB in the Glossary for details). In our example, the FORTRAN statement PROGRAM MAIN labels the ECB.
2. Use the loader's SPLIT command to declare segment '4000 as a split segment. Use MIX so SEG will know that both procedure and data may go into the same segment.
3. Complete the load using the prefixes S/, D/, and P/ as required. Once a segment has been split, it is addressable only with the S/ or P/ command (or with D/ following an S/ or P/ command), since split loading must use absolute segment numbers. These commands are presented in Chapter 6.
4. Return to SEG via the RETURN subcommand and use SEG's SHARE command to create an unsegmented runfile.

MANAGING COMMON BLOCKS

Large common blocks are generally the easiest targets for saving space. Uninitialized common blocks (those that have not been given a value within a program) can be removed from the runfile entirely, thus decreasing runfile size. Such common blocks can also be placed within segment '4000. Placement within segment '4000 allows the program to use some of the many pages that SEG requires be allocated but that are not normally used. Be careful, however, that they are placed at an address high enough to avoid overwriting SEG or RUNIT's stack in segment '4000.

If a common block has already been created by a compiler, use SYMBOL to place it. If not, allocate room with A/SYMBOL, which specifies the size.

SYMBOL defines uninitialized common blocks without reserving space for them, thus keeping down the runfile size. Such an uninitialized block, if loaded into a segment that has other contents (such as '4001 or '4002) does have space reserved for it. If you place it in a segment of its own, however, it does not actually take up runfile space (and does not appear on the map). It is, nevertheless, used when referenced. An example of this technique is given in Chapter 3 under **Finding Why There Are No More Available Segments**.

A common block, once defined with SEG or in a module that is loaded, cannot be redefined as larger. If it is redefined as smaller, a warning message is displayed at Rev. 19.2. To suppress the warning, use the VLOAD subcommand NSCW. To reactivate the warning, use SCW.

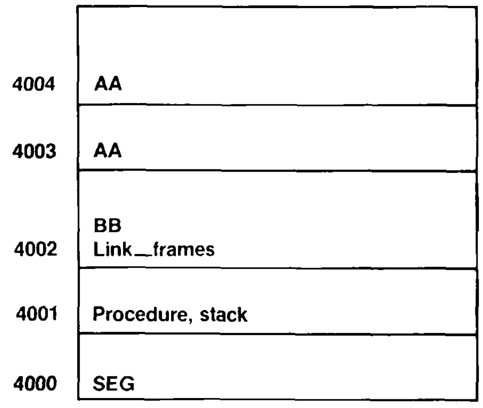
The libraries CBLLIB, PASLIB, and PL1GLB include some common blocks. Therefore, whenever you are loading all of a program into segment '4000, you must manage these common blocks with the command **COMMON ABS 4000** in the load.

An Example Using SYMBOL

This example shows how to build a default runfile that contains large common blocks, then inspect its map for ways to optimize. The program (Example 9 in Appendix D) contains two large uninitialized common areas, AA and BB. The basic default load is:

```
OK, SEG -LOAD
[SEG rev x.x]
$ LO LARGE
$ LI
LOAD COMPLETE
$ MAP
*START  4002  000004  *STACK  7777  000000  *SYM    000032

SEG. #    TYPE        LOW       HIGH       TOP
  4001    PROC      001000    001343    001343
  4002    DATA      000000    040265    040265
  4003    DATA      177777    000000    177777
  4004    DATA      177777    000000    143423


ROUTINE         ECB          PROCEDURE     ST. SIZE  LINK FR.
  MAIN      4002  000004     4001  001061     000062  000054  4002  177400
  F$ERX     4002  040240     4001  001262     000020  000026  4002  037640

DIRECT ENTRY LINKS
  EXIT      4001  001310     TNOU      4001  001314    TNOUA     4001  001320
  F$IFW     4001  001324     F$XFR     4001  001330    F$CB77    4001  001334
  F$STOP    4001  001340

COMMON BLOCKS
  BB        4002  000054  040164    AA        4003  000000

OTHER SYMBOLS
  F184DONE  4001  001261

$ EXEC
START OF LARGE ARRAY
END OF LARGE ARRAY
START OF ARRAY 2
END OF ARRAY 2
OK,
```

An inspection of the map shows that four segments are used for this small program. In addition, the map does not show it, but segment '4000 is required for SEG. Figure 4-3 diagrams segment allocation in this default load.



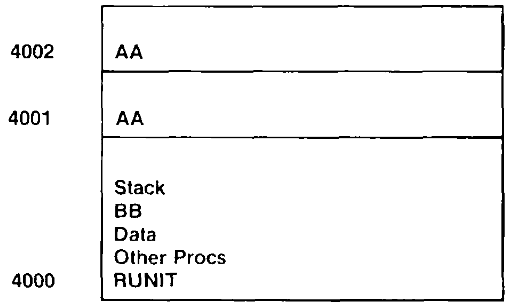
Default Segment Allocation for LARGE.SEG
Figure 4-3

To optimize the runfile, use the commands shown for making RUNIT files in this chapter, plus the SYMBOL command to relocate the common blocks. The block AA is so large that it requires its own segments, but BB can fit into '4000 with everything else.

Now the load map below shows a reduction in size, as diagrammed in Figure 4-4.

```
OK, SEG -LOAD
[SEG rev x.x]
$ SPLIT 4000
$ MIX
$ SY BB 4000 040164
$ S/LO LARGE 0 4000 4000
$ D/LI
LOAD COMPLETE
$ MAP
*START  0266  177425  *STACK  7777  000000  *SYM    000040

SEG. #    TYPE        LOW       HIGH      TOP
  4000    PROC      001000    001703    001703
  4000    DATA      177777    000000    003777
  4001    DATA      177777    000000    177777
  4002    DATA      177777    000000    143423


ROUTINE         ECB          PROCEDURE     ST. SIZE  LINK FR.
  STACK_OV  4000  001212   4000  001152    000030  000024  4000  000612
  MAIN      4000  001520   4000  001313    000062  000054  4000  001114
  F$ERX     4000  001616   4000  001570    000020  000026  4000  001216

DIRECT ENTRY LINKS
  EXIT      4000  001644   MKONU$    4000  001650    TNOU      4000  001654
  TNOUA     4000  001660   F$IFW     4000  001664    F$XFR     4000  001670
  F$CB77    4000  001674   F$STOP    4000  001700
```

```
COMMON BLOCKS
  STACK$     4000  001006             BB         4000  040164
  AA         4001  000000

OTHER SYMBOLS
  RUNIT      4000  001000    RESUME     4000  001053    F184DONE  4000  001570

$ RETURN
# SHARE
TWO CHARACTER FILE ID: LC
CREATING LC4000
# DELETE
# Q
OK, R LC4000
START OF LARGE ARRAY
END OF LARGE ARRAY
START OF ARRAY 2
END OF ARRAY 2
OK,
```



Controlled Segment and Common Allocation for LARGE.SEG
Figure 4-4

Note

This compression of space is not the most efficient strategy in many cases. Often a programmer would prefer to load each common block on a page boundary, if not in a separate segment, for faster access.

PREPARING PROCEDURES FOR SHARING

Shared procedures, as discussed in Chapter 1, use one copy of segments below '4000 for all users. (Chapter 1 also discusses the notion of pure code, which is usually necessary in a shared procedure.) The advantages of sharing are:

- Saving of memory — only one copy of code is necessary for all users.
- Decreased restore time — one copy is restored.
- Reduced paging.

Large procedures that use relatively small amounts of data and that are run by several jobs simultaneously are excellent candidates for shared procedures. Examples include Prime's EDITOR or a user-written order entry system. In general, however, programs that are small or that will normally only be run by one user at a time are not candidates for sharing.

Note

This section assumes that you want to run a program from segment '4000, as well as share it. It is possible also to share a program whose data (impure) area starts in segment '4001 or above. This requires using an interlude program to run the shared program. Usually you would only follow this more complicated method if the linkage section of the program were too large to fit into one segment.

Steps in Creating a Shared Program

The program to be shared must be named MAIN. The procedure and linkage of the program must each occupy no more than one segment.

1. Obtain the shared segment number and address range from the System Administrator. In Rev. 19.2, segments '2140 to '2167 are reserved for Prime-supplied shared subsystems (Editor, FORMS, etc.). Segments '2030 to '2037, '2170 to '2177, and '2300 to '2317 are available as public shared segments. See the Rev. 19.1 Update to the **System Administrator's Guide** for a complete list of shared segments and their current assignments.
2. Use SPLIT to load the impure part of the procedure (data and link frames) into segment '4000 along with RUNIT. At the same time, load the pure procedure part of the program into the segment address range assigned by the System Administrator. MIX is not necessary because procedure and data will be separated.
3. Use SEG's SHARE command to create one unsegmented runfile for each segment under '4001 that contains initialized data.
4. The segments numbered lower than '4000 (public segments) must then be shared by the System Administrator with the PRIMOS command SHARE. It should be noted that these segments need to be reinitialized in every cold start of PRIMOS. The System Administrator should, therefore, also include in the cold start command file (CMDNC0>C_PRMO or CMDNC0>PRIMOS.COMI) the PRIMOS SHARE commands necessary to reload these segments, and include the program in the UFD named SYSTEM. See the **System Administrator's Guide**.

The next example shows how to prepare a procedure for sharing. The source file is RHELP.PL1G, which is Program 8 in Appendix D. It lists phone numbers of people who can give emergency help on a product. Putting it in segment '4000 requires relocation of common blocks, since PL1G treats some of its library routines as common blocks.

On a default load, the load map for RHELP looks like this:

```
OK, SEG -LOAD
[SEG rev x.x]
$ LO RHELP
$ LI PL1GLB
$ LI
LOAD COMPLETE
$ MAP
*START  4002  000004  *STACK  7777  000000  *SYM    000213

SEG. #    TYPE       LOW      HIGH      TOP
  4001    PROC      001000    045025    045025
  4002    DATA      000000    057371    057371


ROUTINE        ECB          PROCEDURE      ST. SIZE  LINK FR.
  MAIN      4002  000004    4001  002077     000340  000133  4002  177400
  ####      4002  000027    4001  002165     000232  000133  4002  177400
  P$STOP    4002  000674    4001  007140     000030  000032  4002  000274
  P$EINF    4002  000732    4001  007301     000102  000201  4002  000326
  ####      4002  000762    4001  007361     000732  000201  4002  000326
  P$EOUTF   4002  002150    4001  010123     000102  000165  4002  001544
.
.
.
COMMON BLOCKS
  SYSPRINT  4002  000134  000257    SYSIN      4002  000414  000257
  P$CURSOR  4002  001130  000001    P$FCBC     4002  001132
  P$STAT    4002  001136  001006    P$BUSY     4002  003464  000002

OTHER SYMBOLS
  P$ENTP    4001  043434    P$APPC     4001  044042    P$MDLB     4001  044104
  F191RETS  4001  044606    P$MXLB     4001  044606    P$MNLB     4001  044622

$ Q
```

To make a file for sharing, assume that the System Administrator has assigned segment '2037, and use the following steps.

```
OK, SEG -LOAD
[SEG rev x.x]
$ COMMON ABS 4000              /* NOTE WELL FOR PL1GLB
$ SPLIT 167777 4000 150000     /*SPLIT SEGMENT '4000 AT '167777,
                               /*STACK AT '150000
$ S/LO RHELP 0 2037 4000       /*LOAD RHELP:  PROC IN '2037, DATA IN '4000
$ D/LI PL1GLB                  /*DITTO FOR PL1G LIBRARY
$ D/LI                         /*DITTO FOR SYSTEM LIBRARIES
```

```
LOAD COMPLETE
$ MAP
*START  4035  175324  *STACK  7777  000000  *SYM    000222


SEG. #    TYPE       LOW       HIGH       TOP
  2037    PROC      001000    045031    045031
  4000    PROC      001000    060623    060623
  4000    DATA      017000    076371    076371


ROUTINE          ECB           PROCEDURE      ST. SIZE  LINK FR.
  STACK_OV  4000  001212    4000  001152    000030  000024  4000   000612
  MAIN      4000  001236    2037  002077    000340  000133  4000   000632
  ####      4000  001261    2037  002165    000232  000133  4000   000632
  P$STOP    4000  002126    2037  007140    000030  000032  4000   001526
  P$EINF    4000  002164    2037  007301    000102  000201  4000   001560
  ####      4000  002214    2037  007361    000732  000201  4000   001560
  P$EOUTF   4000  003402    2037  010123    000102  000165  4000   002776
.
.
.
COMMON BLOCKS
  STACK$    4000  001006            SYSPRINT  4000  001366  000257
  SYSIN     4000  001646  000257    P$CURSOR  4000  002362  000001
  P$FCBC    4000  002364            P$STAT    4000  002370  001006
  P$BUSY    4000  004716  000002

OTHER SYMBOLS
  P$ENTP    2037  043434    P$APPC    2037  044042    P$MDLB    2037  044104
  F191RETS  2037  044606    P$MXLB    2037  044606    P$MNLB    2037  044622
  RUNIT     4000  001000    RESUME    4000  001053

$ RETURN                    /*RETURN TO SEG LEVEL
# SHARE                     /*CREATE SAM FILE
FILE ID:  RH
CREATING RH2037
CREATING RH4000
# DELETE                    /*DELETE SEGMENTED FILE RHELP.SEG
# Q
```

Now the System Administrator must use the SHARE command on RH2037:

```
    OPR 1
    SHARE SYSTEM>PANIC2037 2037 /*SHARE SEGMENT 2037 FOR EXECUTE ONLY
    OPR 0
```

The System Administrator should also insert RH2037 in SYSTEM and the share commands in CMDNC0>C_PRMO or CMDNC0>PRIMOS.COMI. Then when anyone enters RESUME RH4000, the same copy of RH2037 is used. (This is the copy in segment '2037.)

Appendix E contains a CPL file that loads a program and then shares it if the segment number entered by the user indicates a shared segment.

Sharing Two Programs in the Same Segment

If two small programs are to be shared, you may want to load them into the same shared segment to save space. You must be sure that the programs do not overlap. Since the programs will be loaded in separate loading sequences, SEG will not load one of them automatically at a high address in the segment. To load one program at a high address, first use A/SYMBOL to reserve the low addresses in the segment.

The next example uses the program CHECK.PASCAL (listed below in the section on shared data) and the set of PL1G programs listed in Example 10 in Appendix D. Suppose that a user has loaded CHECK in segment '2037 with the procedure shown in the example above. The map for CHECK shows that its highest address in segment '2037 is '3721:

```
*START  0266  177064  *STACK  4000  150000  *SYM    000062

SEG. #    TYPE       LOW      HIGH      TOP
  2037    PROC     001000    003721    003720
  4000    PROC##   001000    003061    003061
  4000    DATA##   177777    000000    016776
```

The two programs do not interact, but it is possible that different users may call the two programs simultaneously. The user knows from the map that CHECK ends at address '3721 in segment '2037. Therefore, the command **A/SYMBOL DUMMY PR 2037 5000** reserves enough space to protect the part of CHECK that is in segment '2037. Note that it is necessary to include the specification PR (procedure segment). Otherwise SEG assumes that a segment with a symbol in it is a data segment, and will not load the procedure for Example 10 into the same segment.

It is not necessary to reserve any space in segment '4000 because each program will use a different copy of that segment.

```
OK, SEG -LOAD
[SEG rev x.x]
$ COMMON ABS 4000            /*NECESSARY TO PLACE PL1GLB
$ A/SY DUMMY PR 2037 5000    /*SKIP PART OF '2037, MAKE IT A PROC SEGMENT
$ SPLIT 167777 4000 150000   /*THE REST IS STANDARD SHARING
$ S/LO MAIN 0 2037 4000
$ D/LO SUB1
$ D/LO SUB2
$ D/LI PL1GLB
$ D/LI
LOAD COMPLETE
$ MAP 1
*START  0266  107200  *STACK  4000  150000  *SYM    000177

SEG. #    TYPE       LOW      HIGH      TOP
  2037    PROC     006000    033177    033177
  4000    PROC##   001000    057251    057251
  4000    DATA##   177777    000000    167776
```

```
$ RETURN
# SHARE
ENTER FILE ID: LADD
CREATING LADD2037
CREATING LADD4000
# QUIT
```

It is not recommended that you try to share two programs in the same segment if one of them requires base areas. Such programs require that three separate areas be reserved — one for the low offsets, another for the high offsets, and a third for the procedure code. Then the second symbol must be expunged.

CREATING EXTERNAL COMMANDS

External commands are PRIMOS-level commands, all located in the UFD named CMDNC0 on the master disk. Examples are ED, COBOL, and SEG itself. You can create your own external commands to customize your system, or to make often-used programs run faster. You must create an unsegmented runfile, since the operating system recognizes only runfiles that can be RESUMEd and executed from segment '4000.

There are two ways to do this. One is to use SEG commands: SPLIT to load RUNIT into location '1000 of segment '4000 and SHARE to create a single-segment runfile as shown in the first section of this chapter. The other is to put into CMDNC0 an interlude program that runs the other program. The interlude may be a CPL file or one created through CMDSEG, which is discussed in Appendix B.

The single-segment runfile or the CPL file must be installed in CMDNC0 by the System Administrator.

The following sequence demonstrates preparation of an external command using the first method. The source file is RHELP.PL1G (Example 8 in Appendix D), which was also used in the first example above for preparing shared programs. The default load with its load map is shown above. To make an external command with SEG requires four steps. The first two are the same as in the previous example for preparing shared procedures.

It is not required that an external command be shared as in this illustration. For example, the command NSED brings the nonshared editor into memory for use in preparing files for installing programs before a system has been shared. But for everyday use it would be hard to imagine an external command that would be better unshared.

1. Get from your System Administrator the number of a shared segment that you may use. This example uses '2037.
2. Prepare a single-segment file using SPLIT and SHARE. The MIX command is not necessary because here you want to keep procedure and data in different segments. The data will go into your private segment '4000 and the procedure will be shared in segment '2037.
3. Have your System Administrator transfer the resulting runfile named xx4000 to CMDNC0 and share the lower-numbered file with the PRIMOS command SHARE. Probably you will want the file in CMDNC0 to have a more descriptive name than the one created by SHARE. Also the Administrator must install the file named xx2037

in the UFD SYSTEM and the appropriate share command in CMDNC0>C_PRMO or CMDNC0>PRIMOS.COMI.

4. Now everyone on the system can use this program simply by invoking its new name as a PRIMOS command line.

```
OK, SEG -LOAD
[SEG rev x.x]
$ COMMON ABS 4000
$ SPLIT 167777 4000 150000
$ S/LO RHELP 0 2037 4000
$ D/LI PL1GLB
$ D/LI
LOAD COMPLETE
$ MAP RHMAP          /*MAP WILL BE SAME AS IN LAST EXAMPLE
$ RETURN
# SHARE
TWO CHARACTER FILE ID: RH
CREATING RH2037
CREATING RH4000
# DELETE
# Q
```

Now the System Administrator copies RH4000 to CMDNC0 and RH2037 to SYSTEM, giving them more memorable names, such as PANIC.SAVE and PANIC2037. The System Administrator then shares the procedure segment as shown above for shared programs.

After PANIC2037 has been shared, anyone on the system may enter

```
PANIC
```

and display a list of people who can help.

CREATING SHARED DATA

It can be desirable for several users to have access to the same shared data. Some examples are user records that must be read for several purposes, as well as accessed for updates. This data can be installed in a shared segment. However, the data must be reinstalled each time the system is brought up from a cold start.

Note that COBOL does not support shared data areas because it does not support common blocks.

Here are the steps for creating shared data.

1. The segment must be shared before any reading or writing of data in the segment. Sharing must be done by the System Administrator from the system supervisor terminal. The following commands share a segment for writing as well as reading.

```
OPR 1
SHARE 2031 700 /*SHARE SEGMENT 2031 FOR READING AND WRITING
OPR 0
```

2. The user program must then place all of its shared data within a named common area. A FORTRAN program does this with COMMON. Pascal uses the indication {$E+}. PL1 uses the EXTERNAL attribute. PMA uses the pseudo-op COMM.
3. At load time, the named common area must be defined with SYMBOL so that it refers to the shared segment.

Example of Shared Data

The following FORTRAN 77 program puts a number into a shared segment.

```
PROGRAM SHRDTA
INTEGER*2 SIGNAL
COMMON/SHARED/SIGNAL
SIGNAL = 0
CALL EXIT
END
```

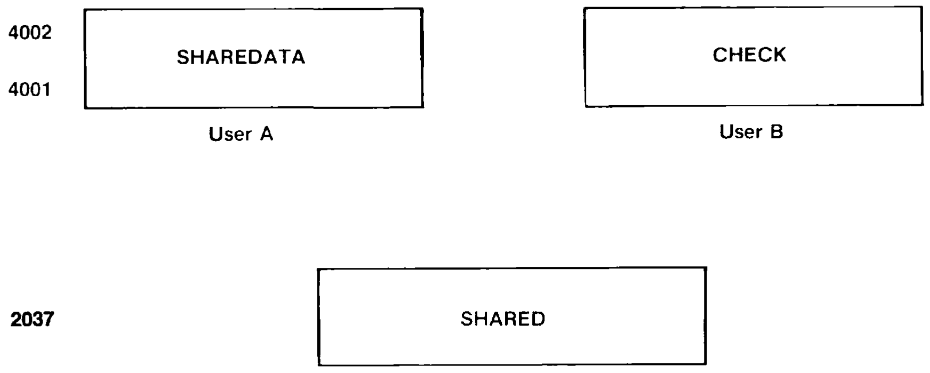
The next program, written in Pascal, reads the number.

```
PROGRAM CHECK;
    VAR
      {$E+}
         SHARED: INTEGER;
      {$E-}
    BEGIN
     SHARED := 1;
     WHILE SHARED = 1 DO; {NOTHING}
     {WAIT FOR F77 PROGRAM TO CHANGE SIGNAL TO 0}
     WRITELN ('OK TO PROCEED')
    END
END.
```

The following load procedures use the SYMBOL command to define placement of the common area for the preceding two programs. In the load maps, segment '2037 is shown in COMMON BLOCKS AND OTHER SYMBOLS, but not at the beginning of the map because the data is not initialized.

```
OK, SEG -LOAD               /* FOR FORTRAN PROGRAM
[SEG rev x.x]
$ SYMBOL SHARED 2037 0      /*2037 is allocated by the System Administrator
$ LO SHRDTA
$ LI
LOAD COMPLETE
$ MAP
*START  4002  000004  *STACK  7777  000000  *SYM    000023

SEG. #    TYPE       LOW      HIGH      TOP
  4001    PROC      001000    001062    001061
  4002    DATA      000000    000061    000061
.
.
.
COMMON BLOCKS
  SHARED    2037  000000

OTHER SYMBOLS
  F184DONE  4001  001013

$ EXEC
OK, SEG -LOAD            /*FOR PASCAL PROGRAM
[SEG rev x.x]
$ SYMBOL SHARED 2037 0
$ LO CHECK
$ LI PASLIB
$ LI
LOAD COMPLETE
$ MAP
*START  4002  000004  *STACK  7777  000000  *SYM    000052

SEG. #    TYPE       LOW       HIGH      TOP
  4001    PROC      001000    003711    003710
  4002    DATA      000000    001627    001627
.
.
.
COMMON BLOCKS
  P$AINP    4002  000054  000313    P$AOUT    4002  000370  000313
  P$ASET1   4002  000704  000020    P$ASET2   4002  000724  000020
  P$ASETI   4002  000744  000001

OTHER SYMBOLS
  F184DONE  4001  003624    SHARED    2037  000000

$ EXEC                   /*LOADS THE DATA INTO SEGMENT 3027
OK TO PROCEED
```

The data is now in a shared segment, while the programs run in the user's own space:



In this example, the Pascal program loads the correct data into the shared segment. In many applications, the System Administrator must be sure that the correct shared data is loaded at system start.

