HP 3000 Computer Systems

MPE SEGMENTER

Reference Manual

HEWLETT PACKARD

19447 PRUNERIDGE AVENUE, CUPERTINO, CA 95014

Part No. 30000-90011
U0886

Printed in U.S.A. 11/82

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the date of the current edition, and lists the dates of all pages of that edition and all updates. Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. If an update is incorporated when an edition is reprinted, these bars and dates remain. No information is incorporated into a reprinting unless it appears as a prior update.

Third Edition.....November 1982
Update #1.....August 1986

Effective Pages	Date
Title	8/86
ii to vi	8/86
vii	ORIGINAL
viii to x	8/86
1-1	8/86
1-2 to 1-9	ORIGINAL
1-10 to 1-11	8/86
2-1	8/86
2-2 to 2-6	ORIGINAL
2-7 to 2-8	8/86
2-9	ORIGINAL
2-10	8/86
2-11 to 2-12	ORIGINAL
2-13 to 2-16	8/86
2-17 to 2-20	ORIGINAL
2-21	8/86
2-22	ORIGINAL
2-23 to 2-24	8/86
2-25	ORIGINAL
2-26	8/86
2-27	ORIGINAL
2-28 to 2-31	8/86
2-32 to 2-33	ORIGINAL
2-34 to 2-35	8/86
2-36 to 2-38	ORIGINAL
2-39 to 2-45	8/86
2-46 to 2-47	ORIGINAL
3-1 to 3-2	ORIGINAL
3-3 to 3-4	8/86
3-5 to 3-9	ORIGINAL
3-10 to 3-11	8/86
4-1 to 4-3	8/86
4-4 to 4-5	ORIGINAL
4-6 to 4-9a	8/86
4-10 to 4-11	ORIGINAL
4-12	8/86
4-13 to 4-18	ORIGINAL
4-19	8/86
4-20	ORIGINAL
4-21	8/86
4-22	ORIGINAL
4-23 to 4-23c	8/86
4-24	ORIGINAL
4-25 to 4-28	8/86
4-29	ORIGINAL
4-30 to 4-38	8/86
4-39	ORIGINAL
4-40	8/86
4-41 to 4-44	ORIGINAL
4-44a to 4-44b	8/86
4-45 to 4-46	ORIGINAL
4-46a to 4-46d	8/86
4-47	ORIGINAL
5-1 to 5-2	ORIGINAL
5-3	8/86
5-4	ORIGINAL
5-5	8/86
5-6 to 5-9	ORIGINAL
5-10	8/86
5-11	ORIGINAL
A-1	ORIGINAL
B-1 to B-2	ORIGINAL
C-1	ORIGINAL
D-1	ORIGINAL
E-1 to E-4	ORIGINAL
F-1	ORIGINAL
G-1	ORIGINAL
I-1 to I-3	8/86
Reader Comment Sheets (3)	8/86
Back Cover	8/86

PRINTING HISTORY

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover of the manual changes only when a new edition is published. When an edition is reprinted, all the prior updates to the edition are incorporated. No information is incorporated into a reprinting unless it appears as a prior update. The edition does not change.

First Edition . . . . . JUN 1976
Second Edition. . . . . FEB 1977
Third Edition . . . . . NOV 1982
Update #1 . . . . . AUG 1986

PREFACE

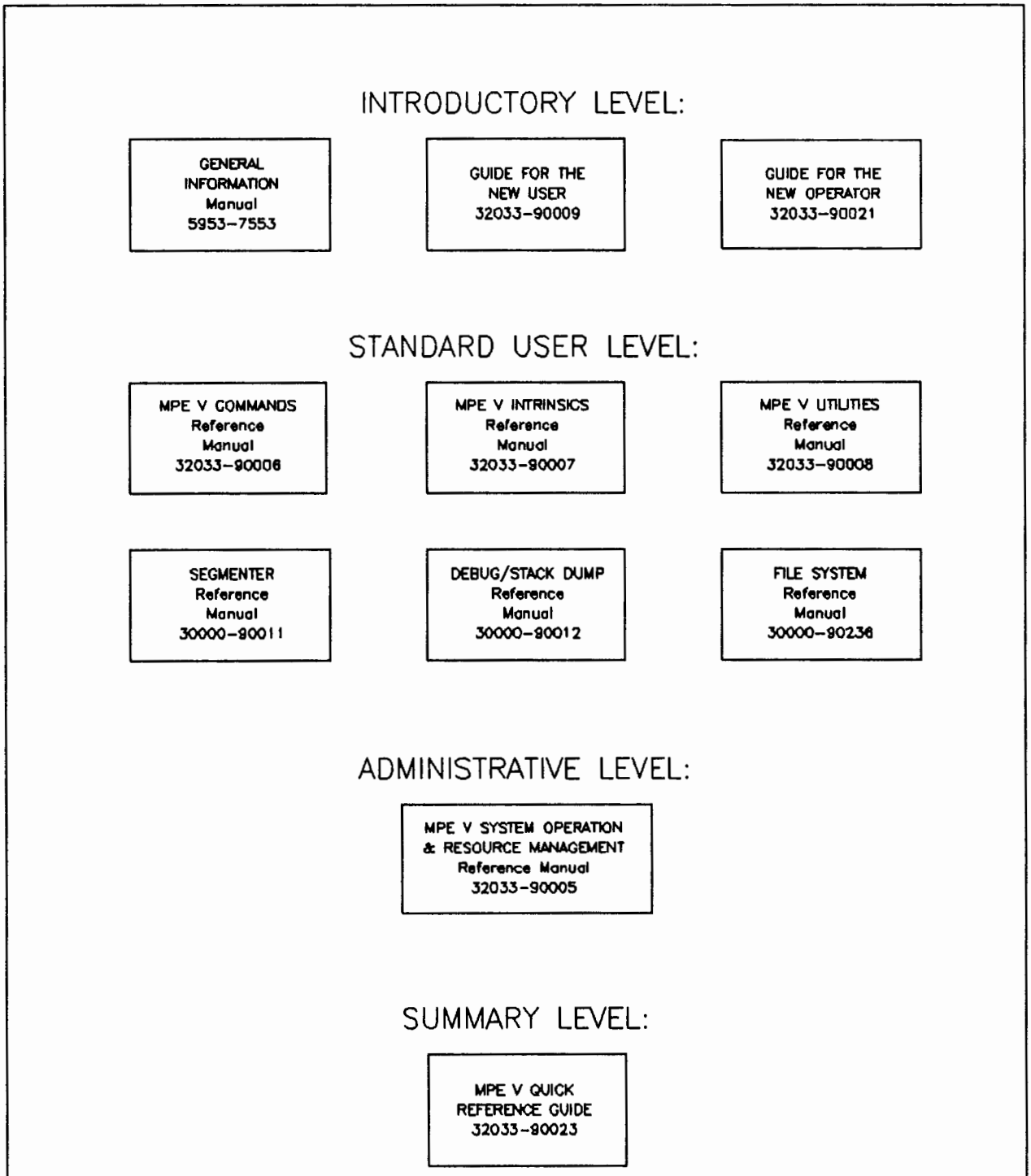
This manual describes how you can use the MPE Segmenter to manage shared code stored in library files and to control the segmentation of program code. In addition to specifying individual commands and intrinsics used within the subsystem (Section V), the manual discusses what the Segmenter is and how it works, and provides strategies for effective Segmenter use. This update to the third edition of the MPE Segmenter Manual includes information on FPMAP and SL Expansion.

Although the Segmenter is a powerful tool, many programmers never need to access it explicitly. The manual, therefore, is arranged so that those needing general information can find it without getting lost in technical detail. Conversely, readers requiring more complex information should be able to locate it quickly. Section I (INTRODUCTION TO THE SEGMENTER) is intended for readers who need an overview or a quick review. Sections II and III (USING THE SEGMENTER and STRATEGIES FOR USING THE SEGMENTER) are intended for users who have become familiar with the Segmenter and plan to use it heavily for the management of stored code and for the control of program segmentation.

Although the manual provides basic as well as higher-level information, it is assumed that you have some familiarity with programming, with the HP 3000, or both. If you need further help or information, the following documentation will provide any in-depth discussions you may require:

- Using the HP 3000: An Introduction to Interactive Programming (03000-90121)
- MPE File System Reference Manual (30000-90236)
- MPE V Intrinsics Reference Manual (32033-90007)
- MPE V Commands Reference Manual (32033-90006)

MPE V MANUAL PLAN



There are many more manuals applicable to the HP 3000. A complete list may be found in every issue of the MPE V Communicator. Please contact your System Manager.

CONVENTIONS USED IN THIS MANUAL

NOTATION	DESCRIPTION
COMMAND	Commands are shown in CAPITAL LETTERS. The names must contain no blanks and be delimited by a non-alphabetic character (usually a blank).
KEYWORDS	Literal keywords, which are entered optionally but exactly as specified, appear in CAPITAL LETTERS.
parameter	Required parameters, for which you must substitute a value, appear in ***bold italics***.
parameter	Optional parameters, for which you may substitute a value, appear in *standard italics*.
[ ]	An element inside brackets is optional. Several elements stacked inside a pair of brackets means the user may select any one or none of these elements. Example: [A] [B] user may select A or B or neither. When brackets are nested, parameters in inner brackets can only be specified if parameters in outer brackets or comma place-holders are specified. Example: [*parm1*[,*parm2*[,*parm3*]]] may be entered as *parm1*,*parm2*,*parm3* or *parm1*,,*parm3* or ,,*parm3* ,etc.
{ }	When several elements are stacked within braces the user ***must*** select one of these elements. Example: { A } { B } user must select A or B or C. { C }
...	An ellipsis indicates that a previous bracketed element may be repeated, or that elements have been omitted.
user input	In examples of interactive dialog, user input is underlined. Example: NEW NAME? ALPHA1
superscript^{c}	Control characters are indicated by a superscript^{c}. Example: Y^{c}. (Press Y and the CNTL key simultaneously.)
RETURN	RETURN indicates the carriage return key.

CONTENTS

CONTENTS (Continued)

ILLUSTRATIONS

TABLES

INTRODUCTION TO THE SEGMENTER

SECTION I

SITUATION CHECKLIST

You need to study this manual in detail under these circumstances:

- When you start developing programs if your facility requires large, complex applications programs.
- If you've inherited programs from another programmer and don't understand what the Segmenter commands are doing.
- If you can't prepare a program because your code segment is too large.
- When you have exceeded the limit of 255 code segments in your program file.
- If you've made changes or wish to make changes to a procedure used frequently in your facility and you wish to put it back into a relocatable library (RL) file or a segmented library (SL) file.
- If you wish to take a common procedure residing in a user subprogram library (USL) file and build it in with your code or put it into an RL file or an SL file.
- When you know you have infrequently-used procedures mixed into segments with those frequently used; or you have procedures which call each other in different segments and you realize you could increase run-time efficiency by moving code around within or among segments.
- When you frequently exhaust available Code Segment Table (CST) entries.

VIRTUAL MEMORY AND SEGMENTATION

Because memory capacity is limited, all computer systems need some method of separating code and data into units and moving these units in and out of main memory as they are needed. HP has developed a solution based on "virtual memory" and "segmentation."

Virtual Memory

Virtual memory is a memory management scheme which uses disc storage as secondary memory, allowing the system to reference a virtual memory space many times larger than main memory so that you can write programs much larger than the physical memory available could contain. As programs are executing, only those pieces of each program required at a particular time actually reside in main memory. The other related segments remain on disc until they are in turn required. Then the system makes space available for them and brings them into main memory. Thus, segments within a program which are idle do not take up space along with those that are actually executing, possibly preventing the loading of code segments needed for another program. This design allows the HP 3000 to run multiple programs concurrently.

The process of bringing code segments from disc memory to main memory is called "swapping". Excessive swapping slows down program execution and, in general, makes heavy demands on system resources. The number of times swapping occurs depends on how efficiently a program is segmented with respect to:

- Program logic. If procedures in one segment frequently call procedures within another segment, the operating system may have to make frequent swaps and transfers of control.

- Memory size available. If segments are too large, segments which should be in main memory together because of their logical relation will have to be swapped in and out. If segments are uneven in size, the system will spend much time seeking appropriate space for each segment.

The HP 3000 allows you to tailor segmentation to the logic of your program and the memory space available in your system.

Segmentation

Segmentation is the separation of code into various-sized pieces, or "segments," according to logical, rather than physical, considerations. It is the most efficient means of implementing the HP 3000's virtual memory design. The HP 3000 can follow system defaults to segment your program. You can also handle the segmentation yourself, using embedded control statements to the compiler or commands to the Segmenter, which is a subsystem of the MPE operating system. Several features contribute to the flexibility of the segmentation design.

SEPARATION OF CODE AND DATA. Code consists of the executable instructions that make up a program or subprogram. Data is the values and arrays used by the program or subprogram. In most computer systems, prepared programs consist of intermixed code and data. For example, within a subprocedure there are program locations reserved by the compiler for the return addresses of other subroutines and space set aside for the storage of local variables.

The HP 3000 system separates programs into those elements that do not need to be altered and those that do. Thus, prepared HP 3000 programs consists of separate segments for code and for data. The two are never intermixed (with the exception that program constants may be present in code segments). Since data changes dynamically during execution, it must be written back to disc storage after each modification. Code, on the other hand, is unchanging during execution and needs only to be read into main memory, never written back to disc. When a code segment is no longer needed, it is simply overlaid by another code segment. Should the segment be needed again, another copy can be read in from the original on disc.

Because it separates code from data, the HP 3000 reduces the amount of material that must be swapped. If code were intermixed with data, the system would have to swap material that had not changed along with material that had.

Although code is not modifiable during execution, you can use the Segmenter to create alternate versions of programs from the compiled source code.

SHARABLE CODE ENVIRONMENT. Because the HP 3000 maintains code and data in strictly separate environments, and because code is non-modifiable during execution, HP 3000 code is sharable among many users. HP 3000 code is also re-entrant: when a program is interrupted during

completely protected against modification, and will be returned intact to the previous user's execution. Hewlett-Packard's design for code handling uses main memory with optimum efficiency.

VARIABLE SEGMENT SIZE. In the HP 3000 design, segment sizes are not fixed, as are the code and data entities in some designs, but vary according to the logical needs of each unit of code or data: code segments may be up to 16,383 words in length, and data segments may be up to 32,767 words in length. Thus, a particular subprogram can always be contained within one segment rather than arbitrarily divided between two physical pages. The amount of swapping necessary is reduced, and memory space is not wasted with partially-filled pages.

Segmentation is one of the key features in the design of the HP 3000. Good segmentation can enhance your program's execution efficiency as well as lessening the overall load on system resources.

Although the management of code segments (i.e., the transfer of segments from disc to main memory) is completely transparent to your program, the steps you take to control segmentation affect how efficient that management can be. The Segmenter subsystem is a powerful tool which allows you to manipulate code and to tailor segmentation, assuring efficient individual programs and effective use of your system's resources.

THE SEGMENTER

The Segmenter is a subsystem of the MPE operating system. It performs all intermediate functions between source code compilation and program execution. One of its primary responsibilities is to gather and link into pieces, or segments, most of the resources needed to form an executable program file. The Segmenter gives programmers considerable control over the arrangement of code within segments, which in turn affects the efficiency of individual programs and the economical use of system resources in general.

The Segmenter In Context: The Program Development Process

To get a clear idea of what the Segmenter is and how it works, it helps to have in mind the process of program creation. Figure 1-1 provides an overview of the entire process.

An analogy will help to summarize the process and to highlight some particularly important points.

Think of the various elements and events of the program development process, taken together, as forming a multi-floor condominium, with the elements corresponding as follows:

- Relocatable Binary Modules (RBMs): rooms.
- Entry points: doors.
- Segments: floors (groups of rooms).
- Procedure libraries: files of pre-designed common areas, such as hallways, bathrooms, kitchens.

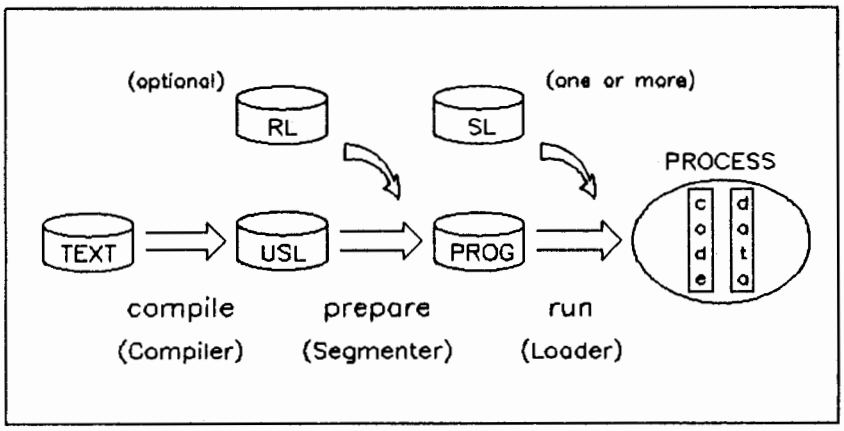


Figure 1-1. Program Development Overview

- User Subprogram Library (USL): preliminary plans.
- Program file: final plans.

The program development process is analogous to the architectural process of designing the condominium. The architect takes her ideas and translates them into a preliminary plan. This floorplan may contain one or more rooms per floor, and each room may have one or more doors.

Since these are preliminary plans, the architect can rearrange rooms and even move rooms to different floors. She can also go to her portfolios for copies of already-created rooms or groups of rooms to incorporate into one of her floors.

When the architect is finally satisfied with the design, she prepares the final plan, inking in the lines and in effect locking the doors, rooms, and floors into a permanent arrangement.

Since a copy of the preliminary plans still exists, the architect can get them out and go through the process as often as she wishes to create other, slightly different final plans to be followed during actual construction (program loading and execution).

In the following explanation, the boxed portions of the accompanying diagrams indicate which parts of the process are active for that step.

COMPILATION. The first step, illustrated in figure 1-2, is to translate the source program units (sometimes called procedures, functions, or subroutines) into blocks of machine instructions called "relocatable binary modules", or RBMs. This is done by the various MPE language compilers, which automatically store the RBMs in a specially-formatted file called the user subprogram library, or USL.

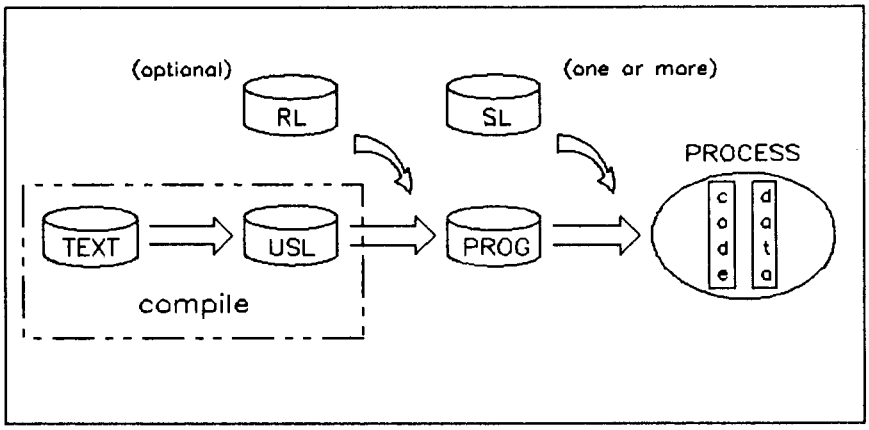


Figure 1-2. Compilation

Relocatable Binary Modules. An RBM is the smallest unit of object code generated by a compiler. RBMs for the various subprograms contain program instructions and external references (references to procedures in other RBMs or in library files). In addition to these subprogram RBMs, the compiler constructs a main, or outer block, RBM, which contains instructions for the main program as well as program constants. The main RBM may also contain fixed addresses for locating items in the data stack; this information will be used later in the program development process.

The different compilers have their own specific conventions for constructing program units into RBMs. See Appendices A through F for a discussion of these.

Some programming languages (FORTRAN and SPL) allow you to specify parameters within RBMs as "entry points". These are points within the code unit which you can selectively instruct the system to use as starting locations for a particular action.

When the compiler places the RBMs in the USL file, it "associates" them with particular code segments. Actual code segments do not yet exist, but you can think of the RBMs as "belonging" to their associated segments.

The RBMs are relocatable, which means that using the Segmenter you can move RBMs around and associate them with different code segments. You can also copy RBMs from other USLs, add new RBMs to a USL, or purge RBMs.

An analogy will help clarify some of the important characteristics of RBMs.

Suppose you are designing posters for a presentation. Each poster will contain one or more pictures or diagrams. You lay the pictures out on the poster board but leave them unglued, so you can rearrange them on the boards or move them from board to board as often as you wish. Once they are glued down, however, their positions are fixed, and they can no longer be moved around.

Think of RBMs as the diagrams and the code segment names as the poster board. While in the USL, the RBMs can be rearranged and associated with one code segment name (board) or another. At preparation time, the Segmenter looks at the current arrangement and applies the "glue" to form the final poster, or code segment, which is output to the program file.

However, unlike the pictures which were glued to the poster board, the pieces of code used to form the code segments still exist in the USL, waiting to be further manipulated or changed in another cycle of segment design. The picture, or code, glued into the code segment was just a copy of the code modules found in the USL.

User Subprogram Libraries (USLs). Users often think of code as residing only in program files, but in Hewlett-Packard's design for code-handling, code can also be stored, maintained, and managed in three different kinds of specially formatted files called "libraries." The user subprogram library, or USL, is the first of the procedure libraries used in the program development process. It is the file used for compiler output and forms the basis for the other two libraries ("Relocatable Libraries," or RLs; and "Segmented Libraries," or SLs). The USL is the file you can manipulate to achieve effective segmentation.

In addition to an RBM for the main program and each subprogram successfully compiled, the compiler generates and places the following into the USL:

- A directory to keep track of the RBMs stored in the USL.
- Data stack information that will be required at a later stage.

A major advantage of this library is that once code is compiled into the USL, the programmer can manipulate it and even copy it into another USL without having to perform time-consuming recompilation.

Users can also compile several versions of a procedure into a USL, using the Segmenter's indexing capability to select at execution time the version they wish to use. Although USLs are usually created by the compiler, the programmer can create them, using commands within the Segmenter subsystem. Segmenter commands also allow users to list the contents of the USL they are currently working with.

PREPARATION. The second step in the program development process is to "prepare" the USL. This step, which is illustrated in figure 1-3, is performed by the MPE Segmenter subsystem, and results in a "program file" containing:

- Loadable code segments.
- A skeleton data segment, or "stack".

The Segmenter may bring in procedures from a "relocatable library" (RL) to resolve external references made within the RBMs, such as a call to a procedure which finds the cosine of a number generated within the program.

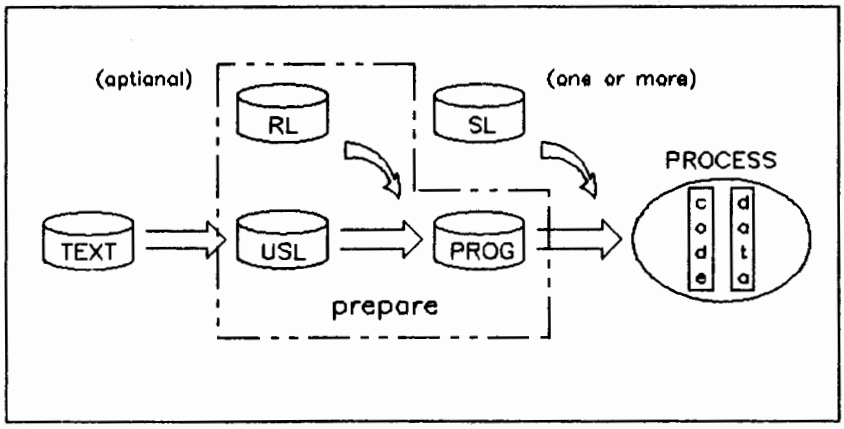


Figure 1-3. Preparation

Code Segments. During preparation, the Segmenter uses the associations found in the USL to bind the RBMs into one or more code segments. It is important to realize that no segments actually existed in the USL. By associating RBMs with segments, the compiler was, in effect, looking forward to segments which would eventually be created. That creation is the sole responsibility of the Segmenter.

As it establishes the necessary linkages between RBMs, the Segmenter creates an "external reference list", which contains information about those RBMs not present in the program file, but which are referred to within it.

The Segmenter also generates a Segment Transfer Table (STT) for each code segment it creates. This table contains linkage information, used during execution when control has to transfer from one RBM within the segment to another. If control has to transfer to an RBM in another segment, the STT will keep track of that linkage as well.

Code segments may contain only non-modifiable material; that is, material not subject to change during execution. Therefore, code segments may contain program instructions, but they may not contain data. The only exception to this rule is that program constants (which are non-modifiable data) may be contained in code segments.

Although you cannot change the code segments themselves, you can manipulate the original copy of the code, which resides in the USL, and re-prepare it into a variation of your first code segment.

In summary, each code segment contains the following:

- The instructions of the program itself.
- Program constants.
- Addresses for locating items in the data stack.
- An external reference list.
- An STT for keeping track of intra-segment and inter-segment transfers.