EXTENDING THE STACK

If the message STACK_OVF$ is displayed during execution, the stack can be extended with the STACK subcommand of VLOAD. To force use of a whole segment, set the **size** operand to '177774. If more than one segment is needed, use the SK subcommand of MODIFY discussed in the next section.

The default stack size may be changed during normal loading, or later by modifying a runfile.

The following example extends the stack of a previously created runfile. The program consists of the PL1G modules in Example 10 in Appendix D.

The first map below shows normal default stack placement in the first segment above '4000 that contains 2048 free addresses. In this case, that segment is '4001. The stack size is determined only at run time, but there is room for ('177776 - '026255) or '151521 16-bit addresses.

If you might need more room, the following procedure gives the stack a whole segment.

```
OK, SEG
[SEG rev x.x]
  RESTORE MAIN                  /*RESTORE EXISTING RUNFILE
  MAP 1                         /*LOOK AT ORIGINAL STACK
*START  4002  000004                          *SYM    000166

SEG. #    TYPE       LOW      HIGH      TOP
  4001    PROC##    001000    026255    026255
  4002    DATA      000000    056025    056025

$ QUIT
# VLOAD *                       /*DON'T OVERWRITE ANYTHING
$ ST 177774                     /*STACK NEEDS A WHOLE SEGMENT
$ SAVE                          /*SAVE SO WE CAN SEE STACK ON MAP
$ MAP 1                         /*LOOK AT NEW STACK
*START  4002  000004                          *SYM    000166

SEG. #    TYPE       LOW      HIGH      TOP
  4001    PROC      001000    026255    026255
  4002    DATA      000000    056025    056025

$ QUIT
```

Another example is in Chapter 3 on mapping.

RELOCATING THE STACK

The stack is relocated automatically with the SPLIT command of VLOAD. It may also be necessary to relocate the stack if it requires more than one segment, or if it overwrites segment '4035, which is needed for the symbol table in Rev. 19.

The SK subcommand of MODIFY is used if it is necessary to relocate the stack or to extend it into more than one segment. The following example demonstrates this procedure on the runfile already created and modified in the example above.

```
OK, SEG
[SEG rev x.x]
# RESTORE MAIN          /*GET AN EXISTING RUNFILE
# MODIFY
$ SK 4010 10 4011       /*PUT STACK IN 4010, EXTENSION IN 4011 AND BEYOND,
                        /*LEAVING FIRST '10 LOCATIONS FOR STACK HEADER
$ RETURN
# MAP 1
*START  4002  000004                        *SYM    000166

SEG. #    TYPE       LOW       HIGH       TOP
  4001    PROC      001000    026255    026255
  4002    DATA      000000    056025    056025

# QUIT
OK,
```

REPLACING PROGRAM MODULES

If a large runfile requiring a long load time must be modified, modules in it can be replaced quickly with the RL command of MODIFY. This technique avoids a lengthy reloading after a patch. It is recommended only for temporary tests, as it uses extra space.

The following example executes a runfile containing the modules in Program 10 in Appendix D. Then it replaces the first subroutine. The new subroutine must have the same internal program name (ECB name) as the one it replaces.

```
    OK, SEG MAIN

    this is main
    this is sub1
    this is sub2
    end of run

    OK, PL1G SUB1A
    0000 ERRORS (PL1G-REV 19.0.0)

    OK, SEG
    [SEG rev x.x]
    # RESTORE MAIN          /*GET EXISTING RUNFILE
```

```
# VLOAD *                  /*DON'T OVERLAY ANYTHING
$ RL SUB1A                 /*PUT SUB1A IN PLACE OF ECB WITH SAME NAME
LOAD COMPLETE
$ Q
OK, SEG MAIN

this is main
this is replacement
this is sub2
end of run
OK,
```

This new runfile should not be used permanently. The substitute subroutine is added at the top of the procedure and data segments, but the storage that was occupied by the old subroutine is not made available for reuse. Thus, with repeated updates, the runfile can become unwieldly and inefficient.

Note

In PL1G or Pascal programs with initialized common blocks, RL will not change the initialization.

CREATING AND USING TEMPLATES

A user or group may want to have a customized load file that always contains certain libraries and subroutines. This file can then be used as the basis for many application runfiles that need the same support libraries and routines. This kind of customized file is called a **template**. It is a general purpose procedure (usually shared) that must be completed with specific applications before being run.

The major use of templates is to move pure code into shared procedure segments, thus lowering memory usage. Unlike the system libraries that use direct entry links, templates do not allow reloading of the shared code without reloading all programs that use this code. The advantages of templates, on the other hand, are the ease of creation and the simplification of procedures for loading user programs. In a very large application program, the use of templates may reduce the execution time by sixty percent over conventional techniques.

A private library may also be created with the EDB utility described in the **Subroutines Reference Guide**.

There are three steps in creating shared templates:

1. Create the shared procedure segments.
2. Create the template — a runfile with shared segments. Specify that the main routine will be, not the first one in the template, but the first application routine to be loaded later.
3. Load an application program into a copy of the shared template.

Creating the Shared Procedure Segments

To create a template, use SEG without the -LOAD option to create a new, empty runfile. Then load the procedure, data, and common blocks that you have so far. Probably you will want to load procedure into a shared segment ('2xxx). It may also be desirable to load data into segment '4000 to keep the template small. In the example below, segment '2030 is allocated as the user's shared segment. Procedure and common blocks are loaded into segment '4000, which is split at '150000 so that all of SEG can also be in segment '4000. This model runs with Rev. 19.1 or 19.2, but in a later software revision, SEG may be enlarged so that the template has to be reloaded.

```
OK, SEG
[SEG rev x.x]
# LO #KIDA                    /* CREATE A SEGMENTED RUNFILE
$ SP 4000 150000              /* SEG ENDS BELOW '150000 AT REV.19.2
$ CO ABS 4000                 /* ALL DATA IN '4000
$ S/F/LI VKDALB 0 2030 4000 /* PROCEDURE IN '2030, LINKAGE IN '4000
$ D/PL                        /* LOAD PURE LIBRARIES LAST
$ MAP MAPFILE
LOAD COMPLETE
$ RE
# SH                          /* CREATE SEGMENT FOR SHARING
FILE ID: KI                 /*THROUGH 19.1 THIS IS ONLY TWO CHARACTERS
CREATING KI2030
$ QUIT
```

Segment '4000 is laid out with the SPLIT command without RUNIT, to permit NEW to be used to copy the template and CMDSEG to be used to set up runfile invocation. S/F/LI specifies the segments for the procedure and linkage. The user should always load the pure libraries after loading all routines that are to go in the template library.

The prefix F/ causes all modules in a library to be loaded. Most libraries are created in such a way that when they are loaded with the LIBRARY command, only the modules that are explicitly referenced by the user program will be loaded. In normal loading this technique is effective, but with templates the user program is loaded after the library, so all modules in a library most be loaded to assure that whatever is called will be there at runtime.

The SHARE command creates an unsegmented runfile for segment '2030. Nothing is in segment '4000 yet, so no new runfile is created for it. The part of the file that goes in segment '2030 may be prepared for sharing in this step or in later steps, as this part of the file will not be changed by subsequent modifications.

Segment '2030 must be shared at system start, so the share command should be included in C_PRMO.

Creating the Template

The next example illustrates the procedure for creating the template used for loading application programs. It involves getting the segmented file created above and creating a new copy with the start address set to '7777 0 with the START command of MODIFY. When SEG finds this address in an existing runfile, the next routine loaded becomes the main routine. (The next routine loaded will be the application routine at a later date.)

You can use NEW or COPY to copy the template before linking specific applications to it. NEW has the advantage that the runfiles are smaller; COPY copies all segments, including the shared segments, while NEW only copies segments '4000 and up. Before Rev. 19, use FUTIL instead of COPY.

NEW's disadvantage is that it restricts the use of segment '4000 when creating the template. In Rev. 19.2, there must be nothing below '120000 or NEW will overwrite SEG when it loads segment '4000. Thus, you cannot use RUNIT to set up runfile invocations. In the example above, format 1 of SPLIT is used so that RUNIT is not loaded and NEW can be used in the next step below.

Most of the time it is desirable to use NEW to set up the template since the template usually loads pure procedures into shared segments and NEW avoids copying them, thus decreasing runfile size.

```
OK, SEG
[SEG rev x.x]
# MODIFY #KIDA   /* GET RUNFILE FROM STEP ONE
$ START 7777 0   /* THIS CAUSES NEXT ROUTINE LOADED TO BE
                    MAIN ROUTINE
$ NEW KIDIMP     /* COPY TEMPLATE FOR LOADING USER PROGRAMS
RESTORING RUN FILE
$ RE
# QUIT
```

Loading Programs into the Template

To use the template, restore its runfile and copy it with NEW. Then use the VLOAD command followed by an asterisk to load in your specific program module without overwriting the other modules.

How the template is used depends upon the size of the specific application. A small application (less than 128K bytes) can be loaded into segment '4000, but a large one requires multiple segments.

The final examples below illustrate the procedure for loading application programs with the template.

Small Application Example: Here the user loads everything into segment '4000. It will run with Rev. 19, but if SEG is enlarged at a later software revision, the template may have to be recreated. This runfile can be executed with the command **SEG KIDSML**, since SEG is still present in segment '4000.

```
OK, SEG
[SEG rev x.x]
# MODIFY KIDIMP       /* GET TEMPLATE
$ NEW KIDSML          /* COPY TO USER'S OWN RUNFILE
RESTORING RUN FILE
$ RETURN
```

```
# VLOAD *              /* NOW THE USER CAN LOAD AS NEEDED
$ COMMON ABS 4000
$ S/LOAD SMALL 0 4000 4000
$ MAP1 MAPFILE
$ D/LIBRARY
LOAD COMPLETE
$ RETURN
# MODIFY
$ SK 4000 172000
$ RETURN
# QUIT
```

Large Application Example: If a large application is to be linked to the template you may not use segment '4000. The large example below loads everything into the user's unshared segments, using '4001-4007 for procedure and '4010 for data. This program can be run from SEG itself with **SEG KIDBIG**, since there is no user procedure or data in segment '4000. If the program is to be run as an external command from CMDNC0, a CPL program or CMDSEG must be used.

```
OK, SEG
# MO KIDIMP            /* SAME AS ABOVE
$ NEW KIDBIG
RESTORING RUN FILE
$ RE
# VLOAD *              /* NOW LOAD AS NEEDED
$ CO ABS 4010          /*THIS IS DIFFERENT FROM THE PREVIOUS EXAMPLE
$ S/LO LARGE 0 4001 4010 /*LEAVE SEGMENTS '4001-'4007 FOR DATA
$ MAP1 MAPFILE
$ D/LO MORE
$ MAP1 MAPFILE
$ D/LO ANDMORE
$ MAP1 MAPFILE
$ D/LI
$ MAP1 MAPFILE
LOAD COMPLETE
$ MAP MAPFILE 7
$ QUIT
```

ALLOCATING BASE AREAS

Base areas are the areas of memory reserved for out-of-range address resolution. They are sometimes required by Prime's older COBOL, which uses many 16-bit memory reference instructions and thus requires large base areas for their resolution. On a default load, base areas are allocated if needed in sector zero of each procedure block. Figure 3-3 in Chapter 3 shows such an allocation. If sector zero is used up, the message BASE SECTOR 0 FULL is displayed and base area size must be increased.

Two commands are available for increasing base area size: AUTOMATIC and SETBASE. The AUTOMATIC command causes procedures greater than '341 (decimal 225) 16-bit halfwords in length to have a base area allocated before and after the procedure code. The base areas are placed, not only around blocks as they are loaded, but between modules within a block (such as a library file or a user file containing more than one procedure). Thus, AUTOMATIC may provide base areas in more convenient places than the user can plan with SETBASE. The disadvantage of the AUTOMATIC subcommand is that it may reserve an unnecessarily large amount of storage as unused base area.

If a COBOL program is very large, AUTOMATIC still may not set base areas close enough together. It may be necessary to break the program into subprograms.

If you have many small programs, use SETBASE to insert a base area of a given size at the top of a segment. The disadvantage of SETBASE is that you can use it only to create base areas between commands in your load. It will not place base areas between modules that you load with one load command, as does AUTOMATIC.

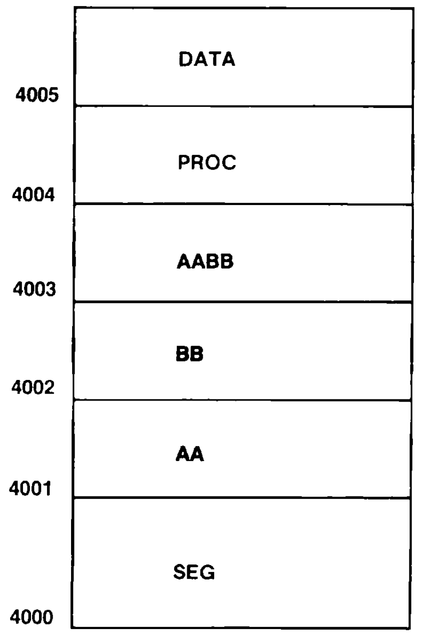
As a rule of thumb, if you get the message BASE AREA OVERFLOW and you have a lot more loading to do, use AUTOMATIC. If you have only a little more, use SETBASE. If you get the message BASE SECTOR 0 FULL, use the AUTOMATIC command.

USING RELATIVE SEGMENT NUMBERS

Occasionally you may want to specify a load into several segments without specifying which segments. For example, you might want common blocks to be loaded into different segments from data, but not care what those segments are. For this purpose, you may use relative segment numbers. These are small numbers (below 2000), which SEG replaces with its own segment numbers above '4000.

As an example, consider the following load, in which the user specifies that the common blocks AA, BB, and AABB should each be in its own segment, and separate from procedure segments. (This might help optimize paging of often-used common blocks by placing them on page boundaries.) The program is Program 1 in Appendix D. The user uses relative segments 3, 4, and 5. The map shows that SEG has translated these relative numbers into absolute segments '4001, '4002, and '4003, with the allocation pictured in Figure 4-5.

```
OK, SEG -LOAD
[SEG rev x.x]
$ R/SYMBOL AA 3
$ R/SYMBOL BB 4
$ R/SYMBOL AABB 5
$ LOAD PGM1
$ LI
LOAD COMPLETE
$ MAP
*START  4005  000004  *STACK  7777  000000  *SYM    000035

SEG. #   TYPE      LOW      HIGH     TOP
  4001   DATA     177777   000000   177777
  4002   DATA     177777   000000   177777
  4003   DATA     177777   000000   177777
  4004     PROC       001000     001363     001363
  4005     DATA       000000     000101     000101


ROUTINE         ECB           PROCEDURE      ST. SIZE  LINK FR.
  MAIN      4005  000004    4004  001121     000056  000054  4005   177400
  F$ERX     4005  000054    4004  001276     000020  000026  4005   177454

DIRECT ENTRY LINKS
  EXIT      4004  001324    TIMDAT     4004  001330    TNOU        4004  001334
  TNOUA     4004  001340    F$IFW      4004  001344    F$XFR       4004  001350
  F$CB77    4004  001354    F$STOP     4004  001360

COMMON BLOCKS
  [illegible] 4001  000000               BB         4002  000000
  AABB      4003  000000

OTHER SYMBOLS
  F184DONE  4004  001276

$ QUIT
```



Specific Translation of Relative Segments
Figure 4-5

5 SEG AND SEG-LEVEL COMMANDS

COMMAND SUMMARY

Following is a summary of all SEG commands, in alphabetical order within four groups:

- The SEG command
- SEG-level commands
- VLOAD or LOADER processor subcommands
- MODIFY processor subcommands

SEG displays the prompt #. The subprocessors display the prompt $.

Discussion of the first two groups above completes this chapter. Chapter 6 lists the subcommands for VLOAD, and Chapter 7 explains the commands within MODIFY.

SEG-level Commands

These commands can access runfiles in memory or on disk. Rust color indicates minimum valid abbreviations.

DELETE	Delete a runfile.
HELP	List SEG commands.
* **LO**ADER	Synonym for VLOAD.
MAP	Print a load map of a saved or current runfile.
* **MO**DIFY	Invoke the MODIFY processor.
PARAMS	Display starting location, stack location, register settings, and keys of a runfile.
PSD	Invoke VPSD debugger.
QUIT	Save runfile, close all files, and exit to PRIMOS.
RESTORE	Restore a runfile to memory.
RESUME	Restore a runfile to memory and begin execution.
* **SA**VE	Synonym for MODIFY.
SHARE	Create unsegmented files for procedure segments below '4001.
SINGLE	Create unsegmented file for any specified segment.
TIME	Display date and time a runfile was last modified.
VERSION	Display version number of SEG.
* **VL**OAD	Invoke the loader subprocessor.

* Invokes a subprocessor

VLOAD or LOADER Processor Subcommands

The loader subsystem performs most of the traditional loading and linking functions. Rust color indicates minimum valid abbreviations. See Chapter 6 for further discussion.

ATTACH	Attach to another UFD or sub-UFD.
AUTOMATIC	Place base areas between procedures.
A/SYMBOL	Reserve space for a defined symbol.
COMMON	Specify location of common blocks.
D/xx	Duplicate previous load parameters.
EXECUTE	Save and execute a program.
F/xx	Force loading of all routines in an object file.
IL	Load the impure FORTRAN library IFTNLB.
INITIALIZE	Restart a load.
LIBRARY	Load files from the UFD named LIB.
LOAD	Load an object file.
MAP	Generate a load map of the current runfile.
MIX	Put procedure and data in the same segment.
MV	Reserved.
NSCW	Suppress warning message when common area is redefined as smaller.
OPERATOR	Control system privileges.
P/xx	Load on a page boundary.
PL	Load the pure libraries PFTNLB and SPLLIB.
QUIT	Save the runfile and exit to PRIMOS.
R/SYMBOL	Reserve space for a defined symbol using relative references.
RETURN	Save a runfile and exit to SEG.
RL	Replace a module in the current runfile.
S/xx	Load a file to a specified absolute segment.
SAVE	Write a runfile to disk.
SCW	Reactivate warning message when common area is redefined as smaller.
SETBASE	Create a base area for indirect addressing.
SPLIT	Specify a segment for the stack and optionally load SEG's execution unit RUNIT.
SS	Protect a symbol from deletion from the load map.
STACK	Set minimum stack size.

SYMBOL	Define a symbol without reserving space for it.
SZ	Control the use of base areas in sector 0 of a procedure segment.
XPUNGE	Delete symbols and base area information from the load map.

MODIFY Processor Subcommands

The modification subsystem modifies existing runfiles. With it you can change runfile parameters. Rust color indicates minimum valid abbreviations. These commands are discussed more fully in Chapter 7.

NEW	Copy segments '4000 and above of a runfile.
PATCH	Write a patch to disk.
RETURN	Save runfile and exit to SEG level.
SK	Specify stack size and location and extension stack segments.
START	Change the starting address of a program.
WRITE	Rewrite to disk all segments above '4000.

OPERANDS FOR ALL COMMANDS

Files and directory names may be specified by pathnames. All numerical values must be octal. The following conventions are followed for parameters.

addr	Address within a segment
segno	Segment number
psegno	Procedure segment number
lsegno	Linkage segment number

THE SEG COMMAND

```
▶ SEG  [ pathname              ]
       [ -LOAD                 ]
       [ [pathname] 1/1        ]
```

Invokes the SEG utility for a number of purposes, according to the format, as explained below. **Pathname** may be as long as 160 characters. The internal name of the ECB, however, should not exceed eight characters — any more will be truncated. See the Glossary for definition of the ECB name in your programming language.

SEG has five formats:

SEG -LOAD	Causes SEG to display the **$** prompt, allowing the user to load, modify, or execute a SEG runfile. An example follows. It loads a binary file called CYPHER.BIN from a UFD named SECRET with a password of CRYPTO, and creates a runfile called CYPHER.SEG.

```
OK, SEG -LOAD
$  LOAD 'SECRET CRYPTO>CYPHER'
$
```

SEG	Causes SEG to display the # prompt, allowing the user to modify, load, or execute a SEG runfile.
SEG pathname	Loads the runfile into memory and starts execution. **Pathname** must be the filename or pathname of a SEG runfile.
SEG pathname 1/1	Restores the SEG runfile **pathname** to memory and tranfers control to the VPSD debugging utility. Control may be returned to SEG by the QUIT command.
SEG 1/1	Allows the current runfile to be examined and modified with the VPSD debugging utility. Control may be returned to SEG by the QUIT command, but the runfile cannot be executed at the SEG command level.

SEG employs two subprocessors, VLOAD or LOAD and MODIFY, which accept further commands. The subprocessors use the **$** prompt character.

If an error occurs during an operation, SEG prints an error message, then the prompt character. Error messages and suggested handling techniques are discussed in Chapter 8. When a system error (FILE IN USE, ILLEGAL NAME, INSUFFICIENT ACCESS RIGHTS, etc.) is encountered, SEG prints the system error and returns the prompt symbol if the error is not fatal.

SEG remains in control until a QUIT command returns control to PRIMOS, or an EXECUTE command starts execution of the loaded program.

SEG-LEVEL COMMANDS

▶ DELETE **[pathname]**

Deletes a saved SEG runfile; if **pathname** is omitted, deletes the current runfile.

▶ **HELP**

Displays a list of commands available.

```
          ⎧ pathname     ⎫
▶ LOADER  ⎨              ⎬
          ⎩ * [pathname] ⎭
```

Same as VLOAD below.

```
       ⎡⎡ pathname-1 ⎤                ⎤
▶ MAP  ⎢⎢            ⎥  [pathname-2]  ⎥   [map-option]
       ⎣⎣ *          ⎦                ⎦
```

Prints a load map of a runfile (**pathname-1**) or of the current runfile (*), either at the user's terminal or to a file (**pathname-2**). If both pathnames are omitted, the current runfile map is printed at the user's terminal.

Map Options	**Load Map Information**
0	Full map (default).
1	Segment use map only (map sections 1 and 2).
2	Segment use and base areas (map sections 1, 2, and 3).
3	Undefined symbols sorted by ascending address (map section 7).
4	Full map (identical to 0).
5	Reserved.
6	Undefined symbols, sorted alphabetically (map section 7).
7	Full map, sorted alphabetically.
10	Symbols sorted by ascending address.
11	Symbols sorted alphabetically.

The full SEG load map consists of seven sections. The selected map option above determines which sections are present; in small loads some, particularly section 3, may be absent. Some examples are:

`MA TEST 3`	Print unsatisfied references in SEG runfile TEST.SEG at the terminal.
`MA TEST ATLAS 7`	Write a full map of SEG runfile TEST.SEG sorted alphabetically into file ATLAS.
`MA 1`	Print a segment use map of the current runfile at the terminal.
`MA * ATLAS`	Write a full map of the current runfile into file ATLAS.

How to read a load map is explained in Chapter 3.

▶ MODIFY [pathname]

Invokes the MODIFY processor to create a new runfile or modify an existing runfile. If **pathname** is omitted, the current runfile is used.

This processor accepts a number of subcommands that are listed in Chapter 7. Modifications permitted are:

- Change starting ECB address.
- Change stack size and/or location and add stack extension segment.
- Save patched runfile to the same or to a new runfile.
- Create a new copy of a shared procedure template file, or of any segments numbered '4000 and above.

▶ PARAMS [pathname]

Displays the parameters of a SEG runfile. If **pathname** is omitted, parameters are displayed for the current SEG runfile.

The parameters are the starting address (ECB address of main program), stack location, contents of the A, B, and X registers, and the keys. The ECB is explained in the glossary. Keys and their values are explained in the **Assembly Language Programmer's Guide**.

Both the starting address and the stack location are given in two fields: the first is the segment number, the second is the location within the segment. All information is displayed in octal.

In the example below, the starting address is at location '2 in segment '4002; the stack is at location '1066 in segment '4001. The contents of the A, B, and X registers are 0. The keys value is '14000, indicating single-precision arithmetic and 64V addressing mode.

```
OK, SEG
# PA TEST1
START(2), STACK(2), A, B, X, KEYS
004002  000002  004001  001066  000000  000000  000000  014000
```

▶ PSD

Invokes VPSD debugging utility. See the **Assembly Language Programmer's Guide**.

▶ QUIT

Saves the runfile, returns to PRIMOS command level, and closes all open files.

▶ RESTORE [pathname]

Restores **pathname** to memory. If **pathname** is omitted, the current runfile is used.

This command allows restoration of a runfile to memory for examination with VPSD, or for patching. After patching, the patched version in memory may be executed; SEG does not restore a fresh copy to memory if a new filename has not been entered since the last RESTORE.

▶ **RESUME [pathname] [1/1]**

Restores **pathname** to memory and begins execution. If **pathname** is omitted, the current runfile is used. If the program is already in memory (the current runfile), then SEG does not bring a new copy from disk.

The directive 1/1 transfers control to the VPSD debugging utility described in the **Assembly Language Programmer's Guide**.

▶ **SAVE [pathname]**

Synonym for MODIFY.

▶ **SHARE [pathname]**

Prepares a segmented runfile for sharing. SHARE copies portions of **pathname** corresponding to segments below '4001 into an unsegmented (SAM) runfile. If **pathname** is omitted, the current runfile is copied. (See Chapter 4 for more information.)

After SHARE is entered, SEG asks for a file-id. The user may enter any characters, starting with an alphabetic character or one of the symbols # $. & * /. In revisions before 19.2, the file-id may be only two characters, and may not begin with a space. Thus, the entry may not be indented in a CPL file or in any other case. Starting with Rev. 19.2, the name entered may be as long as 28 characters. Any more will be truncated.

SEG will then rewrite records from the segmented runfile to new, unsegmented files with the names **CCnnnn** and **CC4000**, where **CC** are the characters supplied by the user and **nnnn** is a segment number in the '2xxx range (if any exist in the original file). If files with these names already exist, they will be overlaid.

The new file or files are now ready either to be run with the PRIMOS command RESUME or, if below '4000, to be shared by the System Administrator.

▶ **SINGLE [pathname] segno**

Creates an unsegmented (SAM) runfile for the segment **segno** of file **pathname**. If **pathname** is omitted, the current runfile is used. SINGLE is similar to SHARE, but can convert a file assigned to any segment number.

SINGLE may be used to create unsegmented (SAM) runfiles for segments '4001 and higher. Like SHARE, this command asks for file-id. Thus, the command **SINGLE 4031** copies segment 4031 of the current runfile to a SAM file named xx4031.

SINGLE must be followed by a segment number. If none is given, a new file is not created, although no error message is displayed. If no pathname is entered after the command, SEG prompts ENTER SAVE FILE NAME. Enter the pathname of the segmented file you want to convert. SEG asks for a file-id, which must follow the rules for the file-id requested by SHARE above.

▶ TIME [pathname]

Prints time and date of last **pathname** modification. If **pathname** is omitted the current runfile is used. This command allows the user to know when the runfile was last modified by anyone.

Example:

```
OK, SEG
# TI TEST
107-2-83    14:13:14
# Q
```

▶ VERSION

Displays SEG version number.

▶ VLOAD $\left\{ \begin{array}{l} \textbf{pathname} \\ \textbf{* [pathname]} \end{array} \right\}$

Defines the runfile name. The first format invokes the virtual loader to create a new runfile. The second format allows appending to an existing runfile. If **pathname** is omitted, SEG uses the current runfile; if there is no current runfile, SEG requests a name.

The asterisk permits addition to or replacement of a module in an existing runfile. This format facilitates the loading of program modules to shared procedure templates. If the asterisk is omitted, the runfile is initialized.

The VLOAD or LOADER command performs three functions:

- Defines (explicitly or implicitly) the name of the SEG runfile. This is unnecessary if the command line was SEG -LOAD. Prime's convention is to use the suffix .SEG with a SEG runfile name (e.g., MYPROG.SEG). Since this convention allows use of the other filename conventions explained in Chapter 2, the user should follow this convention in Rev. 18 and higher unless there are compelling reasons to do otherwise.
- Specifies whether a new file is to be written or an existing file is to be added to.
- Transfers operations to the loader. The loader prints the prompt **$** to differentiate itself from SEG-level commands.

The loader has a large number of subcommands that are described in Chapter 6.

6 THE VLOAD OR LOADER PROCESSOR

This chapter describes the subcommands accepted by SEG's VLOAD (LOADER) processor in response to the $ prompt character.

▶ ATTACH [ufd-name] [password] [ldisk] [key]

Attaches to a directory. Since SEG supports pathnames in all cases, this command is obsolete with Rev. 17 and higher.

ufd-name	Name of the target UFD; the default is the home UFD.
password	Password of target UFD if it is protected by a password.
ldisk	Key of logical disk to search for the specified UFD:

0 (or omitted)	Search logical disk 0.
100000	Search all logical disks.
177777	Search logical disk on which current directory is located.

key — Key for attach/set information:

0	Attach to UFD; do not set it as home.
1	Attach to UFD; set home to new current UFD.
2	Attach to sub-UFD in current UFD; do not set home to new current UFD.
3	Attach to sub-UFD in current UFD; set home to new current UFD.

▶ AUTOMATIC base-area-size

This subcommand is intended to allow more base area space for large procedures. Base areas are the areas of memory reserved for indirect address resolution. **Base-area-size** is the size of each base area. The default is zero and turns this feature off.

This command causes procedures greater than '341 16-bit halfwords in length to have a base area allocated before and after the procedure code. If the sector-zero base area is filled, AUTOMATIC causes the loader to attempt to place base areas where needed, that is, surrounding blocks of procedure. The base areas are placed, not only around blocks as they are loaded, but also between modules within a block (such as a library file or a user file containing more than one procedure). Thus, AUTOMATIC may provide base areas in more convenient places than the user can plan with SETBASE. The disadvantage of the AUTOMATIC subcommand is that it may reserve an unnecessarily large amount of storage as unused base area.

▶ A/SYMBOL sname [segtype] segno [size]

Defines a symbol in memory and reserves space for it using absolute segment numbers.

sname	Name of the symbol.
segtype	Type of segment, either DATA or PROCEDURE; if it is omitted, a data segment is assumed. If the segment does not yet exist, it will be created.
segno	Absolute octal segment number.
size	Number of 16-bit locations (octal) to be reserved for the symbol; if it is omitted, 0 is assumed.

This form of the SYMBOL command is for use only when addressing specific segments. The prefixes S/ or P/ (or D/ if appropriate) should be used with any remaining load commands. A/SYMBOL is particularly valuable for controlling the placement or size of common blocks, and for reserving space in shared segments. Both techniques are illustrated in Chapter 4. See also SYMBOL and R/SYMBOL below.

Cautions

1. Be sure that the number of locations reserved for the symbol is adequate for its intended use.
2. If there is not enough room in the segment or if the segment is not of the correct **segtype**, the next segment will be used.
3. You cannot use this command to satisfy previously identified unresolved references.

In the following examples, TOP+1 is the next available location in a given segment.

`A/SY KELVIN 4002 1000`	Place symbol KELVIN at the current TOP+1 in **data** segment '4002 reserving '1000 locations for it.
`A/SY KELVIN PR 4001 1000`	Place symbol KELVIN at current TOP+1 in **procedure** segment '4001 reserving '1000 locations for it. This is a way of placing a common block in a procedure segment.
`A/SY KELVIN DA 4001 1000`	Place symbol KELVIN at current TOP+1 in **data** segment '4001, reserving '1000 locations for it. If the segment specified did not exist, it would be created and the address of KELVIN would be 0 (a special case of TOP+1).

▶ **COMMON** $\begin{bmatrix} \textbf{ABS} \\ \textbf{REL} \bullet \end{bmatrix}$ **segno**

This command allows the user to specify the segment into which common blocks are loaded. It relocates common blocks using absolute or relative segment numbers. **Segno** is the segment number into which the common blocks are to be loaded. It is either an absolute or a relative octal segment number, depending on whether ABS or REL is specified. The default is REL.

Use the COMMON command for loading initialized common blocks that already exist in the system. Use SYMBOL for uninitialized common blocks.

If relative mode is specified, SEG first tries to use segment '4001, then, if MIX OFF is in effect, other data segments. Even if linkage has been loaded in segment '4000 with SPLIT and MIX, SEG puts common blocks in segment '4001 unless specifically directed to do otherwise. If segment '4001 has no room, SEG tries to load common blocks in each successively numbered segment not assigned to procedure.

When SEG's default segment assignments are used, the COMMON RELATIVE command causes SEG to load the common blocks into a different segment than that used for the link frames. This often decreases the size of the runfile that has to be restored, because only the first page or so of each segment need be used. The user may also desire to specify that certain common blocks be assigned in the same segment with specific link frames. (See SYMBOL, A/SYMBOL, R/SYMBOL.) For example:

`CO REL 1`
`LO MYPROG 0 2 3`
	Declares one of SEG's default segments as relative data segment 1 and uses it for loading COMMON. It assures that the link frame (data portion) is loaded in a different segment from the common blocks.

CO ABS reserves space only for those blocks that have been initialized. Since uninitialized common blocks do not take up space, but are simply entered into the symbol table, this enables SEG to optimize its buffer allocation, thereby decreasing both the size of the runfile and the time to restore it. For example:

`CO ABS 4015`	Causes the loader to load all common blocks into segment '4015 so long as they will fit, then into segment '4016, '4017, etc.

When loading to specific segments via the S/ or P/ prefix to LOAD (or D/ if appropriate), use the COMMON ABS command to assign common blocks.

```
▶ D/   { IL      }
       { LOAD    }
       { LIBRARY }
       { PL      }
       { RL      }
```

Continues a load using parameters of the previous load command. The D/ modifier is especially useful for large loads and in command files. Use of D/ decreases typing and minimizes errors; the creation of command files is made simpler.

D/ may be combined with F/ as either D/F/ or F/D/. It may not be combined with P or S, which by definition modify parameters.

An example will help illustrate use of D/:

`S/LO JUNK 0 4002 4004`	Load JUNK.BIN in procedure segment '4002 and data segment '4004.
`D/F/LO TRASH`	Forceload all of TRASH.BIN in the same segments as above.
`D/F/LI VKDALB`	Forceload all of VKDALB in the same segments as above.
`D/LI`	Load system libraries in the same segments as above.

In all the above cases, procedures are loaded into segment '4002 and link frames into segment '4004, unless either one overflows, in which case loading continues in the next segment.

The next example causes MAIN.BIN and the system libraries to be loaded in the same pair of procedure and linkage segments. SUB1.BIN and B_SUB2 will be loaded into one pair also, but this will be a different pair from those used for MAIN and the system libraries. This might be useful for debugging.

```
LO     MAIN
LO     SUB1 0 1 1
D/LO   B_SUB2
LI
```

Note

The D/ prefix does not cause duplication of a preceding P/ in a load.

▶ EXECUTE

Saves and executes a program.

VLOAD first saves the program, if necessary, and then executes it. After execution, control returns directly to PRIMOS.

▶ F/ { IL | LOAD | LIBRARY | PL | RL } [pathname] [addr psegno lsegno]

Forces loading of all routines in an object file. Otherwise, when library files are loaded, often only those modules that have been called by a previous program are loaded. The command is useful in preparing templates when the calls in the main program are not yet known. See Chapter 4 for a discussion of templates and of forced loading of library files.

The F/ prefix should be used in any instance where a module must be part of a program, but is not referenced in such a way that it appears in the load map.

pathname — Object file to be forceloaded. It is required with some commands and must be omitted with others:

Load Command	**Pathname Requirement**
LOAD or RL	Required.
PL or IL	Omitted.
LIBRARY	Optional (if it is omitted the system libraries are force-loaded).

addr — Starting address in **psegno** for the procedure part of the binary file. If 0 is specified, the current program TOP (as shown in the load map) is used. This is also called the current load point.

psegno — Absolute or relative number of procedure segment, depending on whether or not an absolute prefix (S/, P/, or D/ if appropriate) is included with the subcommand.

lsegno	Absolute or relative number of segment for link frames and data, depending on whether or not an absolute prefix (S/, P/, or D/ if appropriate) is included with the command.

If **psegno** and/or **lsegno** are 0, SEG's default segments starting with '4001 and '4002 are used. If S/ (absolute segment loading) or P/ (load to page boundary) is being used, F/ must also use absolute segment numbers. D/ may also require absolute numbers if it duplicates a P/ or S/ load.

In a simple forced load both the procedure segment number and the linkage segment number are default assignment numbers. The defaults resulting if parameters are omitted are the same as for the commands without the F/ prefix. For example:

`F/LI`	Forceload **all** of each system library into default segments.

Forced loading is very useful in building a shared template or creating a partial load. For example:

`F/LO THING`	Forceload all modules in THING.BIN into default segments.
`F/LI`	Forceload all the system libraries into default segments.

F/ may be combined with P/ as F/P/ or P/F/ to force loading of all routines in an object file to a page boundary. For example:

`F/P/LO JUNK 0 4002 4004`	Forces loading of the procedure portions of JUNK.BIN on page boundaries of segment '4002 and data/link frame portions of JUNK.BIN on page boundaries of segment '4004.

F/ may be combined with S/ as F/S or S/F/ to force loading of all routines in an object file to specific segments. This format is useful for creating shared templates. For example:

`F/S/PL 4000 2000 4002`	Forces loading of all modules in the libraries PFTNLB and SPLLIB into segment '2000 beginning at location '4000, with the linkage area in segment '4002.

D/ may also be combined with F/ as D/F or F/D to minimize typing errors.

▶ IL[addr psegno lsegno]

Loads the impure FORTRAN library IFTNLB. This subcommand is an abbreviation for LI IFTNLB.

addr	Starting address in **psegno** for the procedure part of the binary file. If 0 is specified, the current program TOP is used.
psegno	Relative or absolute number of procedure segment, depending on whether an absolute prefix (S/, P/, or D/ if appropriate) is used.
lsegno	Relative or absolute number of segment for link frames and data, depending on whether an absolute prefix is used.

If **psegno** and/or **lsegno** are 0, SEG's default segments starting with '4001 and '4002 are used.

Use IL when creating shared procedures to load the impure part of the system libraries into the non-shared segment Normally, the compound commands S/, F/, and P/ are used with IL to control library segment assignment

▶ INITIALIZE

Initializes and restarts the VLOAD (LOADER) subprocessor This subcommand is used to abort a bad load or to begin a new load after a SAVE

▶ LIBRARY [filename] [addr psegno lsegno]

This command is an abbreviation for LOAD LIB>filename It loads a library file

filename	Name of the library file to be loaded, if it is omitted, the system library files PFTNLB, IFTNLB, and SPLLIB are loaded
addr	Starting address in **psegno** for the procedure part of the binary file. If 0 is specified, the current program TOP is used
psegno	Relative or absolute number of procedure segment, depending on whether an absolute prefix (S/, P/, or D/ if appropriate) is used.
lsegno	Relative or absolute number of segment for link frames and data, depending on whether an absolute prefix is used

LIBRARY can be combined with D/, F/, P/, and S/ The only difference between **LIBRARY** and **LOAD** is that the former causes the loader to look for a file in the UFD named LIB

▶ LOAD [pathname] [addr psegno lsegno]

Loads and links an object file into a runfile

pathname	Name of the object module
addr	Starting address in **psegno** for the procedure part of the object module If 0 is specified, the current program TOP is used
psegno	Relative or absolute number of procedure segment, depending on whether an absolute prefix (S/, P/, or D/ if appropriate) is used
lsegno	Relative or absolute number of segment for link frames and data, depending on whether an absolute prefix is used

This command links the object module to the other modules in the current runfile If **psegno** and/or **lsegno** are 0, SEG's default segments starting with '4001 and '4002 are used

Lsegno does **not** select a segment for common blocks, the COMMON or SYMBOL commands must be used if common blocks are to be loaded into specific or absolute segments

LOAD can be combined with D/, F/, P/, and S/

▶ MAP [pathname] [map-option]

Creates a load map of the current runfile.

pathname	Name of the file into which the load map is to be written. If **pathname** is omitted, the map is displayed at the user's terminal.
map-option	Type of load map to be generated; options are the same as in SEG's MAP command. See MAP in Chapter 3 or Chapter 5.

MAP outputs the specified map either to the user's terminal or to a file. QUIT, EXECUTE, and RETURN cause the map file, if any, to be truncated. Only one map file can be generated in each session of VLOAD. When a map file is specified it is opened on PRIMOS unit 13. (File units are explained in the **Subroutines Reference Guide**.) That file remains open until the load session is complete, and any additional MAP commands specifying output to a file will use the one already opened. When the user exits from the loader (via EX, QU, or RE) the map file is closed. If the user has a file already open on PRIMOS unit 13 when SEG's loader is invoked this open file will be used for the map. Under these circumstances the loader will **not** close the file. In this respect, the MAP subcommand of VLOAD differs from the MAP command of SEG. If multiple maps are to go to the file, any nonnumeric character may be used for **pathname** after the first reference.

The load map options are identical to those generated by SEG's MAP command and are discussed in detail in Chapters 3 and 5.

▶ MIX $\begin{bmatrix} \text{ON} \bullet \\ \text{OFF} \end{bmatrix}$

Allows loading of linkage and common blocks in procedure segments so that the RUNIT files described in Chapter 4 may be created. When MIX has been invoked all segments will be created as procedure segments.

MIX or MIX ON invokes the MIX feature. MIX OFF turns the feature off and resumes regular loading into separate data and procedure segments. The feature is not reset by the INITIALIZE command. In this way users may elect to save a copy of SEG with MIX turned on and make this the default mode of loading. In general, loading under the MIX option reduces the number of segments required for a program, but debugging such programs may be more difficult. If there are common blocks in the runfile, however, the file may turn out bigger unless special placement is used. See Chapter 4 for examples.

▶ MV [start-symbol move-block dest-segno]

Moves portions of the load file. This command is intended primarily to facilitate the creation of shared libraries by Prime and not as a user command. Code or data moved by the MV command cannot be executed or used in the destination location as the links to the moved area still reflect the original addresses.

Start-symbol	The name of a symbol indicating the start of the move. An example is given with the START prompt below. Information will be moved from **start-symbol** to the current HIGH in the segment.

Move-block	A previously defined symbol corresponding to a five-address block into which information concerning the move will be placed. **Move-block** is optional, but if it is specified, the format after the move will be:

Locations 1,2	Address to which move was made.
Locations 3,4	Address from which move was made.
Location 5	Number of 16-bit locations moved.

Move-block may thus be used to restore the moved information to its original place.

Dest-segno	The segment to which the information is to be moved. It may be either a procedure or a data segment. If there is not enough room in the segment, the next segment with sufficient room will be used.

MV without parameters causes prompts for further input:

Prompt	Response
START:	The response to START may be either a defined symbol from the symbol table or the segment and address of the source location. For example:

```
START:  FOOBAR
```

or

```
START:  4001 1232
```

END:	END may also be specified as a symbol or numeric value. If END is specified symbolically it must be in the same segment as that defined for the start of the move. If END is defined numerically it must be one number representing the first location which is not to be moved in the segment. For example:

```
END:  FOOEND
```

or

```
END:  2000
```

In either case locations up to but not including the end location will be moved.

A value of 0 or no value (CR only) will cause MV to move locations up to and including the current HIGH in the segment specified by START.

DEST. SEGMENT:	A segment number into which the block of information is to be moved. It may be either a procedure or a data segment. If there is not enough room in the segment, the next segment with enough room will be used.
IP VECTOR:	Corresponds to **move-block** above. It is also optional.

▶ NSCW

Starting at Rev. 19.2, suppresses the warning message when common area is redefined as smaller.

▶ OPERATOR option

Gives or removes system privileges. The actual implementation of OPERATOR may change from revision to revision. Consequently, the command is not considered to be a supported function of SEG.

Option	Function
0	Reinstate restrictions.
1	Relax restrictions.

This subcommand allows creators of specialized software to refer to locations in other segments using only the address in the segment. This is dangerous, and is not normally allowed by SEG's loader. The OPERATOR command is used to relax this restriction when necessary.