The Skeleton Data Segment, or Stack. The Segmenter is also responsible for creating a skeleton data segment (or stack), based on references in the code, and placing it in the program file. Each program file will have only one data segment.

To construct the data segment, the Segmenter uses the initial stack information compiled into the USL. That information defines and initializes storage space for data that is considered global (available to all program units directly). This area, usually referred to as the Primary DB, contains information about and pointers into the other parts of the data segment. A secondary storage area accessible to all program units indirectly through local or global pointers is defined as well, and parts of it may also be initialized. The secondary storage area is called the Secondary DB.

Relocatable Libraries. The relocatable library, or RL, is the second of the three libraries which may be accessed during the program development process. RLs contain procedures, in RBM form, needed for program execution. They can be created by the programmer, using Segmenter commands, from the material in a USL. Programmers can also use Segmenter commands to add RBMs from a USL to an already-built RL, to purge RBMs within RLs, and to list the contents of the currently-managed RL.

When a program makes a call to some or all of the RBMs kept in an RL file, the Segmenter copies the RBMs at preparation time and binds them to the calling program as a single segment, known as the "RL segment." Different programs will likely ask for different RBMs or combinations of RBMs and will need to use them in unique ways, so the RL segment must be unique to each program.

No segmentation information accompanies the code in RL RBMs, as it does those in a USL. Since all required RBMs are added to the calling program file as a single segment, individual segment associations for each RBM are unnecessary. The Segmenter binds the single RL segment to the code segment it is constructing from USL material.

Since a copy of requested RL material is prepared into the program file, your program file will have to be re-prepared should the RL code change. In fact, all program files prepared with that code will have to be re-prepared.

RLs are used to keep procedures that are likely to be used with some frequency by more than one programmer. Keeping such common procedures in an RL means that programmers can access them more efficiently: they can use a call to the procedure and receive a copy of it for their program file. They do not have to rewrite it each time it is needed in a new program.

Different programming languages have different rules for the specific kinds of code that may be placed in an RL; these are discussed in the reference manual for each language. All languages allow you to place either global or non-global procedures in an RL. Global procedures reference the global storage area of a program's data stack, so that the system will have instructions on how to handle the RL code that are appropriate to each particular file in which it is used. Non-global (also called local or dynamic) procedures are more general and can be prepped and run without any knowledge of the program's data stack.

EXECUTION. In the third and final step, illustrated in figure 1-4, the MPE Loader allocates entries in the HP 3000 Code Segment Table (CST) for each of the program file's SL code segments and in the code segment table extension for each of its other code segments. It also allocates an entry in the HP 3000 Data Segment Table for the process's data stack. These two tables keep track of all the segments needed for your program's execution, as well as for the execution of all other programs ready to run at that time. The Loader also searches segmented libraries to resolve any external references remaining in the program file.

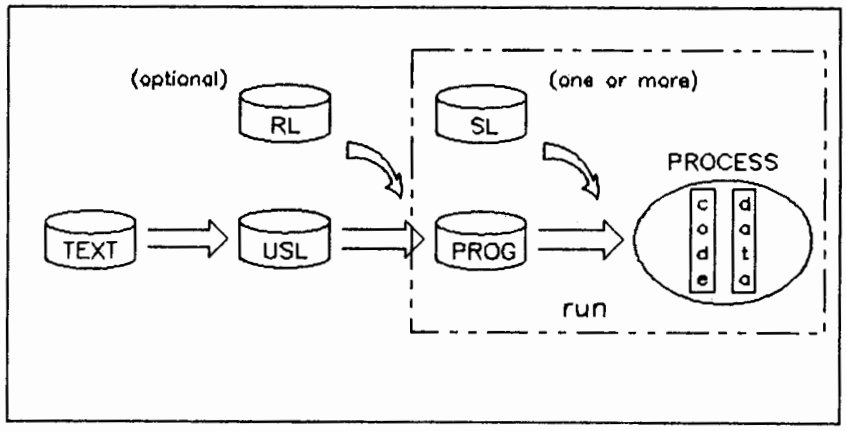


Figure 1-4. Execution

Segmented Libraries. The Segmented Library, or SL, is the third of the three libraries which may be accessed during the program development process. An already-existing system SL contains procedures applicable to all HP 3000 systems, such as the procedure for program termination. Segmenter subsystem commands also allow the programmer to create SLs from the material in one or more USLs, to add segments to an already-built SL, to purge segments within SLs, to list the contents of the currently-managed SL, and to copy the contents of one SL into another SL.

SLs and RLs share three important characteristics. Both:

- Contain procedures needed for program execution.
- Are created by the programmer, using Segmenter commands, from USL material.
- Are used to permit programs to share procedures.

However, the two libraries are intended for different purposes, so there are important differences in how and when they are constructed and used.

First, as its name indicates, the segmented library contains procedures in segmented form, not in RBM form as in the RL. When you use the segmenter to build an SL file, it uses the USL's segment association information to bind the required RBMs into code segments as it places them into the file. Procedures thus exist in the SL as runnable code segments.

SLs are intended for the storage of procedures with wider applicability than those placed in RLs. Examples are general utility procedures such as FOPEN, which are used in exactly the same way by every calling program. Because of this generality, the Segmenter does not have to allow for the unique combinations and uses of procedures that occur when a program calls for procedures from an RL. Instead, it can prepare, or segment, such general procedures at the moment you specify that you want them placed in an SL. Since the segmentation is not altered when different programs reference procedures in an SL, these segments may be shared concurrently by many programs.

While RLs contain code that can be copied and bound to each calling program file, SLs contain code that can be shared; that is, every program references the original version of the code. SL code is not copied or in any way combined with the program file. When a program calls an SL procedure, the procedure is read into memory from its place in disc storage. This method makes economical use of system resources, since each SL procedure exists only once in main memory and does not need to be brought in as part of every program file that requires it.

As is true of RLs, the various programming languages have different rules for what kinds of code may go into an SL. The general requirement, however, is that SLs can only contain non-global procedures. These are procedures that make no references to the global storage area of the data stack on which they will run: all parameters must be passed in explicitly. SL procedures may not read information about the stack's global storage area. Sharability is the primary feature of SL procedures. If they required knowledge of global storage, they wouldn't be sharable, since each program's global storage area is different.

While the Segmenter links RLs to the program file at preparation time, SLs don't enter the program development process until run time. They are used for the final resolution of external references, and their linkage to the program file is handled by the Loader, rather than the Segmenter. The Segmenter reserves space for called SL procedures in the Segment Transfer Table (STT) at preparation time, but the entries are actually made by the Loader when it searches the SL library files at run time.

You may build multiple SLs at each of three levels:

- SL.*group*.*acct*: The Group Library SL. It is the library of the group under which the program file is stored and is readable by any user who can access the group.
- SL.PUB.*acct*: The account's Public Library SL. It is the library of the public group of the account under which the program file is stored. It is readable by any user who can access the account.
- SL.PUB.SYS: The System Library SL. It is the library of the public group of the System account. It can be accessed by all users of the system.

If your program file is a permanent file, then *group* and *account* refer to the group and account where the program file resides, which may or may not be the same as your log-on group and account. However, if your program file is not a permanent file but is a job/session temporary or passed file, then *group* and *account* refer to your log-on group and account. The LOADPROC intrinsic, which you may use at times to dynamically load and unload SL procedures while your program is running, searches the SL libraries according to the user's log-on group and account, or the group and account where the program resides, as specified by the '*LIB*' parameter. See the discussion of the LOADPROC intrinsic in the MPE V Intrinsics Reference Manual (32033-90007) for more information.

The search order and the number of libraries searched depend on the *;LIB* parameter specified as part of either the :RUN or the :PREPRUN commands. The table below illustrates this relationship.

PARAMETER	SEARCH ORDER
;LIB=G	SL.*group*.*acct* SL.PUB.*acct* SL.PUB.SYS
;LIB=P	SL.PUB.*acct* SL.PUB.SYS
;LIB=S (or no LIB parameter)	SL.PUB.SYS

While you can search only one RL during any one program preparation process, you can search one SL at each of the levels (group, account, and system) during any one execution process.

The Code Segment Table (CST) and the Data Segment Table (DST) keep track of the loading and unloading of the necessary segments as program execution proceeds. Although their operation is completely invisible to users, some explanation of what they are and how they work will help complete your picture of the Segmenter.

The Code Segment Table. The CST is a main memory-resident table which is maintained by the MPE operating system. It contains a list of code segments that are being referenced by executing programs and keeps track of whether these segments are present in main memory or are out on disc. Entries in the CST are dynamically allocated by the operating system as programs are loaded and unloaded.

Although it is often referred to as a single table, the CST is actually divided, logically and physically, into two portions:

- The CST contains entries for coded segments in segmented libraries (including the system SL, which contains large chunks of the MPE operating system). Some entries or parts of entries in the CST also contain various service procedures for internal interrupts, external interrupts, system intrinsics, and library procedures.

- The CST Extension (CSTX) contains blocks of code segment entries, one block for each loaded program. Note that "loaded" as used here does not necessarily mean "loaded into main memory." Rather, it means that since the segment is part of an executing program, information about it has been loaded, or entered, into the CSTX. The segment itself may still be waiting out on disc.

The CST contains space for 2048 entries. The number of blocks of entries in the CSTX is configurable and can be set at system configuration time, but any one block has space for no more than 255 code segment entries. Thus, a program may contain as many as 255 code segments. One larger than that would not be runnable, since there would not be enough room in the CSTX to enter all the information the system needs in order to find and manage all the segments needed for execution.

Each segment required for the program receives a unique identifying number (the code segment number), and a CST entry which consists of a single 4-word descriptor providing the following information:

- Control information (such as whether the segment is present in main memory or is out on disc).

- The segment's length.

- The segment's disc (or starting) address if it is not present in main memory.

When the system needs to read the code segment into main memory, it uses this information to determine where to start reading and how much to read.

Since SLs are sharable, two different programs running at the same time may be using one particular SL code segment. If a CST entry has already been made for a segment, a new entry is not made. Instead, the existing entry is used.

The Data Segment Table. Like the Code Segment Table, the Data Segment Table is a main-memory resident table which is maintained by the MPE operating system. It contains a list of data segments currently in use by the operating system and user programs. Each segment receives a four-word entry recording its length, location, presence or absence in main memory, and other characteristics.

The length of the table is determined at system generation time. Entries in the DST are dynamically allocated by the system as programs are initiated or terminated, or special capability processes request or release additional data segments.

Summary: The Program Development Process.

Events which occur both before and after the Segmenter's part in program development affect the final segmentation of your program. To ensure good results, you need to keep the Segmenter's purpose and operation in mind as you begin developing your programs, even if you are not yet planning to control the process explicitly. For instance, what you compile into your USL will determine what is available to be placed into an SL, which in turn will affect what the loader will need to do at :RUN time to run your program, and how efficiently it will do so.

USING THE SEGMENTER

SECTION II

ACCESSING AND EXITING THE SEGMENTER

In an interactive session, you access the Segmenter implicitly whenever you use the MPE :PREPARE command, or when you use any of the combination commands such as:

:PREPRUN *object_code_filename* (prepares and executes in one step)

:BASICPREP *source_code_filename* (compiles and prepares in one step)

:BASICGO *source_code_filename* (compiles, prepares and executes in one step)

If you want to directly manipulate code yourself, you need to access the Segmenter explicitly. Enter the following in response to the MPE colon prompt:

:SEGMENTER [*listfile*]

where *listfile* is an ASCII file from the output set (formal designator SEGLIST) to which is written any listable output generated by Segmenter commands. By default, all such listings are sent to $STDLIST only. If you want to route your listings to the line printer, you must set up a file equation and use a file reference when you issue the :SEGMENTER command. For example:

```
:FILE ELIZ;DEV=LP
:SEGMENTER *ELIZ
```

The designator SEGLIST should not be used as the actual file designator, since it is the formal file designator.

If you decide you want a line printer listing after you are already in the Segmenter subsystem, you cannot use the BREAK key. Instead, you will have to -EXIT, make the file equation, and re-invoke the Segmenter using the file reference.

When you enter the :SEGMENTER command, the Segmenter responds with the following message and displays a dash prompt character:

```
HP32050A.03.00 SEGMENTER/3000 (c) HEWLETT PACKARD CO. 1986
-
```

You can now enter Segmenter commands. To end Segmenter operation, use the EXIT command:

-EXIT or -E

The system responds with the following message and returns you to the MPE colon prompt:

```
END OF SUBSYSTEM
  :
```

NOTE

You may also explicitly call the Segmenter and provide Segmenter commands in batch mode, but since the dash prompt character is supplied by the system as your job is running, it cannot be part of your input.

All examples in this manual were run in interactive mode and thus include the dash prompt. User input is underlined in all dialogue where it is necessary to distinguish the input from computer output. Editorial comments are enclosed in pairs of asterisks (**comment**).

MANIPULATING RBMs

Although the compilers and the Segmenter provide the required RBM management in most circumstances, you may want to take explicit control of RBMs for one of the following reasons:

- To use common code from another USL file.
- To change the number of code segments associated with a program.
- To create more efficient programs by relocating or purging RBMs or activating/deactivating a version of an RBM.

RBMs are units of source code which have been compiled into a User Subprogram Library (USL). More than one version of an RBM may be compiled into the same USL, and the various RBMs may be generated by different compilers. This "subprogram compatibility" is possible because all of the HP 3000 compilers output information in exactly the same format, generated by the same file system. The HP 3000 gives you two options for controlling RBMs: compiler control and Segmenter control.

Compiler Control of RBMs

You can embed control commands ($CONTROL) in your source code which direct the compiler to do the following:

- Assign specified RBMs to specified segments.
- Limit the size of the segments that can be generated.
- Tell the compiler to generate code only for the subprogram(s) supplied in the text file, suppressing generation of the outer block.
- Give the main (outer block) RBM the specified name.

Refer to Appendices A through F for information about the compiler $CONTROL commands, or to the reference manuals for the various languages if you need more detailed information.

To manipulate your code after the compiler has translated it into RBMs and placed the RBMs in a USL, you can use the Segmenter. Segmenter control provides you with different capabilities than compiler control; of the options listed above, the only points of overlap are the assignment of RBMs to specific segments and the specification of segment size.

Further, as the compiler options listed above indicate, the compiler's management of RBMs is limited to control of segmentation. The Segmenter gives you this capability and several others as well. While $CONTROL commands allow you to make some some preliminary decisions about segmentation, Segmenter commands provide you with fine-tuning capabilities, which you would be likely to want after you have run your program at least once and know where it could be improved.

Since the HP 3000 allows you to manipulate RBMs with the Segmenter as well as the compiler, you can do such fine tuning without having to recompile any code.

Managing RBMs With The Segmenter

The Segmenter identifies each RBM by its procedure name (the name you gave it in your source code), which can be fifteen alphanumeric characters with an initial alphabetic character. SPL also allows the apostrophe (') except as initial character. The Segmenter also identifies each RBM by a unique "version index."

THE VERSION INDEX. Since you can compile more than one version of an RBM into the same USL, several Segmenter commands allow you to specify which version of an RBM you wish to access. The index (also called index integer, version number, index parameter, or version index) is the value which lets you state which version you want the Segmenter to use. You can specify the index in all commands which allow you to specify the name of an RBM as a parameter. Its use is always optional: the Segmenter uses only one version of an RBM during any one preparation process, and the default is the most recent active version of the specified RBM. The index designations and defaults are as follows:

	INDEX=n	INDEX=0 (Default)
ALL COMMANDS EXCEPT CEASE AND USE	nth version, active or inactive	most recent active version
CEASE	nth version, active or inactive	most recent active version
USE	nth version, active or inactive	most recent inactive version

Study the -CEASE and -USE commands in Section IV of this manual for more information about why the index works differently with them than with the other Segmenter commands.

In the following example, the programmer has used the -LISTUSL command to verify the contents of a USL. The command lists the segment names and the RBMs associated with each. The USL in the printout contains only the segment SEG'. Four versions of the RBM ABC are associated with SEG'; the oldest version (the first one compiled into the USL) is known to the Segmenter by the index 4 and appears on the bottom in this listing. Note that although the Segmenter knows each RBM by its index, the index numbers do not actually appear on the listing. (Refer to figure 2-4 for an explanation of the fields in this listing.)

```
-USL MYUSL

-LISTUSL


USL FILE MYUSL.PUB.WHITMAN

  SEG'
     ABC                    16  P  A  C  N  R      **1**
     ABC                    16  P  I  C  N  R      **2**
     ABC                    16  P  I  C  N  R      **3**
     ABC                    16  P  I  C  N  R      **4**

  FILE SIZE          144000
  DIR. USED              70              INFO USED          130
  DIR. GARB.              0              INFO GARB.           0
  DIR. AVAIL.         14310              INFO AVAIL.     127050
```

Each RBM retains its particular index until an RBM is deleted from or added to the USL. Then the index is reset. With a deletion, all older versions (those with higher index numbers than the deleted RBM) receive a new index: their old value minus one. With an addition, the newest RBM receives the index 1 and all older versions receive their old value plus one. Thus, while each RBM's index number is unique, it is not fixed. An illustration will help you visualize each situation:

ABC **1**	ABC **1**		ABC **1**
ABC **2**	-------	ABC **1**	ABC **2; previously 1**
ABC **3**	ABC **2**	ABC **2**	ABC **3; previously 2**
ABC **4**	ABC **3**	ABC **3**	ABC **4; previously 3**
Before deletion of version 2.	After deletion of version 2.	Before addition of new version.	After addition of new version.

There will not always be a correspondence between the way the Segmenter accesses the RBM versions and the way they appear in your listing. In fact, the listing order and the access order will correspond exactly only until you move RBMs from one segment to another. When you do this, the RBM in effect takes its index number with it: that is, the third version is still the third version to the Segmenter even if it changes its position within the USL.

This next example shows the currently-managed USL before and after we have used the command

```
-NEWSEG SEGASK, ABC (1)
```

to move version 1 of RBM ABC from the segment SEG' to the segment SEGASK.

```
SEG'                          SEG'
   ABC **1 - to be moved**       ABC **2**
   ABC **2**                     XYZ **1**
   XYZ **1**                     XYZ **2**
   XYZ **2**
                              SEGASK
SEGASK                           ABC **1 - moved**
   ABC **4**                     ABC **4**
   ABC **5**                     ABC **5**
```

As you will see in the following discussion, the index significantly increases your power and flexibility in manipulating RBMs. However, to use it successfully you will need to keep your own records, either on-line or off-line, of what is in the various versions. You will also have to remember how the Segmenter uses the index and how the correspondence between the order of accessing and the order of your USL listings can change.

CONTROLLING AND ALTERING SEGMENTATION. You can override the Segmenter's default manipulations of RBMs, using Segmenter commands to:

- Control segment association.
- Purge RBMs.
- Activate/deactivate RBMs.
- Add new RBMs to a USL.
- Transfer RBMs from other USLs to the one you are currently using.

Note: the final item is discussed under Managing the USL.

Controlling Segment Association. The -NEWSEG command allows you to change the segment name associated with an RBM, thus assigning the RBM to a different code segment the next time it is prepared onto a program file. With the command

```
-NEWSEG SUB, TIMESTWO
```

you are associating the RBM TIMESTWO with the segment named SUB. Whatever segment association TIMESTWO had previously no longer exists, since each RBM can be associated with only one segment.

Purging RBMs. If you have revised a program, you may have completely changed a subroutine or removed it from your program altogether. You can use the Segmenter to purge one or more versions of the RBM, or the entire segment in which the RBM resides. With the command

```
-PURGERBM UNIT,MAIN,2
```

we are purging the second-newest version of the RBM MAIN .

With

```
-PURGERBM UNIT, MAIN
```

the Segmenter will purge the most recent active version of the RBM MAIN, since we specified no index.

If we specify

```
-PURGERBM SEGMENT, MAIN
```

the Segmenter will purge the entire segment in which the RBM MAIN resides.

If we input only

```
-PURGERBM,MAIN
```

without specifying either UNIT or SEGMENT, the Segmenter defaults to UNIT, thus shortening the amount of information you must type when you are managing RBMs as well as protecting you from accidentally purging more RBMs (or more versions of an RBM) than you intend to.

Figure 2-1 illustrates the use of the -PURGERBM command.

Activating/Deactivating RBMs. Segmenter commands allow you to activate or deactivate RBMs according to various "entry points." An entry point is any location in a routine to which control can be passed by another routine. The first executable statement of a main program or a procedure is an implicit entry point. Called the "primary entry point", it is the natural beginning point for execution. The allowance of multiple entry points permits you to begin execution of a main program or procedure at various secondary entry points.

Each RBM is identified to the operating system by the symbolic name, or label, of the primary entry point for the program unit which resides in the RBM. In figure 2-2, we have compiled a simple FORTRAN program and then used the -LISTUSL command to verify the contents of the USL. Since we did not specify a name for our main RBM when we compiled, the operating system gives it the default symbolic name MAIN'. The subroutine is identified as TIMESTWO, the specified name of its primary entry point. Note that the RBMs are further identified by their association with the segment SEG', to which they will belong after preparation.

An "activity bit" is associated with each entry point (the primary entry point and any secondary entry points). This bit determines whether the program unit currently can be entered at the corresponding entry point; that is, the bit determines whether you can start executing your program at that location. When a compiler writes an RBM to a USL file, all entry points are set to the active (entry allowed) state and the compiler deactivates any active versions of a particular RBM already in the USL. With the Segmenter, you can switch any activity bit in the USL to the inactive (entry disallowed) state and back again, if you wish. (See the -CEASE and -USE commands, Section IV.)

```
:SEGMENTER

HP32050A.03.00 SEGMENTER/3000 HEWLETT-PACKARD CO. 1986

-USL $OLDPASS
-LISTUSL

USL FILE $OLDPASS

 SEG'
    TIMESTWO        16  P  A  C  N  R
    MAIN'           32  OB A  C  N

 FILE SIZE       144000
 DIR. USED           70              INFO USED         130
 DIR. GARB.           0              INFO GARB.          0
 DIR. AVAIL.      14310              INFO AVAIL.    127050

-NEWSEG SUB,TIMESTWO
-PURGERBM MAIN'          **Add segment SUB.  Put RBM TIMESTWO
                           into SUB.  Delete RBM MAIN'**

-LISTUSL

USL FILE $OLDPASS
 SUB
    TIMESTWO        16  P  A  C  N  R
 SEG'            **Null segment**

 FILE SIZE       144000
 DIR. USED           76              INFO USED         130
 DIR. GARB.          31              INFO GARB.        112
 DIR. AVAIL.      14302              INFO. AVAIL.   127050

-PURGERBM SEGMENT, SEG'

-LISTUSL

USL FILE $OLDPASS

 SUB
    TIMESTWO             16  P  A  C  N  R

 FILE SIZE       144000
 DIR. USED           76              INFO USED         130
 DIR. GARB.          73              INFO GARB.        130
 DIR. AVAIL       14302              INFO AVAIL.    127050
```

Figure 2-1. Using the Segmenter -PURGERBM Command

```
:FORTRAN SEG8

 PAGE 0001   HP32102B.00.07


00001000  $CONTROL FREE
INTEGER A(4)
ACCEPT A
CALL TIMESTWO(A)
DISPLAY A
STOP
END

SUBROUTINE TIMESTWO(A1)
INTEGER A1(4)
DO 1 I=1,4
1 A1(I)=A1(I)*2
RETURN
END

****       GLOBAL STATISTICS        ****
      NO ERRORS,     NO WARNINGS    ****
TOTAL COMPILATION TIME    0:00:01
TOTAL ELAPSED TIME        0:00:03
 END OF COMPILE


:SEGMENTER

HP32050A.03.00 SEGMENTER/3000 HEWLETT-PACKARD CO. 1986
-USL $OLDPASS

-LISTUSL

USL FILE $OLDPASS

SEG'
   TIMESTWO           16  P  A  C  N  R
   MAIN'              32  OB A  C  N

FILE SIZE          144000
DIR. USED              70              INFO USED         130
DIR. GARB.             73              INFO. GARB.       130
DIR. AVAIL.         14302              INFO. AVAIL.   127050
```

Figure 2-2. Procedure Entry Points

The control given you over the activity/inactivity of entry points is a very important feature, since it allows you to associate many versions of an RBM with one segment, selectively deactivating those you don't want prepared into the program file. For large applications this is an invaluable aid, making costly recompiles unnecessary during test phases. Suppose, for example, that you make a change to a subprogram, compile it into a USL, and then prepare your program from this USL, having deactivated the first-version RBM. If the change turns out to be the source of a serious bug in the program, you could simply deactivate the new version and reactivate the previous version. Your program would be quickly returned to functional status without the need for time-consuming and costly recompilation.

In a similar way, the activate/deactivate control also increases your power and flexibility during the program design stages. You can construct alternative programs, varying your main program and one or more subprograms at a time, and test them without having to recompile the entire program each time you change something.

When the Segmenter prepares a program file from the USL, all RBMs having at least one active entry point are extracted from the USL for segmentation in the program file. Those associated with identical segment names are placed in the same segment. To permit the creation of a program file that can be executed properly, only one outer-block or main-program RBM can have active entry points, along with the RBMs for the subprograms or procedures. The presence in a USL of two active RBMs of the same name will cause a prepare failure. Thus, through your manipulations with the -CEASE and -USE commands, you could have several active RBMs of the same name in a USL file, but must de-activate all but one of the RBMs before trying to prepare the USL into a program file.