This command must be used before any errors are generated. It is not cancelled by the INITIALIZE command.

▶ P/ $\left\{\begin{array}{l}\textbf{IL}\\\textbf{LOAD}\\\textbf{LIBRARY}\\\textbf{PL}\\\textbf{RL}\end{array}\right\}$ [pathname] option [psegno] [lsegno]

Loads an object file on a page boundary. The command was originally intended to load operating system procedures and their linkage areas on page boundaries to minimize wired memory. Users may also find the command helps to save search time by placing frequently-referenced items at the start of a page

pathname	Object file to be loaded. It is required with some load commands and must be omitted with others:

Load Command	Pathname Requirement
LOAD or RL	Required.
PL or IL	Omitted.
LIBRARY	Optional (if it is omitted, the system libraries are loaded).

option	Determines what shall be loaded:

PR	Load just the procedure on a page boundary.
DA	Load just the link frames on a page boundary.
(omitted)	Load both procedure and link frames on a page boundary.

psegno	Absolute octal number of procedure segment.
lsegno	Absolute octal number of segment for link frames and data.

Default segments are those of the current procedure and/or link frame pointers; if necessary, SEG creates new segments. If either PR or DA is specified for **option**, loading in the default segment begins at its current load point. Only the first routine in the file is placed on a page boundary.

Note

A subsequent load with the D/ prefix will not place its operand on a page boundary.

P/ may be combined with F/ to force loading on a page boundary as F/P/ or P/F/.

The P/ prefix incorporates the features of S/; the segments supplied by the user are assumed to be specific (absolute) segments.

For example:

`P/LO JUNK 4002 4004`	Place both procedure and data of JUNK.BIN on a page boundary, the first in segment '4002 and the second in '4004.
`P/LI PFTNLB PR`	Place procedure portion of the first loaded library routine on a page boundary.
`P/LO MUMBLE DA`	Place link frame of first loaded routine of MUMBLE.BIN on a page boundary.

▶ PL [addr psegno lsegno]

Loads the pure libraries PFTNLB and SPLLIB. This is an abbreviation for LI PFTNLB and LI SPLLIB. Use PL when creating shared procedures to load the pure part of the system libraries into reentrant procedure segments (those numbered below '4000).

addr	Starting address in **psegno** for the procedure part of the binary file. If 0 is specified, the current program TOP is used.
psegno	Relative or absolute number of segment into which procedure is to be loaded, depending on whether an absolute prefix (S/, P/, or D/ if appropriate) is used.
lsegno	Relative or absolute number of segment into which link frames are to be loaded, depending on whether an absolute prefix is used.

If **psegno** and/or **lsegno** are 0, SEG's default segments starting with '4001 and '4002 are used. Normally, the compound commands S/, F/, and P/ are used with PL to control library segment assignment.

▶ QUIT

Performs a SAVE and returns to PRIMOS command level. The user remains attached to the last UFD specified in a PRIMOS ATTACH command, or to the UFD specified as home in VLOAD's ATTACH command.

▶ R/SYMBOL sname [segtype] segno [size]

Defines a symbol, **sname**, and reserves space in a relative segment for it using relative segment numbers.

sname	Name of the symbol.
segtype	Type of segment, either DATA or PROCEDURE; if omitted, a data segment is assumed.
segno	Relative segment reference number. If 0 is specified, the first available segment of the current type is used.
size	Number of locations to be reserved for the symbol. If omitted, it is assumed to be 0.

This form of the SYMBOL command is especially useful in controlling the placement and size of named common blocks during a load. If the segment specified does not exist, or does not contain enough room, a new segment is created to locate the symbol. Check to be sure that the number of locations reserved for the symbol is sufficient. See SYMBOL below.

This command may not be used to satisfy unsatisfied references already existing in the load.

In the following examples, TOP+1 is the next available location in a given segment.

`R/SY COUSIN 0 1000`	Places symbol COUSIN at the current TOP+1 in a data segment with no reference number, reserving '1000 locations for it.
`R/SY COUSIN PR 0 1000`	Places symbol COUSIN at current TOP+1 in a procedure segment with no reference number, reserving '1000 locations for it. This is a way of placing a common block in a procedure segment.
`R/SY COUSIN DA 1 0`	Places symbol COUSIN at current TOP+1 in a data segment with reference number 1, reserving 0 locations for it. If a segment with reference number 1 does not exist, it is created and the address of COUSIN is 0 (a special case of TOP+1).

▶ RETURN

Performs a SAVE and returns to SEG command level.

▶ RL pathname [addr psegno lsegno]

Replaces binary modules in the current runfile. RL replaces a routine or routines in a SEG runfile, making it possible to replace a defective subroutine without having to rebuild the runfile completely.

pathname	Name of the module to be replaced.
addr	Starting address in **psegno** for the procedure part of the binary module. If 0 is specified, the current program TOP is used.
psegno	Absolute or relative number of procedure segment, depending on whether or not an absolute prefix (S/, P/, or D/ if appropriate) is used with RL.
lsegno	Absolute or relative number of segment for link frames and data, depending on whether or not an absolute prefix (S/, P/, or D/ if appropriate) is used.

If **psegno** and **lsegno** are omitted, SEG loads the file in the first default procedure and data segments with enough room.

The new module logically and functionally replaces the old module of the same name by patching the entry point. The new module need not be the same length as the old, since it is not physically reloaded on top of the old module; just the subroutine entry points are patched. However, since the old module still occupies space in the runfile, overuse of the RL command may significantly increase runfile size as well as restoration and execution time.

The entry points in the replacement module must include all the entry points that were defined in the old module. In particular, a new ECB must be supplied for each old ECB. New common block names may be added in the replacement module, but redefinition of old common blocks is not permitted.

If a binary module contains more than one procedure, all of the procedures in that module will be replaced (or be loaded, in the case of the new module). The RL command causes the loader to continue loading procedures from a module even after a LOAD COMPLETE signal, in case new unresolved references are introduced.

To use an existing runfile for reloading, use the VLOAD * command. If the runfile contains no segments below '4000, use MODIFY's NEW subcommand to make a copy of the runfile, since a mistake could destroy its integrity.

When reloading a module, remember that SEG's loader has no record of whether the last load was to a specific segment or not. Therefore, appropriate load parameters must be supplied for at least the first module to be replaced.

FORTRAN or Pascal modules with internal functions may cause trouble, since these functions have no ECB name.

In Pascal and PL1G, RL does not change the initialization values of common blocks.

Examples are:

`RL B_MODULE`	Place the routines contained in B_MODULE in SEG's default segments and logically replace the old routines with the new one.
`S/RL FOO 0 4002 4004`	Use the routine in FOO.BIN to replace a loaded routine of the same name using specific segments.
`D/RL MUMBLE`	Use the routine in MUMBLE.BIN to replace another routine using the parameters for FOO above.

Another example is given in Chapter 4.

RL can be combined with D/, F/, P/, and S/.

$$\blacktriangleright\ \mathrm{S/}\quad \left\{\begin{array}{l}\textbf{IL}\\ \textbf{LOAD}\\ \textbf{LIBRARY}\\ \textbf{RL}\\ \textbf{PL}\end{array}\right\}\quad \textbf{[pathname] addr psegno lsegno}$$

Loads an object file to specified absolute segments.

pathname	The object file to be loaded. It is required after some combinations and must be omitted after others:

Load Command	Pathname Requirement
LOAD or RL	Required.
PL or IL	Omitted.
LIBRARY	Optional (if it is omitted, PFTNLB, SPLLIB, and IFTNLB are loaded).

addr	Starting load address (octal) in the procedure segment. If 0 is specified, loading starts at the current pointer position (TOP).
psegno	Absolute octal number of procedure segment.
lsegno	Absolute octal number of segment for link frames and data.

If the segments do not already exist, they will be created, and if the specified segment runs out of room, the next segment will be used.

S/ does not place common blocks; this should be done prior to the load.

Examples are:

`S/LO JUNK 0 4002 4004`	Load object file JUNK.BIN with its procedure beginning at the current load pointer location in segment '4002 and its link frame areas beginning at the current load pointer in segment '4004. Assume that previously any common blocks were located with a CO ABS command.
`S/IL 0 4000 4000`	Load the impure library IFTNLB into the split segment '4000.

S/ may be combined with F/ as either S/F/ or F/S/, but may not be combined with D/. A command including D/ may, however, follow an S/ command.

Caution

If an S/ is specified beginning at an address too high in the segment for the module's size, the load will not extend to a new segment. It will wrap around to the beginning of the same segment and destroy data there.

▶ SA VE [a-reg] [b-reg] [x-reg]

Saves the results of the load. A location for the stack is also assigned.

a-reg	Value of A register (obsolete).
b-reg	Value of B register (obsolete).
x-reg	Value of X register (obsolete).

Writes all buffers to disk and sets stack pointers. SAVE is obsolete except that it sets the stack location, which is useful before getting a map.

▶ SCW

Starting with Rev. 19.2, reactivates the warning message when common area is redefined as smaller.

▶ SETBASE segno length

Creates a base area for address resolution linkages.

Segno is the segment in which the base area is to be located. It must be either a procedure segment, or undefined. **Length** is the length of the base area to be created. The base area is created at the current TOP of the segment. There is no facility in SEG for placing a base area at a specific location in a segment. PMA contains a pseudo-operation, SETB, to create additional base areas.

See Chapter 4 for a discussion of the differences between SETBASE and AUTOMATIC.

▶ SPLIT [segno addr | [addr] • | addr ssegno saddr [esegno]]

```
         ┌segno addr                    ┐
▶ SPLIT  │[addr] •                      │
         └addr ssegno saddr [esegno]    ┘
```

Breaks a segment into procedure and data portions. If there are zero, one, three, or four operands, the loader also loads RUNIT, RESUME, and a stack overflow handler. There are three formats:

Format	Operand	Meaning
Format 1	segno	Absolute octal number of the segment to be split.
	addr	Octal location of the split within the segment (starting address of the stack). Procedure will be loaded at addresses below the split, and data (linkage and common blocks) will be loaded above it.
Format 2	addr	Since **segno** is not supplied, segment '4000 is assumed and the interlude program RUNIT will be loaded beginning at location '1000. No data or code is allowed above the location where RUNIT puts its stack ('150000 in Rev. 19). If **addr** is omitted or 0, then RUNIT is loaded and the segment is **not** split.
Format 3	addr	Octal address of split (beginning of stack) in segment '4000. As in format 2 above, RUNIT is loaded. The location of the stack, however, is defined by the following parameters.
	ssegno	Octal segment number of the stack.
	saddr	Octal address within the segment for the split (start of the stack).
	esegno	Octal segment number for the extension stack.

Chapter 4 explains how loading SEG's small invocation program, RUNIT, into '4000 eliminates the need to have SEG occupy segment '4000. Further, RUNIT permits the program to be invoked either from the user UFD (with RESUME) or from the UFD named CMDNC0 as a PRIMOS-level command. Formats two and three allow execution with RESUME if all loaded information is in segment '4000 and the runfile is reformatted with SEG's SHARE command.

The third format of SPLIT is recommended in order to avoid having RUNIT's default stack placement overwrite data that you load into segment '4000. If you use Format 2, be aware that, at different revisions of PRIMOS, the RUNIT stack may be placed differently. To find out where the default placement of SEG's stack is in your software revision, look in the file SEGSRC>SHARES.PMA on the master disk. Locations 1 and 2 after STACK$ define the stack location. See Appendix F. Such inconvenience is avoided if you use the third option of SPLIT in order to place the stack where you want it. The stack may only be placed in procedure segments.

Examples are:

`SP 40000`	Creates a procedure area in segment '4000 below address '40000 and a link area above '40000. RUNIT will be loaded at '1000 in the procedure portion. The symbol table will contain RUNIT and RESUME as defined symbols and MAIN as an undefined symbol. (The use of MAIN is explained in Chapter 4.) RESUME may be used as the starting address if the user wishes to preserve an existing stack. RUNIT is the normal starting address.
`SP 4001 10000`	Makes locations below '10000 in segment '4001 available for procedures and the remainder of segment '4001 available for data.

The Source Level Debugger ignores the result of SPLIT and uses its own stack to run the program.

▶ SS symbol-name

Saves **symbol-name**, preventing XPUNGE from deleting it. To protect multiple symbols, name each by a separate SS command.

▶ STACK ssize

Sets the minimum stack size.

ssize	Minimum required stack size (octal). An **ssize** of '177774 forces use of an entire segment for the stack.

The location is not changed. To change stack location, use the SK command in the MODIFY processor.

Example:

`ST 100000`	Requires at least '100000 free locations in the segment used for the stack.

$$\blacktriangleright \textbf{ SYMBOL [new-symbol-name]} \left\{\begin{array}{l}\textbf{old-symbol-name}\\ \textbf{segno [addr]}\\ *\end{array}\right\} \textbf{[octal-no]}$$

Defines a symbol but does not reserve space for it.

new-symbol-name	Symbol name. If it is omitted, blank (unlabeled) common is assumed.
old-symbol-name	Previously defined symbol to be replaced.
segno	Absolute octal number of segment in which the symbol is to be located.
addr	Octal address of 16-bit location in the specified segment for the symbol. If omitted, it is assumed to be zero.
*	Store symbol at current load point or TOP of linkage segment. See the chapter on load maps.
octal-no	Constant offset in 16-bit locations. If it is preceded by a negative sign, the value of the offset is negative.

The value assigned to **new-symbol-name** may be a previously defined symbol (**old-symbol-name**), a location (**segno addr**) or the value of the load point (*). Constant offsets may be added to the symbol value (**octal-no**) and may be negative. The offset may be an expression. Thus the offset may be used to define a symbol as a location within a common block, as in the third example below.

It is not necessary that the segment ever be defined by the loader because SYMBOL does not actually assign a segment in SEG's segment table for the **symbol-name**, but only an entry in the symbol table. Hence, the command is useful for defining uninitialized common blocks that will not be restored to memory when a program is invoked; this will decrease restore time prior to execution. SYMBOL defines an address for a symbol that already exists. The prefixes A/ and R/ allocate space as well as defining the symbol. They may be used, for example, to reserve space for large common blocks.

SYMBOL cannot be used to define a common block that will be initialized by a DATA statement or BLOCK DATA subprogram in FORTRAN. Use the COMMON command for this purpose. Symbol names defined by this command cannot be used to satisfy unsatisfied references in a partial load.

Examples are:

`SY CYMBAL 4001 12000`	Locates symbol CYMBAL at segment '4001, location '12000.

`SY 4015 1000`	Defines blank common area as beginning in segment '4015 at location '1000. Here the user has located a blank common block above the other program procedure and data segments so that overflow of blank common (indexes out of range) will not overwrite other code. The user must determine which segments and locations are to be used by examining SEG's load map.
`SY NEWNAME AA 6`	Defines the new symbol NEWNAME as the sixth location in common block AA in segment '4003. Thus, if AA consists of four 32-bit items, NEWNAME references the fourth item.

▶ SZ **segno** $\begin{bmatrix} \textbf{YES} \\ \textbf{NO} \bullet \end{bmatrix}$

Controls the use of sector zero base areas in procedure segments.

Segno	The segment whose sector zero base area is affected.
NO	(Default) causes the given base area not to be used. An error is generated if a link is needed.
YES	Allows the use of the sector zero base area again.

▶ XPUNGE **dsymbols [dbase]**

Expunges the defined symbols indicated by **dsymbols** from the symbol table and deletes base area information indicated by **dbase**. Undefined symbols and symbols referenced in an SS command cannot be removed. XPUNGE allows a program to call two or more routines by the same name. This command is intended for use by the COBOL compiler in overlaying defined symbols in different phases of its compilation.

dsymbols		Action
0	Delete	all defined symbols, including common blocks.
1	Delete	only entry points, leaving common blocks.

dbase		Action
0	Delete	all base area information. (Default)
1	Retain	only sector zero information.
2	Retain	all base area information.

7 THE MODIFY PROCESSOR

The following commands are accepted by SEG's MODIFY processor, in response to the $ prompt.

▶ NEW pathname

Writes a partial copy of the current SEG runfile to disk. NEW duplicates all segments greater than or equal to '4000 under the specified new pathname. The full map and all references to segments below '4000 are preserved. NEW may be used to create a template for further additions (Chapter 4) or to save a patched version of a runfile already in memory. **Pathname** must not already exist.

After NEW is used, all subsequent commands with no pathname specified will use the one specified with NEW as the current runfile.

Caution

If there is a segment '4035 in the runfile, it will overwrite the symbol table and crash SEG.

If NEW is to be used, do not use the formats of SPLIT that load RUNIT. See Chapter 6.

▶ PA TCH segno baddr taddr

Writes a patch to disk.

segno	Absolute octal number of the patched segment.
baddr	Lowest octal location of the patch.
taddr	Highest octal location of the patch.

PATCH writes specified portions of segments '4001 and above in the current runfile to the disk. If the patch spans areas that were not previously on disk, all pointers are corrected so that the patch becomes a permanent part of the runfile. For example:

```
PA 4001 3000 3777
```

writes to disk locations '3000 to '3777 in segment '4001, extending the in-use range of the segment if needed.

▶ RE TURN

Saves the runfile and returns to SEG command level.

▶ SK { ssize | segno addr | ssize 0 esegno | segno addr esegno }

ssize
segno addr
ssize 0 esegno
segno addr esegno

Specifies stack size, location, and an extension stack segment.

Format	Operand	Meaning
Format 1	ssize	Minimum required stack size in octal 16-bit elements. If 0 is specified, the default value of '6000 is used. If **ssize** is '177774, an entire segment is reserved for the stack.
Format 2	segno	Absolute octal segment number for the stack. There must be at least '20 16-bit addresses available in this segment.
	addr	Octal starting address for the stack in the specified segment. **Addr** must be at least 4, as locations 0 to 3 must be reserved for use by stack hardware. An **addr** of at least '10 is recommended.

Format 3	**ssize**	Minimum size for the stack.
	esegno	Absolute octal number of the first segment available for the extension stack.
Format 4	**segno**	Absolute octal number of the segment in which the stack begins.
	addr	Octal starting location of the stack in the starting segment.
	esegno	Absolute octal number of the first segment available for the extension stack. SEG will allocate additional segments sequentially as needed. If an allocated segment is not needed for an extension, it is not included in the runfile.

Examples:

`SK 60000`	Set up a stack of '60000 octal 16-bit addresses.
`SK 4001 122000`	Locate the start of the stack in segment '4001 at address '122000.

▶ ST ART segno addr

Changes the ECB used to start execution (see Glossary).

segno	Absolute octal segment number.
addr	New octal address in the segment for the ECB to be start of execution.

One possible application of this command is the creation of template programs with multiple entry points (programs alike except for the start of execution location). If *START is reset to '7777 '000000 as part of template creation, SEG's loader will reset *START to the starting address of the program using the template.

This command is also used to change the start point to RESUME so that a current stack is used.

▶ WR ITE

Rewrites to disk all segments of the current runfile above segment '4000.

This command saves the runfile on disk without changing any of the segment ranges previously declared. It assures that all patches are written to the disk and that no segment ranges have been changed.

If NEW is given before WRITE, the segments are written into a new runfile; otherwise the current runfile name is used.

8 ERROR MESSAGES

OVERVIEW

Use of SEG can produce error messages from several sources:

- From SEG's own routines
- From PRIMOS
- From the subroutine ERRPR$
- As cryptic messages with no recognizable source

SEG ERROR MESSAGES

- ATTEMPT TO REFERENCE UNDEFINED COMMON

A common area is referenced in a module, but the area has not been defined or loaded. This message may also result from an internal error in a user-written compiler.

- BAD GROUP TYPE

The object file does not meet SEG's expectations. This message usually results from an internal error in a user-written compiler. Make sure that all of your program modules compile with no errors. If the message persists, call Prime Customer Engineering.

- BAD OBJECT FILE

The user is attempting to load a file that has faulty code. The file may not be an object file or it may be incorrectly compiled.

- BAD TREENAME. xxxx

The pathname given by the user is incomplete.

- BASE AREA ZERO FULL

All locations in the sector zero base area have been used. Use the AUTOMATIC command to generate base areas at regular intervals, or use the SETB command to place base areas specifically.

- BLOCK SIZE MISMATCH

This is an internal or compiler-generated error message. Be sure that all of your program modules compile with no errors. If the message persists, call Prime Customer Engineering.

- CAN'T LOAD IN SECTORED MODE

The user is attempting to load an object module compiled or assembled in a mode other than V- or I-mode. Recompile with the -64V option or reassemble in V- or I-mode.

- CAN'T LOAD IN 32R MODE

The user is attempting to load an object module compiled or assembled in 32R mode. Recompile with the -64V option or reassemble in V- or I-mode.

- CHECK SEGMENT

The user attempted to load something into segment '4035, which is reserved for SEG's symbol table. Reload in such a manner as to avoid using this segment.

- xxxxxxxxx. COMMAND_MAKE. CMDSEG

This message is returned by ERRPR$. The first part of the message consists of one of the error messages listed in Appendix D of the **Subroutines Reference Guide**.

- COMMAND ERROR

The command entered is misspelled or does not exist at this level, or the filenames and parameters following the command are incorrect.

- DEBUG GROUP ENCOUNTERED BEFORE A PROC DEF GROUP

A compiler emitted DBG information before the procedure section of the program. Be sure that all of your modules compile with no errors. If the message persists, call Prime Customer Engineering.

- **EMPTY FILE**

There is nothing in the object file.

- EXTERNAL MEMORY REFERENCE TO ILLEGAL SEGMENT

A compiler tried to load a common block into an inappropriate segment. This is an internal error. Be sure that all modules compile with no errors. If the error message persists, call Prime Customer Engineering.

- FILE NOT OPEN

SEG tried to execute a program, but could not open that file. A common source of the message is an EXECUTE command issued when there is no current or established runfile.

- ILLEGAL ADDRESSING MODE

A direct reference has been made to common areas located in another segment.

- ILLEGAL BLOCK TYPE

This is a compiler-generated error message. Be sure that all modules compile with no errors. If the message persists, call Prime Customer Engineering.

- ILLEGAL INDEX OR INDIRECT ON AN ADDRESS CONSTANT

This is usually a compiler-generated error message. Be sure that all modules compile with no errors. If the message persists, call Prime Customer Engineering.

The message may also occur if SEG was unable to find an address because the address was out of range. Most likely the program module last loaded was larger than 128K bytes. In that case, the program should be broken into subprograms and recompiled.

- xxxx: ILLEGAL SMALLER/LARGER REDEFINITION OF COMMON

Once a common block is defined with a size, it may not be redefined in another module as larger. For smaller redefinition, a warning is issued.

- IMPROPER FILE TYPE. SYMCHA.

This message usually indicates that a runfile has not been properly saved due to a control break from SEG.

- MISSING PROCEDURE END GROUP

This is a compiler-generated message. Be sure that all modules compile with no errors. If the message persists, call Prime Customer Engineering.

- MULTIPLE INDIRECT

SEG was unable to follow an address because it was out of range. Most likely the program module last loaded was larger than 128K bytes. In that case, the program should be broken into subprograms and recompiled.

The message may also be caused by a module compiled in 32R mode. It can also happen if code has accidentally been loaded into base areas as the result of a bad load command sequence.

- NEED SECTOR ZERO LINK

A link is required for address resolution, and no base areas are within reach except sector zero. This message is only generated when the SZ command has been given.

- NEGATIVE STRING LENGTH. TEMP$S

This is a compiler-generated error. The message may be displayed if a module was compiled with errors that caused an abort, and then the module was loaded anyway. Recompile. If there are no errors and the load message persists, call Prime Customer Engineering.

- NO FREE SEGMENTS TO ASSIGN

The total pool of segments available to all users has been used. The user may free some segments with the DELSEG command. If this is not successful, wait for more free segments or ask the System Administrator to put more segments in the pool.

- NO ROOM IN SEGMENT

The user is trying to create a base area that is too big for the segment. Not fatal.

- NO ROOM IN SYMBOL TABLE

SEG's symbol table is full. The user may try to reduce the number of symbols used in the runfile. Probably a new version of SEG with a bigger symbol table is required.

- NOT A SEGMENT DIRECTORY

SEG was invoked to execute a file that is not a segmented file.

- OLD OBJECT FILE

The object file was compiled or assembled in an old revision of the compiler or assembler. Recompile or reassemble and restart the load.

- OLD OBJECT MODULE — MIX FAILS

The cause and remedy are the same as for the OLD OBJECT FILE error message.

- SAVE FILE TREE NAME:

This is not an error message, but requests a pathname for the runfile.

- SEG HAS NOT CLOSED ALL THE FILES IT HAS OPENED. CLEAN

The last command entered caused such problems that SEG was not able to do all housekeeping. It may be necessary to abort the load, correct other problems, and restart.

- SEGMENT ALREADY DEFINED. SYMLIT

This message is caused by an attempt to split an assigned segment or when contiguous segments are not available for a large COMMON.

- SEGMENT WRAP AROUND TO 0

Block data in the last word of a segment may have been initialized. In this case, the relevant common block should be moved. The message may also appear if an attempt was made to load a 64R-mode program. Recompile in 64V mode.

- SMALLER REDEFINITION OF COMMON

A common block was defined in one module, and was defined as smaller later during the load. This is only a warning. It may be suppressed with the VLOAD subcommand NSCW.

- SYMBOL xxx ALREADY EXISTS

An attempt was made to define a new symbol, but the symbol is already in the symbol table. Give the symbol a new name or, if the old symbol is not needed, delete it with XPUNGE.

- SYMBOL xxx IS UNDEFINED

An attempt was made to equate two symbols, but the first symbol is not in the symbol table. Not fatal, try again.

- SYMBOL NOT FOUND

The cause is the same as for the preceding message.

- SYMBOL xxx NOT FOUND

The cause is the same as for the preceding message.

- THIS CONTROL ARGUMENT IS NOT IMPLEMENTED

The command SEG was followed by some option other than -LOAD. Reenter the command.

- UNDEFINED SEGMENT. SETSEG

Usually this message means that the SYMBOL command was used in an attempt to allocate initialized common blocks. The R/SYMBOL or A/SYMBOL command must be used instead. The message can also be caused by trying to initialize in FORTRAN a BLOCK DATA variable that has not been previously defined.

- WARNING: LOAD NOT COMPLETE

There still exist unresolved references, that is, calls to subroutines that are not in the runfile. Be sure that all user subroutines and special Prime libraries such as VAPPLB and VSRTLI have been loaded, and in the right order. A calling program must be loaded before the subroutine it calls.

- WARNING: SEG IS NOW LOADING INTO SEGMENT 4035.

  THIS SEGMENT IS RESERVED BY SEG FOR ITS SYMBOL TABLE.
  USAGE OF THIS SEGMENT FOR ANYTHING OTHER THAN UNINITIALIZED
  DATA MAY RESULT IN ERROR WHEN ATTEMPTING TO RESTORE THIS
  PROGRAM INTO MEMORY.

This error message is self-explanatory.

- WRONG FILE TYPE

LOAD was used with no object.

COMMON PRIMOS ERROR MESSAGES

- ACCESS_VIOLATION$

In general, you have attempted to read a segment to which you don't have rights. Within programs, this may be caused by an index value higher than it should be. If the program is generated by the load, one of the modules loaded may be more than 128K bytes in size. This message may also be generated if the RUNIT stack is overlaid by program linkage and data, or if reference is made to a segment that has not been shared.

- ILLEGAL_SEGNO$

The user tried to access a segment that is not allocated. The user may have tried to access a nonshared segment above the current system limit. (The default number is '4040.) Reload using only segments within the limit, or ask the System Administrator to increase the default number of user segments (NUSEG in the CONFIG file).

Another source of this message is the attempt to access a segment in the '2xxx range that has not yet been shared by the System Administrator. If your runfile includes a segment in this range, make sure that the code to share the program is included in the system build file. How to include it is explained in Chapter 4.

- LINKAGE_FAULT$

An unresolved call was encountered but the called program could not be found. Usually this means that a dynamic entry link could not be "snapped" or resolved. Be sure that all required libraries and user subroutines were loaded. If necessary, start to reload and enter MAP 3 to list unresolved calls. If a RUNIT file is being created, the main program must be named MAIN.

- NO_AVAIL_SEG$

The system has run out of available segments. Use DELSEG ALL, or contact your System Administrator.

- NO FREE SEGMENTS

Paging space is unavailable. Contact your System Administrator.

- POINTER_FAULT$

An attempt was made to transfer control to a subprogram or direct entry link, but it was not successful. Usually the problem is that not enough arguments or arguments of the wrong type were passed to the called routine.

- RESTRICTED_INST$

This message may occur if the user forgot the MIX instruction in a split load.

- STACK_OVERFLOW$

There was not enough room in the stack. See Chapter 4 on how to extend the stack.

- UNDEFINED SEGMENT

The user tried to access a segment that is not available.

MESSAGES FROM ERRPR$

Usually SEG invokes the subroutine ERRPR$ to display its error messages. This is particularly useful because many of the error codes are those produced by other subroutines called by SEG. The use of ERRPR$ and a listing of codes it interprets are explained in Appendix D of the **Subroutines Reference Guide** for Rev. 19. Most error messages include a brief message and the name of the reporting routine. These messages are usually self-explanatory. They use this format:

message 1 [message 2]   reporting routine

Example:

```
Bad key in call          (SEGPRW)
```

In this case, the user probably was using the SHARE command to write a non-segmented file, but had forgotten to use the MIX command first.

CRYPTIC MESSAGES

Sometimes the load procedure, or subsequent execution of the program, may abort with garbled displays on the screen. The cause is frequently overlaying of a stack to the extent that SEG or the operating system is not able even to return a message to that effect. In making RUNIT files, be sure that the first operand of the SPLIT command is high enough to avoid overlaying all of the linkage and data of the runfile.

The cause may also be overwriting of segment '4035. Another cause may be use of S/LO that makes a segment overflow. In this case, the data being loaded wraps around to the beginning of the segment and destroys data already there.

PART II

LOAD

9 LOADING R-MODE PROGRAMS

INTRODUCTION

The LOAD utility is used for loading object files compiled in 32R or 64R mode. Most programs compiled or assembled on Prime's 50 series of computers, however, are produced in V-mode and are loaded with the SEG utility described in the first part of this guide.

The following description emphasizes the loader commands and functions that are of most use to a programmer. For a complete description of all loader commands, including those for advanced system-level programming, see Chapter 10.

USING LOAD

The PRIMOS command LOAD transfers control to the R-mode loader, which prints a $ prompt character and awaits a loader subcommand. After executing a command successfully, the loader repeats the $ prompt character.

If an error occurs during an operation, the loader prints an error message, then the $ prompt character if the error is not fatal. Loader error messages and suggested handling techniques are discussed in Chapter 12. Most of the errors encountered are caused by large programs for which the user is not making full use of the loader capabilities.

When a system error (FILE IN USE, ILLEGAL NAME, NO RIGHT, etc.) is encountered, the loader prints this system error and returns the $ prompt symbol if the error is not fatal.

The loader remains in control until a QUIT or PAUSE subcommand returns control to PRIMOS, or an EXECUTE subcommand starts execution of the loaded program.

Load subcommands can be used in command files. Comment lines must be preceded by an asterisk (*), or they will result in a CM (command error) message.

NORMAL LOADING

Loading is normally a simple operation with only a few straightforward commands needed. The loader has many additional features to optimize runfile size or speed, perform difficult loads, and deal with possible complications. This chapter presents the most frequently used load commands and operations first to enable immediate use of the loader. Advanced features are then described. The next chapter has a summary of all loader commands.

The following commands accomplish most loading functions:

PRIMOS-Level Commands	**Function**
FILMEM	Initializes user space in preparation for load.
LOAD	Invokes loader for entry of subcommands.
RESUME	Starts execution of a loaded, saved runfile.

LOAD Subcommands	**Function**
MODE option	Sets runfile addressing mode. Default is D32R.
LOAD pathname	Loads specified object file.
LIBRARY [filename]	Loads library object files from UFD named LIB. (Default is FTNLIB.)
MAP [option]	Prints load map. Option 6 shows unresolved references.
INITIALIZE	Returns loader to starting condition in case of command errors or faulty load.
SAVE pathname	Saves loaded memory image as runfile.
QUIT or PAUSE	Returns to PRIMOS.

Most loads can be accomplished by the following procedure:

1. Use the PRIMOS FILMEM command to initialize memory.
2. Invoke LOAD.
3. Use the MODE command to set the addressing mode, if necessary. (The default is 32R mode.)
4. Use the loader's LOAD subcommand to load the object file and any separately compiled subroutines. If the object filename is of the format **program.BIN**, only the part of the name before the suffix BIN need be entered.
5. Use loader's LIBRARY subcommand to load subroutines called from libraries (the default is FTNLIB in the UFD LIB). Other libraries, such as SRTLIB or APPLIB, must be named explicitly.
6. If the message LOAD COMPLETE is not displayed, enter a MAP 6 to identify the unsatisfied references, and load them. If no LOAD COMPLETE is displayed the second time, enter LI again.
7. SAVE the runfile. If no filename is specified, the runfile is automatically named **program.SAVE**.
8. Enter QUIT to return to PRIMOS, or EXECUTE to execute the program and return to PRIMOS.

If Step 6 produces a LOAD COMPLETE message, then loading was accomplished. If there is a problem, it will become apparent by the absence of a LOAD COMPLETE message or by a loader error message. (See Chapter 12 for a complete list of all loader error messages and their probable cause and correction.)

After a successful load, you can either start runfile execution from LOAD command level, or exit from the loader and start execution through the PRIMOS command RESUME. The following example assumes that the source program SYM.FTN was compiled in 32R mode to produce the object file SYM.BIN.

```
OK  LOAD
$ LOAD SYM
$ LI
LOAD COMPLETE
$ SAVE
$ QUIT
```

Order of Loading

The following loading order is recommended:

- Main program.
- Separately compiled user subroutines (preferably in order of frequency of use).
- Other Prime libraries (LI filename).
- System FORTRAN library (LI).

Loading Library Subroutines

Standard FORTRAN mathematical and input/output functions are implemented by subroutines in the library file FTNLIB in the UFD named LIB. The appropriate subroutines from this file are loaded by the LIBRARY command given without a filename argument. If subroutines from other libraries are used, such as MATHLB, SRTLIB, or APPLIB, additional LIBRARY commands are required that include the desired library as an argument.

Load Maps

During loading, the loader collects information about the results of the load process, which can be printed at the terminal (or written to a file) by the MAP command:

```
MAP [pathname] [option]
```

The information in the map can be consulted to diagnose problems in loading, or to optimize placement of modules, linkage areas, and common blocks in complex loads.

Load information is printed in four sections, as shown in Chapter 11. The amount of information printed is controlled by MAP option codes:

Option	Load Map Information
None, 0, or 4	Load state, base area, and symbol storage; symbols sorted by address.
1	Load state only.
2	Load state and base areas.
3	Unsatisfied references, sorted by address.
6	Unsatisfied references, sorted in alphabetical order.
7	Full map, symbols sorted in alphabetic order.
10	Special symbol map for PSD (to a file).

ADVANCED LOADING TECHNIQUES

When standard loading goes well, the user can ignore most of the loader's advanced features. However, situations can arise where some detailed knowledge of the loader's tasks can optimize size or performance of a runfile, or even make a critical load possible. From that viewpoint, the main tasks of the loader are:

- Convert block-format object code into a runtime version of the program (executable machine instructions, binary data, and data blocks).
- Resolve address linkages (symbolic names of variables, subroutine entry points, data items, etc.) into appropriate binary address values.
- Perform address resolution.
- Detect and flag errors such as unresolved external references, memory overflow, etc.
- Build (and, on request, print) a load map. The map may also be written to a file.
- Reserve common blocks as specified by object modules.
- Keep track of runfile's hardware execution requirements and make user aware of a need to load subroutines from the Unimplemented Instruction Interrupt (UII) library.

Virtual Loading

The loader occupies the upper 64K bytes of the user's 128K-byte virtual address space. Programs up to 64K bytes are loaded directly into the memory locations from which they execute. Programs loaded in this manner can be started by the loader's EXECUTE command without being saved. For larger 64R-mode programs, the loader uses the available memory as buffer space and transfers loaded pages of memory to a temporary file that accomodates a full 128K-byte memory image. When loading is complete, the file must be assigned a name by the loader's SAVE command; it can then be executed either through the loader's EXECUTE command or the PRIMOS command RESUME.

The loader remains attached to the working directory throughout loading for access to the temporary file. Files in other directories can be loaded by giving their pathname in a LOAD command.

Object Code

Inputs to the loader are in the form of object code — a symbolic, block-format file generated by all of Prime's language translators. Prime's system library files consist of subroutines in this format.

The loader combines the user's main program object file with the object files of all referenced subroutines into a single runfile. The runfile is then ready for execution, either directly through the loader's EXECUTE command or through the PRIMOS command RESUME.

Runfiles

A runfile consists of a header block followed by the runfile text in memory image format. The header contains information that enables the runfile to be brought into memory by the PRIMOS command RESTORE or RESUME. Contents of the header can be examined after a RESTORE by the PM command. (See the **PRIMOS Commands Reference Guide**.)

Selecting the Addressing Mode

The 32R addressing mode is retained as the loader's default for compatibility with existing command files. The only significant difference between 32R and 64R for small programs is that 32R permits multiple indirect links, while 64R allows only one level of indirection. In certain situations such as processing of multi-dimensional arrays, 32R mode may enable the compiler to produce a runfile that is somewhat more compact or runs slightly faster. However, for programs that approach the 64K byte boundary, 64R mode ensures successful loading with no significant penalties of size or speed. Thus MODE D64R is recommended for most applications.

Address Resolution (Base Areas)

The loader resolves out-of-range address references. These arise because 16-bit memory reference instructions cannot directly reference all of memory. The loader compensates for this by generating a pointer to the operand and then modifying the instruction to reference through the pointer. These pointers are located in reserved areas called base areas.

The default base area is from memory location '200 to '777. For many programs, this area will be sufficient. However, for larger programs this area might be inadequate. The loader has a number of commands to enlarge the default base area and create local base areas.

The base area below location '1000 can be used to resolve any instruction, no matter what its location. Local base areas (above location '1000) can be used only to resolve instructions in a window around the local base area. The window extends approximately '400 locations above and below the base area.

The loader uses local base areas automatically when possible in preference to the base area below location '1000. Base area locations are not available for any other use during program loading or execution.

The following error messages indicate a condition usually encountered when loading large programs:

```
BASE SECTOR 0 FULL

symbolname XXXXXX NEED SECTOR 0 LINK
```

This condition can be avoided in several ways:

- Use the SETBASE command to use addresses '100 – '777. Do not set the lower bound below '100.
- Give the AUTOMATIC command to enable the loader to assign local linkage areas before and after individual subroutines.
- Use setbase parameters with a LOAD or LIBRARY command to insert local linkage areas where they are needed.
- Use the SETBASE command to designate a base area where it is required.
- During compilation of FORTRAN modules, use the -DEBASE option. During assemblies use the SETB pseudo-operation.

Locating Common Blocks

By default, the loader sets the high end of common blocks at '077777 (the 64K-byte boundary) and allocates common area downward from there. If a PROGRAM-COMMON OVERLAP message occurs, common blocks can be moved higher by the COMMON or DC (Defer Common) subcommands. DC is recommended. (If DC is used, a LOAD COMPLETE message will not occur until a SAVE or EXECUTE command is given.)

Unimplemented Instruction Interrupt (UII) Handling

The loader can keep track of the CPU hardware required to execute the instructions generated by the modules already loaded. This is shown in the UII entry in the load-state section of a load map. The codes are:

UII Value	CPU Required
100	Prime 450 and up
57	Prime 350 or 400
17	Prime 300 with floating-point hardware
3	Prime 300
1	Prime 100 or 200 with HSA
0	Prime 100 or 200

If the UII code on the load map is greater than the value for the target CPU specified by the HARDWARE command then it will be necessary to load part of the UII library to make execution possible. When a CPU encounters an instruction not implemented by hardware, a UII occurs and control is transferred to the appropriate UII routine. This routine simulates the missing hardware with software routines.

However, the UII routine must be loaded by the command LI UII. This should be the last LOAD command before the program is saved. The appropriate routines will be selected from this library to satisfy the additional hardware requirements of the program.

To make sure that only the required subroutines are loaded, the user can "subtract" hardware features that are present in the CPU by entering a HARDWARE command. For example, assume a load session produces a load map UII value of 17. The target CPU is a Prime 300 (UII value 3). The command:

```
HA 3
```

reduces the load state UII value to 14 and ensures that the high-speed arithmetic subroutines do not occupy space in the runfile.

If, after a HARDWARE command, the load state UII value is 0, the UII library need not be loaded.

System Programming Features

The following commands are primarily of interest to assembly language and systems programmers. They are described in more detail in Chapter 10.

Command	Meaning
F/	Prefix to LOAD and LIBRARY that forceloads unreferenced modules.
P/	Prefix to LOAD and LIBRARY that starts loading on next page boundary. (Can reduce paging time).
PB	Program break. Resume loading at a new location.
CH, SS, SY, XP	Symbol control commands.
EN	Command to save a copy of entire load session, for building of program overlays.
ER	Command to control action taken by loader following errors.
SZ	Command to control use of sector 0.

10 LOAD COMMANDS

Following is a summary of all LOAD commands, in alphabetical order. All file names may be specified by pathnames. All numerical values must be octal.

▶ ATTACH [ufdname [password]] [ldisk [key]]

Attaches to specified directory. This command is obsolete with Rev. 18, as LOAD accepts full pathnames for files in other directories.

▶ AUTOMATIC base-length

Instructs the loader to insert linkage area automatically, helping to reduce the number of loads that use sector 0 link space.

Whenever the loader detects the end of a routine, and more than '300 (octal) locations have been loaded since the last base area was inserted, the loader inserts a base area of the size specified by **base-length** (octal).

The value of base-length may be changed between load files. This automatic feature may be turned off with an **AU 0** command.

▶ CHECK [symbol-name] [param-1]...[param-9]

Checks the value of the current PBRK against the value of a symbol or number. (**PBRK** is the current load point, or address at which the next module will automatically be loaded.) This is useful when it is necessary to load modules out of order and below previously loaded information, or when a module should not exceed a certain size.

symbol-name is an optional symbol that must be defined in the symbol table. **param-1** through **param-9** are octal parameters that are summed to form either an address or an offset from symbol-name. Each number may be preceded by a dash, -, in which case it will be negated.

For example if the value (location) of OVRFLW is '17777 and PBRK is '20010, then:

```
CH OVRFLW
```

will report '000011 WDS OVERLAP'. If on the other hand PBRK was '17770, then CHECK would report '000007 WDS TO SPARE'.

Similarly, **CH 50000 - 20** compares PBRK with '47760, and **CH OVRFLW 10000** compares PBRK with '27777.

▶ COMMON address

Moves the top or starting location of FORTRAN-compatible common blocks to the octal **address** specified. Space for common items is allocated downward from, but not including, the starting location. The top of common is the last location used for common by the loader. The default load address for common blocks is '077777.

To specify a common load point, give the location desired plus 1. For example, **CO 0** specifies '177777 as the top location in common.

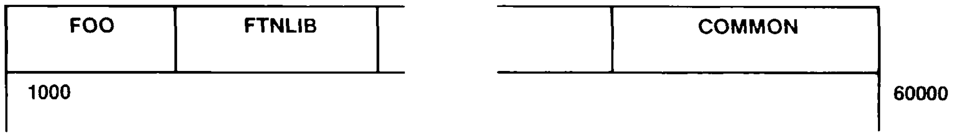
Note

At Rev. 16 and above, the DC (Defer common) command is the preferred method of moving common. The COMMON command is retained for compatibility with previous releases of LOAD.

▶ DC [END]

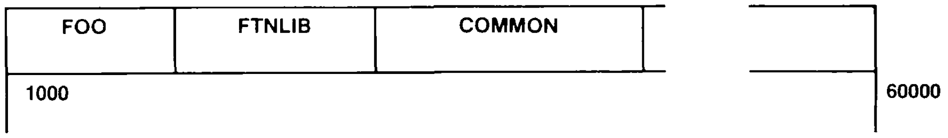
Defers definition of common blocks until convenient or until a SAVE or EXECUTE command is given.

A DC command without the END option enables deferred common, so that common is created immediately following the loaded program instead of being defined as "down from CMHIGH". For example, if a user has loaded FOO and the library with LOAD, memory allocation looks like Figure 10-1.



Common Blocks Loaded Without the DC Command
Figure 10-1

If the program were loaded under the DC option, the memory allocation could be represented as Figure 10-2.



Common Blocks Loaded With the DC Command
Figure 10-2

Common blocks are defined (allocated) as follows:

- When the user gives the **DC END** command, all common area is defined.
- When the loaded program attempts to initialize a common block, that block is defined.
- When a PMA module creates a common block, it is defined.
- When a SAVE or EXECUTE command is given, all common area is defined.