In figure 2-3, we use the -CEASE command to deactivate the two most recent versions of the RBM ABC, and the -USE command to activate a previous version. Then we use -LISTUSL to verify the contents of the USL.

Note that index information is not part of the data provided by the -LISTUSL command. You must remember the listing order and the associated index numbers.

With both the -CEASE and -USE commands, the index 0 is assumed if you do not specify an index number. For the -CEASE command, 0 indicates the most recent *active* version. For the -USE command, it indicates the most recent *inactive* version.

As with the -PURGERBM command, you can use -CEASE and -USE to activate/deactivate a single entry point within a specified RBM, all entry points in the specified RBM, or all entry points in all RBMs associated with the specified segment name. The default is the single entry point associated with the specified name.

Putting Additional RBMs in a USL. As you are designing and changing programs, you may need to put additional RBMs in a USL, either by copying already-existing code from another USL or adding new RBMs. The procedures for copying code are covered in the discussion following this one (Managing User Subprogram Libraries). To add new RBMs, you simply create a source file with the new subroutine(s) and compile it into the USL. With the compiler command

```
:FORTRAN SUB72, MASTRUSL
```

we instruct the compiler to compile the new subroutine SUB72 into the previously-created USL MASTRUSL.

```
:SEGMENTER

HP32050A.03.00 SEGMENTER/3000 (C) HEWLETT-PACKARD CO. 1986

-USL $OLDPASS
-LISTUSL

USL FILE $OLDPASS.PUB.SPL

SEG'
                                          **INDEX**
   ABC                 1  P  A  C  N  R **1, 0  Most recent active**
   ABC                 1  P  I  C  N  R **2**
   ABC                 1  P  I  C  N  R **3**
   ABC                 1  P  I  C  N  R **4**

FILE SIZE     144000(  620.  0)
DIR. USED        307(    1.107)     INFO USED          4(    0.  4)
DIR. GARB.         0(    0.  0)     INFO GARB.         0(    0.  0)
DIR. AVAIL.    14071(   60. 71)     INFO AVAIL.   127374(   535.174)

-CEASE ABC(1)
-USE ABC(2)
-LISTUSL

USL FILE $OLDPASS.PUB.SPL

SEG'
                                        **INDEX**
   ABC                 1  P  I  C  N  R **1**
   ABC                 1  P  A  C  N  R **2, 0  Most recent active**
   ABC                 1  P  I  C  N  R **3**
   ABC                 1  P  I  C  N  R **4**

FILE SIZE     144000(  620.  0)
DIR. USED        307(    1.107)     INFO USED          4(    0.  4)
DIR. GARB.         0(    0.  0)     INFO GARB.         0(    0.  0)
DIR. AVAIL.    14071(   60. 71)     INFO AVAIL.   127374(   535.174)
```

Figure 2-3. Using the Segmenter -CEASE and -USE Commands

If you wish, you can embed the compiler command SEGMENT in your source file to assign the new RBM to a specific segment:

```
$CONTROL SEGMENT=SEG2
    SUBROUTINE SUB72
      .
      .
      .
```

Or you could allow the compiler defaults to operate and later use the Segmenter to alter segmentation, if necessary.

MANAGING USER SUBPROGRAM LIBRARY FILES (USLs).

A USL is one of the three procedure libraries used in the program development process. It is the file used for compiler output. Most frequently, you will use the USL as input for the Segmenter to use in preparing a program file. However, you may also use it for the following purposes:

- To share code from another USL or program.
- To solve code management problems which may occur if you exceed the size limitation established for a USL.
- As an aid in program design and code management. The USL eliminates time-consuming recompilation when you are testing various versions of a procedure or changing code storage methods.
- As a basis for constructing relocatable libraries (RLs).
- As a basis for constructing segmented libraries (SLs).

The last two items are discussed later in this section.

Invoking The USL

A command you will use often as you manage USLs with the Segmenter is

-USL *filereference*

This command identifies the USL specified by *filereference* as the "currently-managed", or "currently-referenced", USL. These interchangeable terms mean the specified USL is the one to which the Segmenter will apply any commands you give which have USL as a parameter, and will do so until another -USL command is entered.

Listing The USL

Another command you will frequently use to verify the results of your code manipulations is

-LISTUSL [*segmentname*]

With this command you can list all or part of the contents of the currently managed USL. If you specify a segment name, only that segment is listed. The output is written on the file designated in the *listfile* parameter of the MPE :SEGMENTER command, or on $STDLIST if the *listfile* parameter is omitted.

In figure 2-4, the Segmenter prints information about all segments in the USL, since we didn't specify a particular segment name. Significant entries are indicated with item numbers (**number**), which are explained following the listing. All numbers appearing in the listing are octal values.

```
-USL SEARCHUSL                **1**
-LISTUSL

USL FILE SEARCHUSL.SEGMENT.SUB3000

WRITESEG          **2**              **4** **5** **6** **7** **8** **9**
   WRITENUMSONLY **3**          16  P  A  C  N  R
SEARCHSEG             .
   SEARCHLINE          .     27  P  A  C  N  R
   SEARCHLINE            27  P  I  C  N  R
ASKSEG
   ASKFORMAT             15  P  I  C  N  R    **4**   **5**
SEG'
   WRITENUMSONLY         16  P  I  C  N  R    **6**   **7**
   WRITENUMSONLY         16  P  I  C  N  R    **8**   **9**
   OB'                  255  OB A  C  N
   ASKCHAR               17  P  A  C  N  R
   ASKNAME               13  P  A  C  N  R

FILE SIZE   2000(     10.  0)                              **10**
DIR. USED    574(      2.174)**11**   INFO USED     1132(   4.132)
DIR. GARB.     0(      0.  0)         INFO GARB.       0(   0.  0)  **12**
DIR. AVAIL.    4(      0.  4)         INFO. AVAIL.    46(   0. 46)  **13**
```

Figure 2-4. Using the Segmenter -LISTUSL Command

ITEM NO	MEANING
1	Name of USL.
2	Segment name.
3	RBMs associated with segment.
4	Size of RBMs in words (octal).
5	Entry point type, where: P =Procedure primary entry point. SP =Secondary entry point for procedure. OB =Outer block primary entry point. SO =Secondary entry point for outer block. BD =Block data entry point. In =Interrupt procedure entry point (n is the interrupt procedure type, ranging from 0 through 2). CP =COBOL secondary entry point.
6	Active/Inactive bit.
7	Callable/Uncallable bit.
8	Not privileged/Privileged bit.
9	Reveal/Hide flag.
10	File size: number of words in file (overhead + directory + info) followed by number of records in file (in parentheses).
11	Directory used + avail. = total directory space in words and in records (in parentheses).
12	Garbage: unusable portion "used" space in words and in records (in parentheses).
13	Info used + avail. = total info space in words and in records (in parentheses).

Figure 2-4. Using the Segmenter -LISTUSL Command (Continued)

Building New USLs With The Segmenter

Although new USLs are usually built automatically by compilers and managed with the Segmenter, you can use the Segmenter to both build and manage USLs. You will need this capability when you want to copy RBMs from another USL using the -COPY command and do not want to place the copied RBMs into a USL containing other material.

You might also wish to create an empty USL into which you can compile code after you have exited the Segmenter subsystem. The command

```
-BUILDUSL MYUSL,200,1
```

creates a new USL file with space for 200 records and one disc extent.

A new USL built with the Segmenter becomes the currently-managed USL. An important point is that, unlike compiler-built USL files, those built with the Segmenter are "job/session-temporary"; that is, they will be lost when you log off unless you take special action to save them. In figure 2-5, we use the Segmenter to build the session-temporary USL MYUSL, and then use the MPE :SAVE command to make MYUSL permanent.

Copying RBMs

You may often find that you would like to use a subroutine from one of your programs in another program. With the Segmenter, you can copy RBMs and segments directly from one USL to another. Without this capability, you would have to create a source file containing the procedure(s) and compile it into the desired USL.

To use this capability, specify the source USL (the one you wish to copy *from*) with the -AUXUSL command:

-AUXUSL *filefrom*

The destination USL (the one you wish to copy *to*) is the currently-managed USL, which you specify with the -USL command:

-USL *fileto*

Note that while the destination USL may be one you just created with the -BUILDUSL command, the source or auxiliary USL must already exist. The -LISTAUX command, which functions like the -LISTUSL command, allows you to view the contents of the auxiliary USL so you can select the segments or RBMs you wish to copy.

Once you have specified a source USL and a destination USL, you can use the -COPY command to copy a single RBM and all associated entry points, or all RBMs associated with a particular segment name, from the auxiliary USL to your currently-managed USL. In figure 2-6, the -LISTAUX command used after the transfer shows that the -COPY operation leaves the auxiliary USL file intact; that is, -COPY transfers a duplicate of the specified material without moving or altering the original.

```
:SEGMENTER

HP32050A.03.00 SEGMENTER/3000 (C) HEWLETT-PACKARD CO. 1986

-BUILDUSL MYUSL,100,10
-LISTUSL

USL FILE MYUSL.PUB.SPL

FILE SIZE  31000(  144.  0)
DIR. USED    200(    1.  0)  INFO              0(   0.   4)
DIR. GARB.     0(    0.  0)  INFO GARB.        0(   0.   0)
DIR. AVAIL. 3000(   14.  0)  INFO AVAIL.   25600( 127.   0)

-EXIT

END OF SUBSYSTEM

:SAVE MYUSL
:LISTF MYUSL,2
ACCOUNT=   SPL         GROUP=   PUB

FILENAME   CODE  ------------LOGICAL RECORD------------  ----SPACE----
                   SIZE  TYP         EOF       LIMIT R/B   SECTORS #X MX

MYUSL      USL     128W  FB           14         100   1         22  2 10
```

Figure 2-5. Building a USL with the Segmenter -BUILD Command and Saving It as a Permanent File

```
:SEGMENTER

HP32050A.03.00 SEGMENTER/3000 (C) HEWLETT-PACKARD CO. 1986

-USL MYUSL
-AUXUSL SEARCHOLD
-LISTAUX

USL FILE SEARCHOLD.PUB.SPL

SUB
   TIMESTWO            16 P  A  C  N  R
SEG'
   MAIN'               32 P  A  C  N  R
SEG2
   INPUT               32 P  A  C  N  R
   GOFOR               16 P  A  C  N  R
   GOFOR               16 P  I  C  N  R

                .
                .
                .
-COPY MAIN'          **Copy the RBM MAIN' and the segment
-COPY SEGMENT, SUB     SUB from $SEARCHOLD to MYUSL**
-LISTUSL

USL FILE MYUSL.PUB.SPL

SUB
   TIMESTWO            16  P  A  C  N  R
SEG'
   MAIN'               32  P  A  C  N  R

-LISTAUX

SUB
   TIMESTWO            16 P  A  C  N  R
SEG'
   MAIN'               32 P  A  C  N  R
SEG2
   INPUT               32 P  A  C  N  R
   GOFOR               16 P  A  C  N  R
   GOFOR               16 P  I  C  N  R
                .
                .
                .
```

Figure 2-6. Using the Segmenter -COPY, -AUXUSL, and -LISTAUX Commands

Copying An Entire USL

As you are developing your programs, you may receive an error message stating that the space in the currently-managed USL is exhausted. The Segmenter may give you the message when, for example, you are copying RBMs from an auxiliary USL to your currently-managed USL. Compilers have an equivalent message which they may send when you are compiling code into a USL.

Two more copy commands, -COPYUSL and -CLEANUSL, are available for you to use when you need to copy an entire USL to a different file because of such space problems.

-COPYUSL copies all of the currently-managed USL to a new USL file, while giving you control over the size of the new USL. Assume that SEARUSL is the currently-managed USL. With the command

```
-COPYUSL 100, SEARUSL5
```

we are instructing the Segmenter to copy all of MYUSL into a new USL called SEARUSL5, giving SEARUSL5 100% more space than the space needed to hold SEARUSL5's current contents. For example, if the information in SEARUSL occupied 400 records, SEARUSL5 would get 400 plus 100% of 400, or 800 total.

If we specify only

```
-COPYUSL 100
```

The Segmenter automatically creates a new file, purges the currently-managed file after copying the material in it, and gives its name to the new USL.

In either case, the new USL becomes the currently-managed USL. Figure 2-7 demonstrates the operation of this command.

The -CLEANUSL command, which calls the CLEANUSL intrinsic, copies only the active RBMs to a new USL file, cleaning out any inactive entries and garbage. Since with this command you have no control over the new file's size, the new file is the same size as the old file. However, it has more usable space since the inactive RBMs are not copied.

Assume that SEARUSL is the currently-managed USL. With the command

```
-CLEANUSL SEARUSL7
```

the Segmenter will copy the active RBMs from SEARUSL to a new USL called SEARUSL7, which becomes the currently-managed USL. (See figure 2-8.) Note in figure 2-8 that the segment name ASKSEG, which had no active RBMs associated with it, is still present. Because of the possibility that you may inadvertently purge files with these two commands by omitting a filename for the new USL, you must have Save Files capability to use them.

Other Methods of Obtaining More USL Space

A USL file with insufficient space cannot simply be FCOPY'd to a physically larger file to make room for more entries. Look again at the information provided in the response to the -LISTUSL command in figure 2-4. Notice that maintenance information (such as the size of the USL file, the amount of space used, and the amount of space available), which the Segmenter must have to manage the USL, resides within the USL file itself. Since FCOPY is not aware of the internal structure of a USL file, it cannot make any modification to the size or structure of the file, but can only make a copy identical to the original.

Recompiling into a larger USL is the only other way (besides -CLEANUSL and -COPYUSL) to obtain more USL space. Within the Segmenter subsystem, use the -BUILDUSL command to create a larger USL than the one in which space is exhausted. Then recompile the material from the old USL into the new, larger USL. Note that this alternative requires you to have your original source code available. If you don't, you will have to rewrite it.

Managing USLs: Special Considerations

The -COPY command will allow you to copy both active and inactive RBMs with the same name as some already in the currently-managed USL. Unlike the compilers, the Segmenter does not inactivate already-present RBMs of a particular name when it copies other RBMs of the same name into a USL. Further, it has no way to distinguish between different RBMs with the same name. Instead, it indexes all RBMs with the same name as if the RBMs themselves were the same. (See the discussion of the version index.) Since the Segmenter can use only one active RBM of a particular name per preparation, the presence of two active RBMs of the same name (even though they contain different procedures) will cause a prepare failure.

To avoid this problem, make your procedure names unique. Then make a habit of using the -LISTAUX and -LISTUSL commands before you -COPY the RBMs. If you see potential duplication problems, you will have to change the procedure's name in your source code and recompile your program. (All other programs that use the procedure will also have to be recompiled, and re-prepared as well.)

If you -COPY in another active RBM of the same name which contains another version of the *same* procedure, the solution is simpler: use the Segmenter -CEASE command to inactivate one of the versions before you prepare your program file.

Another point to be aware of when you are managing USLs is that, although similar in function, the Segmenter subsystem -COPY commands work differently than the MPE operating system :FCOPY commands, and the two types are not compatible. The MPE file system is concerned with the file label, while the Segmenter is concerned with contents and does not read MPE file labels. Further, MPE creates files of a different size than those built by the Segmenter, so that the Segmenter could not copy such files even if it could read the labels.

You can avoid management problems by using only the Segmenter -COPY commands with files you have built and want to continue to manage with the Segmenter.

```
-USL TUSL1
-LISTUSL

USL FILE TUSL1.HSU.KSE
        .
        .
        .
FILE SIZE    144000(  620.  0)
DIR. USED       716(    3.116)   INFO USED       225(    1. 25)
DIR. GARB.        0(    0.  0)   INFO GARB.        0(    0.  0)
DIR. AVAIL.   13462(   56. 62)   INFO AVAIL.  127153(  534.153)


-COPYUSL 100,TUSL7        **Segmenter copies all of TUSL1 to new file
                            TUSL7, giving TUSL7 100% more than thespace presently
                            needed for the contents of TUSL1.**


-LISTUSL

USL FILE TUSL7.HSU.KSE
        .
        .
        .
FILE SIZE      2600(   13.  0)
DIR. USED       716(    3.116)   INFO USED       225(    1. 25)
DIR. GARB.        0(    0.  0)   INFO GARB.        0(    0.  0)
DIR. AVAIL.     662(    3. 62)   INFO AVAIL.     553(    2.153)


-USL TUSL1
-COPYUSL 0,TUSL8          **Segmenter copies all of TUSL1 to new file
                            TUSL8, giving TUSL8 no more than the mimimum amount
                            of space presently needed to hold the present
                            contents of TUSL1.**


-LISTUSL
USL FILE TUSL8.HSU.KSE
        .
        .
        .
FILE SIZE      1400(    6.  0)
DIR. USED       716(    3.116)   INFO USED       225(  1. 25)
DIR. GARB.        0(    0.  0)   INFO GARB.        0(  0.  0)
DIR. AVAIL.      62(    0. 62)   INFO AVAIL.     153(  0.153)


          **Note the difference in the amount of free space in TUSL7 vs. TUSL8.**
```

Figure 2-7. Using the Segmenter -COPYUSL Command

```
-USL SEARUSL
-LISTUSL

USL FILE SEARUSL.SEGMENT.SUB3000

SEARCHSEG
   SEARCHLINE          27  P  A  C  N  R
   SEARCHLINE          27  P  I  C  N  R
ASKSEG
   ASKFORMAT           15  P  I  C  N  R
SEG'
   WRITENUMSONLY       16  P  I  C  N  R
   WRITENUMSONLY       16  P  I  C  N  R
   OB'                255  OB A  C  N
   WRITENUMSONLY       16  P  A  C  N  R
   ASKCHAR             17  P  A  C  N  R
   ASKNAME             13  P  A  C  N  R

FILE SIZE      2000(   10.  0)
DIR. USED       563(    2.163)    INFO USED     1132(  4    .132)
DIR. GARB.        0(    0.  0)    INFO GARB.       0(  0    .  0)
DIR. AVAIL.      15(    0. 15)    INFO. AVAIL.    46(  0    . 46)

-CLEANUSL SEARUSL7

-LISTUSL

USL FILE SEARUSL7.SEGMENT.SUB3000

SEARCHSEG
   SEARCHLINE          27  P  A  C  N  R       **SEARUSL7, the new file,
ASKSEG                                          contains only the active
SEG'                                            RBMs of SEARUSL.  Note
   OB'                255  OB  A  C  N  R       that the segment ASKSEG,
   WRITESNUMSONLY      16  P  A  C  N  R       which contained no active
   ASKCHAR             17  P  A  C  N  R       RBMs in SEARUSL7, is
   ASKNAME             13  P  A  C  N  R       still present.**

FILE SIZE      2000(  10.   0)
DIR. USED       435(   2.  35)    INFO USED     1016(   4. 16)
DIR. GARB.        0(   0.   0)    INFO GARB        0(   0.  0)
DIR. AVAIL.     143(   0.143)    INFO AVAIL.     162(   0.162)

              **Note that SEARUSL7, the new USL file,
                has more free space than SEARUSL.**
```

Figure 2-8. Using the Segmenter -CLEANUSL Command

Using MPE Intrinsics To Manipulate USL Files

MPE Intrinsics provide you with an alternate means of altering the size of USLs. Using them, you can perform the following functions programmatically:

- Initialize a buffer for a USL file to the empty state with the INITUSLF intrinsic.
- Move the information block in a USL file with the ADJUSTUSLF intrinsic.
- Change the length of a USL file with the EXPANDUSLF intrinsic.

Note that for each of these intrinsics you must have FOPENed the file so that you can provide the file number (the parameter *uslf*), rather than the name, of the USL you wish to manipulate. Refer to the MPE V Intrinsics Manual (32033-90007) for more information.

The above intrinsics are most useful to programmers writing compilers. As you have seen from the preceding discussion, the Segmenter gives you powerful tools for controlling USL size and contents. For ordinary purposes, you will probably prefer to use the Segmenter to manage your USLs.

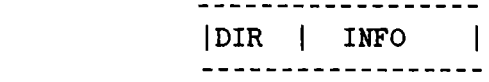
INITIALIZING A BUFFER FOR A USL FILE TO AN EMPTY STATE. The INITUSLF intrinsic initializes an already-existing USL to an empty state. Within a program, you could write a call to this intrinsic as follows:

```
ERRNUM:=INITUSLF(uslf, buf);
```

where *uslf* is the file number, obtained from the FOPEN intrinsic, of the file you wish to empty. This intrinsic actually clears the area of the USL which contains pointers to the remainder can provide the file number of the material. Once this "roadmap" has been cleared, the specified USL appears empty to the compiler, which overlays any existing material in the USL with your new source code.

Remember that you can use the Segmenter command -BUILDUSL when you need an empty USL. One situation where you might prefer the intrinsic is with a system where memory space is so severely limited that re-using files, rather than building new ones, is efficient and perhaps necessary. However, this is an unlikely circumstance.

CHANGING THE DIRECTORY BLOCK/INFORMATION BLOCK SIZE IN A USL FILE. In each USL file, separate areas are reserved for the directory, which keeps track of which RBMs are in the USL; and for information (the RBMs themselves).



As long as space is available, compilers expand the directory as required. Although some compilers (e.g., FORTRAN) also automatically copy USL material to a new, larger USL file when more information room is needed, you may receive one of the following messages while you are manipulating code:

AVAILABLE DIRECTORY SPACE EXHAUSTED

or

AVAILABLE INFO SPACE EXHAUSTED

With the ADJUSTUSLF intrinsic, you can move the information block forward or backward, thereby increasing or decreasing, respectively, the space available for the directory block. For example, the ADJUSTUSLF intrinsic call could be written as follows:

errnum:=ADJUSTUSLF(*uslfnum*,-80)

In the example below, we are are using this intrinsic call to move the information block backward in the USL file, increasing its space by 80 records and decreasing the directory block space by the same amount.

With the call

errnum:=ADJUSTUSLF(*uslfnum*,80)

we are moving the information block forward in the USL file, decreasing its space by 80 records and increasing the directory block space by the same amount.

CHANGING THE SIZE OF A USL FILE. While ADJUSTUSLF changes the relative amount of space available for code, the EXPANDUSLF intrinsic changes the actual size of the USL. The EXPANDUSLF intrinsic call could be written as follows:

filenum:=EXPANDUSLF(*uslfnum*,*80*);

This intrinsic creates a new USL file whose length is 80 records longer than the USL file specified by the parameter *uslf*. The specified USL file is copied to the new USL file with the same file name and the specified file is deleted. The file number for the new USL is returned to NEWFNUM.

Remember that three Segmenter commands (-BUILDUSL, -COPYUSL, and -CLEANUSL) will suit most of your needs for controlling the size of your USLs as well as or better than the intrinsics.

Preparing A Program File

When you have finished manipulating your USL, you are ready to prepare the active RBMs residing in it into a program file. You can do this in two ways:

- With the MPE command

  :PREP *uslfile*, *programfile*

  after exiting the Segmenter subsystem.

- With the Segmenter command

  -PREPARE *programfile*

  while still in the Segmenter subsystem.

The operation of the two commands is the same. In the syntax for the Segmenter -PREPARE command, however, we don't need the USL filename, since it is already defined: it is the currently-managed USL.

Although most of the time you will probably use the MPE :PREP command to prepare your program files, the Segmenter -PREPARE command is very helpful if you want to manipulate a USL file before preparing it. -PREPARE gives you the freedom to make changes and to experiment with different segment structures: you can resegment your USL material, -PREPARE it into a program file, resegment, -PREPARE into another program file, etc., without exiting the Segmenter subsystem.

With either :PREP or -PREPARE, you can specify a PMAP (which produces the locations of procedures and their entry points within a given code segment) or FPMAP (which stores an internal copy of the PMAP information in the program file). You should always specify FPMAP if you are going to use APS/3000 or HPToolset. The -SETFPMAP command allows you to invoke FPMAPing at the system or session level. The -SHOW command displays the current status of the system and session FPMAP flags. See Appendix G for further discussion and an example.

Figure 2-9 illustrates the use of -PREPARE. Note that if the program file is not an already-existing program file, the Segmenter sets up a new temporary file. Use the MPE :SAVE command before logging off if you want to keep a temporary program file.

```
:FORTRAN SEG8,$NEWPASS,$NULL

****   NO ERRORS,     NO WARNINGS     ****
TOTAL COMPILATION TIME  0:00:01              **Compile source
TOTAL ELAPSED TIME      0:00:03                code into USL
                                               file.**

:SEGMENTER

HP32050A.03.00 SEGMENTER/3000 (C) HEWLETT-PACKARD CO. 1986

-USL $OLDPASS
-LISTUSL

USL FILE $OLDPASS

 SEG'
    TIMESTWO      16  P  A  C  N  R
    MAIN'         32  OB A  C  N

 FILE SIZE     377600( 1777.  0)
 DIR. USED        270(    1. 70)   INFO USED        130(     0.130)
 DIR. GARB.         0(    0.  0)   INFO GARB.         0(     0.  0)
 DIR. AVAIL.    37510(  176.110)   INFO AVAIL.   337450(  1576. 50)

-NEWSEG TIMESTWOSEG,TIMESTWO
-LISTUSL

USL FILE $OLDPASS                    **Optionally reorganize
                                       USL file**
 TIMESTWOSEG
    TIMESTWO             16  P  A  C  N  R
 SEG'
    MAIN'                32  OB A  C  N

 FILE SIZE     377600( 1777.  0)
 DIR. USED        302(    1.102)   INFO USED        130(    0.130)
 DIR. GARB.         0(    0.  0)   INFO GARB         0(    0.  0)
 DIR. AVAIL.    37476(  176. 76)   INFO AVAIL.  337450( 1576. 50)

 -PREPARE SEG243P
-EXIT
END OF SUBSYSTEM            **New program file prepared
                              from currently-referenced
 :RUN SEG243P                 USL file.**
      .
      .
      .
 END OF PROGRAM
 :SAVE SEG243P               **The program file is saved
                               as a temporary file by the
                               Segmenter.  Here, the MPE
                               :SAVE command is used to
                               save it as a permanent file.**
```

Figure 2-9. Using the Segmenter -PREPARE Command

MANAGING RELOCATABLE LIBRARY FILES (RLs)

The Relocatable Library is the second of the three libraries which may be part of the program development process. It contains procedures, in RBM form, that are needed for program execution, and its use is optional. The Segmenter accesses the RL at preparation time to resolve references within the code it is preparing.

Procedures that are likely to be used with some frequency by more than one programmer can be stored in an RL. RBMs in any USL may reference the RBMs in an RL. At preparation time, the Segmenter searches the specified RL (one per preparation) and resolves the references by copying the appropriate RBMs and binding them to the program file as a single segment, referred to as the RL segment. Although the RBMs in the RL are sharable, the RL segment is unique; that is, not sharable by any other program.

Note that each of your programs can reference RBMs residing in only one RL, since the Segmenter will search only one RL during a particular preparation. References to any additional RLs would remain unresolved, and your program would be unrunnable.

Invoking The RL

You can invoke the desired RL by including its name with the ;RL= parameter in either the Segmenter -PREPARE command or the MPE :PREPARE command.

A command you will use often as you manage RLs with the Segmenter is

```
-RL filereference
```

This command identifies the RL specified by *filereference* as the currently-managed or currently-referenced RL. These interchangeable terms mean that the specified RL is the one to which the Segmenter will apply any RL modification commands until another -RL *filereference* command is entered or until a -BUILDRL command creates a new RL, thus effectively superseding the first -RL *filereference* command.

Listing The RL

Another command you will frequently use verifies the results of your RL manipulations. It is

```
-LISTRL
```

This command lists all procedure entry points and external references within the currently-managed RL. Figure 2-10 is an example of the use of the -LISTRL command and the output produced. Significant entries are indicated with item numbers keyed to the discussion following. All numbers appearing on the listing are octal values.

Because RLs are written in a special format, Segmenter commands are required to create and change them. Once an RL is created, however, it can be referenced by other MPE commands and intrinsics, just like any other standard, formatted file. For example, you can purge the entire file with :PURGE, rename it with :RENAME, or read it with the FREAD intrinsic.

Besides its role in accessing the RL at preparation time, the Segmenter allows you to directly manage your RLs. Using it, you can:

- Build RLs.
- Add RBMs to an RL.
- Purge RBMs within an RL.

```
:SEGMENTER

HP32050A.03.00 SEGMENTER/3000 (C) HEWLETT-PACKARD CO. 1986

-LISTRL

RL FILE RLPROC.PUB.WILBUR        **1**

ENTRY POINTS      CHECK    ADR    LOC    NUM    CODE    INFO

                           **4**          **6**          **8**

START  **2**        3      54    400      1     577     677
                   **3**          **5**          **7**

EXTERNALS         CHECK    ADR    LOC


TFORM'              0
FMTINIT'            0
IIO'                0
SIO                 0

**9**       **10**  **11**  **12**

USED                       1500          AVAILABLE          27300
                           **13**                           **14**

-EXIT

END OF SUBSYSTEM
```

Figure 2-10. Using the Segmenter -LISTRL Command

ITEM NO.	MEANING
1	The name of the RL file (*filename.groupname.accountname*).
2	The procedure entry point name.
3	The parameter checking-level for the entry point.
4	The entry point address (relative displacement within the code module).
5	The RL file address of the procedure information block.
6	The number of entry-points in the procedure. (If this field is blank, the entry point field (Item 2) names a secondary entry point If this field contains a number, the entry point name field contains a primary entry point name.
7	The length of the code module, in words. (If this field is blank, the entry point name field names a secondary entry point. If this field contains a number, the entry point name field contains a primary entry point name.)
8	The length of the procedure information block, in words. (If this field is blank, the entry-Point name field names a secondary entry point. If this field contains a number, the entry point name field contains a primary entry point name.
9	The names of the external references.
10	The parameter checking-level for the external reference.
11	The entry point address of the external procedure. If this field is blank, the procedure is external to the RL. If this field contains a number, the procedure is not external to the RL but is called by (and thus external to) some other procedure in the RL.
12	The RL file address of the procedure information block. If this field is blank, the procedure is external to the RL. If this field contains a number, the procedure is not external to the RL but is called by (and thus external to) some other procedure in the RL.
13	The number of words presently used in the RL file.
14	The number of words presently available in the RL file.

Figure 2-10. Using the Segmenter -LISTRL Command (Continued).

Building RLs

The Segmenter creates RLs from RBMs that have previously been compiled into a User Subprogram Library (USL). With the Segmenter command

```
-BUILDRL MYRL.MYGROUP.MYACCOUNT,100,10
```

we create an RL named MYRL with 100 records and 10 extents. As indicated by the use of our :LISTF,2 command in figure 2-11, RLs are always saved as permanent files by the Segmenter. Therefore, you must have SAVE access to the group to which you assign the RL (it is assigned by default to the logon group and account). The RL created with the -BUILDRL command automatically becomes the currently-managed RL. The Segmenter initializes the RL files it builds with information it will need each time it uses the RL. Although you can also create RLs with the MPE :BUILD command, such RLs will not be properly initialized and you will therefore be unable to manage them with the Segmenter. They would be useful only if you planned to use FCOPY with them to make an exact copy of an already-usable RL. As a general rule, therefore, build your RLs with the Segmenter to be sure you will be able to manage them successfully.

```
:SEGMENTER

HP32050A.03.00 SEGMENTER/3000 (C) HEWLETT-PACKARD CO. 1986

-BUILDRL MYRL,100,10  **Builds and initializes RL**

-LISTRL

RL FILE MYRL.CLASS.MAL    **RL becomes currently-referenced RL**

  ENTRY POINTS

          **The RL is empty**

  EXTERNALS

USED              400        AVAILABLE        30400
-EXIT

END OF SUBSYSTEM

:LISTF MYRL,2   **The new RL is saved as a permanent file by the Segmenter**

ACCOUNT=   MAL     GROUP=  CLASS

FILENAME   CODE ------------LOGICAL RECORD------------ ----SPACE----
                  SIZE  TYP        EOF      LIMIT R/B  SECTORS #X MX

MYRL       RL     128W  FB           2        100   1       11  1 10
```

Figure 2-11. Using the Segmenter -BUILDRL Command

Adding Procedures To An RL

Once you have built an RL, you can add a procedure to it by copying the RBM containing the procedure from the currently-managed USL to the RL. You must have WRITE access to the designated RL. Suppose we have specified USL1 as our currently-managed USL and MYRL (the empty RL which we created in figure 2-11) as our currently-managed RL. With the command

```
-ADDRL START (2)
```

we are copying the second-latest version of the RBM START from USL1 to MYRL. Figure 2-12 illustrates the use of the command and of -LISTRL to verify the results.

```
:SEGMENTER

HP32050A.03.00 SEGMENTER/3000 (C) HEWLETT-PACKARD CO. 1986

-USL USL1

-LISTUSL

USL FILE USL1.CLASS.MAL
SEG1
   START              577  P  I  C  N  R
   START              577  P  A  C  N  R

FILE SIZE    377600( 1777.  0)
DIR. USED        42(    0. 46)  INFO. USED         24(    0.   24)
DIR. GARB.        0(    0.  0)  INFO. GARB          0(    0.    0)
DIR. AVAIL.   37510(  176.110)  INFO. AVAIL.   337450( 1576.   50)

-RL MYRL

-ADDRL START (2)

-LISTRL

RL FILE MYRL.CLASS.MAL

ENTRY POINTS  CHECK   ADR   LOC   NUM   CODE   INFO

START           3      54   400     1    577    674

EXTERNALS     CHECK   ADR   LOC

TFORM'          0
FMTINIT'        0
IIO'            0
SIO'            0
USED              1500            AVAILABLE        27300
```

Figure 2-12. Using the Segmenter -ADDRL Command

Purging RL Entries

You may find that you no longer use a particular RBM, or that you would like to modify it by removing an entry point within it. You can deal with either of these situations without exiting the Segmenter subsystem or rewriting and recompiling the procedure from which the RBM was created. With the command

```
-PURGERL ENTRY,TIMESTWO
```

we delete the entry point TIMESTWO from the RBM in which it resides. If we specify

```
-PURGERL UNIT,TIMESTHREE
```

the Segmenter will delete the entire RBM whose primary entry point name is TIMESTHREE and any secondary entry points within TIMESTHREE.

When we delete the last, or only, entry point from an RBM, the entire RBM is deleted. In figure 2-13, we would have deleted the entire RBM named TIMESFOUR even if we had specified

```
-PURGERL ENTRY,TIMESFOUR
```

Since there is only one entry point (the primary entry point which gives TIMESFOUR its name), deleting it deletes the entire RBM.

```
:SEGMENTER

HP32050A.03.00  SEGMENTER/3000 (C) HEWLETT-PACKARD CO. 1986

-RL MYRL

-LISTRL

ENTRY POINTS    CHECK    ADR    LOC    NUM    CODE    INFO

TIMESFOUR         3        0    500      1      16      30

EXTERNALS       CHECK    ADR    LOC

USED                1340       AVAILABLE        27440

-PURGERL TIMESFOUR

-LISTRL

RL FILE MYRL.CLASS.MAL

ENTRY POINTS          CHECK    ADR    LOC    NUM    CODE    INFO

EXTERNALS             CHECK    ADR    LOC

USED                  400        AVAILABLE        28380
```

Figure 2-13. Using the Segmenter -PURGERL Command

Recall that purging procedures from a USL "garbages", or renders unusable, some of the space which the purged code had occupied. (You could recover the lost space by using either -CLEANUSL or -COPYUSL.) In contrast, when you use the -PURGERL command, the space becomes immediately reusable.

If you want to delete an entire RL, rather than selectively deleting entry points or RBMs within the RL you can use the MPE :PURGE command. :PURGE wipes out the file itself as well as all the material within it. See the MPE V Commands Reference Manual (32033-90006).

Managing RLs: Special Considerations

Following are several things you should keep in mind when working with RLs.

PREPARATION. As mentioned previously, the Segmenter will search only one RL per preparation. You must be careful when you are writing a program or adding material to a USL to include references to only one RL. Otherwise, unresolved external references will remain after preparation and loading, and the program will be unrunnable.

RL SIZE. Unlike USLs, RLs are not directly expandable: there are no commands or intrinsics to increase the size of an RL. Further, there are no commands to move RBMs, either individually or collectively, from one RL to another. The only way to obtain more RL space is to build another, larger RL and place the same material in it. You can do this easily with the -ADDRL command if you have available the USL from which you originally built the RL. Otherwise, you would have to recompile (and possibly also rewrite!) your original source code.

CODE SEGMENT SIZE. Since all requested RLs are placed in a single segment for addition to the program file at preparation time, that segment may become larger than the configured maximum size for code segments. You have two options when this happens:

- Increase the configured code segment size.
- Reduce the number of RBMs referenced.

The first option is not the best one because it increases the demands on disc storage and, more importantly, on main memory.

You can implement the second option in several ways, each of which involves modification of your code on some level. You cannot simply move RBMs from an RL back to a USL because the Segmenter is designed for one-way transfer only. This means that a library cannot communicate directly with another library which enters the program development process earlier. Figure 2-14 illustrates the point at which the three procedure libraries enter the program development process. It indicates that the Segmenter can move material directly from a USL to an RL or an SL, since that is a case of forward referencing, but not from a USL back to an RL, from an SL back to a USL, or between an RL and an SL.

One way to reduce the RL RBMs is to rewrite your source code to eliminate references to RL RBMs, and then recompile it.

Another choice is to move the RBMs in question to the USL being prepared. You can do this easily if you took the RL RBMs from a USL which is still available. You can designate that original USL as

the auxiliary USL and move the desired RBMs to your currently-managed USL, making references to the RL unnecessary.

If the USL from which you originally built the RL is not available but you do have your source code, you can recompile it into a USL and use that USL to prepare your program, skipping the intermediate step of moving the RBMs to an RL. You could move the RBMs to an SL, so that they will not need to be added to the program file in a code segment but instead will be resolved by the Loader at run time. Remember that both RLs and SLs are built from USLs; the Segmenter cannot move material directly between these two types of libraries. If you have available the USL from which you originally built the RL, you can make this change easily using the -ADDSL (and perhaps also the -BUILDSL) commands. If not, you will have to rewrite your source code (if you have not saved it), recompile it into a USL, and then place in an SL the RBMs you originally had in the RL.

DUPLICATE ENTRY-POINTS. Since each name in a library file must be unique, you cannot have more than one active entry point of a given name in an RL file. Therefore, you cannot add to an RL an RBM containing active entry point names which already exist in the RL. The Segmenter will disallow such duplication, sending you the following error message: ENTRY POINT ALREADY DEFINED.

Make a habit of using the -LISTUSL and -LISTRL commands before you try to move RBMs from a USL to an RL. If you see potential duplication problems, you can:

- Leave the RBMs in the USL and prepare them from there (as part of your object code).
- Place the RBMs into an SL instead of the intended RL.
- Rewrite the source code, changing the procedure name, and then move the newly-named RBM into the RL.
- Inactivate one of the entry points if it performs the same function as its duplicate, as well as having the same name.

REFERENCING ORDER. Because the Segmenter is designed for forward referencing only, procedures in an RL cannot reference procedures in a program file. Functionally and chronologically, the program file enters the program development process before the RL does: the USL material is prepared into the program file, and then the specified RL is searched to satisfy external references within the program file. Therefore, a reference from an RL to a program file would be a backward reference (see figure 2-14). For example, if subroutine A in the RL calls subroutine B in the program file, B is considered an unresolved external for A when the RL segment is added to the program file. The Segmenter has already dealt with the program file and will not look at it again. At run time, the Loader will search the Segmented Libraries (SLs) looking for a routine called B. If it found one, it would almost certainly not be the same as the B in the program file which you intended to use.

RL FILE PROTECTION. Whenever you specify an RL as your currently-managed RL, the Segmenter assumes you are going to modify it and automatically calls the FLOCK intrinsic to unconditionally lock the file, protecting its integrity while it is in use. Any other -PREPARE (or :PREPARE) request specifying that RL will result in a -PREPARE (or :PREPARE) failure, and any other request to access the RL for management purposes waits until the first user is finished and the

Segmenter removes the FLOCK, which it does automatically when you access another RL or when you exit the subsystem.

This protective mechanism cannot be overridden. However, unless you have many programmers heavily using the same RLs, locking is unlikely to cause any inconvenience for you. A good general

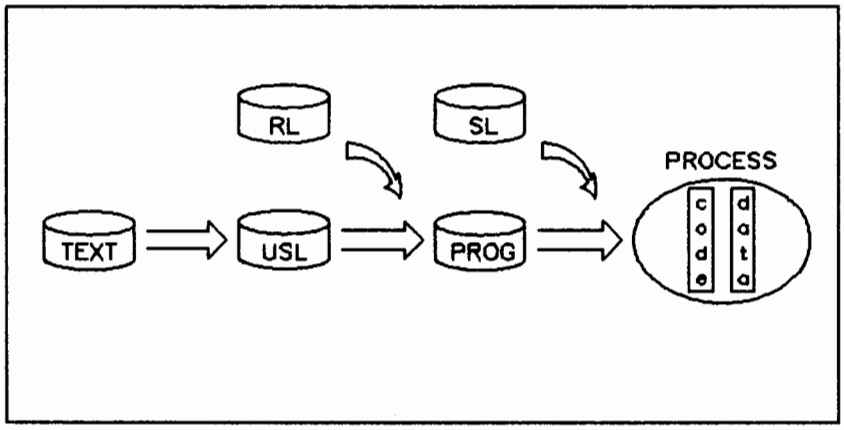


Figure 2-14. Procedure Library Access Order

rule is to access an RL for as short a time as possible to avoid interfering with its use by another programmer.

RETAINING SOURCE CODE OR USLs. Efficiently solving problems which may arise as you are managing RLs requires you to have available the USL or the source code from which you originally constructed the RL. If you save the USL, you avoid the extra step of recompiling. However, if you save the source code, you can easily make changes in your code as well as changing your method of storing shared code. (You cannot alter code in a USL.)

Make it a habit to retain either USLs or source code, particularly if you anticipate that you will heavily use the Segmenter to manage code, and if you foresee:

- Writing programs that will include many RL references, making it likely that your RL segment will exceed the allowable code segment size.
- Heavy usage of a particular RL by yourself and other programmers, making it likely that at some point you will exceed the size limitations established when you built the RL and need to build another, larger one.
- Heavy RL usage which could result in duplicate entry point names and thus the need to modify code or find an alternate means of storing and referencing it.

MANAGING THE SEGMENTED LIBRARY

The Segmented Library (SL) is the third of the three libraries which may be part of the program development process. Like the Relocatable Library (RL), it contains procedures needed for program execution and its use is optional. The Loader accesses the SL to resolve any external references still remaining at run time.

General utility routines (with wider applicability than the procedures stored in RLs) are stored in the SL as prepared code segments. SL procedures must be non-global, meaning they are so general that they require no knowledge of the data stack on which they will run. Because they are so general, SL code segments are completely sharable: every calling program references the same original code. Since the SL code is never bound to any program file, the segmentation never changes; thus, an SL segment can be in use by more than one program at a time.

At run time, the Loader searches one SL at each of three levels: group, account, and system. The Loader references the requested code segments; that is, it copies them into main memory from their place on disc rather than combining them with the program file.

The Loader can only search SLs actually named "SL". Thus, although your SLs can have any name you wish for storage and manipulation purposes, you must use the MPE command :RENAME to change the name of each of the SLs you want the Loader to search at run time.

Invoking the SL

You invoke the desired SL(s) by specifying a library parameter with the MPE :RUN command or one of the combination commands such as :FORTGO. Refer to a discussion of the library search order in Section I. If you don't supply a ;LIB= parameter, the Loader will search only the System Library. If you're uncertain which SL you have stored a procedure in, specifying ;LIB=G will instruct the Loader to search all three libraries. With the ;LMAP parameter as part of the :RUN command, you can obtain a loader map. This is a listing providing information about the external references for a loaded program, including the type of segmented library in which the external procedure resides. See Appendix G for a sample LMAP listing.

Although the Loader accesses SLs, they are written in a special format and must therefore be created and managed with the Segmenter. Once an SL is created, however, it can be referenced by other MPE commands and intrinsics just like any other standard, formatted file. For example, you can purge the entire file with the :PURGE command, change its name with :RENAME, or read it with the FREAD intrinsic.

A command you will use often as you manage SLs with the Segmenter is

-SL *filereference*

This command identifies the SL specified by *filereference* as the currently-managed or currently-referenced SL. These interchangeable terms mean that the specified SL is the one to which the Segmenter will apply all SL-modification commands until another -SL *filereference* command is entered or until a -BUILDSL command is used, thus effectively superseding the -SL command.

Listing the SL

Another command you will frequently use verifies the results of your SL manipulations. The command

```
                [segmentnumber ]
-LISTSL [[SEGMENT,]segmentname ]
        [ENTRY,procedurename ]
```

lists all entry points and external references in the SL segment. If no *segmentname* is specified, all segments in the SL are listed. If you specify a procedure, the segment containing that procedure is listed. Figure 2-13 is an example of the use of the -LISTSL command and the output produced. Significant entries are indicated with numbers keyed to the discussion following. All numbers appearing in the listing are octal values.

```
:SEGMENTER

HP32050A.03.00 SEGMENTER/3000 (C) HEWLETT-PACKARD CO. 1986

-USL USL1

-LISTUSL

USL FILE USL1.PUB.TECHPUBS

 SEG'
    DISP                24   P  A  C  N  R

 FILE SIZE       377600( 1777.  0)
 DIR. USED           42(    0. 46)  INFO. USED          24(    0.   24)
 DIR. GARB.           0(    0.  0)  INFO. GARB           0(    0.    0)
 DIR. AVAIL.      37510(  176.110)  INFO. AVAIL.    337450( 1576.  50)

 -BUILDSL DPSL,100,1

-ADDSL SEG'

-LISTSL    **1**

 SL FILE DPSL.PUB.TECHPUBS     **2**

 SEGMENT    0    SEG'                        LENGTH        30
          **3** **4**                         **5**

    ENTRY POINTS     CHECK  CAL  STT  ADR
    DISP               3     C     1    0
    **6**            **7** **8**  **9** **10**

    EXTERNALS        CHECK  STT  ADR

    **11**          **12** **13** **14**

1  **15**
NUMBER OF SEGMENTS IN SL: 1 DECIMAL

SL FREE SPACE SUMMARY

USED              3574400(     16762.   0)     **16**
AVAILABLE        12717400(     53476.   0)     **17**

 -EXIT

 END OF SUBSYSTEM
```

Figure 2-15. Using the Segmenter -LISTSL Command

Item No.	Meaning
1	Lists the currently-managed SL file.
2	The name of the SL file (filename.groupname.accountname)
3	The logical segment number (relative to this SL).
4	The segment name.
5	The segment length, in words.
6	The entry point name in the segment.
7	The parameter checking-level of the entry point.
8	The callability of the entry point, where C=Callable U=Uncallable
9	The Segment Transfer Table (STT) number of the entry point.
10	The entry point address (relative displacement within the segment).
11	External references (blank in this case).
12	The parameter checking-level of external references (blank in this case).
13	The Segment Transfer Table (STT) number of the external references (blank in this case).
14	If a number appears in this field, it is a (logical) segment number and indicates that this reference has been bound to an entry point within the SL.
15	A bit list of the segments referenced within the SL by this segment. The left-most bit corresponds to the first (0) logical segment number. For each bit, 1=Segment referenced. 0=Segment not referenced.
16	The number of words presently used in the SL file.
17	The number of words not used, at present, in the SL file.

Figure 2-15. Using the Segmenter -LISTSL Command (Continued)

With the Segmenter, you can:

- Build SLs.
- Add procedures to an SL.
- Delete entry points or entire segments.
- Copy the SL to a new, larger SL.
- Copy the SL to a new SL, cleaning out fragments of free space in each of the segments.
- Protect certain entry points within an SL from general use.

Building SLs

The Segmenter creates SLs from RBMs that have previously been compiled into a User Subprogram Library (USL). With the command

```
-BUILDSL MYSL.PUB.MYACCT,100,10
```

we create an account SL named MYSL with room for 100 records and 10 extents. As indicated by the use of the :LISTF,2 command in figure 2-16, SLs are always saved as permanent files by the Segmenter; therefore, you must have SAVE access to the group and account to which you assign the SL (it is assigned by default to the logon group and account). If your program references an SL, the SL must reside in the same group and account as the program file. Remember that the search order of the libraries is determined by the program file's group and account, not the logon group and account. An SL residing in some group and account other than the program file would therefore not be found when the Loader performed its library search. The SL created with the -BUILDSL command becomes the currently-managed SL.

```
-BUILDSL MYSL,100,10   **Builds and initializes SL**

-LISTSL

SL FILE MYSL.CLASS.MAL    **SL becomes currently-referenced SL**
   **The SL is empty**
 USED                37510(  176.110)        AVAILABLE          57200(  275.   0)

 :END OF SUBSYSTEM
 :LISTF MYSL,2  **The new SL is saved as a permanent file by the Segmenter**

 ACCOUNT=   MAL     GROUP=  CLASS

 FILENAME   CODE ------------LOGICAL RECORD------------ ----SPACE----
                  SIZE  TYP        EOF      LIMIT R/B  SECTORS #X MX

 MYSL       SL     128W  FB           2        100   1       11  1 10
```

Figure 2-16. Using the Segmenter -BUILDSL Command

The Segmenter initializes each SL file it builds with information it will need each time it uses the SL. Although you can also create SLs with the MPE :BUILD command, such SLs will not be properly initialized and you will therefore be unable to manage them with the Segmenter. They would be useful only if you planned to use FCOPY with them to make an exact copy of an already-usable SL. As a general rule, therefore, build your SLs with the Segmenter to be sure you will be able to manage them successfully .

Adding Procedures To An SL

Once you have built an SL, you can add procedures to it by preparing the segment with which the procedures are associated and placing it in the SL. In figure 2-17 we have specified PROCUSL as our currently-managed USL and MYSL as the currently-managed SL. With the command

```
-ADDSL SUBPROGX',PMAP
```

we are preparing all procedures associated with the segment SUBPROGX and adding the prepared code segment to MYSL. We have also requested a PMAP of the prepared segment (see Appendix G), so we can make a note of the segment number of our added code for future debugging.