▶ EN pathname

Saves the entire state of the loader, complete with the temporary file. The current running copy of the loader is saved as a runfile, together with all buffers and data bases. The current copy of the temporary file is copied into a new temporary file and the original is closed. (The new temporary file also has a name of the form T$XXXX.) The **pathname** is the name of the saved copy of the loader.

The EN command is provided as a convenience in overlay building, once the main program and common blocks have been created. The current copy of the loader and the new temporary file are used to create the first overlay. Subsequent overlays are created using the saved copy.

A commented example of the use of EN follows. The example creates a new file from object files named MAIN.BIN and OVER1.BIN.

```
OK, LOAD
$ LO MAIN 10000        Load main program (MAIN.BIN) above overlays.
$ LI                   Satisfy its library references.
$ SAVE                 Save it as MAIN.SAVE.
$ EN MAINLDR           Create saved copy of loader.
$ LO OVER1 1000        Build first overlay.
$ LI
$ SAVE                 Save it as OVER1.SAVE.
$ QUIT

OK, R MAINLDR
$ EN MAINLDR           Preserve state of temporary file.
$ LO B_OVER2 1000      Create second overlay.
$ LI
$ SAVE OVER2.SAVE      Save it with new filename convention.
$ QUIT

OK, R MAINLDR
$ LO B_OVER3 1000      Only three overlays.
$ LI
$ SAVE OVER3.SAVE
$ QUIT

OK,
```

In the example above, overlays are sandwiched between the main program and its links to avoid sector zero link conflicts. When the second of the three overlays is to be created, an EN command is the first loader command given. This preserves the "incoming" temporary file for use in building the next overlay.

The save range for the overlays includes all locations loaded up to the time of the save. The save ranges can be changed using the RR command prior to the EN command in order to decrease the size of the runfile.

▶ ER error-code

Tells the loader what action to take when an error occurs — in particular, errors of the multiple indirect or sector-zero overflow class.

error-code	Action
1	"Multiple Indirect" errors are noted on the terminal, but loading continues. All other errors terminate loading of the offending object file, and LOAD returns to its command level. (Default)
0	Errors generated by the SZ command and multiple indirect errors are treated as above. All other errors abort loading of the object file.
2	All errors cause a return to PRIMOS.

▶ **EXECUTE [a] [b] [x]**

Starts execution of the loaded program with optional values preset into the A, B, and X registers. Execution starts at the location specified by the *START entry of the load map.

All 32R, 16S, and 32S programs can be executed directly, without being saved. However, if an EXECUTE is attempted on a 64R-mode program that contains information outside of the virtual loader buffer space, LOAD will respond with "CAN'T - PLEASE SAVE". Once the image has been saved, it may be executed.

EXECUTE closes and deletes the temporary file if it has been saved. If not, the file remains open and loading can continue.

▶ **F/ { LIBRARY | LOAD } pathname [parameters]**

Forceloads all of the modules in an object file regardless of whether they have been referenced and regardless of whether the load is complete. **Parameters** are the same as for LOAD.

▶ **HARDWARE definition**

Defines the hardware available in the CPU on which the load module is intended to execute. The **definition** parameter is an octal code with the following values:

CPU	Definition
P450 and up	100
P350,P400	57
P300/FP	17
P300	3
P200/HSA	1
P100/HSA	1
P200	0
P100	0

(FP is floating-point hardware and HSA is High Speed Arithmetic.)

The definition value is subtracted from any UII requirements accumulated during loading. If the UII value on a subsequent load map is 0, the UII library need not be loaded.

See the discussion of UII handling in Chapter 9 for more information.

▶ INITIALIZE [pathname] [parameters]

Initializes the loader and then optionally performs the same functions as a LOAD command without returning to PRIMOS level. In the loader's initialized state, the load state parameters return to their initial values. If no filename is provided, the loader repeats its prompt character ($). **Parameters** are the same as for LOAD.

▶ LIBRARY [filename] [loadpoint]

Attaches to the UFD named LIB, loads from the specified library (**filename**), and then re-attaches to the original UFD.

Filename is the name of a specific library in LIB. The default is FTNLIB. **Loadpoint** is the address where loading is to start. The default is *PBRK.

The LIBRARY command accepts the P/ and F/ prefixes.

▶ LOAD [pathname] [parameters]

Loads the object file specified by **pathname**. The arguments may be entered in three formats:

1. loadpoint [setbase-1]...[setbase-8]
2. * [setbase-1]...[setbase-9]
3. symbol [setbase-1]...[setbase-9]

In form 1, **loadpoint** is the starting location of the load. In form 2, the load starts at the current PBRK location (*). In form 3, the load address can be stated symbolically (**symbol**).

The **parameters** (**setbase-1**, etc.) are octal values that specify the size of linkage areas to be inserted before and after modules during loading. If the last parameter is '177777, the loader requests more setbase values.

As an example of linkage area placement, assume that there are four modules (subroutines) in an object file named SUB4.BIN; the command:

```
LO SUB4 2000 10 20 0 40 50
```

causes loading to begin at '2000, where a base area of length '10 is created before the first routine, one of length '20 after the first routine, none after the second routine, one of length '40 after the third routine, and one of length '50 after the fourth routine.

In form 2, the numeric parameters are interpreted as base area lengths as above. For example:

```
LO B_BIG * 30 30
```

starts loading at the current PBRK value and places base areas of length '30 before and after the first routine in object file B_BIG.

In form 3, if the symbol FSTEND has the value (location) '10000, then:

```
LO B_MIDDLE FSTEND
```

causes LOAD to begin loading at '10000. The only requirement is that FSTEND be a defined symbol. This can be accomplished either by the SYMBOL command or through symbol definition from an object module. As in the other two forms of the command, any numeric parameters are interpreted as base area lengths.

If the load arguments are not assigned by the command string, the following default values apply:

Loadpoint	*PBRK (initially '1000)
Base-start	'200
Base-range	'600

If all of the symbols in the load module have been previously defined, the loader skips the module. A load module is defined to terminate with an "END" statement. To force loading of a module that contains only previously defined symbols, use the F/ (forced load) prefix with the LOAD command.

▶ MAP [pathname] [option]

Generates a load state map on the terminal, or in a file if **pathname** is specified.

Option	Meaning
0	Load state, base area, symbol storage map; symbols sorted by address (Default)
1	Load state only
2	Load state and base area
3	Unsatisfied references, sorted by address
4	Same as 0
5	Reserved
6	Unsatisfied references, sorted alphabetically
7	Full map, symbols sorted alphabetically
10	Special symbol map for PSD (to a file)

Map formats and conventions are described in detail in Chapter 11.

▶ MODE $\left\{ \begin{array}{l} \text{D64V} \\ \text{D32R} \\ \text{D64R} \\ \text{D16S} \\ \text{D32S} \end{array} \right\}$

Specifies address resolution mode for next load module (32K relative, or D32R, is the default). If used, MODE must precede other LOAD commands. The mode set by this command may be overridden by mode control statements in object files.

▶ PAUSE

Returns to PRIMOS for execution of an internal command (such as LISTF). The loader's temporary file is left open and the load state is preserved, so that loading can continue after a PRIMOS START command.

Note

QUIT also returns to PRIMOS, but the temporary file is closed and deleted, so loading cannot continue.

▶ PBRK $\left\{\begin{array}{ll}\textbf{[symbol-name]} & \textbf{[offset-1]...[offset-9]}\\ \textbf{[*]offset-1} & \textbf{[offset-2]...[offset-9]}\end{array}\right\}$

Sets PBRK (see Glossary) to the value of a symbol or an octal value, with optional offset values. Like the CHECK command, it is intended to facilitate complicated loads.

Symbol-name, if present, must be a defined symbol. **Offset-1**, etc., are octal parameters that will be summed. If **symbol-name** is present, the sum of the numbers is treated as an offset from the specified symbol. If the asterisk (*) is used, the sum is treated as an offset from the current PBRK. If neither is present, the sum is the actual value to which PBRK is set. For example:

```
PB 10000 10
```

moves the value of PBRK to '10010. Similarly, if OLDEND is a symbol with the value '17456, then:

```
PB OLDEND 10
```

sets PBRK to '17466.

If PBRK is currently '1000, then:

```
PB * 10000 -77
```

sets PBRK to '10701.

▶ P/ $\left\{\begin{array}{l}\textbf{LIBRARY}\\ \textbf{LOAD}\end{array}\right\}$ **[pathname] [parameters]**

Begins loading at the next page boundary. See LOAD for the meaning of the other arguments.

The P/ and F/ prefixes may be concatenated, as in:

```
P/F/LO MODULE
```

▶ QUIT

Deletes the temporary file, closes the map file (if the loader opened it), and returns to PRIMOS. The user is reattached to the home UFD. (Also see PAUSE.)

▶ RR [start-addr] [end-addr]

Resets the runfile save range. The **start-addr** default is -1 and the **end-addr** default is 0. For example:

```
OK, FILMEM ALL

OK, LOAD
$ LO POO 11000     /* LOADS POO.BIN
$ LI
LOAD COMPLETE
$ MA 2
*START  011000    *LOW    000200    HIGH   020144   *PBRK 020145
*CMLOW  077777    *CMHGH 077777    *SYM   000062   *UII  000003

*BASE  000200  000231  000777  000777
*BASE  011524  011571  011574  011575
*BASE  012574  012655  012656  012657
*BASE  013655  013710  013716  013716
*BASE  014701  014722  014724  014724

$ VI 200 10
$ RR 10200 20144
$ SA               /* SAVES POO.SAVE
$ Q

OK, CO TTY
OK, RESTORE POO.SAVE
OK, PM
SA,EA,P,A,B,X,K=
10200 20144 11000 0 0 0 6 00
```

▶ SAVE pathname [a] [b] [x]

Saves the loaded memory image from *LOW to *HIGH, including all initialized common areas, under the name **pathname**. At this time, the user has the option to store new values in the A, B, and X registers. Also saved with the program are the low, high, start, and keys parameters obtained from the loader. The RR command can be used before the SAVE command to store new low and high values.

▶ SETBASE $\begin{bmatrix} \textbf{base-start} & \textbf{[base-range]} \\ * & \textbf{base-range} \end{bmatrix}$

Defines a base area that begins at **base-start** and includes the number of locations specified by **base-range**. If the range is not specified, the end of the area is location '777 of the sector containing the base-start location. Multiple base areas are allowed. A command to create a base area that overlaps a previously defined area is ignored. The default values are:

Base-start	'200
Base-range	'600

Base-start can be set at the current location by using the asterisk (*). Thus, if PBRK (**base-start**) is '1765, the command **SETBASE * 20** creates a set-base area of length 20 at '1765 and PBRK is set at '2005 after the command has been executed.

The user may wish to increase the size of the sector zero base area by the command **SE 100** at the start of a load session. The beginning of the sector zero base area should not be made lower than '100.

▶ SS **symbol-name**

Specifies a symbol name that will not be deleted by the XPUNGE command. **Symbol-name** must be defined in the symbol table. All symbols thus referenced are not deleted if the symbol table is expunged.

▶ SYMBOL **symbol-name** $\left\{ \begin{array}{l} \textbf{old-name [offset-1]...[offset-9]} \\ \textbf{address [offset-2]...[offset-9]} \\ \textbf{* offset-1 [offset-2]...[offset-9]} \end{array} \right\}$

Establishes locations in the memory map for common blocks or to provide relocation points for the course of the load. They may also be used to satisfy references.

Symbols follow the same rules as filenames, but are restricted to eight characters.

The first form allows the user to equate two symbols or to equate the new symbol to an offset from the old. For example:

```
    SY SNAME OLDSYM
```

equates SNAME to OLDSYM, which must be a defined symbol in the symbol table.

The second form allows the user to equate a symbol to an octal value. For example:

```
    SY SNAME 1300 20 - 7 10
```

sets SNAME to '1321

The third form of the command permits the user to equate the symbol to the value of the current PBRK plus the sum of the numeric parameters. For example, the sequence of commands:

```
    SY OVRFLW *
```

```
LO TEST   * 10 20
CH OVRFLW 10 567 20
```

allows the user to be sure that the module TEST.BIN does not occupy more that '567 locations.

▶ **SZ** $\begin{bmatrix} \textbf{YES} \\ \textbf{NO} \bullet \end{bmatrix}$

Prevents routines from using sector zero links.

If **SZ** or **SZ NO** is given, LOAD will not put links in sector zero. Instead, it will flag any attempt to do so, giving the location of the instruction attempting to link. This will normally terminate the loading of the object file. However, if an **ERROR 0** command has been given, loading will continue and thus will generate a list of sector zero link attempts. A sector zero base area will be created, but no links will be put into it while the **SZ NO** is in effect.

If **SZ YES** is given, linking is again permitted in sector zero.

▶ VI**RTUALBASE base-start to-sector**

Copies the base sector (from the **base-start** location to the end) to the corresponding locations of **to-sector**. This command is intended for use in building RTOS modules using dedicated sector zero or base sector relocation.

▶ XP**UNGE d-symbols d-base**

Controls the deletion of symbols and base areas.

d-symbols An octal value that controls the deletion of symbols:

0	Deletes all but undefined symbols.
1	Deletes all symbols except undefined symbols and symbols for common areas.

d-base An octal value that controls the deletion of base areas:

0	Deletes all defined base areas from the symbol table.
1	Deletes all defined base areas except sector zero.
2	Retains all defined base areas.

▶ * ***comments***

Comments may be included in a command file when an asterisk and a blank or two blanks precede the comment. The rest of the line is not processed by the loader.

11 R-MODE LOAD MAPS

This chapter describes the sections of an R-Mode load map. See Figures 11-1 and 11-2 for examples of a full map.

Load State

The load state area shows where the program has been loaded, the start-of-execution location, the area occupied by common blocks, the size of the symbol table, and the UII status. All locations are octal numbers.

*START	The location at which execution of the loaded program will begin. The default is '1000.
*LOW	The lowest memory image location occupied by the program. Executable code normally starts at '1000, but sector 0 address links (if any) normally begin at '200.
*HIGH	The highest memory image location occupied by the program (excluding any area reserved for uninitialized common blocks).
*PBRK	"Program Break": The next available location for loading. It normally is the location following the last loaded module, but can be moved by PBRK or the LOAD family of commands.

```
*START  001000    *LOW    001000    *HIGH   001020    *PBRK   001021
*CMLOW  077777    *CMHGH  077777    *SYM    000004    *UII    000000


  MAIN  001000  EXIT  001016

  COMMON BLOCKS

  LIST   000001
```

Full Map (MA)
Figure 11-1

```
*START  001000    *LOW    001000    *HIGH   001020    *PBRK   001021
*CMLOW  077777    *CMHGH  077777    *SYM    000004    *UII    000000


  EXIT  001016  MAIN  001000

  COMMON BLOCKS

  LIST   000001
```

Full Map, Symbols Sorted Alphabetically (MA 7)
Figure 11-2

*CMLOW	The low end of the common area.
*CMHGH	The top of the common area.
*SYM	The number of symbols in the loader's symbol table. This is usually of no concern unless the symbol space crowds out the last remaining runfile buffer area. (There is room for about 4000 symbols before crowding is a risk.)
*UII	A code representing the hardware required to execute the instructions in loaded modules. Codes and other information are described in Chapter 9.

Base Areas

The base area map includes the lowest, highest and next available locations for all defined base areas. Each line contains four addresses as follows:

```
*BASE     AAAAAA     BBBBBB     CCCCCC     DDDDDD
```

AAAAAA	Lowest location defined for this area
BBBBBB	Next available location if starting up from AAAAAA
CCCCCC	Next available location if starting down from DDDDDD
DDDDDD	Highest location defined for this area

Symbol Storage

The symbol storage listing consists of every defined label or external reference name, printed four per line in the following format:

```
  namexxxx NNNNNN
```

or

```
**namexxxx NNNNNN
```

NNNNNN is a six-digit octal address. The ** flag means the reference is unsatisfied (has not been loaded).

Symbols are listed by ascending address (default) or in alphabetical order (MAP 0 or MAP 7). The list may be restricted to unsatisfied references only (MAP 3 or MAP 6). Since the symbol table is already in alphabetical order, MAP 6 or MAP 7 is faster. Figure 11-2 shows a map created with MAP 7.

Common Blocks

The low end and size of each common area are listed, along with the name (if any). Every map includes a reference to the special common block, LIST, defined as starting at location 1. This is provided to allow FORTRAN programs to address any location in the segment easily, using the construct:

```
INTEGER*2 LOCNO (1)
COMMON/LIST/LOCNO
```

Reference to LOCNO(N) refers to the location identified by the value of N.

12 LOAD ERROR MESSAGES

The following are the LOAD error messages:

- ALREADY EXISTS !

An attempt is being made to define a new symbol; however, the symbol name is already a defined symbol in the symbol table. Not fatal; try again.

- ATTEMPT TO LOAD IN LOCS. 0-10

The program has loaded into the last location in memory and has wrapped around to load in location 0. The program size must be decreased. Alternatively, compile in 64V mode and use SEG. Fatal; abort load.

- BAD OBJECT FILE

The object text is not recognizable. This usually occurs when an attempt is made to load source code or when the object text was compiled or assembled for segmented loading. Fatal; abort load.

- BASE SECTOR 0 FULL

All locations in the sector zero base area have been used. Use the AU command to generate base areas at regular intervals, or use the SETB or LOAD commands to place base areas specifically. Fatal; abort load.

- sname xxxxxx CAN'T DEFER COMMON, OLD OBJECT TEXT

The DC command has been given and a module created with a pre-Rev. 14 compiler or assembler has been encountered. It is not possible to defer common block **sname** at load point **xxxxxx** in this case. The module must be recreated with the current compiler or assembler. Fatal; abort load.

- CAN'T - PLEASE SAVE

The EXECUTE command has been given for a runfile that has required virtual loading. Not fatal. Save the runfile and give the EXECUTE command.

- CM$

Command line error. Unrecognized command given. Not fatal; try again.

- COMMON OUT OF REACH

COMMON above '100000 is out of reach of the current load mode (16S, 32S, or 32R). Use the MODE command to set the load mode to 64R. Fatal; abort load.

- sname xxxxxx COMMON TOO LARGE

Definition of common block **sname** at load point **xxxxxx** causes common to wrap around through zero. Moving the top of common — with the COMMON command — may help. Fatal; abort load.

- D PREFIX MISSING

For the address mode specification, the first character must be a D. Not fatal; try again.

- WARNING! sname xxxxxx ENTRY/COMMON NAME CONFLICT

The load will continue without being affected. However, if the entry is to a subroutine, the user probably inadvertently used the name of a library routine for a user-named common block.

- sname xxxxxx ILLEGAL COMMON REDEFINITION

An attempt is being made to redefine common block **sname** at load point **xxxxxx** to a longer length. The user's program should be examined for consistent common definitions. At the very least the longest definition for a common block should be first, or common definition could be deferred. Fatal; abort load.

- 64R LOAD MODE INVALID

A module compiled or assembled to run in only 32K of memory is being loaded in 64R mode. Recompile or reassemble or change the load mode with the loader's MODE command. Fatal; abort load.

- xxxxxx MULTIPLE INDIRECT

A module loading in 64R mode requires a second level of indirection at location **xxxxxx**. This message usually results when an attempt is made to load code compiled or assembled for 32R mode in 64R mode. It can also happen if code has accidentally been loaded into base areas as the result of a bad load command sequence. This results in a link without an address. The load continues.

- sname xxxxxx NEED SECTOR ZERO LINK

At location **xxxxxx** a link is required for address resolution. No base areas are within reach except sector zero. The last referenced symbol was **sname**. This message is only generated when the SZ command has been given. **Sname** may be the name of a common block, the name of the routine to which the link should be made, or the name of the module being loaded. Fatal; abort load.

- NO POST BASE AREA, OLD OBJECT TEXT

A post base area has been specified for a module that was created with a pre-Rev. 14 compiler or assembler. No base area is created. Recreate the object text with the current compiler or assembler. Fatal; abort load.

- NO ROOM IN SYMBOL TABLE

The symbol table has been expanded to its limit. Reduce the number of symbols in the load. Fatal; abort load.

- PROGRAM-COMMON OVERLAP

The module being loaded is attempting to load code into an area reserved for common blocks. Use the loader's COMMON command to move common up higher. Fatal; abort load.

- sname xxxxxx REFERENCE TO UNDEFINED COMMON

An attempt is being made to link to common block **sname**, which has not been defined at load point **xxxxxx**. This usually happens to users creating their own translators. Fatal; abort load.

- SECTORED LOAD MODE INVALID

A module compiled or assembled to load in R mode has been loaded in S mode. Use the MODE command to reset the load mode. It might be a good idea to be sure that all modules are correctly written, since the default load mode is 32R. Fatal; abort load.

- SYMBOL NOT FOUND

An attempt is being made to equate two symbols with the SYMBOL command and the old symbol does not exist. Not fatal; try again.

- SYMBOL UNDEFINED

An attempt is being made to equate two symbols; however, the old symbol is an undefined symbol in the symbol table. Not fatal; try again.

APPENDIXES

A SEG'S FUNCTIONS

INTRODUCTION

This Appendix describes each of SEG's functions, expanding upon the list in Chapter 1.

Produce Optimized Runfiles

Runfiles can be optimized in the following ways.

- If you don't want the default load of instructions and data into separate segments, you can use the MIX command of SEG to create runfiles with no division of function. These files are usually smaller than runfiles created with the default loading method. The process is described in Chapter 4.
- An optimized runfile may have a smaller execution unit than SEG. SEG includes a small execution unit called RUNIT. This unit can be loaded into a runfile, and the whole file can occupy segment '4000 instead of putting SEG in '4000 and the runfile in other segments. Execution is faster, and allows execution with RESUME as well as execution of the program as a PRIMOS command. RUNIT is loaded with the SPLIT command of VLOAD, and its uses are discussed in Chapter 4. The resulting runfiles are sometimes called **single-segment runfiles**, **sequential runfiles**, or **R-mode-like runfiles**, but none of these names accurately describes all files created by this method. This manual calls them **RUNIT files**. These files may also be run as PRIMOS commands.

- Optimization can include relocating data and symbols in the runfile. Certain blocks of data, called common blocks, are put by the loader in separate segments from those used for other program data. You can use the COMMON, SYMBOL, SS, A/, S/, and R/ subcommands of VLOAD to relocate that data more efficiently. The commands are described in Chapter 6. Examples of COMMON, SYMBOL, and S/ are in Chapter 4.
- An object file can be loaded into a specified segment. Use the S/ subcommand of VLOAD, described in Chapter 6. Examples are in Chapter 4.
- Data or a procedure can be loaded on a page boundary to reduce search time. Use the P/ subcommand of VLOAD, described in Chapter 5.