Note that the Segmenter adds prepared code segments, not individual RBMs, to the SL. The SL code segments are ready for execution, except for their own external references. These are resolved from other code segments within the SL or from other SLs.

References in code segments within the same SL are considered resolved. Although such a reference appears as unresolved in the PMAP for the single code segment which makes the reference, the -LISTSL output (which shows all segments in the SL) shows the reference to be resolved. (See figure 2-18.)

The -ADDSL command allows you to specify only segment names, not individual procedure names. The only way you can add a single procedure is if it resides by itself in a segment. Conversely, you may have a procedure residing alone which you would like to add to your SL along with several other procedures. With Segmenter commands you can manipulate your USL to produce whatever procedure arrangement you wish before creating the SL. In figure 2-19, we wish to add a COBOL subprogram to an SL but we first want to combine the initialization segment and the procedure division segment before preparing the subprogram.

```
-USL PROCUSL  **Designate PROCUSL file**

-LISTUSL  **Display the USL file contents.**

USL FILE PROCUSL.GROUP1.PAYACCT

SUBPROGX

   SUBPROGX'          510  P  A  C  N  R
   SUBPROGX           174  P  A  C  N  R
   SUBPROGX'S             CP A  C  N  R
   .
   .
   .
-SL MYSL  **Designate SL**

-ADDSL SUBPROGX,PMAP  **Prepare segment and add it to the SL file.  Request
                        a PMAP so you know the segment number for future
                        debugging.**

SUBPROGX            0  **The segment number.**

   NAME              STT  CODE  ENTRY  SEG
   SUBPROGX'           1     0      0
   SUBPROGX            2   510    510
   SUBPROGX'S          3   510    701
   SEGMENT LENGTH          710

-LISTSL  **Display the SL file contents.**

SL FILE MYSL.GROUP1.PAYACCT

SEGMENT   0   SUBPROGX       LENGTH   710

   ENTRY POINTS      CHECK  CAL  STT  ADR
   SUBPROGX            2     C     2   510
   SUBPROGX'S          1     C     3   701
   SUBPROGX'           0     C     1     0

   EXTERNALS         CHECK  STT  SEG

    **No externals referenced.**
1   **Bit list of segments referenced.**

NUMBER OF SEGMENTS IN SL:  1 DECIMAL

SL FREE SPACE SUMMARY :

USED           3574400(      16762.   0)
AVAILABLE     12717400(      53476.   0)

-EXIT
```

Figure 2-17. Using the Segmenter -ADDSL Command

```
-ADDSL SEG', PMAP

SEG'                0
   NAME              STT  CODE  ENTRY  SEG
   TIMESTWO           1     0      0
   SEGMENT LENGTH           20

-ADDSL SHOWINTRNL,PMAP  **Add code segments to SL file.**

SHOWINTRNL          1
    NAME             STT  CODE  ENTRY  SEG
    TIMESSIX          1     0      0
    TIMESTWO          3                  ?  **TIMESTWO is an unresolved external
    TIMESTHREE        2    12     12          for this segment.**
    SEGMENT LENGTH                  34

-ADDSL SHOWEXTRNL
-LISTSL

SL FILE SL.CLASS.MAL

SEGMENT   0 SEG'        LENGTH   20

   ENTRY POINTS   CHECK  CAL  STT  ADR
   TIMESTWO         3     C    1     0

   EXTERNALS      CHECK  STT  SEG
100

SEGMENT   1 SHOWINTRNL  LENGTH   34

   ENTRY POINTS    CHECK  CAL  STT  ADR
   TIMESTHREE        3     C    2    12
   TIMESSIX          3     C    1     0

   EXTERNALS       CHECK  STT  SEG
   TIMESTWO          3     3    0   **TIMESTWO is a resolved external when
110                                    we consider the entire SL.**

SEGMENT   2 SHOWEXTRNL  LENGTH   20
   ENTRY POINTS     CHECK  CAL  STT  ADR
   TIMESEIGHT        3     C    1     0

   EXTERNALS       CHECK  STT  SEG
   TIMESFOUR         3     3    ?   **TIMESFOUR is an unresolved external: it is
   TIMESTWO          3     2    0     not present in any segment in this SL.**

NUMBER OF SEGMENTS IN SL:  3 DECIMAL

SL FREE SPACE SUMMARY :

USED           3574400(      16762.   0)
AVAILABLE     12717400(      53476.   0)
```

Figure 2-18. External References in an SL

```
-USL PROCUSL  **Designate PROCUSL file.**

-NEWSEG SUBPROGX,SUBPROGX'  **Change the segment name associated with the
                              SUBPROGX' RBM to the segment name SUBPROGX.**

-LISTUSL

USL FILE PROCUSL.GROUP1.PAYACCT

SUBPROGX'
SUBPROGX
   SUBPROGX'          510  P  A  C  N  R
   SUBPROGX           174  P  A  C  N  R
   SUBPROGX'S           CP A  C     R
   .
   .
-PURGERBM SEGMENT, SUBPROGX'  **You can delete the SUBPROGX' segment
                                from the USL with the PURGERBM command.**

-LISTUSL

USL FILE PROCUSL.GROUP1.PAYACCT

SUBPROGX    **Now there is only one segment in the USL file.**
   SUBPROGX'          510  P  A  C  N  R
   SUBPROGX           174  P  A  C  N  R
   SUBPROGX'S           CP A  C  N  R
   .
   .
-SL MYSL  **Designate SL**

-ADDSL SUBPROGX,PMAP   **Prepare segment and add it to the SL file.  Request
                         a PMAP so you know the segment number
                         for future debugging.**

SUBPROGX            0      **The segment number.**
   NAME              STT  CODE  ENTRY  SEG
   SUBPROGX'           1     0      0
   SUBPROGX            2   510    510
   SUBPROGX'S          3   510    701
   SEGMENT LENGTH        710
```

Figure 2-19. Altering Code Segmentation Before Preparation into an SL

```
-LISTSL **Display the SL file contents.**

SL FILE MYSL.GROUP1.PAYACCT

SEGMENT   0   SUBPROGX        LENGTH   710

   ENTRY POINTS       CHECK  CAL  STT  ADR
   SUBPROGX             2     C    2   510
   SUBPROGX'S           1     C    3   701
   SUBPROGX'            0     C    1     0

   EXTERNALS          CHECK  STT  SEG

1      **Bit list of segments referenced.**

NUMBER OF SEGMENTS IN SL:  131 DECIMAL

SL FREE SPACE SUMMARY :

USED           3574400(      16762.   0)
AVAILABLE     12717400(      53476.   0)

-EXIT
```

Figure 2-19. Altering Code Segmentation Before Preparation into an SL (Continued)

Purging SL Entries

If you no longer use a particular code segment in an SL or would like to hide an entry point in a code segment, you can do so with the Segmenter subsystem. The command:

```
-PURGESL SEGMENT,segmentname
```

removes a code segment from the SL file. The command:

```
-PURGESL ENTRY,entrypointname
```

removes an entry from the directory of the SL file.

When an entry point is removed, it is accessible only to procedures that reside in the same code segment; programs run with the SL or code segments added to the SL are unable to find a purged entry point. Note that if you purge an entry point and it is the only entry point in a given code segment, the entire code segment is purged from the SL. For example, the command:

```
-PURGESL ENTRY, DISP
```

purges the entry point DISP from the currently managed SL. Since DISP is the only entry point for this code segment, the entire segment is purged from the SL (see Figure 2-20).

```
:SEGMENTER
HP32050A.03.00 SEGMENTER/3000 (C) HEWLETT-PACKARD CO.  1986

 -SL DPSL

 -LISTSL

 SL FILE DPSL.PUB.TECHPUBS

 SEGMENT   0 SEG'            LENGTH    30
    ENTRY POINTS    CHECK  CAL  STT  ADR
    DISP              3     C    1    0

    EXTERNALS       CHECK  STT  SEG

NUMBER OF SEGMENTS IN SL:  131 DECIMAL

SL FREE SPACE SUMMARY :

USED           3574400(      16762.   0)
AVAILABLE     12717400(      53476.   0)


 -PURGESL ENTRY,DISP

 -LISTSL

 SL FILE DPSL.PUB.TECHPUBS

NUMBER OF SEGMENTS IN SL:  131 DECIMAL

SL FREE SPACE SUMMARY :

USED           3574400(      16762.   0)
AVAILABLE     12717400(      53476.   0)


 -EXIT

 END OF SUBSYSTEM
 :
```

Figure 2-20. Using the Segmenter -PURGESL Command

Recall that purging procedures from a USL "garbages", or renders unusable, some of the space which the purged code had occupied. (You could recover the lost space by using either -CLEANUSL or -COPYUSL.) In contrast, when you use the -PURGESL command, the space becomes reusable after the Segmenter performs another binding operation (thinks was performed when the code was put into the SL) to re-establish the links among segments. This re-binding will occur as soon as you use any of the following three commands: -EXIT, -LISTSL, -SL.

If you want to remove an entire SL file, rather than selectively deleting entry points or segments while preserving the space within the file, you can use the MPE :PURGE command. :PURGE wipes out the file itself as well as all the material in it.

Copying an Entire SL

If you and other programmers heavily use an SL, you may at some time receive a message from the Segmenter stating that the space limits established with the -BUILDSL command have been exhausted. Like RLs, SLs are not directly expandable: there are no equivalents of the intrinsics used to adjust the space in the USL. However, for SLs (although not for RLs!) the Segmenter provides two -COPY commands to help resolve space problems.

-COPYSL copies all of the currently-managed SL to a new SL file, while giving you control over the size of the new SL. Assume that MYSL is the currently managed SL. With the command

```
-COPYSL 90,NEWSL
```

we are instructing the Segmenter to copy the contents of MYSL to a new SL called NEWSL, giving NEWSL 90% more space than the space needed to hold MYSL's current contents. For example, if MYSL had space for 400 records, NEWSL would get space for 400 plus 90% of 400, or 760 total.

If we specify only

```
-COPYSL 90
```

the Segmenter automatically creates a new file, purges the currently-managed file after copying the material in it, and gives its name to the new SL.

In either case, the new SL becomes the currently-managed SL.

The -CLEANSL command copies the contents of the currently-managed SL file to a new SL file, eliminating any fragments of free space in each of the segments. Since with this command you have no control over the new file's size, the new file is the same size as the old file. However, it has more space since unused areas are not copied.

Assume that MYSL is the currently-managed SL. With the command

```
-CLEANSL NEATSL
```

the Segmenter will copy the contents of MYSL to a new SL called NEATSL, eliminating any fragments of free space in each of MYSL's segments.

There is a possibility that you may inadvertently purge files with these two commands by omitting a filename for the new SL. To prevent this, be sure you have SAVE FILES capability.

Protecting SL Entry Points

The Segmenter allows you to selectively protect SL entry points from general use, making them accessible only to other procedures residing in the same SL. The -HIDE and -REVEAL commands accomplish this when they are invoked for RBMs within the currently-managed USL. When the RBMs are added to an SL, they are hidden or revealed, depending upon which command you have used.

In addition to the activity bit associated with each entry point in an RBM, there is also an internal flag that determines whether or not that entry point will be placed in the SL directory - and thus made available to other segments within the SL and to general users - when the RBM is segmented and added to an SL. The -HIDE command sets the flag on; -REVEAL sets it off. For SPL, the OPTION INTERNAL statement in the declarations of applicable procedures does an implicit HIDE at compile time.

Since the internal flag must either be set from within your source code (for SPL) or used on RBMs in the currently-managed USL, changing the flag from one condition to another involves:

- Purging the procedure from the SL and then rewriting it, recompiling it, and re-adding the required RBMS (with the flag for each set the way you want it) into the SL.

- Using the -HIDE or -REVEAL command on the RBM in the USL and then re-adding it into the SL, skipping the steps of rewriting and recompiling.

The second is clearly the much simpler option. Note that you must have saved the USL from which you originally created the SL in order to use this option.

As an example of how the HIDE/REVEAL facility can be used, suppose you are writing a complicated utility that you want to add to an SL. You write the routine as a set of procedures to be placed in the same code segment, but want only entry points of certain procedures made available to other users; you do not want them to use the entry points of the support procedures for the main routine. You can effectively hide these private entry points from other users (and other segments) by using OPTION INTERNAL if you are programming within SPL, or by setting the internal flag on with the -HIDE command.

The HIDE/REVEAL facility can also be useful if you want to keep two procedures of the same name in an SL. Since only one procedure receives a directory entry, the Segmenter does not "see" the second and will not prevent you from adding it to the SL. The main purpose, however, is to protect entry points from general use.

Managing SLs: Special Considerations

Following are several things you should keep in mind when working with SLs.

CHANGING CODE STORAGE METHODS. At some point as you are managing your code libraries, you may decide you would prefer not to store certain code in an SL. If you have not saved either your source code or your USL, you will have to rewrite and recompile the code before deciding on another option for managing the procedure modules which had been in the SL. As is true of RLs, you cannot simply move code from an SL back to a USL. The Segmenter is designed for one-way referencing, which means that a library cannot communicate directly with another library which enters the program development process earlier. The Segmenter can move code directly from a USL to an RL or an SL, since that is a case of forward referencing, but not from an RL back to a USL or from an SL back to a USL. Further, both RLs and SLs are created from USLs; the Segmenter cannot move material back and forth between these two types of libraries.

DUPLICATE ENTRY POINTS. Since each name in a library file must be unique, you cannot have more than one active entry point of a given name in an SL file. If you try to add a segment containing two active entry points of the same name to an SL, the Segmenter will disallow the operation, sending you the following error message: DUPLICATE ACTIVE ENTRY POINTS.

Nor can you add to an SL a segment containing an active entry point name which already exists in the SL. In this case, the Segmenter will disallow the operation and you will receive the following error message: ENTRY POINT ALREADY DEFINED.

Make a habit of using the -LISTUSL and -LISTSL commands before you try to move code from a USL to an SL. If you see potential duplication problems, you can:

- Leave the code in the USL and prepare it from there along with the rest of your object code. If necessary, use the -AUXUSL and -COPY commands to transfer the code from the USL it resides in to the one that contains your object code.

- Rewrite the source code, changing the procedure names, and then move the newly-named code to the SL.

- Inactivate one of the entry points if it performs the same function as its duplicate, as well as having the same name.

COPY COMMANDS. Although similar in function, the Segmenter subsystem -COPY command work differently than the MPE operating system :FCOPY commands, and the two types are not compatible. The file system is concerned with the file label, while the Segmenter is concerned with file contents and does not read MPE file labels. Further, the MPE file system creates files of a different size than those built by the Segmenter, so that the Segmenter could not copy such files even if it could read the labels.

You can avoid management problems by using only the Segmenter -COPY commands with files you have built and want to continue managing with the Segmenter.

REFERENCING ORDER. The Segmenter's forward-referencing design prevents SL procedures from referencing program file procedures. The Loader would consider any such calls to be unresolved externals for the code segment or the SL. It would search the other SLs specified with the RUN command looking for a procedure of the specified name. If it found one, it would almost certainly not be the same as the one you intended to use from your program file. If the Loader found no such procedure, your program would contain an unresolved external and would be unrunnable.

SL FILE PROTECTION. When you specify an SL as your currently-managed SL, the Segmenter assumes you are going to modify the SL and unconditionally FLOCKs the file, protecting its integrity while it is in use. The Segmenter automatically removes the FLOCK when you access another SL or when you exit the subsystem. Any other request to access the same SL waits until the first user is finished and the Segmenter FUNLOCKs the file.

Any process that executes the RUN command against an FLOCKed SL also waits until the SL is FUNLOCKed by the Segmenter. This is because the Loader also FLOCKs SL files to prevent their modification while they are in use by a particular process. If the file is already FLOCKed by the Segmenter or by some other process, the Loader suspends until the file is unlocked and can be locked again for the Loader's purposes. While the Loader is waiting for the SL to be unlocked, any other processes queued on the Loader have to wait also. Thus, the Loader as well as the particular SL are unavailable to users.

A particularly important consequence of this protective mechanism is that other users may be prevented from normal operation during the time that you are accessing SL.PUB.SYS. Not only does virtually every program have at least one external reference which must be resolved from this SL (the system procedure TERMINATE), but many other system functions also require access to SL.PUB.SYS. Since you cannot override this protective mechanism, it is important for you to avoid potential interference with other users by accessing an SL, particularly SL.PUB.SYS, for as short a time as possible.

RETAINING SOURCE CODE OR USLs. Note that efficiently solving problems which may arise as you are managing SLs requires you to have available the USL or the source code from which you originally constructed the SL. If you save the USL, you avoid the extra step of recompiling. However, if you save the source code, you can easily make changes in your code as well as changing your method of storing shared code. (You cannot alter code in a USL.)

Make it a habit to retain either USLs or source code, particularly if you anticipate that you will heavily use the Segmenter to manage code, and if you forsee heavy SL usage which could:

- Increase the possibility of duplicate segment names and thus the need to modify code or find an alternate means of storing it.
- Repeatedly exhaust the available CST entries.

STRATEGIES FOR USING THE SEGMENTER

SECTION III

When we discuss strategies for using the Segmenter, we have two major areas of concern:

- Managing libraries so that heavily-used code can be shared among programmers.
- Controlling segmentation or altering the contents of a segment.

A very strong caution applies whether you will use the Segmenter heavily or just occasionally for either of these purposes: keep either your source code or your USLs. If you have not done so and then need to alter code or change your storage strategy, you will have to rewrite your code. Attempting to reconstruct the code as it existed before is difficult and time-consuming.

Many experienced Segmenter users have found that saving source code rather than USLs provides more flexibility, since you can then change code or alter storage arrangements. If you have saved only the USL, you are limited to changes in storage arrangements, since you cannot alter the code in a USL. Keeping a copy of all your source code requires a relatively small investment in disc space or in magnetic tapes for a significant return in efficiency.

ALTERNATIVES FOR STORING SHARED CODE

Different languages differ both in the amount of control they give the programmer (SPL provides relatively more, with, for example, all external procedure calls made explicitly) and in their conventions for what code may be placed in libraries. However, given any language, various alternatives are available for you to use when you want to store code that will be shared among programmers. Although there are arguments for and against each method, there are few technical constraints. The method you choose is more a function of your account structure, control procedures, and personal preference than a technical issue.

Table 3-1 presents the advantages and disadvantages of five methods for storing shared code.

RLs vs. SLs

If you have eliminated the other alternatives for storing shared code and are trying to decide whether to use a Relocatable Library (RL) or a Segmented Library (SL), consider the following:

- Does the routine require access to global storage? If so, it cannot be stored in an SL.
- Is the procedure likely to be changed or enhanced? If so, what would be the impact of having to re-prepare all the programs that use the procedure?
- If you put the procedure in an SL (other than the system SL), will you need it in other groups and accounts? If you do, will it be a problem for you to maintain several copies of the procedure?

Table 3-2 summarizes for you some of the differences between RLs and SLs.

Table 3-1. Alternatives for Storing Shared Code

METHOD	ADVANTAGE	DISADVANTAGE
COMMON SOURCE CODE (Use the JOIN command of the EDITOR or the equivalent facility of your programming language (such as COPYLIB in COBOL) to include the common source procedures as part of the source code for your main program.)	Easy to use. Requires no use of the Segmenter subsystem. Allows non-dynamic subroutines.	Changes to a common routine may require recompiling many programs. Compiler time is increased since the common routine is recompiled. Possible conflicts with data or paragraph names in main program. Does not allow mixing of source languages.
AUXILIARY USL FILE (Compile the procedure and keep the USL file. Using the Segmenter, reference the procedure's USL file as the AUXUSL and copy the required RBMs into another USL that contains the RBMs for the main program.)	Allows complete flexibility in segmentation. Allows mixing of source languages for main and subroutines. Allows non-dynamic subprograms.	Requires extra commands in JCL jobstream. Requires more Segmenter expertise on part of programmers. Requires more control, good documentation of shop standards and how to use them. No easy way to determine if a program calls the routine, since all references are internal to the object code.
USE AN RL (Place the procedure in an RL. All other programs that need the procedure are then simply prepared with the ;RL=*filename* parameter.)	Easy to use. (:PREP *uslfilename*; **RL=***rlfilename*.) Allows storage of non-dynamic procedures. Flexible: the RL can contain all the common procedures for your shop; only the referenced procedures are linked to the object code at PREP time.	All referenced procedures are brought into the program file as one segment. For some programs, this segment may approach or exceed the configured maximum segment size. A change to the RL procedure requires Segmenter manipulation and may require that all programs which use the procedure be re-PREPed. If the program USL is not available, you will also have to recompile.

Table 3-1. Alternatives for Storing Shared Code (Continued)

METHOD	ADVANTAGE	DISADVANTAGE
USE AN RL (Continued)		There is no way to clean or expand an RL file. If you run out of space you must rebuild the RL from the USL or from the source code, which implies recompilation.
USE SL *.group.account* or SL.PUB.*account* (Place the procedure in an SL in the group where the program file will reside or in the PUB group of that account. Programs that need the procedure must be run with the ;LIB=G (or P) parameter on the :RUN statement, unless the parameter is part of a UDC.)	Allows for sharing of code segments, i.e., every program does not have its own copy of the procedure. If an SL procedure is changed, no recompilation is necessary. Program files are smaller.	Uses CST entries. (System limit is 2048.) Requires ;LIB= parameter each time an SL procedure is used, although the various ;LIB= parameters could be placed in UDC files. Depending upon your account structure, may require you to generate many SLs containing identical procedures.
USE SL.PUB.SYS (Place the procedure in SL.PUB.SYS. The program will not require the ;LIB= parameter.)	Same advantages as SL.*group.account* but no ;LIB parameter required. Allows for sharing of code on a system-wide rather than just an account-wide level, minimizing duplicate code modules. Somewhat easier to control than many SL files. More sharing of common code could result in less swapping for some environments.	Difficult to make changes to the system SL. It should be done only via the SYSDUMP procedure and may require you to shut down your system to install any new procedures or changes. You should make a new cold load tape after any change. You will have to re-apply your changes every time there is an update to MPE. The entry point names of your procedures may conflict with those of MPE. Even if you have no problem currently, there is no guarantee that the next version of MPE will not have a name which conflicts with your procedure(s).

Table 3-2. RLs vs. SLs

	RL	SL
ALLOWS NON-GLOBAL PROCEDURES	YES	YES
ALLOWS GLOBAL PROCEDURES	YES	NO
RESOLVED AT PREP TIME	YES	NO
RESOLVED AT RUN TIME	NO	YES
CAN CONTAIN MORE THAN 1 SEGMENT	NO	YES
FILE CAN HAVE ANY NAME	YES	NO*
CAN REFERENCE COMMON DATA	YES	NO
PERMANENT PART OF PROGRAM FILE	YES	NO

*An SL file can be built and maintained with any name, but it cannot be referenced at RUN time with the ;LIB= parameter unless the name of the file has been changed to "SL".

Using SLs: Special Concerns

A good alternative for storing shared code is one that makes efficient use of system resources as well as making it easy for programmers to access and manage the code. Among the most critical system resources are the Code Segment Table entries. The CST contains descriptive information (whether the segment is in main memory or out on disc; whether running it requires privileged mode capabilities) for each active SL segment. The CST has space for 2048 entries, many of which are reserved for MPE

itself. (Other program code segments are kept track of in the CST extension, or CSTX, a less valuable resource since it has more space available.)

Entries in the CST are dynamically allocated by the operating system as programs are loaded and overwritten. Execution of a program cannot proceed if no room is available in the CST for the entry of the program's SL code segments. Excessive usage of SLs could therefore exhaust the currently available CST entries and cause serious disruption of all executing processes.

You will probably not have problems with CST entries until your installation is fairly large. But the groundwork for future difficulties is laid when you and other programmers first begin to put code into SLs, and a CST entry problem can be very difficult and time-consuming to resolve by the time it becomes evident. The following preventive measures are well worth the extra time and effort it takes to set them up and train yourself and other programmers to use them consistently.

- Decide on installation-wide standards for what procedures are to be kept in SLs. Consider both the type and the frequency of use.

- When several programmers are writing code, there is a tendency for similar routines to proliferate, each in a different SL. Periodically examine your SLs to see if they contain very similar procedures. Decide on one procedure that serves everyone's purposes and purge the others.

- Consolidate your procedures, placing those that reference each other in one segment in the same SL.

- Make a habit of using the *;LMAP* option on the :RUN command to see where your SL references are being resolved from. If they are scattered, coming from many different SLs, you need to make better use of your SLs, and, in turn, of the CST entries available for executing programs.

- If a procedure is common to several applications, consider putting it in SL.PUB.SYS, instead of repeating it in many group and account SLs.

- Remember that the Segmenter (as well as the Loader) protects the integrity of SLs by locking them while they are being accessed, preventing their use by other users until the first user is finished. Since SL.PUB.SYS is critical to the functioning of the entire system, it is particularly important that you minimize the amount of time this library is locked. The time involved for a reference to the library is minimal compared to the time involved for the entry and management of code. You can reduce this overhead and protect this critically important library by designating one person to manage it and scheduling entry and maintenance for off-hours.

- If you use multiple SLs, check each of them periodically and purge any procedures duplicated elsewhere.

SEGMENTATION STRATEGIES

Remember that, although the Segmenter is a powerful tool, most programmers never need to access it explicitly. In almost all situations, the segmentation accomplished by the MPE operating system is quite adequate. If you decide to control or alter segmentation, it will usually be for one of the following reasons:

- You are involved in program design and are testing different procedures or versions of a procedure. Resegmenting or altering the contents of a segment gives you freedom to experiment without having to rewrite and recompile your source code.

- You have a problem to solve, such as the inadvertent duplication of an RL procedure name; or the use of an excessive number of RL references, resulting in an RL segment which exceeds your system's maximum configured code segment size.

The problems which would make altering segment contents or controlling segmentation necessary (instead of optional, as it is with program design) are relatively infrequent.

Because segmentation is so adequately handled by the operating system and because problems with it are infrequent, your control of segmentation will rarely have a significant impact on performance. Resegmentation will not help if you have major problems with your use of system resources; and if you want a program with exceptional speed and efficiency, a slight change in program logic or I/O structure will have more effect than any amount or kind of resegmenting you can do.

Nevertheless, if you have chosen to control segmentation, you will want to follow some guidelines to make your management as effective as possible. As you use the Segmenter, you will develop effective strategies suited to your system and the kinds of programs you and other programmers are writing. However, there are some general guidelines you can follow. An understanding of some of the design characteristics of the HP 3000 and of how the Segmenter works will help you to make effective use of these guidelines and to develop your own strategies.

The Operating System Environment

The Multiprogramming Executive (MPE) environment is a dynamic one where programs are run on the basis of "processes". A process is the basic executable entity of MPE. It is not a program, but the unique execution of a program by a particular user at a particular time.

When you execute a program, a private data segment called a "stack" is created for that particular execution. The stack and the program's code segments together constitute the process, along with any external references made by your program code. For example, if several users access the BASIC interpreter, a separate process is created for each of them. They all use the same code, since there is only one BASIC interpreter; but each user has a unique data stack created by MPE.

Two MPE components have primary responsibility for managing process execution: the process dispatcher and the memory manager. The dispatcher allocates CPU time to all the executing processes. The memory manager's function is to fit code and data segments into main memory as they are required. This operation often requires the dispatcher to decide which already-present segments it must delete in order to make space available.

When the time-slice for your process begins, the stack and one code segment are brought into main memory and control is passed to the program. As the program proceeds, it will call procedures which are not in the current segment. At this point your process is suspended while MPE arranges to make the required segment present. In the meantime, the dispatcher tries to run the process with the next highest priority which already has a data stack and any required code segments resident in memory. When the missing segment for the first process has been made present, control is passed back to the procedure which called the segment, and the second process is suspended.

There is no way for you to tell which code segments will be present in main memory at any particular moment. All you can be sure of is that the memory manager will keep the most popular code segments available and will overwrite those which are rarely used. If a process needs an overwritten segment, another copy of the segment will be brought in from disc memory. Frequently-used segments remain in main memory, while those rarely used are in disc memory most of the time.

To you as a user, calling a procedure seems equally easy whether or not it is in the current segment, since MPE handles process management without your assistance. However, there are system constraints which can be aggravated by poor segmentation. One of these is the physical size of main memory, which limits the number of code segments which can be present at any one time. Another is the amount of time involved in swapping absent segments in from disc memory: while it may not be noticeable to you, the time involved does affect the efficient use of system resources and the execution of your process. A third constraint is that many processes are competing for CPU time, and a process which is well-segmented will execute more smoothly, making better use of CPU time, than one which makes many calls to procedures in absent segments.

Although processes themselves are created, maintained and deleted by MPE, you do have control over the following process elements:

- Which code goes into which segment.
- The size of each segment.
- The number of code segments.
- The size of the data stack (which you control with the compiler rather than the Segmenter).

In almost all circumstances, the first element is the only one you need to be concerned about: although your programs are unlikely to approach the segment size and number limitations, which code you place in each segment can affect the amount of work the memory manager must do to make code segments present in main memory for executing processes, thus affecting how quickly and smoothly the CPU can handle your executing program.

Segmentation Guidelines

Your concerns for achieving good segmentation should focus on "locality", which is the degree to which control remains in the same general area of code. This focus doesn't change even in systems with physical memory space problems. Indeed, locality becomes even more of a concern in such systems.

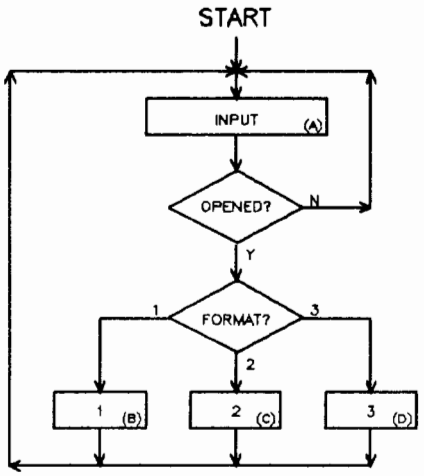
ACHIEVE LOCALITY. Poor locality means that a program branches wildly and rapidly from place to place, so that when it is segmented and run, there will be many external references and control will have to pass frequently from segment to segment. Inter-segment transfers which require control to be passed to a segment not present in main memory are particularly wasteful of time and disruptive to your process and other processes as the memory manager reorganizes main memory to make space available for the requested segment.

To work efficiently, the HP 3000 design needs programs that have good segment locality. Good segment locality exists when inter-segment transfers are minimized: that is, when a program is segmented so that once the system is working within a particular segment, it stays there as long as possible, and when it is out, it stays out as long as possible. (Note that the degree of locality within a segment is not important.)

When they make segmentation decisions, most programmers intuitively feel that they will achieve localization if they segment according to function: that is, they want to put all the command decoding procedures together in one segment, all the command executors together in another segment, and so on. But you will achieve much better results if you segment according to temporal considerations: that is, consider how you can group your procedures so that when your program is executing, the

system will be able to stay within each segment for as long as possible and pass control smoothly from segment to segment as the program progresses.

For example, consider a small utility program which dumps a file to the line printer in some special format. Suppose the operator can choose the name of the file and which of those possible formats to use. The program has four procedures: A, B, C, and D.



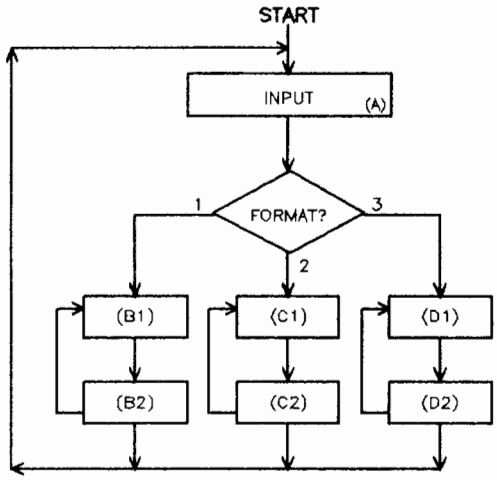
Procedure (A) asks user for file name and dump format.

File is opened successfully, or error message is generated.

Dump routine is selected from user's choice during input.

Procedure (B), (C), or (D) produces the dump in Format 1, 2, or 3.

Suppose, further, that each dump procedure has a procedure to fetch a record from its file and a procedure to format a print line:



Procedures (B1), (C1), and (D1) get records.

Procedures (B2), (C2), and (D2) format output lines.

You would probably be tempted to put all the formatting routines in one segment and the record-fetching routines in another. But such functional segmentation would cause a segment boundary to be crossed twice for every record dumped. During this transfer time, your program is doing no useful work and other processes are disrupted, since they must wait for the CPU to finish with your process.

If you segment temporally, however, with A in its own segment and B1/B2 together, C1/C2 together, etc., then only two segment boundaries are crossed for a whole dump.

block must reside wholly within a segment. If it becomes necessary to move a block of code into another segment, it will only be possible if the code is a procedure: you cannot move an arbitrary set of instructions and place them in another segment but must move the whole RBM.

You will also want to be aware of the conventions of your particular programming language for constructing RBMs from source code and compiling those RBMs into segments. Appendices A through F will provide you with some of the major conventions for each language. Refer to the appropriate language reference manual for more help.

Routinely using the ;PMAP option when you prepare your programs will help keep you aware of each program's environment: the size of each segment, which procedures are in which segment, and the names of the externals called by each segment. Such information will help you minimize inter-segment transfers as well as to follow the other guidelines for code manipulation. You can find more detailed information about ;PMAP in Appendix G.

ELIMINATE NON-ESSENTIAL MATERIAL FROM YOUR SEGMENTS. Minimizing the amount of unused material in any segment, file, account, etc., is a good general principle, as well as a way to reduce contention for memory resources.

You can clean up segments by examining the code within them and removing code which executes only infrequently. Very often, this will be code which does error-handling. Instead of handling detected errors on-line, write an error-message generating procedure and call it with a parameter indicating which message to output. You can put this procedure in a separate segment so that it does not clutter up main memory while the system is doing normal, error-free processing of your program.

As another example, suppose you have a program that does an FWRITE and then checks the condition code for end-of-file. If required, the program executes a somewhat elaborate sequence to extend the file by building a new one, copying the old one into it, and then purging the old file. If this condition is likely to occur only once in every 500 program runs, there is little point in holding the procedure in main memory with other procedures which do execute frequently. You can move this code to an auxiliary segment and let MPE bring it in only when it is needed. (Remember that you can only move such code out of a segment if it is a procedure in itself, rather than a series of instructions which are part of a procedure.)

If your program will be run from multiple terminals, then the code segments will automatically be shared by the multiple processes, but each process will have its own data stack. If your program design requires data which is never altered, such as error messages, look-up tables, etc., put the data in your code so that only one copy will be required for all processes. Otherwise, all users who run the program will receive space in their data stack for the code, thus making inefficient use of main memory resources.

In the following SPL program, the array MESSG is present in the stack of every process running the program.

```
BEGIN
  BYTE ARRAY MESSGINIT (0:22):="TOO MANY TIMES ENTERED";
  .
  .
  .
    PROCEDURE MESSOUT;
    BEGIN
    BYTE ARRAY MESSG (0:22)
    MOVE MESSG:=MESSGINIT,(23);
    PRINT (MESSG,-23,0);
    .
    .
    .
    END;
      .
      .
      .
  END.
```

We will use main memory space more efficiently if MESSG only exists while MESSOUT executes. In the next example, SPL stores the string in quotes in the code segment.

```
BEGIN
  .
  .
  .
    PROCEDURE MESSOUT;
    BEGIN
    BYTE ARRAY MESSG(0:22);
    MOVE MESSG:="TOO MANY VALUES ENTERED";
    PRINT (MESG,-23,0);
    END;
  .
  .
  .
  END.
```

This guideline and the accompanying examples illustrate the point that the steps you take to achieve good segmentation are not limited to moving material from segment to segment or altering material within segments. Rather, you need to keep in mind the Segmenter's function and the effect of segmentation on execution, even during phases of the program development process that precede the Segmenter's involvement.

Obtaining Machine-Readable PMAPs: The FPMAP

The Segmenter's PMAPs, produced by the PMAP option of the -PREPARE and -ADDSL commands and explained in Appendix G, are printed listings which show where a program's procedures are located in the code segments of a program or SL file. When you prepare a program file or SL segment, you may also request that a machine-readable form of the PMAP be stored in the program or SL file. This machine-readable form of the PMAP is known as the FPMAP.

FPMAPs are useful to subsystems which wish to refer to code locations by procedure names and offsets within procedures, rather than by segment numbers and code segment addresses. For instance, they allow HP TOOLSET to set and interpret breakpoints based on source-code locations, and allow APS/(SAMPLER) to break down program execution profiles by procedures rather than by arbitrary code segment addresses.

There are two ways to control whether or not the Segmenter will generate FPMAPs with your program files or SL file segments. The first is explicit, and is done by including either the FPMAP or NOFPMAP keyword on a -PREPARE or -ADDSL command. The second is implicit, and involves the setting of the system-wide and job/session FPMAP flags with the -SETFPMAP command. These flags allow you to control FPMAP generation without using the FPMAP/NOFPMAP keywords on -PREPARE and -ADDSL commands, thereby providing a means for obtaining FPMAPs without modifying your existing Segmenter job or command files.

SYSTEM-WIDE FPMAP FLAG. The system-wide FPMAP flag can be used to force FPMAPs in all program files and SL segments prepared on the system, and is also used to set the value of the job/session FPMAP option flag when a user logs onto the system. To set the system-wide FPMAP flag, you must have System Manager (SM) capability. The system-wide FPMAP flag may assume one of the following values:

UNCONDITION (ON)	Causes FPMAPs to be generated for all program files and SL segments prepared on the system, regardless of the setting of the job/session FPMAP flag or presence of the FPMAP/NOFPMAP keywords on -PREPARE and -ADDSL commands. This value also causes the job/session FPMAP flag to be initialized to ON at the beginning of each job and session.
CONDITION (ON)	Causes the job/session FPMAP flag to be initialized to ON at the beginning a job or session.
OFF	Causes the job/session FPMAP flag to be initialized to OFF at the beginning of a job or session.

When a system is started or loaded, the system-wide FPMAP flag is initialized to OFF.

JOB/SESSION FPMAP FLAG. Each job or session also has its own FPMAP flag, which controls FPMAP generation for -PREPARE and -ADDSL commands performed within that job or session as follows:

ON	Causes FPMAPs to be generated for all program and SL files prepared in the current job or session, unless overridden by the NOFPMAP keyword on -PREPARE and -ADDSL commands.
OFF	Inhibits FPMAP generation for all program and SL files prepared in the current job or session, unless overridden by either the system-wide FPMAP flag (if set to UNCONDITION (ON)), or the FPMAP keyword on -PREPARE and -ADDSL commands.

For programs compiled by HP TOOLSET, the FPMAP will always be generated if symbolic debugging is requested, regardless of the FPMAP flags and FPMAP/NOFPMAP keywords. When preparing such programs, only the NOSYM keyword can suppress the FPMAP information.

FPMAPs will increase program and SL file size by 3 to 10 percent, and will cause the CPU time for preparing a program file or SL segment to increase by 5 to 20 percent. Since the FPMAP information is not stored as part of the code segments themselves, it will have no effect on program execution

performance. Because of the potential benefit of having the FPMAP information available for use by other subsystems, it is a good idea to include it with all program and SL files.

Three intrinsics are provided for programmatic access to the FPMAP: FINDPMAPNAME translates segment names and procedure names to segment numbers and offsets; FINDPMAPADDR translates segment numbers and offsets to segment names and procedure names, and DUMPPMAP extracts the entire FPMAP from a program or SL file into an external file for use by other applications. Each of these uses the external FPMAP record format shown in Figure 3-1.

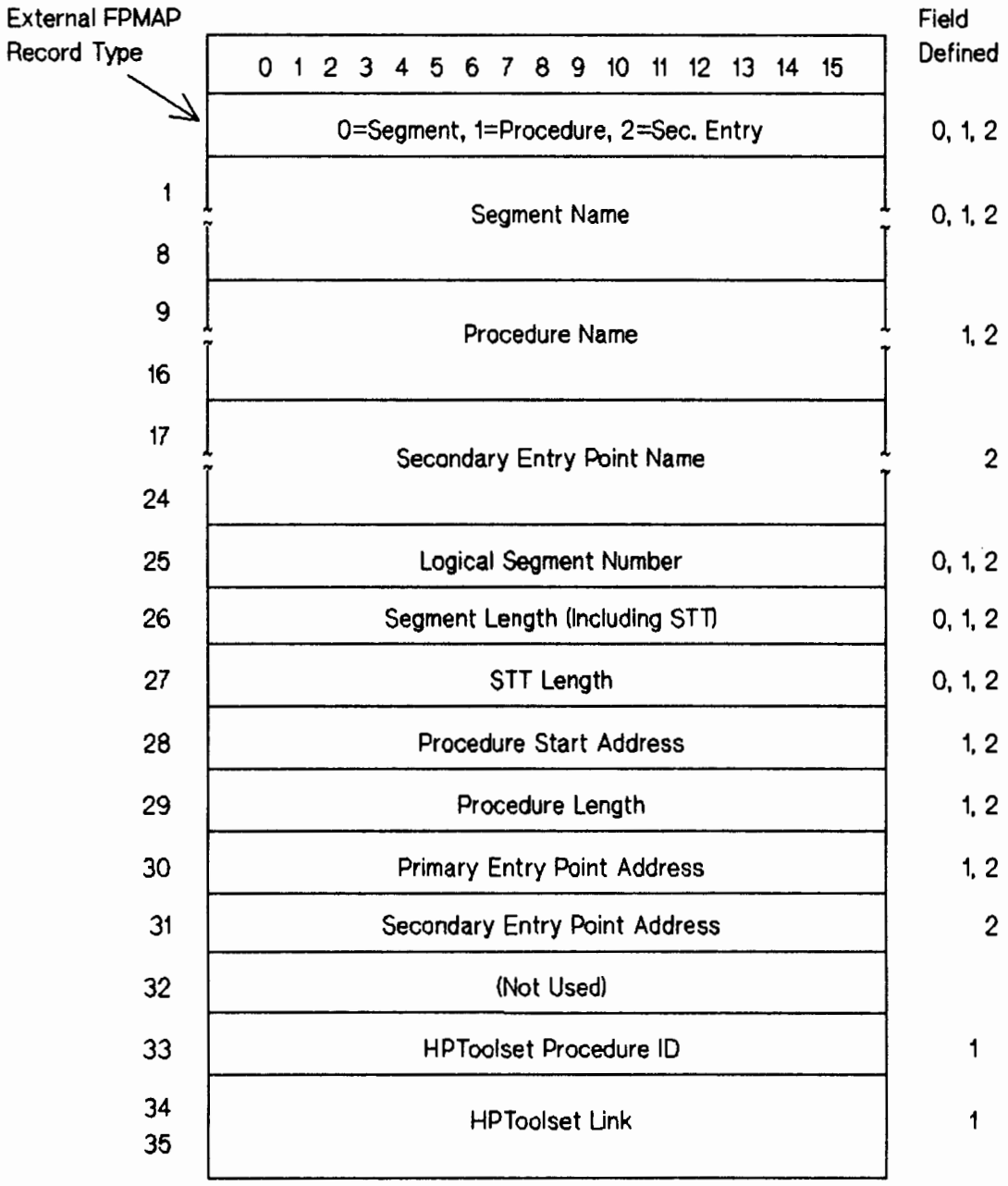


Figure 3-1. External FPMAP Record Format

Using the Application Program SAMPLER/3000 (APS/3000)

APS/3000 is an interactive performance tool for tuning applications on the HP 3000. You may wish to use it to facilitate your identification and correction of segmentation problems.

APS/3000 monitors the execution of software (a single program or the multiple execution of one or more shared program files) and produces histograms showing the amount of CPU time spent by various programs, or portions of one monitored program. You can study the histograms and then fine-tune your software by optimizing the execution time of:

- All processes associated with a single program file.
- A single process.
- A segment.
- A procedure.
- An address range.

In a typical application of this performance tool, the user runs and monitors software for a period of time, studies APS/3000's histograms, and learns which code consumes the most CPU time. After optimizing the code, the user repeats this process until the software performance is acceptable.

For resegmentation purposes, APS/3000 will allow you to estimate the number of transfers of control between segments during program execution. Because it is a statistical sampling technique, APS/3000 assesses system operation at pre-set time intervals, rather than continually. Thus it will not provide the exact number of transfers, nor can it tell you specifically what to do to make your segmentation more efficient. However, it can offer you valuable help in identifying problem areas.

You should always specify FPMAP when running APS/3000 since it automatically generates valuable information that can assist you in diagnosing program failures.

For more information, see the Application Program SAMPLER/3000 (APS/3000) Reference Manual and User Guide (32180-90001).

COMMAND AND INTRINSIC SPECIFICATIONS

SECTION IV

In table 4-1, the commands and intrinsics used with the Segmenter are arranged in functional order. Many of the commands appear in more than one category (e.g., -LISTUSL appears both in the Managing USLs category and the Listing Files category) so that you will find the appropriate commands while considering the Segmenter from various functional perspectives.

Following the table, the command and intrinsic specifications are presented in alphabetical order.

COMMANDS

The command specifications contain the following information:

- The command name.
- The type of command (Segmenter or MPE command). This information is shown in italics directly under the command name.
- A brief summary of the command's function.
- Syntax. The command syntax is highlighted by enclosure in a box.
- Parameter definitions, including meaning, constraints, and defaults.
- Examples.
- Text discussion. (The page in this manual where usage of the command is discussed.)

For more information about MPE commands, see the MPE V Commands Reference Manual (32033-90006). Segmenter commands consist of the following elements:

- The command name. In an interactive session, the Segmenter displays a dash (-) as a prompt when it is ready to accept a command. Note that the prompt character is not allowed when entering Segmenter commands in a batch job. The command name is shown in this section with a preceding prompt character.
- Parameters, if any, follow the command name. A blank character must be used between the end of the command name and the first parameter.

INTRINSICS

The intrinsic specifications contain the following information:

- The intrinsic name. The word *intrinsic*, in italics, directly under the intrinsic name identifies it as an intrinsic.

- A brief summary of the function of the intrinsic.
- The complete intrinsic call description, highlighted by enclosure in a box. The format is as follows:

```
          I                  IV     IA
errnum:=INITUSLF(uslfnum,rec0);
```

- For those intrinsics which return a value to the calling process (type procedures), the return is described. For example, INITUSLF is an integer procedure which returns an error number to *errnum* if an error occurs. The abbreviation, I, over *errnum* identifies INITUSLF as an integer procedure.
- All parameters are described. Required parameters, such as ***uslfnum*** above, are shown in ***bold face italics***. Superscripts (such as IV) are used to denote the types of parameters and whether they must be passed by value, instead of by reference (the default case). See Section I of the MPE V Intrinsics Reference Manual (32033-90007) for a discussion of passing parameters by value and by reference.

  Superscripts have the following meanings:

  I Integer

  IA Integer array

  IV Integer by value

  BA Byte array

- Condition codes are included for each intrinsic.
- Text discussion. (The page in this manual where use of the intrinsic is discussed.)

Table 4-1. Functional List of Commands and Intrinsics

FUNCTION	COMMAND OR INTRINSIC
Accessing the Segmenter:	:SEGMENTER Combination commands such as :COBOLGO, :COBOLPREP, :PREPRUN
General Segmenter commands:	
Terminating interaction with the Segmenter.	-EXIT or -E
Editing or repeating a Segmenter command.	-REDO
Obtaining help on a Segmenter command.	-HELP
Managing USL files:	
Open a USL file as the currently-managed USL file.	-USL
Build a USL file and open it as the currently- managed USL file.	-BUILDUSL
List contents of the USL file.	-LISTUSL
Move code modules within the USL file.	-NEWSEG
Activate/inactivate code modules.	-CEASE/USE
Hide or reveal RBMs which you will place in an SL file.	-HIDE/REVEAL
Delete code modules.	-PURGERBM
Add code modules from a USL file to an RL file.	-ADDRL
Add code modules from a USL file to an SL file.	-ADDSL
Specify auxiliary USL file.	-AUXUSL
List contents of auxiliary USL file.	-LISTAUX
Copy code from one USL file to another.	-COPY

Table 4-1. Functional List of Commands and Intrinsics (continued)

FUNCTION	COMMAND OR INTRINSIC
Clean the USL file.	-CLEANUSL (Calls the CLEANUSL intrinsic)
Copy and enlarge the USL file.	-COPYUSL
Initialize a buffer for a USL file to the empty state.	INITUSLF intrinsic
Move the information block in a USL file.	ADJUSTUSLF intrinsic
Change the length of a USL file.	EXPANDUSLF intrinsic
Managing RLs:	
Open an RL file as the currently-managed file.	-RL
Build an RL file and open it as the currently- managed RL file.	-BUILDRL
List contents of an RL file.	-LISTRL
Add code modules from a USL file to an RL file.	-ADDRL
Delete code modules from an RL file.	-PURGERL
Managing SLs:	
Open an SL file as the currently-managed file.	-SL
Build an SL file and open it as the currently-managed SL file.	-BUILDSL
List contents of the SL file.	-LISTSL
Add code modules from a USL file to an SL file.	-ADDSL

Table 4-1. Functional List of Commands and Intrinsics (continued)

FUNCTION	COMMAND OR INTRINSIC
Delete code modules from an SL file.	-PURGESL
Clean an SL file.	-CLEANSL
Clean and enlarge an SL file.	-COPYSL
Building Files:	
USL file	-BUILDUSL
RL file	-BUILDRL
SL file	-BUILDSL
Cleaning and enlarging files:	
Clean a USL file.	-CLEANUSL (calls the CLEANUSL intrinsic)
Clean and enlarge a USL file.	-COPYUSL
Initialize a buffer for a USL file to the empty state.	INITUSLF intrinsic
Move the information block in a USL file.	ADJUSTUSLF intrinsic
Change the length of a USL file.	EXPANDUSLF intrinsic
Transferring code modules from one file to another:	
From a USL file to an RL file.	-ADDRL
From a USL file to an SL file.	-ADDSL
From an auxiliary USL file to the currently- managed USL file.	-COPY

Table 4-1. Functional List of Commands and Intrinsics (continued)

FUNCTION	COMMAND OR INTRINSIC
Adding Code Modules:	
To an RL file from a USL file.	-ADDRL
To an SL file from a USL file.	-ADDSL
Deleting code modules:	
From a USL file.	-PURGERBM
From an RL file.	-PURGERL
From an SL file.	-PURGESL
Listing File Contents:	
USL	-LISTUSL
Auxiliary USL file	-LISTAUX
RL file	-LISTRL
SL file	-LISTSL
Managing FPMAP Information:	
To list FPMAP information.	-LISTPMAP
To set or clear the system and the job/session FPMAP flags.	-SETFPMAP
To display the status of the FPMAP flags.	-SHOW
Translate segment names and procedure names to segment numbers and offsets.	FINDPMAPNAME
Translate segment numbers and offsets to segment names and procedure names.	FINDPMAPADDR
Extract FPMAP from a program or SL file.	DUMPPMAP

Prepares program from a User Subprogram Library (USL) file onto a program file. Implicitly calls the Segmenter.