Control Base Area Allocation

The **base area** is that part of a procedure segment used for reference in indirect addressing instructions. It is normally at the beginning of each procedure segment, and the user need not be concerned with it unless SEG returns the message SECTOR ZERO BASE AREA FULL. To allow more space or rearrange it, use the AUTOMATIC, SETBASE, and SZ commands in Chapter 5. To delete base area information, use XPUNGE.

Prepare Shared Programs

If you have a program to be used concurrently by several users, you can share it with your System Administrator's authorization. The SHARE or SINGLE command prepares a runfile for sharing. See Chapter 4 for a description of sharing, and Chapter 5 for lists of these commands.

Change Stack Space and Location

If the stack runs out of space (PRIMOS error message STACK_OVF$), or if too much space is allocated for it in a small program, you can change its size or location with the SK subcommand of MODIFY, or change its location with the SPLIT command of SEG. You can change minimum stack size with the STACK subcommand of VLOAD. SK and SPLIT can also specify extension segments. Chapters 6 and 7 list these commands; Chapter 4 gives examples.

Produce Load Maps

A **load map** is a list of the segments being used by a particular runfile, with the address within each segment of the procedures and data sections that were loaded. SEG allows different map options, including one that lists only unresolved procedure calls (references) and two that list only symbols. The load map is created by the MAP command of SEG, and by the MAP subcommand of VLOAD. Chapter 3 discusses map creation and map reading in detail.

Make Templates

A template is a group of routines that you plan to use with several application programs. How to make and share templates is discussed in Chapter 4. (The EDB utility discussed in the **Subroutines Reference Guide** may also be used if sharing is not necessary.) The template procedure can include:

- Forced loading of all routines in a binary file. This process, initiated by the subcommand F/ of VLOAD, is discussed in Chapter 6.
- Loading IFTNLB, PFTNLB, and SPLLIB where desired for reentrancy of templates listed in Chapter 6. Use the IL and PL subcommands of VLOAD.

Execute Runfiles

The command **SEG runfilename** starts execution of a default runfile previously created. To execute a runfile just after creating or modifying it, use the EXEC command within VLOAD. For runfiles containing RUNIT discussed above, use the PRIMOS command **RESUME runfilename**. See Chapter 5.

Get Help

The HELP command of SEG (Chapter 5) causes the screen to display a digest of available commands.

Debug Runfiles

Various debugging procedures are available through SEG:

- Invoke VPSD for debugging. VPSD is a symbolic debugger described in the **Assembly Language Programmer's Guide**. It may be invoked with the PSD command of SEG (Chapter 5).
- Check for unresolved references. MAP 6 or MAP 3 show these. Examples are shown in Chapters 2 and 3.
- Restart a load after an error, overlaying any work already done. Use the INITIALIZE subcommand of VLOAD (Chapter 6).
- Restart a program at a certain address with the PRIMOS command START after having interrupted it with CONTROL-P (BREAK). How to do this is discussed in the subsection on RUNIT in Chapter 4.

Build Runfiles

- Load an object file into a runfile. Use the LOAD subcommand of VLOAD.
- Load library files. Library files needed for COBOL or other compilers are stored in the UFD named LIB. The LIBRARY subcommand of VLOAD loads these libraries if you give only the filename instead of the whole pathname. LI with no pathname loads the files PFTNLB, IFTNLB, and SPLLIB in LIB, which contains most operating system subroutines.
- Duplicate the parameters of the previous load. The purpose is to avoid retyping. Use D/ (Chapter 6).

Modify Runfiles

To patch, save, restore, or copy a runfile without leaving SEG, use the subcommands of the MODIFY subprocessor listed in Chapter 7. To add or replace modules in an existing runfile, use the RL subcommand of VLOAD (Chapter 6).

Create a New Runfile Starting from One That Already Exists

If no segments in the runfile are below '4000, use the NEW subcommand of MODIFY (Chapter 7).

Name a Runfile

Use VLOAD or the command line **SEG –LOAD** discussed with new filename conventions in Chapter 2. NEW may also name a runfile.

Save Runfiles

The QUIT and SAVE commands of SEG write the current runfile to disk, as do the RETURN subcommands of VLOAD and MODIFY. So does EXEC, and the WRITE subcommand of MODIFY. The NEW command also writes segments '4000 and above of a runfile back to disk, copying it to a new file if a new name is given.

Restore Runfiles in Order to Modify Them Without Executing

Use the RESTORE command of SEG (Chapter 5).

Delete Symbols

Use the XPUNGE subcommand of VLOAD (Chapter 6).

Delete Runfiles

Use the DELETE command of SEG, discussed in Chapter 5.

Check Attributes of a Runfile

Use the VERSION, TIME, and PARAMS commands of SEG to check version, date last modified, and runtime parameters. All are listed in Chapter 5. The **runtime parameters** are the starting address, stack location, keys, and contents of the A, B, and X registers.

Leave SEG and Return to PRIMOS

Use QUIT.

B USE OF CMDSEG

CMDSEG

CMDSEG is a command file that handles runfiles larger than one segment. It may be used only if segment '4000 has nothing above '160000; CMDSEG puts the V-mode interface there. It must be used if NEW was used on the runfile or if the runfile has multiple segments.

CMDSEG creates an R-mode program called **program.SAVE** that contains a V-mode interface routine. When **program.SAVE** is executed it loads the runfile and transfers control to the V-mode interface routine. This routine loads the stack base and calls the program. For example, the following routine creates an R-mode program, MAIN.SAVE, for the V-mode program MAIN.SEG. Note that only the commands in rust color are entered by the user.

```
OK, R SEG>CMDSEG MAIN
[CMDSEG version 19.x]
0000 ERRORS [<.DATA.>FTN-REV x.x]
[SEG rev x.x]
#   vload ENA$RJGQCZBBBBCN.SEG.T
$   co abs 4000
$   mi
$   sz 4000
$   s/li share4 130000 4000 4000
$   xp 1 2
$   sy map 4000 126000
$   s/lo b_ENA$RJGQCZBBBBCK.FTN.T 0 4000 4000
$   au 3
$   d/lo <4>seg>cmdlib.bin
$   au 0
$   d/li vapplb
$   au 1
$   d/li
LOAD COMPLETE
$   re
#   sh
TWO CHARACTER FILE ID: CN
CREATING CN4000
#   delete
#   q
```

Now the new program, MAIN.SAVE, should be installed in CMDNC0 and/or shared. Then it may be invoked to run MAIN.SEG. During the above procedure, CMDSEG does not check that MAIN.SEG exists, so that the R-mode interlude may be created for a program that is yet to be developed.

In order to have CMDSEG to check for the existence of the segmented file, enter the command as **R CMDSEG**. CMDSEG will then ask for the name of the file. If it cannot find the file, it will display the message LINKAGE_FAULT$. If the file pathname does not contain a necessary password, CMDSEG will display the message BAD PASSWORD.

THE OLDER VERSION

Below is an example of CMDSEG before Rev. 19. This program takes the user program, MAIN.SEG, and creates a program called *TEST. This new file must then be installed in CMDNC0 and/or shared.

```
OK, R SEG>CMDSEG
R SEG>CM.CFI.SAVE
RUN FILE NAME: MAIN
FTN S$$SEG -64V -DCL
0000 ERRORS [<MAIN  >FTN-REV18.4]
FILMEM ALL
SEG
  LOAD SEG.SHARE.SEG
  CO ABS 4000
  MI
  SZ 4000
  S/LI SHARE4 130000 4000 4000
  XP 1 2
  SY MAP 4000 126000
  S/LO S$$SEG.BIN 0 4000 4000
  au 3
  D/LO SEG>CMDLIB.BIN
  AU 0
  D/LI VAPPLB
  AU 1
  D/LI
  RE
  SH
SE
  DELETE
  Q
[SEG rev x.x]
#   LOAD SEG.SHARE.SEG
$   CO ABS 4000
$   MI
$   SZ 4000
$   S/LI SHARE4 130000 4000 4000
$   XP 1 2
$   SY MAP 4000 126000
$   S/LO S$$SEG.BIN 0 4000 4000
$   au 3
$   D/LO SEG>CMDLIB.BIN
$   AU 0
$   D/LI VAPPLB
$   AU 1
$   D/LI
LOAD COMPLETE
$   RE
#   SH
TWO CHARACTER FILE ID: SE
CREATING SE4000
#   DELETE
#   Q
FUTIL
  C SE4000 *TEST
  DELETE SE4000
  Q
[FUTIL rev x.x]
>   C SE4000 *TEST
>   DELETE SE4000
>   Q
REST *TEST
SAVE *TEST 2/130000
DELETE S$$SEG.BIN
DELETE S$$SEG.FTN
```

C
OCTAL TABLES

To convert an octal number to decimal with Table C-1, locate each digit in the correct column position and add the decimal equivalents of all digits. For example, **675** in octal equals **384** + **56** + **5** in decimal, or **445**. To convert from decimal to octal, locate the largest decimal value in the table that is still smaller than the number to be converted. Subtract that value from your number, and look for the result in the column to the right. For example, 95 in decimal is 1 in the third octal column, 3 in the fourth column, and 7 in the last column; 95 decimal equals 137 in octal.

Table C-1
Octal and Decimal Conversion

8^4		8^3		8^2		8^1		8^0	
OCT	DEC	OCT	DEC	OCT	DEC	OCT	DEC	OCT	DEC
0	0	0	0	0	0	0	0	0	0
1	4096	1	512	1	64	1	8	1	1
2	8192	2	1024	2	128	2	16	2	2
3	12288	3	1536	3	192	3	24	3	3
4	16384	4	2048	4	256	4	32	4	4
5	20480	5	2560	5	320	5	40	5	5
6	24576	6	3072	6	384	6	48	6	6
7	28672	7	3584	7	448	7	56	7	7

For octal addition with Table C-2, find the numbers in the first lines across and down that correspond to the two numbers you want to add. Then find the sum at the intersection of the lines that start with those two numbers. If the sum has more than one digit, the left digit may be carried or added to the next column. Thus, to add '45 and '56, first sum 5 and 6 to get '13. Carry the 1. Add 4 and 5 to get '11, and then add the 1 that was carried, to get '12. Thus the sum of '45 and '56 is '123.

Table C-2
Octal Addition Table

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	10
2	3	4	5	6	7	10	11
3	4	5	6	7	10	11	12
4	5	6	7	10	11	12	13
5	6	7	10	11	12	13	14
6	7	10	11	12	13	14	15
7	10	11	12	13	14	15	16

D SAMPLE PROGRAMS

Example 1

This program is stored as TMDT.F77. It is used as the example for the map in Figure 3-10. It is also used in Chapter 3 as the example in **Looking for Wasted Space** and in Chapter 4 in **PERFORMING A MIXED LOAD**.

```
      PROGRAM MAIN
      INTEGER*2 ARRAY(129)
      INTEGER*2 STRING(28)
      INTEGER*2 NUM, DATE(3)
      INTEGER*2 TIME, TIME1, TIME2, NAME(16)
      EQUIVALENCE (STRING(1), DATE)
      EQUIVALENCE (STRING(4), TIME)
      EQUIVALENCE (STRING(5), TIME1)
      EQUIVALENCE (STRING(6), TIME2)
      EQUIVALENCE (STRING(13), NAME)
      COMMON /AA/STRING, /BB/NUM, /AABB/ARRAY
      NUM = 28
      CALL TIMDAT(STRING, NUM)
      WRITE (1,300) DATE
      WRITE (1,400)
      WRITE (1,200) TIME, TIME1, TIME2
      WRITE (1,150) NAME
200   FORMAT (I6, I6, I6)
```

```
300   FORMAT ('DATE IS ', 3A2)
150   FORMAT ('USER IS ', 28A2)
400   FORMAT ('TIME SINCE MIDNIGHT IN MINUTES+SECONDS+TICKS: ')
      CALL EXIT
      END
```

Example 2

This is filed as MINDLESS.F77. It is a recursive program that soon causes stack overflow. It is used in Chapter 3 as the example of **Looking at the Stack**.

```
      PROGRAM MAIN
C          MINDLESS PROGRAM TO BLOW UP THE STACK
C          COURTESY OF L. B.
C
      INTEGER C
      CALL PAGE(200, C)
      CALL EXIT
      END

      SUBROUTINE PAGE(COUNT, I)
C
C          THIS SUBROUTINE CONTAINS 2048 BYTES OF STACK SPACE, SO IT
C          CAUSES ONE PAGE OF STACK SPACE TO BE RESERVED PER INVOCATION
C
      INTEGER COUNT, I
      INTEGER JUNK(976)
      I = I + 1
      IF(I .EQ. COUNT) RETURN
      WRITE (1,100) I
      CALL PAGE(COUNT, I)
100   FORMAT ('VALUE: ',I3)
      RETURN
      END
```

Example 3

This program is filed as LADD.CBL. It is a COBOL 74 program used in Chapter 3 to illustrate a split load in **Refining Storage Allocation**.

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  MAIN.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77 TEST        PIC X.
       PROCEDURE DIVISION.
           DISPLAY 'SO FAR SO GOOD'.
           STOP RUN.
```

Example 4

This program is filed as CALLER.F77. It calls TMDT.WRONG (Example 5 below). It is used in the example for **Locating Runtime Errors** in Chapter 3.

```
      INTEGER*2 STRING(10), STRING2(12)
      STRING(10) = 'FIRST STEP'
      STRING(12) = 'SECOND STEP'
      WRITE (1,100) STRING
      CALL TMDTWR
      WRITE (1,200) STRING2
100   FORMAT (10A2)
200   FORMAT (12A2)
      CALL EXIT
      END
```

Example 5

This program is filed as TMDT.WRONG. It is used in the subsection **Locating Runtime Errors** in Chapter 3. It is called by the program in Example 4 above. In turn, it calls the subroutine TIMDAT, but without enough arguments.

```
      PROGRAM TMDTWR
      INTEGER*2 STRING(28)
      INTEGER*2 NUM, DATE(3)
      INTEGER*2 TIME, TIME1, TIME2, NAME(16)
      EQUIVALENCE (STRING(1), DATE)
      EQUIVALENCE (STRING(4), TIME)
      EQUIVALENCE (STRING(5), TIME1)
      EQUIVALENCE (STRING(6), TIME2)
      EQUIVALENCE (STRING(13), NAME)
      NUM = 28
      CALL TIMDAT(STRING)
      WRITE (1, 100)
      WRITE (1,300) DATE
      WRITE (1,400)
      WRITE (1,200) TIME, TIME1, TIME2
      WRITE (1,150) NAME
100   FORMAT (' ')
150   FORMAT ('USER IS ', 28A2)
200   FORMAT ('        ', I6, I6, I6)
300   FORMAT ('DATE IS ', 3A2)
400   FORMAT ('TIME SINCE MIDNIGHT IN MINUTES+SECONDS+TICKS: ')
      CALL EXIT
      END
```

Example 6

This program also is filed as TMDT.F77. It is used in Chapter 4 as the example for **MAKING A RUNIT FILE.**

```
      PROGRAM MAIN
      INTEGER*2 STRING(28)
      INTEGER*2 NUM, DATE(3)
      INTEGER*2 TIME, TIME1, TIME2, NAME(16)
      EQUIVALENCE (STRING(1), DATE)
      EQUIVALENCE (STRING(4), TIME)
      EQUIVALENCE (STRING(5), TIME1)
      EQUIVALENCE (STRING(6), TIME2)
      EQUIVALENCE (STRING(13), NAME)
      NUM = 28
      CALL TIMDAT(STRING, NUM)
      WRITE (1, 100)
      WRITE (1,300) DATE
      WRITE (1,400)
      WRITE (1,200) TIME, TIME1, TIME2
      WRITE (1,150) NAME
100   FORMAT (' ')
150   FORMAT ('USER IS ', 28A2)
200   FORMAT ('          ', I6, I6, I6)
300   FORMAT ('DATE IS ', 3A2)
400   FORMAT ('TIME SINCE MIDNIGHT IN MINUTES+SECONDS+TICKS: ')
      CALL EXIT
      END
```

Example 7

This example consists of two programs, both used in the CPL file in Appendix E. Both are named MAIN internally, so they can be shared if necessary.

The first program is filed as ACCT.F77:

```
      PROGRAM MAIN
      WRITE (1,100)
      CALL EXIT
100   FORMAT ('THIS IS ACCT-NO')
      END
```

The next program is filed as LOGIN.F77:

```
      PROGRAM MAIN
      WRITE (1,100)
      CALL EXIT
100   FORMAT ('SO FAR SO GOOD')
      END
```

Example 8

This program is filed as RHELP.PL1G. It is used in Chapter 4 as the example for sharing and for creating external commands.

```
main:  proc options (main);
/**/
/*
/*      panic button program for users of system
/*                    mjk     7/20/82
/*
/**/
/* "^233..." indicates use of the non-printing ESC character, to take
/* advantage of the visual attributes of a PT45 terminal.  Use ^233
/* (four characters) to represent the ESC character when entering
/* program text with ED.
/**/
/* "^233d" is the PT45's A-SET function key.  For instance:
/*    A-SET R - reverse blink       A-SET S - reverse half-blink
/*    A-SET B - blink               A-SET @ - normal display
/*    A-SET A - half intensity      A-SET P - reverse video
/**/

dcl (ext, name, prob_num, prob) char (50) var;

dcl (graf_on,     /* Turn on special graphics character set */
     graf_off,    /* Turn off special graphics character set */
     ul_cor,      /* Upper left corner (graphics) */
     ur_cor,      /* Upper right corner (graphics) */
     ll_cor,      /* Lower left corner (graphics) */
     lr_cor,      /* Lower right corner (graphics) */
     v_bar,       /* Vertical bar (contains graphics mode!) */
     curs_up,     /* Move cursor up */
     rev_vid,     /* Turn on reverse video */
     u_rev_vid,   /* Turn on underline-reverse video */
     blink_vid,   /* Turn on blinking video */
     norm_vid     /* Revert to normal video */
              ) char (8) var;

dcl (h_bar) char (70) var;  /* Horizontal bar (graphics) */

/* Set up the values for a PT45 terminal */
/* (Different values would be required for a PST100 or other
/* terminal.) */
graf_on    = '^233R';
graf_off   = '^233S';
ul_cor     = '@';
ur_cor     = 'D';
ll_cor     = 'H';
lr_cor     = 'L';
v_bar      = '^233R d^233S';
curs_up    = '^233A';
```

~~~
rev_vid   = '∧233dP';
u_rev_vid = '∧233dp';
blink_vid = '∧233dB';
norm_vid  = '∧233d@';
h_bar     = '````````````````````````````````' ¦¦
            '````````````````````````````````';   /* 64 chars */

put skip(3) list
 (' ' ¦¦ graf_on ¦¦ ul_cor ¦¦ h_bar ¦¦ ur_cor ¦¦ graf_off);

call main1;
main1: proc;
  put skip edit(v_bar, v_bar) (col(1),a, col(70),a);
  put skip edit(v_bar ¦¦ '  What is the nature of the problem:', v_bar)
               (col(1),a, col(70),a);
  put skip edit(v_bar, v_bar) (col(1),a, col(70),a);
  put skip edit (v_bar ¦¦
    '   1. Cartridges           6. Software         11. Tape',
                 v_bar) (col(1),a, col(70),a);
  put skip edit (v_bar ¦¦
    '   2. Communications       7. Power supply     12. Hardware',
                 v_bar) (col(1),a, col(70),a);
  put skip edit (v_bar ¦¦
    '   3. CPU                  8. Memory',
                 v_bar) (col(1),a, col(70),a);
  put skip edit (v_bar ¦¦
    '   4. Disk                 9. Miscellaneous',
                 v_bar) (col(1),a, col(70),a);
  put skip edit (v_bar ¦¦
    '   5. Disk controller     10. Operational',
                 v_bar) (col(1),a, col(70),a);
  put skip edit (v_bar, v_bar) (col(1),a, col(70),a);
  put skip edit (v_bar ¦¦
    '    Problem #' ¦¦ blink_vid ¦¦ ':' ¦¦ norm_vid) (col(1),a);
  get skip(0) edit (prob_num)(a);
  put edit (curs_up ¦¦ v_bar)(col(66),a);
  put  edit (v_bar, v_bar) (col(1),a, col(70),a);

if prob_num = '1' then do;
   prob = 'streamer tape cartridges,';
   name = 'Geoffrey Chaucer';
   ext = '3030';
  end;
else if prob_num = '2' then do;
   prob = 'communications problems,';
   name = 'Wife of Bath';
   ext = '4138 & 4002';
  end;
else if prob_num = '3' then do;
   prob = 'CPU problems,';
   name = 'Marie de France';
   ext ='3274 & 3033';
~~~

```
  end;
else if prob_num = '4' then do;
   prob = 'disk problems,';
   name = 'Hildebrand';
   ext = '3027';
  end;
else if prob_num = '5' then do;
   prob = 'disk controller problems,';
   name = 'Hildebrand';
   ext = '3027';
  end;
else if prob_num ='6' then do;
   prob = 'software problems,';
   name ='Geoffrey Chaucer';
   ext = '3274 & 3033';
  end;
else if prob_num = '7' then do;
   prob = 'power supply problems,';
   name = 'Chretien de Troyes';
   ext = '3037 & 3400';
  end;
else if prob_num = '8' then do;
   prob = 'memory problems,';
   name = 'Marie de France';
   ext ='3274 & 3033';
  end;
else if prob_num = '9' then do;
   prob = 'miscellaneous problems,';
   name = 'Chretien de Troyes';
   ext = '3037, 3030, & 3031';
  end;
else if prob_num = '10' then do;
   prob = 'general operational problems,';
   name = 'Chretien de Troyes';
   ext ='3037 & 3031';
  end;
else if prob_num = '11' then do;
   prob = 'tape problems,';
   name = 'Geoffrey Chaucer';
   ext = '3030';
  end;
else if prob_num = '12' then do;
   prob = 'general hardware problems,';
   name = 'Pearl Poet';
   ext = '3036';
  end;
else do;
  put skip edit (v_bar ||
    '          ***' || blink_vid || 'ERROR' || norm_vid || '***',
                v_bar) (col(1),a,  col(76),a);
                  /* 76 is 70 + length(blink_vid || norm_vid) */
  put skip edit (v_bar ||
    '  ' || prob_num || ' is not a legitimate selection.', v_bar)
                (col(1),a,  col(70),a);
  call main1;
end;

end main1;

put skip edit (v_bar, v_bar) (col(1),a,  col(70),a);
put skip edit (v_bar || '   For information on ', prob, v_bar)
              (col(1),a,    col(29),a,     col(70),a);
put skip edit (v_bar ||
  '   call' || u_rev_vid || ' ' || name ||' at extension ' ||
             ext || '.' || norm_vid, v_bar)
            (col(1),a,  col(76),a);

put skip edit (v_bar, v_bar) (col(1),a,  col(70),a);
put skip list
    (' ' || graf_on || ll_cor || h_bar || lr_cor || graf_off);
end main;
```

Example 9

This example is filed as LARGE.F77. It is used in Chapter 4 as an example of managing common blocks.