SYNTAX

```
:PREP uslfile,progfile
        [;ZERODB]
        [;PMAP]
        [;MAXDATA=segsize   [;PATCH=patchsize]
        [;STACK=stacksize]
        [;DL=dlsize]  [{;FPMAP   } ]
                      {;NOFMPMAP}
        [;CAP=caplist]
        [;RL=filename]
        [;NOSYM]
```

PARAMETERS

uslfile Actual designator of USL file into which program has been compiled.

progfile Actual designator of program file into which prepared program segments are written. Can be any binary output file. This program file must be created in one of two ways:

1. By creating a new file with the MPE :BUILD command with a *filecode* of PROG or 1029, and a *numextents* parameter value of 1.

2. By specifying a non-existent file in the *progfile* parameter, in which case a file of the correct size and type is created. This file is a job temporary file.

ZERODB Request to initialize to zero the initially-defined, user-managed (DL-DB) area of the stack, and uninitialized portions of the DB-Q (initial) area of the stack. Default is that these areas are not affected.

PMAP Request to produce a descriptive listing of the prepared program on file whose formal designator is SEGLIST. If no :FILE equation is found referencing SEGLIST, listing is produced on $STDLIST. Default is no listing. See Appendix G for a listing of a prepared program.

segsize

Maximum stack area (Z-DL) size permitted, in words. This parameter is included when it is expected that the size of DL-DB or Z-DB areas will be changed during program execution. Default is that MPE assumes that these areas will not change. Regardless what you specify, MPE may change the *segsize* to accommodate table overflow conditions.

If you prepare your program with *segsize* less than the configured minimum, the value is rounded up to the minimum or the amount needed by the program (as calculated by the Segmenter). The maximum actual *segsize* permitted a program is 32,767 words. You may prepare your program with a *segsize* larger than necessary so long as this maximum is not exceeded. If the specified *segsize* does exceed the maximum, it will be rounded down to 32,767 words.

patchsize

Specifies the size of the patch area. This size will apply to all segments within the program file. The value you specify must be within -1 and 16,380 words.

stacksize

Size of initial local data area (Z-Q initial) stack, in words. If specified, this value must be between 511 and 32,767 words. This parameter overrides default *stacksize* estimated by Segmenter.

dlsize

DL-DB area to be initially assigned to stack. This area is of interest mainly in programmatic applications. Due to system logging considerations, the DL-DB area is always rounded upward so that the distance from the beginning of the stack data segment to the DB-address is a multiple of 128 words. The value you specify must be within -1 and 32,767 words. The default is estimated by the MPE Segmenter.

FPMAP or NOFPMAP

Includes or excludes the internal PMAP information. FPMAP is a request to have internal PMAP information included in the program. NOFPMAP excludes PMAP information from the program when the system FPMAP or job/session FFPMAP is on. If the symbolic debug option is invoked, default is FPMAP. Otherwise the default is NOFPMAP.

caplist

Capability-class attributes associated with program, specified as two-character mnemonics. If more than one mnemonic is specified, each must be separated from its neighbor by a comma. The mnemonics are:

IA Interactive access
BA Local batch access
PH Process handling
DS Data segment management
MR Multiple resource management
PM Privileged mode

Note that you can specify only those capabilities that you possess (through assignment by the Account Manager or System Manager). Default is IA, BA (if you possess these capabilities).

filename Actual designator of RL file to be searched to satisfy external references during preparation. This can be any permanent binary file of type RL. It need not belong to log-on group, nor have a reserved, local name. This file, to which you must have READ and LOCK access, yields a single segment that is incorporated into the segments of the program file. See Section II for a discussion of RL files.

Default is that no library is searched.

NOSYM Suppresses the symbolic debug option. Refer to the Toolset Reference Manual (32350-90001).

USE

Available	In Session?	YES
	In Job?	YES
	In Break?	NO
	Programmatically?	NO
Breakable?		YES (Suspends)

OPERATION

The :PREP command prepares a compiled source program for execution. Unless you prepare the program onto a previously created program file, this command will create a program file of the appropriate format for you in the job/session temporary file domain. In fact, it is recommended that you specify a non-existent program file in the :PREP command. This allows MPE to create a file of optimum size and characteristics. (See Example below.)

A compiled program is prepared by searching a relocatable library (RL) file to satisfy references to external procedures required by the program. When the program is prepared, such procedures are linked to the program in the resulting program file. To use an RL file via the :PREP command, the user requires READ and LOCK capability.

EXAMPLE

To prepare a program from the USL file USLX to the new program file PROGX, enter:

:PREP USLX,PROGX	Prepares program into program file PROGX.
:SAVE PROGX	Saves program file.

You can create a program file in the permanent file domain by using the :BUILD command. When you do this, you must be sure to specify a *filecode* of PROG (or 1029) for this file, and to limit the file to one extent (program files are not allowed to span more than one extent). The following :BUILD command creates such a file, which is then used by the :PREP command:

```
:BUILD PFILE;CODE=PROG;DISC=,1
:PREP UFILE,PFILE
```

To prepare a program from the USL file named UFILE and store it in a program file named PROGFILE, list the prepared program, assign a *stacksize* of 511 words, and assign batch-access capability only for the program, enter:

```
:PREP USEFILE,PROGFILE;PMAP;STACK=511;CAP=BA
```

ADDITIONAL DISCUSSION

Page 2-22.

MPE Commands V Reference Manual (32033-90006).
Using Files (30000-90102).
MPE File System Reference Manual (30000-90236).

:SEGMENTER

MPE Command

Accesses the Segmenter.

SYNTAX

```
:SEGMENTER [listfile]
```

PARAMETERS

listfile — An ASCII file (formal designator SEGLIST) to which is written any listable output generated by Segmenter commands. The designator SEGLIST should not be used as the actual file designator. If *listfile* is omitted, the standard job/session list device ($STDLIST) is assigned. The only Segmenter commands which use the *listfile* parameter are LISTUSL, LISTAUX, LISTSL, LISTRL, PREPARE, and ADDSL.

USE

Available	In Session?	YES
	In Job?	YES
	In Break?	NO
	Programmatically?	NO
Breakable?		YES (Suspends)

EXAMPLE

:FILE OUTPUT;DEV=LP	Equates the file OUTPUT to the line printer.
:SEGMENTER *OUTPUT	Accesses the Segmenter and back references the file OUTPUT, sending any listings to the line printer.

ADDITIONAL DISCUSSION

Page 2-1.

Adds a procedure to the currently-managed Relocatable Library (RL) file. You must have READ and WRITE access to the specified RL file.

SYNTAX

```
-ADDRL name[(index)]
```

PARAMETERS

name	The name (primary entry point) of the Relocatable Binary Module (RBM) containing the procedure to be added to the RL file.
index	An integer further identifying the RBM. The index may be used when the currently-managed USL file contains more than one active RBM of this name. If *index* is omitted, a value of 0 (the most recent active occurrence) is assigned by default.

EXAMPLE

-ADDRL XBM	Adds an RBM containing one procedure named XBM.

ADDITIONAL DISCUSSION

Page 2-29.

-ADDSL

Segmenter Command

Adds procedure to a Segmented Library (SL) file.

SYNTAX

```
-ADDSL name[;PMAP][{;FPMAP   }][;NOSYM][;PATCH=patchsize]
                   {;NOFPMAP}
```

PARAMETERS

name	The name of the segment to be added to the SL file.
PMAP	An indication that a listing describing the prepared segment will be produced on the file specified in the MPE :SEGMENTER command parameter *listfile*, which defaults to $STDLIST. If this parameter is omitted, the prepared segment is not listed.
FPMAP or NOFPMAP	Includes or excludes the internal PMAP information. FPMAP is a request to have internal PMAP information included in the program. NOFPMAP excludes PMAP information from the program when the system FPMAP or job/session FPMAP is on. If the symbolic debug option is invoked, default is FPMAP. Otherwise the default is NOFPMAP.
NOSYM	Suppresses the symbolic debug option. Refer to the HP Toolset Reference Manual (32350-90001).
patchsize	Specifies the size of the patch area. This size will apply to all segments within the program file. The value you specify must be within -1 and 16,380 words.

EXAMPLE

-ADDSL SL1;PMAP	Adds segment SL1 to the currently-managed SL file and produces a listing of the segment.

ADDITIONAL DISCUSSION

Page 2-38.

Opens an auxiliary USL file for management.

SYNTAX

```
-AUXUSL filereference
```

PARAMETERS

filereference The name (and optional group and account name) of the auxiliary USL file from which RBMs may be transferred to the current USL file.

EXAMPLE

```
-AUXUSL FILE1
```

Opens the auxiliary USL file FILE1 for management.

ADDITIONAL DISCUSSION

Page 2-14.

-BUILDRL

Segmenter Command

Creates a permanent, formatted Relocatable Library (RL) file. You must have SAVE capability in the group to which you are assigning the RL file.

SYNTAX

```
-BUILDRL filereference,records,extents
```

PARAMETERS

filereference	The name, and optional group and account name, of the RL file.
records	The total maximum file capacity, specified in terms of 128-word binary logical records.
extents	The total number of disc extents that can be dynamically allocated to the file as logical records are written to it. The *extents* value must be an integer from 1 to 32.

EXAMPLE

`-BUILDRL R1.G1.A1,500,1`	Creates an RL file named R1.G1.A1. The RL file can contain 500 records, and is allocated one disc extent.

ADDITIONAL DISCUSSION

Page 2-28.

Creates a permanent, formatted Segmented Library (SL) file. You must have SAVE capability in the group to which you are assigning the SL file.

SYNTAX

```
-BUILDSL filereference,records,extents
```

PARAMETERS

filereference Any file whose local name is SL. You can create an SL file with a local name other than SL, but such a file cannot be searched by the MPE :RUN command or the CREATE or LOADPROC intrinsics unless it is renamed SL.

records The total maximum file capacity, specified in terms of 128-word binary logical records.

extents The total number of disc extents that can be dynamically allocated to the file as logical records are written to it. The size of each extent is determined by the *records* parameter value divided by the *extents* parameter value. The *extents* value must be an integer from 1 to 32.

EXAMPLE

```
-BUILDSL SEARCH,300,1
```

Creates an SL file named SEARCH with 300 records and one disc extent.

ADDITIONAL DISCUSSION

Page 2-37.

-BUILDUSL

Segmenter Command

Creates a new (job/session temporary) User Subprogram Library (USL) file. Must use :SAVE to keep as permanent file.

SYNTAX

```
-BUILDUSL filereference,records,extents
```

PARAMETERS

filereference The name (file designator) assigned to the new USL file. This is a fully-qualified file reference that may include file name, group name, and account name, plus a lockword.

records The length of this file, specified in terms of 128-word binary logical records.

extents The total number of disc extents that can be dynamically allocated to the file as logical records are written to it. The size of each extent is determined by the *records* parameter value divided by the *extents* parameter value. The *extents* value must be an integer from 1 to 32.

EXAMPLE

```
-BUILDUSL FILE2,200,1
```

Creates a new temporary USL file, FILE2, with 200 records and one disc extent.

ADDITIONAL DISCUSSION

Page 2-14.

Deactivates one or more entry points in the currently-managed User Subprogram Library (USL) file.

SYNTAX

```
        [ENTRY,   ]
-CEASE  [UNIT,    ] name[(index)]
        [SEGMENT,]
```

PARAMETERS

ENTRY	Deactivates the entry point indicated by *name* [(*index*)]. (Default.)
UNIT	Deactivates all entry points in the Relocatable Binary Module (RBM) indicated by *name* [(*index*)].
SEGMENT	Deactivates all entry points in all RBMs associated with segment *name*.
name	The name of the entry point, RBM, or segment.
index	An integer further identifying the entry point name. The index may be used when the USL file contains more than one entry point of the same name. If SEGMENT is specified, the *index* parameter is ignored. If *index* is omitted, a default value of 0 (the most recent active occurence) is assigned. An *index n* means the *n*th occurence, whether active or inactive. However, there is no point in using this command with an entry point you know to be inactive, since its purpose is to deactivate active entry points.

EXAMPLE

-CEASE ENTRY,BEGIN2	Deactivates entry point BEGIN2.

ADDITIONAL DISCUSSION

Page 2-3, 2-9.

-CLEANSL

Segmenter Command

Copies the contents of the currently-managed Segmented Library (SL) file to a new SL file, eliminating any fragments of free space in each of the segments. The new SL file will be of the same size as the old one; that is, it may hold up to the same maximum number of records.

The new SL file becomes the currently-managed SL file.

SYNTAX

```
-CLEANSL [filename]
```

PARAMETERS

filename — The name of the new SL file. If you omit this parameter, the Segmenter will purge the old SL file, giving its name to the new file. In either case, the new SL file is a permanent file.

EXAMPLE

```
-SL MYSL
```

Opens the SL file MYSL as the currently-managed SL file.

```
-CLEANSL NEWSL
```

Copies the contents of the SL file MYSL into the new SL file NEWSL, eliminating any fragments of free space in each of the segments. NEWSL is now the currently-managed SL file.

ADDITIONAL DISCUSSION

Page 2-44

Copies only the active Relocatable Binary Modules (RBMs) of the currently-managed User Subprogram Library (USL) file to a new USL file. The new USL file will be the same size as the old one; that is, it will hold up to the same maximum number of records.

When executing this command, the Segmenter calls the CLEANUSL intrinsic.

The new USL file becomes the currently-managed USL file.

SYNTAX

```
-CLEANUSL [filename]
```

PARAMETERS

filename	The name of the new USL file. If you omit this parameter, the Segmenter will purge the old USL file, giving its name to the new USL file. In either case, the new USL file is a permanent file.

EXAMPLE

-USL SEARUSL	Opens SEARUSL as the currently-managed USL file.
-CLEANUSL SEARUSL7	Copies only the active contents of the USL file SEARUSL into the new USL file SEARUSL7. SEARUSL7 is now the currently-managed USL file.

ADDITIONAL DISCUSSION

Page 2-17.

-COPY

Segmenter Command

Copies one or more Relocatable Binary Modules (RBMs) from the auxiliary User Subprogram Library (USL) file to the currently-managed USL file.

SYNTAX

```
-COPY [UNIT,    ] name[(index)]
      [SEGMENT,]
```

PARAMETERS

UNIT	Transfers the RBM identified by *name* [(*index*)] from the source USL file. (Default.)
SEGMENT	Transfers *all* RBMs associated with the segment *name* from the source USL file. (The source USL file is the one specified by an AUXUSL command.)
name	Identifies the RBM to be transferred if UNIT is specified. If SEGMENT is specified, *name* identifies the segment from which all RBMs are transferred.
index	An integer further specifying the RBM name. The *index* may be used when more than one RBM of this name exists in the USL file. If *index* is omitted, a default value of 0 (the most recent active occurrence) is assigned. The *index* parameter is ignored if SEGMENT is specified.

CAUTION

The Segmenter will not deactivate already existing active RBMs of the same name, and such a duplication will cause a -PREPARE failure. Therefore, check your USLs and deactivate RBMs as necessary before you use -COPY.

EXAMPLE

-COPY SEGMENT,SEGNAME	Copies all RBMs associated with the segment name SEGNAME.

ADDITIONAL DISCUSSION

Page 2-14.

Copies the contents of the currently managed Segmented Library (SL) file to a new SL file, eliminating any fragments of free space while allowing you to control the size of the new file. This command is different from -CLEANSL in that you may change the amount of file space used by the file. The new SL file becomes the currently managed SL file.

SYNTAX

```
-COPYSL percent[,filename][;USERFORMAT]
```

PARAMETERS

percent	The amount of extra file space you want the new file to have, expressed as a percentage of the minimum amount of space needed to hold the segments. This parameter must be an integer, not less than 0 or greater than 9900.
filename	The name of the new SL file. If you omit this parameter, the Segmenter will purge the old SL file, giving its name to the new file. In either case, the new SL file is a permanent file.
USERFORMAT	Specifies conversion during copy from system SL format to user SL format. The resulting SL will have a group/account SL structure provided all active segment numbers are less than %377. If the currently managed SL is already a group or account SL, this parameter is ignored. If this parameter is omitted, the new SL structure will be the same as the old SL structure.

EXAMPLE ONE

-SL MYSL	Opens the SL file MYSL as the currently managed SL file.
-COPYSL 40,NEWSL	Copies the contents of the SL file MYSL into the new SL file NEWSL, building NEWSL with 40% more free space than the minimum presently needed to hold the segments.

EXAMPLE TWO

-SL SL.PUB.SYS	Opens the system SL as the currently managed SL file.
-COPY 1,MYSYSSL;USERFORMAT	Copies the contents of the sytem SL into the new SL file MYSYSSL, building MYSYSSL with 1% more free space than the minimum needed to hold the segments. If the system SL has no segment numbers greater than %376, MYSYSSL will have the group/account SL structure, otherwise it will retain the system SL structure.

ADDITIONAL DISCUSSION

Page 2-43.

-COPYUSL

Segmenter Command

Copies the entire currently-managed User Segmented Library (USL) file to a new USL file, while allowing you to control the size of the new USL file. Unlike the -COPYSL command, the -COPYUSL command does no "cleaning." This command will copy inactive RBMs to the new file as well as active RBMs.

The new USL file becomes the currently-managed USL file.

SYNTAX

```
-COPYUSL percent[,filename]
```

PARAMETERS

percent	The amount of extra file space you wish the new USL file to have, expressed as a percentage of the minimum space needed to hold the file. This parameter must be an integer not less than 0 or greater than 9900.
filename	The name of the new USL file. If you omit this parameter, the Segmenter will purge the current USL file, giving its name to the new USL file.

EXAMPLE

-USL SEARUSL	Opens the USL file SEARUSL as the currently-managed USL file.
-COPYUSL 100,SEARUSL5	Segmenter copies all of SEARUSL to new file SEARUSL5, giving SEARUSL5 100% more free space than the minimum presently needed to hold the contents of SEARUSL.

ADDITIONAL DISCUSSION

Page 2-17.

Terminates Segmenter operation.

SYNTAX

```
-EXIT
-E
```

ADDITIONAL DISCUSSION

Page 2-1.

-HELP

Segmenter Command

Accesses the Help subsystem.

SYNTAX

```
-HELP  [HELP                ]
       [tablecontents       ]
       [command[,keyword]   ]
       [ALL                 ]
       [EXIT                ]
```

PARAMETERS

HELP — Displays information for the HELP command. Treated the same as entering -HELP with any other command (because HELP is a command).

tablecontents — Any of several items for which information may be obtained. These items are:

HELPMENU
USLS
RLS
SLS
SHOW
TERMINATE

Thus, if you want information on managing USL files, you would enter:

```
-HELP USLS
```

command — Any Segmenter command. Segmenter displays the command name and syntax. In addition, a list of keywords for that command is displayed. Keywords for all commands are:

PARMS
EXAMPLE

Where:

PARMS	Lists all parameters of the specified command.
EXAMPLE	Displays an example showing usage of the specified command.

Entering the command -HELP, ALL causes MPE to display all information for that command (syntax, parameters, and example).

keyword — One of the keywords described under the command parameter. The keyword parameter can be entered with a command as in -HELP LISTSL,PARMS. In this case, PARMS is a keyword and must be separated from the command with a comma. The keyword parameter also can be entered alone if you are running the Help subsystem in "subsystem" mode and have accessed a command, then received the prompt (>) from Help.

For example,

```
-HELP AUXUSL

    -AUXUSL
```

opens an auxiliary USL file for management.

SYNTAX

```
    -AUXUSL filereference
```

KEYWORDS: PARMS,EXAMPLE

>PARMS (keyword entered by itself in response to prompt)

ALL — Displays entire table of contents for -HELP command. If entered with a command, as in

```
    -HELP LISTSL,ALL
```

displays all information for that command.

EXIT — Exits Help subsystem.

OPERATION

The -HELP command accesses the Help subsystem. If entered with no parameters, as in:

```
    -HELP
```

Help enters the "subsystem" mode, displays a table of contents, a "greater than" (>) prompt, and awaits your input. Entering any table of contents item such as SESSIONS produces a listing of all the commands used in running sessions. Entering any command name produces the syntax for that command and a list of keywords. Entering a keyword, such as PARMS produces a listing of all items for that keyword (all parameters in this case). Entering EXIT terminates the Help subsystem. Entering Control-Y stops the display. Entering Control-S stops the display and entering Control-Q resumes the display (useful if the screen in full if using a CRT terminal).

Entering a carriage return in "subsystem" mode causes HELP to display information up to the next keyword or command. For example, after entering AUXUSL, HELP displays AUXUSL syntax, the keyword list (PARMS, EXAMPLE) and prompts (>). Entering carriage return again causes HELP to display all PARMS information for the -AUXUSL command, and so on. (This is similar to page turning through a manual.)

Entering -HELP with a parameter causes Help to enter the "immediate" mode. Information pertaining to that parameter is displayed immediately. For example:

```
-HELP AUXUSL
```

causes Help to display:

```
-AUXUSL
```

Opens an auxiliary USL file for management.

SYNTAX

```
-AUXUSL filereference
```

EXAMPLE

To obtain information concerning the managing of SL files, enter:

```
-HELP SLS
```

-HIDE

Segmenter Command

Hides the named entry point. This entry point will not appear in the list of known entry-points when the RBM containing it is moved to an SL file. References to this procedure from any procedures except those in the same segment will not be resolved. Cannot use on a secondary entry point or for material which will be placed in an RL file. Can be used on a primary entry point, RBM, or segment.

SYNTAX

```
            [ENTRY,   ]
-HIDE       [UNIT,    ] name[(index)]
            [SEGMENT,]
```

PARAMETERS

ENTRY	Hides the entry point indicated by *name* [(*index*)]. (Default.)
UNIT	Hides all entry points in the Relocatable Binary Module (RBM) indicated by *name* [(*index*)].
SEGMENT	Hides all entry points in all RBMs associated with segment *name*.
name	The name of the entry point in the currently-managed USL file which is to be hidden.
index	An integer further specifying the entry point *name*. The index may be used when there is more than one entry point of this name. If *index* is omitted, 0 (the most recent active occurrence) is assigned by default.

EXAMPLE

-HIDE XBM	Hides the RBM entry point named XBM.

ADDITIONAL DISCUSSION

Page 2-44.

Lists the procedures in the currently-managed auxiliary User Subprogram Library The format for the -LISTAUX listing is the same as that generated by the -LISTUSL command.

SYNTAX

```
-LISTAUX [segmentname]
```

PARAMETERS

segmentname	The name of the segment you would like to see listed. Default is all (that is, if no segment name is specified, all segments are listed).

EXAMPLE

-AUXUSL MYAUX	Specifies the USL file MYAUX as the currently-managed auxiliary USL file.
-LISTAUX MYSEG	Lists the procedures in the segment MYSEG, which is in the auxiliary USL file MYAUX.
-LISTAUX	Lists all segments in the auxiliary USL file.

ADDITIONAL DISCUSSION

Pages 2-14, 2-34.

-LISTPMAP

Segmenter Command

Lists all segments in specified program file.

SYNTAX

```
-LISTPMAP progname   [{;segname          }]
                     [{;procedurename    }]
```

PARAMETERS

progname — The name of the program whose PMAP you want to list.

segname — The name of the segment whose PMAP you want to list. If omitted, all segments in the program file are listed.

procedurename — The name of the procedure whose PMAP you want to list. If the program file has a segment and a procedure that both have the same name, the entire segment is listed.

EXAMPLE

```
-LISTPMAP MYPROG

PROGRAM FILE MYPROG.MYGROUP.MYACCT

MAIN          0 **1**      4056 **2**

NAME          TYPE          CODE              ENTRY            LENGTH
PROC1          P               0                 10                45
PROC11        SP                                21
PROC2          P              45                 45              3000
PROC3          P            3045               3050              1000
```

Item Number	Meaning
1	Logical Segment Number
2	Segment Length

ADDITIONAL DISCUSSION

Page 3-10a

Lists the specified procedure entry points and external references in the currently-managed Relocatable Library (RL) file.