```
C  THIS PROGRAM SETS UP TWO ARRAYS.  THE ONE CALLED 'ARRAY' IS LARGER
C  THAN ONE SEGMENT. BOTH ARE PUT INTO COMMON BLOCKS.
C
      PROGRAM MAIN
      INTEGER*2 ARRAY(116500), ARY2 (16500)
      CHARACTER*20 STRING1, STR2, STR3, STR4
      EQUIVALENCE(STRING1, ARRAY(1))
      EQUIVALENCE(STR2, ARRAY(116470))
      EQUIVALENCE(STR3, ARY2(1))
      EQUIVALENCE(STR4,ARY2(16470))
      COMMON/AA/ARRAY
      COMMON/BB/ARY2
      STRING1 = 'START OF LARGE ARRAY'
      STR2 = 'END OF LARGE ARRAY'
      STR3 = 'START OF ARRAY 2'
      STR4 = 'END OF ARRAY 2'
      WRITE(1,100) STRING1
      WRITE(1,100) STR2
      WRITE(1,100) STR3
      WRITE(1,100) STR4
100   FORMAT(A20)
      CALL EXIT
      END
```

Example 10

This example contains four routines. The first three are used in Chapter 4 as examples of sharing two programs in the same segment, extending the stack, and relocating the stack. All four are used in the section on **REPLACING PROGRAM MODULES**, also in Chapter 4. SUB1.PL1G and SUB1A.PL1G have the same ECB label (sub1) so that one may replace the other in a load.

This is MAIN.PL1G:

```
main:    procedure options(main);
         dcl sub1 external entry;
         dcl sub2 external entry;
         put skip list ('this is main');
         call sub1;
         call sub2;
         put skip list ('end of run');
end main;
```

This is SUB1.PL1G:

```
sub1:    procedure;
         put skip list ('this is sub1');
         end sub1;
```

This is SUB2.PL1G:

```
sub2:    procedure;
         put skip list ('this is sub2');
         end sub2;
```

This is SUB1A.PL1G:

```
sub1:    procedure;
         put skip list ('this is replacement');
         end sub1;
```

E
A CPL PROGRAM FOR SHARING

```
* *********************************************************************
* THIS CPL PROGRAM ALLOWS YOU TO COMPILE, LOAD, AND SHARE AN EXTERNAL
* LOGIN PROGRAM AND ACCOUNT PROGRAM (EXAMPLE 7 IN APPENDIX D).  IT ASKS
* FOR A SEGMENT NUMBER, AND YOUR CHOICE OF SEGMENT NUMBER DETERMINES
* WHETHER YOU GET A SHARED OR A NONSHARED RUNFILE.  BEFORE RUNNING THE
* FILE, YOU MUST DEFINE AN EMPTY GLOBAL VARIABLE FILE.  THANKS TO J. B.
* *********************************************************************
*
COMO INSTALL.COMO
&SET_VAR .CMDPAS. := [TRIM [RESPONSE 'CMDNC0 UFD PASSWORD']]
&SET_VAR .SYSPAS. := [TRIM [RESPONSE 'SYSTEM UFD PASSWORD']]
*
*
&SET_VAR .SEGNO. := 7777
* ************************************************************
* ACCEPT SEGMENT NUMBER FROM CONSOLE OR FILE
* ************************************************************
TYPE
TYPE PROCEDURE SEGMENT NUMBER MAY BE ONE OF:
TYPE
TYPE   4000         -- NON-SHARED VERSION
TYPE   2015         -- SHARED VERSION, OVERRIDE DPTX SEGMENT (DEFAULT)
TYPE   2030-2037    -- SHARED VERSION, CHOOSE AVAILABLE USER SEGMENT
TYPE   2170-2177    -- SHARED VERSION, "           "    "     "
```

```
TYPE
TYPE PRESS RETURN TO GET THE DEFAULT
TYPE
&DO &UNTIL ( %.SEGNO.% = 4000 ) ¦ ( ( %.SEGNO.% >= 2030 ) & ~
           (%.SEGNO.% <= 2037 ) )  ¦ ( %.SEGNO.% = 2015 ) ¦ ~
           ( ( %.SEGNO.% >= 2170 ) & ( %.SEGNO.% <= 2177 ) )
  &SET_VAR .SEGNO. := [TRIM [RESPONSE 'PROCEDURE SEGMENT NUMBER' 4000]]
&END
*
&SET_VAR .LOC1. := 0
&SET_VAR .LOC2. := 0
&SET_VAR .SPLIT. := 77777
&IF %.SEGNO.%= 4000 &THEN &DO
  &S .LOC1. := [TRIM [RESPONSE 'STARTING WORD# FOR WELCOME' 1000]]
  &S .LOC2. := [TRIM [RESPONSE 'STARTING WORD# FOR ACCOUNT' 10000]]
  &S .SPLIT. := 3777
&END
*
* ************************************************************
* COMPILE THE WELCOME AND ACCOUNT PROGRAMS
* ************************************************************
TYPE FTN WELCOME -64V
FTN WELCOME -64V
TYPE FTN ACCOUNT -64V
FTN ACCOUNT -64V
* ***************************************************************
* MAKE ONE- OR TWO-SEGMENT FILE FOR WELCOME, DEPENDING ON VALUE
* OF .SEGNO.
* ***************************************************************
TYPE SEG -LOAD
&DATA SEG -LOAD
   SPLIT %.SPLIT.%
   AUTOMATIC 10
/*             THE NEXT LINE CREATES A ONE- OR TWO-SEGMENT FILE,
/*             DEPENDING ON WHETHER .SEGNO. IS 4000 OR NOT
   S/LOAD WELCOME %.LOC1.% %.SEGNO.% 4000
   D/LIBR VAPPLB
   D/LIBR
   SAVE
   MAP 2
   MAP 3
   MAP WELCOME.MAP
   RETURN
   SHARE
LG         /*IN REV. 19, THIS MAY BE 28 CHARACTERS, PRECEDED BY SPACES
   DELETE
   QUIT
   &TTY
&END
```

```
*
* ***********************************************************
* DO THE SAME FOR ACCOUNT
* ***********************************************************
TYPE SEG -LOAD
&DATA SEG -LOAD
   SPLIT %.SPLIT.%
   A/SYMBOL DUMMY PR 2030 1200  /*FOR CASE OF TWO PROCS IN ONE SEGMENT
   AUTOMATIC 10
   S/LO ACCOUNT %.LOC2.% %.SEGNO.% 4000
   D/LIBR VAPPLB
   D/LIBR
   SAVE
   MAP 2
   MAP 3
   MAP ACCOUNT.MAP
   RETURN
   SHARE
AC         /*IN REV. 19, THIS MAY BE 28 CHARACTERS, PRECEDED BY SPACES
   DELETE
   QUIT
   &TTY
&END
*
* ***********************************************************
* COPY THE UNSEGMENTED PROGRAMS TO CMDNC0
* ***********************************************************
&DATA FUTIL              /*USE COPY IN REV. 19
   FROM *
   TO CMDNC0 [UNQUOTE %.CMDPAS.%]
   COPY LG4000 WELCOME.SAVE
   COPY AC4000 ACCOUNT.SAVE
   DELETE LG4000, AC4000
   DELETE WELCOME.BIN, ACCOUNT.BIN
   FROM CMDNC0
   PROTECT WELCOME.SAVE 7 1
   PROTECT ACCOUNT.SAVE 7 1
   QUIT
   &TTY
&END
*
* ******************************************************************
* IF THIS WAS A SINGLE-SEGMENT FILE ALL IN '4000, WE HAVE FINISHED
* ******************************************************************
&IF %.SEGNO.%= 4000 &THEN &DO
   &DATA MESSAGE -1 NOW
   ****NONSHARED VERSIONS OF EXTERNAL LOGIN AND ACCOUNT REINSTALLED****
   &TTY
&END
COMO -END
&RETURN
&END
```

```
*
* ************************************************************
* NEXT PART IS EXECUTED ONLY IF .SEGNO. IS <4000 (PROGRAM MUST
* BE SHARED)
* ************************************************************
*
&SET_VAR DATE := [DATE -DOW], [DATE -CAL], [DATE -AMPM]
&SET_VAR DATE := [UNQUOTE %DATE%]
*
&DATA ED
* SHARE EXTERNAL WELCOME AND ACCOUNTING PROGRAM
* REINSTALLED ON %DATE%
   OPR 1
   SHARE SYSTEM>LG%.SEGNO.% %.SEGNO.%
   SHARE SYSTEM>AC%.SEGNO.% %.SEGNO.%
   OPR 0
   CO -CONTINUE 6
   CO -TTY

   FILE C_SHAREWELCOME
   &TTY
&END
*
&DATA FUTIL
   FROM *
   TO SYSTEM [UNQUOTE %.SYSPAS.%]
   COPY LG%.SEGNO.%
   COPY AC%.SEGNO.%
   COPY C_SHAREWELCOME
   FROM SYSTEM [UNQUOTE %.SYSPAS.%]
   PROTECT AC%.SEGNO.%7 1
   PROTECT LG%.SEGNO.%7 1
   QUIT
   &TTY
&END
*
&DATA MESSAGE -1 NOW
 *YOU MUST RESHARE EXTERNAL WELCOME PROGRAM!  CO SYSTEM>C_SHAREWELCOME*
 &TTY
&END
COMO -END
&RETURN
```

F
LOCATING THE DEFAULT SPLIT

This appendix lists the first part of the code for SEGSRC>SHARES.PMA. This file is supplied with every Prime machine. However, your System Administrator may have removed it from the computer in order to save space. Before Rev. 19, the file was named SEG>SHARES.PMA. The lines that are shaded are the ones that give the default split address for SEG in segment '4000. This address may change in any version of the software.

```
* SHARES.PMA, SEGSRC, CEH-LSS-KJC, 01/15/79
* SHARE4 Library - Loaded by the 'SP' command to start SEG runfiles
* Copyright (c) 1981, Prime Computer, Inc., Natick, MA 01760
*
*
       REL
       RLIT
       D64V
       SYML
       C64R
*
       ENT   RUNIT
       ENT   RESUME
       EXT   STACK_OVF$
       EXT   MKONU$
       EXT   MAIN
       EXT   EXIT
       EXT    ERRPR$
*
RUNIT  EQU    *
       ELM
       JMP    RUNIT1
*
*      DEFINE DEFAULT STACK AND STACK EXTENSION (STAK$)
*      THEN COPY CERTAIN PARTS TO ALLOW STACK OVERFLOW HANDLER
*      TO MODIFY COPY OF STACK EXTENSION, THEREBY
*      PROVIDING S 1000 CAPABILITY.

STAK$  EQU    *
       OCT    4000   SEGMENT NUMBER
       OCT    150000 WORD NUMBER
       OCT    0      EXTENSION SEGMENT NOT DEFINED
       OCT    4      WHEN DEFINED WILL BEGIN HERE

STACK$ EQU    *
       ENT    STACK$
       OCT    0     CURRENT STACK SEGM NUM (WILL BE COPIED FROM STAK$+0)
       OCT    2     PTR TO EXTENSION PTR IN CURRENT STACK SEGMNT
       OCT    0     NEXT EXTENSION SEGMENT(COPIED FROM STAK$+2)
       OCT    4     WILL BEGIN HERE

*
*
RUNIT1 STL    ISAVE
       STX    ISAVE+2
*
       LDA    STAK$           COPY THE ORIGINAL STAK$ INFORMATION
       STA    STACK$          SO THAT ON-UNIT HANDLER WON'T CLOBBER
       LDA    STAK$+2         ORIGINAL PARAMETERS.
       STA    STACK$+2
       LDL    STAK$          GET SEGMENT NUMBER AND ADDRESS
       STLR   13                SAVE START OF STAK
       LDA    DUMP              STACK ROOT IN SEG 4000
       STA    SB%+1             AN INCESTUOUS STORE
       EAXB   DUMP,*
       LDL    STAK$            SET UP WORD 2 IN STACK SEGMENT
       ADL    =44L
       STL    XB%
       CRL                      NO STACK EXTN TO START
       LDX    =-8               ZERO OUT SOME WORDS
LOOP   STA    XB%+10,1          IN AND NEAR STACK ROOT
*
          .
          .
          .
          .
```

G GLOSSARY

- Absolute prefix

The prefixes S/, P/, or D/ when preceded by one of the first two. They cause the segment number next entered to be taken as an absolute segment number.

- Absolute segment number

A segment number that is assigned to the same virtual segment number. To be absolute, the number must have been entered with COMMON ABS, the S/ or P/ prefix, or the D/ prefix after an S/- or P/-prefixed command. See the discussions in Chapters 1 and 4.

- Address

Prime memory is addressable in 16-bit offsets. An address is a pointer to a 16-bit location.

- Base area

A memory area used for indirect addressing references. The base area is mostly required by Prime's older COBOL, which uses many 16-bit instructions requiring an indirect reference.

- Binary file

The file produced by a compiler or assembler. Binary files must be loaded into runfiles by SEG or LOAD.

- Common area

See Common block

- Common block

A block of data that may be loaded separately from its program. Common blocks are defined with the COMMON pseudo-op in PMA, the COMMON statement in FORTRAN, the $E+ switch in Pascal, and the EXTERNAL attribute in PL1G. Common blocks cannot be defined in COBOL.

- Current directory

The directory to which the user process is attached.

- Current load point

The next available 16-bit address in the procedure or data segments of a V-mode runfile. It corresponds to PBRK in an R-mode file created by LOAD.

- Current runfile

The runfile (executable file) into which SEG will load the next object file. The current runfile may be set with the VLOAD or RESTORE commands, and changed with the NEW subcommand of MODIFY.

- Data segment

See Linkage segment.

- Direct entry link

A label in the PRIMOS operating system code that is used as a subroutine call.

- DTAR

A descriptor table address register, which designates whether its associated segments are unique or shared. See Chapter 1.

- Dynamic entry point

See Direct entry link.

- ECB

The entry control block for an object file. It includes the following information, where **address** is the 16-bit relative offset:

Address	
0-1	Pointer to Called Procedure
2	Stack Frame Size
3	Stack Root Segment Number
4	Argument List Displacement
5	Number of Arguments
6-7	Link Base Reg. of Called Proc
8	**Keys**
9-15	Reserved

An ECB is usually part of the linkage section of a program. ECBs are defined with the ECB directive in PMA, with PROGRAM, SUBROUTINE, or ENTRY in FORTRAN, with PROGRAM-ID in COBOL, with PROCEDURE or ENTRY in PL1G, and with PROGRAM, PROCEDURE, or FUNCTION in Pascal.

- Entry point

An ECB.

- External Reference

A reference or call to a name that is not in the same binary file, such as an external subroutine or common block.

- Force load

To load all modules of an object file, whether or not they have been referenced. Forced loading is needed for library modules in a template or other runfile where not all calls are known when the libraries are loaded. The F/ prefix is used for forced loading.

- Home directory

Normally, the directory specified as home by the user's login name. However, file-handling subroutines and the ATTACH command of SEG may set another UFD as home. This is explained in the **Subroutines Reference Guide**.

- Impure FORTRAN library

The library IFTNLB in the UFD named LIB. It contains some non-reentrant system subroutines.

- Initialized data

A data area to which a value has been assigned.

- Interlude

A program that has its procedure and linkage in segment '4000, and invokes another program that has a different format. Interlude programs are most often used to run programs that cannot themselves be run from CMDNC0 but that either are shared or are external commands.

- Link base

The register that holds the address of the linkage area.

- Link frame

See Linkage.

- Linkage

The part of a program that contains static variables, ECBs, and link frames.

- Linkage segment

A segment reserved for linkage and common blocks, also called a data segment. By default, SEG assigns segment '4002 as the first linkage segment, and assigns higher numbers as necessary.

- Linking loader

A loader that resolves external references as it loads.

- LOAD

Prime's old load utility, for R-mode files only.

- Load point

See Current load point above.

- Location

An address consisting of a segment number plus an offset. The offset is normally expressed in 16-bit units.

- Object file

A file containing one or more object modules.

- Object module

A binary module created by compilation or assembly of one source program.

- Pathname

The name of a file, which may include the MFD name, the UFD name, and one or more sub-UFD names, as well as the filename.

- PBRK (program break)

In a file created by the LOAD utility, the current load point or next available location for loading. See also Current load point.

- Procedure

The part of a program that contains instructions.

- Procedure base

The register that holds the address of the next executable instruction.

- Procedure segment

A segment reserved for procedure code and the stack. By default, SEG assigns segment '4001 as the first procedure segment and then assigns higher numbers as available.

- Pure FORTRAN library

The library PFTNLB in the UFD named LIB. It contains reentrant system subroutines.

- R-mode runfile, R-mode image

A file produced by the old LOAD utility. For similar files produced by SEG, see RUNIT file. An R-mode runfile uses only R-mode instructions and cannot take advantage of virtual memory.

- Recursion

The attribute of a program that can call itself.

- Reentrancy

The attribute of code that keeps it from modifying itself.

- Reference

A call to another program, or use of a symbol name. If the program or name is not in the same binary module, it is called an external reference.

- Relative segment number

A segment number that is reassigned by SEG with its default numbers. These numbers usually begin with '4001 for procedure segment and '4002 for linkage and common blocks, unless the relative assignment includes a different number after COMMON REL. All segment numbers entered by users are taken by SEG as relative numbers unless they are preceded by COMMON ABS, the S/ or P/ prefix, or the D/ prefix after an S/ or P/ command. See the discussion in Chapter 1.

- Relocating loader

A loader that assigns arbitrary addresses to modules, regardless of the addresses assigned by the compiler or assembler that produced the modules.

- Resolution

The determining of addresses for symbol names and program names.

- Runfile

An executable version of a program, consisting of the loaded binary files, subroutines, and library modules used by the program, and any common areas. If produced by SEG with default parameters, the runfile is also a segment directory (see below).

- RUNIT file

A runfile produced by SEG containing V-mode instructions but with both procedure and linkage in segment '4000, together with the execution unit RUNIT. This file is small and fast, may be run with RESUME, and may be used in the UFD named CMDNC0 as a PRIMOS command. It is a SAM file.

- SAM file

A sequential access file, not a segment directory.

- SEG

Prime's segmented loading utility for V-mode and I-mode files.

- Segment

A block of address space consisting of 131,072 bytes. SEG works with virtual segments.

- Segment directory

A directory divided into unnamed subfiles, suitable for copying into Prime's virtual address space.

- Sharing

Installing procedure or data in a segment numbered below '4000 and declaring that segment to be shared. Then more than one user may access the same copy of the code simultaneously.

- Snapping a link

Resolving a direct entry link with PRIMOS. This is done at runtime, whereas traditional subroutine calls are resolved at load time.

- Source file

A file containing programming statements in the format recognized by PMA or one of Prime's high-level language compilers.

- Split

To divide a segment into procedure and linkage parts, with the procedure stack allocated in it.

- Split load

A load with procedure and data in the same segment.

- Stack frame

Storage that is dynamic, that is, assigned when a routine is called and released upon return from the routine. For each routine called, the stack frame header has the following information. **Address** is the relative 16-bit offset.

Address	
0	Invocation Flags
1	Stack Root Segment Number
2-3	Return Pointer
4-5	Caller's Stack Base Reg
6-7	Caller's Link Base Reg
8	Caller's Keys
9	Location Following Call
10-12	Fault Code & Address
13-15	Reserved

- Symbol

A name in a binary file or in a runfile.

- Symbol table

SEG's table in which it keeps track of external references and whether they have been resolved.

- System libraries

The libraries IFTNLB, PFTNLB, and SPLLIB, all in the UFD named LIB. They contain operating system subroutines described in the **Subroutines Reference Guide**, which are called by all Prime compilers.

- Template

A partial runfile into which some libraries, supplied by Prime or the user, have been loaded. The nonshared part (segment '4000 and above) is then copied to a new file. An individual application program that is to run with the previously loaded libraries may then be loaded into the new copy. Using the template will save loading time.

- Unsnapped link

An unresolved reference to a direct entry point (see above). These links are resolved or snapped at runtime rather than load time.

- V mode

The addressing mode normally used on Prime's compilers. Programs in V mode can take full advantage of virtual address space and of the V-mode instruction set.

- Virtual segment

A segment stored on disk, which may occupy any physical segment when it is brought into live memory.

INDEX

READER RESPONSE FORM

DOC 3524-192 SEG and LOAD Reference Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

   ___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

   Readability: ___hard to understand ___average ___very clear

   Technical level: ___too simple ___about right ___too technical

   Technical accuracy: ___poor ___average ___very good

   Examples: ___too many ___about right ___too few

   Illustrations: ___too many ___about right ___too few

3. What features did you find most useful?________________________________

   ____

   ____

   ____

4. What faults or errors gave you problems?________________________________

   ____

   ____

   ____

Would you like to be on a mailing list for Prime's current documentation catalog and ordering information? ___yes ___no

Name:____________________ Position:____________________

Company: ____

Address: ____

________________________________ Zip: ________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760

READER RESPONSE FORM

DOC 3524-192 SEG and LOAD Reference Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

   ___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

   Readability: ___hard to understand ___average ___very clear

   Technical level: ___too simple ___about right ___too technical

   Technical accuracy: ___poor ___average ___very good

   Examples: ___too many ___about right ___too few

   Illustrations: ___too many ___about right ___too few

3. What features did you find most useful?______________________________

   ____

   ____

   ____

4. What faults or errors gave you problems?______________________________

   ____

   ____

   ____

Would you like to be on a mailing list for Prime's current documentation catalog and ordering information? ___yes ___no

Name:______________________ Position:______________________

Company: ____

Address: ____

______________________________________ Zip: __________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PR1ME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760

READER RESPONSE FORM

DOC 3524-192 SEG and LOAD Reference Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

   ___excellent ___very good ___good ___fair ___poor

2. Please rate the document in the following areas:

   Readability: ___hard to understand ___average ___very clear

   Technical level: ___too simple ___about right ___too technical

   Technical accuracy: ___poor ___average ___very good

   Examples: ___too many ___about right ___too few

   Illustrations: ___too many ___about right ___too few

3. What features did you find most useful?______________________________

   ____

   ____

   ____

4. What faults or errors gave you problems?______________________________

   ____

   ____

   ____

Would you like to be on a mailing list for Prime's current documentation catalog and ordering information? ___yes ___no

Name:______________________ Position:______________________

Company: ____

Address: ____

______________________________________ Zip: __________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PR1ME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760