SYNTAX

EXAMPLE

-RL SMALLRL	Opens the RL file SMALLRL for management.
-LISTRL	Lists all procedure entry points and external references in the RL SMALLRL.

ADDITIONAL DISCUSSION

Page 2-25.

Lists all procedure entry points and external references in the currently-manage Segmented Library (SL) file.

SYNTAX

```
                [ segmentnumber]
-LISTSL [[SEGMENT,]segmentname]
        [ENTRY,procedurename]
```

PARAMETERS

segmentnumber	The number of the segment you would like to see listed. The *segmentnumber* is interpreted as an octal number.
segmentname	The name of the segment you would like to see listed. If omitted, all segments are listed.
SEGMENT	If specified, Segmenter will only list the segment that is identified by *segmentname*.
ENTRY	If specified, Segmenter will list the segment that contains the procedure identified by *procedurename*.
procedurename	The name of the procedure you would like to see a listing for, with information as to what segment it resides in.

EXAMPLE

-SL YOURSL	Opens the SL file YOURSL for management.
-LISTSL	Lists all procedure entry points and external references in the SL file YOURSL.
-LISTSL MYSEG	Lists all procedure entry points and external references in the segment MYSEG, which is in the SL file YOURSL.
-LISTSL SEGMENT,MYSEG	Same as -LISTSL MYSEG.
-LISTSL 57	Lists all procedure entry points and external references in segment number 57.
-LISTSL ENTRY,BINARY	Lists the procedure entry point BINARY and the segment name that contains the BINARY procedure.

ADDITIONAL DISCUSSION

Page 2-34.

-LISTUSL

Segmenter Command

Lists the specified segments in the currently-managed User Subprogram Library (USL) file.

SYNTAX

```
-LISTUSL [segmentname]
```

PARAMETERS

segmentname	The name of the segment you would like to see listed. If omitted, all segments in the USL are listed.

EXAMPLE

-USL SEARCHUSL	Opens the USL file SEARCHUSL for management.
-LISTUSL LOCKSEG	Lists the segment LOCKSEG, which is in the USL SEARCHUSL.

ADDITIONAL DISCUSSION

Page 2-12.

Changes the segment name associated with a Relocatable Binary Module (RBM) in the currently-managed USL file.

SYNTAX

```
-NEWSEG newsegname,rbmname[(index)]
```

PARAMETERS

newsegname The new segment name to be associated with the RBM. If it does not already exist, a new segment is created.

rbmname The name of the RBM whose segment name is to be changed.

index An integer further specifying the RBM name. The index may be used when more than one RBM of the same name exists in the User Subprogram Library (USL) file. If *index* is omitted, a default value of 0 (the most recent active occurrence) is assigned.

EXAMPLE

```
-NEWSEG NEWNAME,RB3
```

Changes the segment name associated with RBM RB3 to NEWNAME.

ADDITIONAL DISCUSSION

Page 2-5.

Prepares the active Relocatable Binary Modules (RBMs) in the currently-managed User Subprogram Library (USL) file into a program file.

SYNTAX

```
-PREPARE progfile
            [;ZERODB]
            [;PMAP]
            [;MAXDATA=segsize] [;PATCH=patchsize]
            [;STACK=stacksize]
            [;DL=dlsize] [{;FPMAP  }]
                         {;NOFPMAP}
            [;CAP=caplist]
            [;RL=filename]
            [;FPMAP]
            [;NOSYM]
```

PARAMETERS

progfile — The name of the program file on which the prepared program segments are to be written. If the named program file does not exist, the Segmenter will build a job temporary file for you.

NOTE

Code segments in a program file cannot cross disc extent boundaries. Thus, all segments in such files must be constructed within one extent. See the MPE File System Reference Manual (30000-90236) for a discussion of disc extents.

ZERODB — Request to initialize to zero the initially-defined, user-managed (DL-DB) area of the stack, and uninitialized portions of the DB-Q (initial) area of the stack. Default is that these areas are not affected.

stacksize — The size of the user's initial local data area (Z to Q initial) in the stack, in words. This overrides the *stacksize* estimated by the Segmenter, which applies if the *stacksize* parameter is omitted. (The default is a function of estimated stack requirements for each program unit in the program.) Since it is difficult for the system to predict the behavior of the stack at run time, you may want to override the default by supplying your own estimate with *stacksize*. A value of -1 denotes the default, which is equivalent to omitting the parameter.

dlsize — The DL-DB area to be initially assigned to the stack. This area is of interest mainly in programmatic applications. Due to system logging considerations, the DL-DB area is always rounded upward so that the distance from the beginning of the stack data segment to the DB-address is a multiple of 128 words. The value you specify must be within -1 and 32,767 words. The default is estimated by the MPE Segmenter.

PMAP — Request to produce a descriptive listing of the prepared program on file whose formal designator is SEGLIST. If no :FILE equation is found referencing SEGLIST, listing is produced on $STDLIST. Default is no listing. See Appendix G for a listing of a prepared program.

segsize — Maximum stack area (Z-DL) size permitted, in words. This parameter is included if you expect to change the size of the DL-DB or Z-DB areas during process execution. If omitted, MPE assumes that these areas will not be changed. A value of -1 denotes the default, which is equivalent to omitting the parameter. Regardless what you specify, MPE may change the *segsize* to accommodate table overflow conditions.

If you prepare your program with *segsize* less than the configured minimum, the value is rounded up to the minimum or the amount needed by the program (as calculated by the Segmenter). The maximum actual *segsize* permitted a program is 32,767 words. You may prepare your program with a *segsize* larger than necessary so long as this maximum is not exceeded. If the specified *segsize* does exceed the maximum, it will be rounded down to 32,767 words.

patchsize — Specifies the size of the patch area. This size will apply to all segments within the program file. The value you specify must be within -1 and 16,380 words.

FPMAP or NOFPMAP — Includes or excludes the internal PMAP information. FPMAP is a request to have internal PMAP information included in the program. NOFPMAP excludes PMAP information from the program when the system FPMAP or job/session FPMAP is on. If the symbolic debug option is invoked, default is FPMAP. Otherwise the default is NOFPOMAP.

caplist

Capability-class attributes associated with program, specified as two-character mnemonics. If more than one mnemonic is specified, each must be separated from its neighbor by a comma. The mnemonics are:

IA Interactive access
BA Local batch access
PH Process handling
DS Data segment management
MR Multiple resource management
PM Privileged mode

Users who issue the -PREPARE command can only specify capabilities that they possess (through assignment by the Account Manager). If the user does not specify any capabilities, only IA and BA (if the user possesses them) will be assigned to this program.

filename

The name of a Relocatable Library (RL) file to be searched to satisfy external references during preparation. This can be any permanent file of type RL. It need not belong to the log-on group, nor does it have a reserved, local name. This file yields a single segment that is incorporated into the segments of the program file prepared. If *filename* is omitted, no library is searched. READ and LOCK access to the RL file are necessary during PREPARE.

NOSYM

Suppresses the symbolic debug option. Refer to the HP Toolset Reference Manual (32350-90001).

EXAMPLE

```
-USL OURUSL
```

Opens the USL file OURUSL for management.

```
-PREPARE PFILE;PMAP;
 STACK=10;CAP=IA,PM;
 RL=MYRL;FPMAP
```

Prepares the active Relocatable Binary Modules (RBMs) in the USL file OURUSL into a program file called PFILE. The file does not exist prior to preparation. We have requested a PMAP describing the prepared program. The initial local data area will be 10 words (STACK=10). PFILE will have two capabilities associated with it: interactive access and privileged mode.

During preparation, the Segmenter will search the RL file MYRL to satisfy external references within the source code. In addition to asking for a display of information about the prepared program file (;PMAP), we have used the ;FPMAP parameter to request that PMAP information be made a part of the program file.

ADDITIONAL DISCUSSION

Page 2-22.

Deletes one or more Relocatable Binary Modules (RBMs) from the currently-managed User Subprogram Library (USL) file.

SYNTAX

```
-PURGERBM [UNIT,    ] name[(index)]
          [SEGMENT,]
```

PARAMETERS

UNIT	Deletes the RBM identified by *name* [(*index*)].
SEGMENT	Deletes all RBMs associated with the segment *name*. The default parameter is UNIT.
name	The name of the RBM to be deleted if UNIT is specified. If SEGMENT is specified, this is the name of the segment in which all RBMs are to be deleted.
index	An integer further specifying the RBM name. The index may be used when more than one RBM of this name exists in the USL file. If *index* is omitted, a default value of 0 (the most recent active occurrence) is assigned. If SEGMENT is specified, this parameter is ignored.

EXAMPLE

-PURGERBM UNIT,XPOINT	Deletes the RBM named XPOINT.

ADDITIONAL DISCUSSION

Page 2-6.

-PURGERL

Segmenter Command

Deletes an entry point of a procedure, or the entire procedure, from the currently-managed Relocatable Library (RL) file.

SYNTAX

```
-PURGERL [UNIT, ] name
         [ENTRY,]
```

PARAMETERS

UNIT	Deletes the entire procedure identified by *name*.
ENTRY	Deletes the entry point identified by *name*. (Default.)
name	The name of the procedure to be deleted if **UNIT** is specified, or the entry point to be deleted if **ENTRY** is specified.

EXAMPLE

-PURGERL UNIT,PROC1	Deletes the entire procedure named **PROC1**.

ADDITIONAL DISCUSSION

Page 2-30.

Deletes a segment entry point, or the entire segment, from the currently-managed Segmented Library (SL) file.

SYNTAX

```
-PURGESL [ENTRY,  ] name
         [SEGMENT,]
```

PARAMETERS

ENTRY	Deletes the entry point identified by *name* from the directory or the SL file. (Default.)
SEGMENT	Deletes the entire segment identified by *name* from the SL file.
name	The name of the entry point to be deleted from the directory of the SL file if **ENTRY** is specified or the name of the segment to be deleted from the SL file if SEGMENT is specified.

EXAMPLE

-PURGESL ENTRY,ENT1	Deletes the entry point named ENT1 from the directory of the SL file.

NOTE

PURGESL ENTRY,ENT1 does not remove the procedure that contains entry point ENT1 from the SL file. It only removes ENT1 from the directory of the SL file. So, the entry ENT1 becomes an internal entry point (that is, procedures in the same code segment can access it but procedures in other code segments cannot). However, if ENT1 is the only entry point in the code segment, this command is equivalent to PURSESL SEGMENT,*segname* and deletes the code segment.

ADDITIONAL DISCUSSION

Page 2-42.

-REDO

Segmenter Command

Allows you to edit a command entry.

SYNTAX

```
-REDO [edit request]
```

PARAMETERS

edit request Any valid edit request string as described under operation of the -REDO command. The edit request is applied to the last command that was entered and the resultant command is then executed. If an edit request is entered as a parameter to the -REDO command, the user is not prompted for any more changes.

OPERATION

The -REDO command allows you to correct certain kinds of errors in an incorrect command entry or to change a correct command entry. The -REDO command only applies to the last command entered. When the -REDO command is entered, Segmenter enters a mode similar to the Editor and displays the command to be modified.

To modify the command output by SEGMENTER, position the cursor (using the space bar on the terminal) under the character(s) to be modified, then enter one of the following sub-commands:

D Delete. Deletes the character above the cursor. If D is repeated, each character above each D is deleted.

I Insert. Inserts one or more characters immediately preceding the character above the cursor. The D and I sub-commands can be used in conjunction to delete characters, then insert new characters.

R Replace. Replaces the characters above the cursor with new characters. If one character is entered, the character above the cursor is replaced; if two characters are entered, two characters, (the character above the cursor and the character to the right) are replaced; and so forth for additional characters. R is the default sub-command.

U Undo. Cancels the effect of the previous D, I, or R sub-command. Entering a U, carriage-return, then another U cancels all previous sub-commands for this -REDO command and restores the line being corrected to its original form.

EXAMPLE

```
-BUILDS MYSL,2000,1
*** ERROR ***
ILLEGAL COMMAND NAME
REDO                          (Request to enter command string)
BUILDS MYSL,2000,1            (SEGMENTER displays command)

    IL                        (Insert L)
BUILDSL MYSL,2000,1           (Corrected command displayed)
```

Note that the letter S can be entered without the sub-command **R**, because **R** is the default sub-command. For example,

```
-BUILDXL MYSL,2000,1
      S                       (Replace X with S)
-BUILDSL MYSL,2000,1          (Corrected command displayed)
```

Reveals entry point. Cannot use on a secondary entry point or on material which will be placed in an RL file. Can be used on a primary entry point, RBM, or segment.

SYNTAX

```
          [ENTRY,   ]
-REVEAL   [UNIT,    ] name[(index)]
          [SEGMENT,]
```

PARAMETERS

ENTRY	Reveals the entry point indicated by *name* [(*index*)]. (Default.)
UNIT	Reveals all entry points in the Relocatable Binary Module (RBM) indicated by *name* [(*index*)].
SEGMENT	Reveals all entry points in all RBMs associated with segment *name*.
name	The name of the entry point in the User Subprogram Library (USL) file which is to be revealed.
index	An integer further specifying the entry point name. The index may be used when there is more than one entry point of this name. If *index* is omitted, a value of 0 (the most recent active occurrence) is assigned by default.

EXAMPLE

-REVEAL RBM1	Reveals the RBM entry point named RBM1.

ADDITIONAL DISCUSSION

Page 2-44.

Opens a Relocatable Library (RL) file for management.

SYNTAX

```
-RL filereference
```

PARAMETERS

filereference The name, and optional group and account name (optional), of the RL file.

EXAMPLE

```
-RL R2.G1.A1
```

Opens the RL file R2.G1.A1 for management.

ADDITIONAL DISCUSSION

Page 2-25.

-SETFPMAP

Segmenter Command

Allows you to set or clear the System and the Job/Session FPMAP flags. Setting either of these flags allows you to generate FPMAP information without having to modify your existing Segmenter job or command files.

SYNTAX

```
                {SYSTEM} [;{CONDITION   }   ]  }
-SETFPMAP [                {UNCONDITION}        ][;    {;ON }   ]
                {SESSION                           }   {;OFF}
```

PARAMETERS

SYSTEM — Sets the system-wide FPMAP flag and initializes the job/session flag at logon. You must have System Manager (SM) capability to set the system-wide FPMAP flag.

- UNCONDITION (ON) — Generates FPMAP information for all program files and SL segments prepared on the system (regardless of the setting of the Job/Session FPMAP flag or the presence of the FPMAP keyword). Sets the Job/Session FPMAP flag ON at the beginning of each job and session.
- CONDITION (ON) — Initializes the Job/Session FPMAP flag to ON at the beginning of a job or session.
- OFF — Initializes the Job/Session FPMAP flag to OFF at the beginning of a job or session.

SESSION — Controls FPMAP generation for -PREPARE and -ADDSL commands performed within that job or session.

- ON — Generates FPMAPs for all program and SL files prepared in the current job or session, unless overridden by the NOFPMAP keyword in a -PREPARE or -ADDSL command.
- OFF — Inhibits FPMAP generation for all program and SL files prepared in the current job or session, unless overridden by -SYSTEM; UNCONDITION;ON or by the FPMAP keyword in a given -PREPARE or -ADDSL command.

EXAMPLE

```
-SETFPMAP SYSTEM;ON

-SETFPMAP SYSTEM;UNCONDITION;ON

-SETFPMAP SESSION;ON
```

ADDITIONAL DISCUSSION

Page 3-10a.

Displays the current status of the System and the Job/Session flags and lists those files currently opened by the Segmenter.

SYNTAX

```
-SHOW
```

EXAMPLE

```
USL FILE          :       MYUSL.MYGROUP.MYACCT
AUX USL FILE      :       NONE
SL FILE           :       MYSL.MYGROUP.MYACCT
RL FILE           :       NONE
SYSTEM FPMAP      :       ON (CONDITION)
SESSION FPMAP     :       OFF
```

ADDITIONAL DISCUSSION

Page 3-10a.

-SL

Segmenter Command

Opens Segmented Library (SL) file for management.

SYNTAX

```
-SL filereference
```

PARAMETERS

filereference	The name of the SL file.

EXAMPLE

-SL SL1	Opens the SL file named SL1 for management.

ADDITIONAL DISCUSSION

Page 2-34.

Activates one or more program unit (Relocatable Binary Module) entry points in the User Subprogram Library currently designated for management. Note that, in the order the parameters are listed below (ENTRY, UNIT, SEGMENT), the control given you by the USE parameters becomes less specific. If you want control over a single entry point, use the most specific and limited parameter, ENTRY. (You will probably also use the optional *index* whenever you use this specific parameter.) Either of the other two parameters would activate all entry points within an RBM or a segment, including some you might want to leave inactive.

SYNTAX

```
        [ENTRY,   ]
-USE    [UNIT,    ] name[(index)]
        [SEGMENT,]
```

PARAMETERS

ENTRY	Activates the entry point indicated by *name* [(*index*)].
UNIT	Activates all entry points in the RBM indicated by *name* [(*index*)].
SEGMENT	Activates all entry points in all RBMs associated with the segment *name*.
	The default value is ENTRY.
name	The name of the entry point if ENTRY is specified, or the name of the RBM if UNIT is specified, or the name of the segment if SEGMENT is specified.
index	An integer further identifying the entry point name. The *index* may be used when the USL file contains more than one entry point of the same name. If SEGMENT is specified, the *index* parameter is ignored.
	If *index* is omitted, a default value of 0 (the most recent inactive occurrence) is assigned. An *index n* means the *n*th occurrence, whether active or inactive. However, there is no point in using this command with an entry point you know to be active, since its purpose is to activate inactive entry points.

EXAMPLE

-USE UNIT,RB20	Activates all entry points in the RBM named RB20.

ADDITIONAL DISCUSSION

Page 2-3, 2-9.

-USL

Segmenter Command

Opens a User Subprogram Library (USL) file for management.

SYNTAX

```
-USL filereference
```

PARAMETERS

filereference	The name, and group and account name (optional), of the USL file.

EXAMPLE

-USL FILE1	Opens the USL file FILE1 (in the user's log-on group and account) for management.

ADDITIONAL DISCUSSION

Page 2-11.

Adjusts directory space in a User Subprogram Library (USL) file.

SYNTAX

```
   I                          IV          IA
errnum:=ADJUSTUSLF(uslfnum,records);
```

FUNCTIONAL RETURN

This intrinsic returns an error number if the condition code is equal to CCL.

PARAMETERS

uslfnum *integer by value.*
A word supplying the file number of the USL file (as returned by FOPEN).

records *integer by value.*
A word supplying a signed record count. If *records* is greater than zero, the information block is moved toward the end-of-file in the USL file, increasing the space available for the directory block and decreasing the space available for the information block. If *records* is less than zero, the information block is moved toward the start of the USL file, decreasing the directory-block space and increasing the information-block space.

CONDITION CODES

CCE — Request granted.

CCG — Not returned by this intrinsic.

CCL — Request denied. One of the following error numbers is returned:

Error No.	Meaning
0	The file specified by *uslfnum* was empty, or an unexpected end-of-file was encountered when reading the old *uslfnum*, or an unexpected end-of-file was encountered when writing on the new *uslfnum* .
1	Unexpected I/O error occurred. This can occur on the old *uslfnum* or the new *uslfnum* to which the intrinsic is copying the information.

4	Attempt to exceed maximum directory size (32K words).
5	Insufficient directory space.
6	Insufficient space was available in the USL file information block.

ADDITIONAL DISCUSSION

Page 2-22.

Copies only the active entries of the currently-managed User Segmented Library (USL) file into a new, "clean" USL file.

This intrinsic requires 2900 words of user stack space.

CLEANUSL creates the new USL file as a job temporary file. To make the file permanent, you must call the FCLOSE intrinsic.

SYNTAX

```
    I                       IV          BA
filenum:=CLEANUSL(uslfnum,filename);
```

FUNCTIONAL RETURN

This intrinsic returns the new file number. If an error occurs, the error number is returned instead of the new file number. The condition code therefore must be tested immediately on return from this intrinsic. If an error number were to be used as a file number, unpredictable results would occur.

PARAMETERS

uslfnum *integer by value.*
A word identifier supplying the file number of the file.

filename *byte array.*
The name to be given to the new USL file. It may be fully qualified.

CONDITION CODES

CCE — Request granted. The new file number is returned.

CCG — Not returned by this intrinsic.

CCL — Request denied. CLEANUSL returns one of the following error numbers in *filenum* :

Error No.	Meaning
0	Unexpected end of file marker on either the old or the new USL file.

1	Unexpected I/O error on either the old or the new USL file.
7	Unable to open new USL file.
12	Invalid USL file.

ADDITIONAL DISCUSSION

Page 2-17.

Fills a file with external FPMAP records representing either all of the FPMAP data stored in a program or SL file, or only that for a specified segment. One external record is created as each FPMAP record is encountered.

SYNTAX

```
                IV        IV          IV          I      IV        I
DUMPPMAP (progfnum,pmapfnum,xpmapreclen,reccount,segnum,status);
```

PARAMETERS

progfnum *integer value*
The number of the program or SL file whose FPMAP is to be dumped, as returned from the FOPEN intrinsic. The caller is responsible for opening the file with MULTIRECORD, READ access.

pmapfnum *integer value*
The number of the file that is to contain the external FPMAP records, as returned from the FOPEN intrinsic. The caller is responsible for opening the file with write access and records of size at least *xmapreclen* words.

xpmapreclen *integer value*
The number of words in each external FPMAP record to be written to file *pmapfnum*. If the caller requests more words than are currently defined for an external FPMAP record, the extra words in each record written are set to zero.

reccount *integer*
The number of external FPMAP records which were (or could have been) generated from the FPMAP data found in the program or SL file. If the caller's *pmapfnum* file is large enough, it is also the number of records written to that file.

segnum *integer value*
The number of the segment whose FPMAP data is to be dumped. When *segnum* is -1, all FPMAP records for all segments in the file are dumped.

status *integer*
A word that contains a Condition Code that tells you if the procedure call was successful and, if not, why it failed.

CONDITION CODES

	Error No.	Meaning
CCE	0	No errors detected.
CCG	4	External PMAP file was too small.
CCL	10	Program/SL file did not contain a PMAP.
	11	Program/SL file code was not that of a program/SL file, or, SL loader ID was not compatible with this version of the PMAP intrinsic.
	12	File system error on the program/SL file.
	13	File system error on the external PMAP file.

ADDITIONAL DISCUSSION

Page 3-10a.

Changes length of a User Subprogram Library (USL) file.

SYNTAX

```
   I                          IV          IV
filenum:=EXPANDUSLF(uslfnum,records);
```

FUNCTIONAL RETURN

This intrinsic returns the new file number. If an error occurs, the error number is returned instead of the new file number. The condition code therefore must be tested immediately on return from this intrinsic. If an error number were to be used as a file number, unpredictable results would occur.

PARAMETERS

uslfnum *integer by value.*
A word identifier supplying the file number of the file.

records *integer by value.*
A signed integer specifying the number of records by which the length of the USL file is to be changed. If *records* is positive, the new USL file is longer than the old USL file. If *records* is negative, the new USL file is shorter than the old USL file.

CONDITION CODES

CCE — Request granted. The new file number is returned.

CCG — Not returned by this intrinsic.

CCL — Request denied. One of the following error numbers is returned:

Error No.	Meaning
0	The file specified by *uslfnum* was empty, or an unexpected end-of-file was encountered when reading the old *uslfnum*, or an unexpected end-of-file was encountered when writing on the new *uslfnum*.

1	Unexpected I/O error occurred. This can occur on the old *uslfnum* or the new *uslfnum* to which the intrinsic is copying the information.
3	Your request attempted to exceed the maximum file size (32,768 words).
6	Insufficient space was available in the USL file information block.
7	The intrinsic was unable to open new USL file.
8	The intrinsic was unable to close (purge) the old USL file.
9	The intrinsic was unable to close (save) the new USL file.
10	The intrinsic was unable to close $NEWPASS.
11	The intrinsic was unable to open $OLDPASS.

ADDITIONAL DISCUSSION

Page 2-22.

FINDPMAPADDR

Intrinsic

Searches the FPMAP in a program or SL file for a specific address in a particular segment. An external FPMAP record corresponding to the nearest entry point at or preceding *address* is returned. The caller must have MULTI-RECORD READ access to open the file.

SYNTAX

```
                    IV       IV       IV       V
FINDPMAPADDR (progfnum,segnum,address,xpmaprec,

                        IV         I
                    xpmapreclen,status);
```

PARAMETERS

progfnum *integer value*
The number of the program or SL file of the FPMAP to be searched, as returned from the FOPEN intrinsic. Caller is responsible for opening the file with MULTI-RECORD READ access.

segnum *integer value*
The number of the logical segment in which the address is to be found.

address *integer value*
The PB-relative address in segment *segnum* with which the external FPMAP record returned is associated.

xpmaprec *value*
The external FPMAP record corresponding to the first entry point at or preceding *address*. If the entry point located has more than one name, the first one encountered in the FPMAP will be returned.

xpmapreclen *integer value*
The number of words of the external FPMAP record returned in *xpmaprec*. If the call requests more words than are currently defined for an external FPMAP record, zeros are returned in the extra words.

status *integer*
A word that contains a Condition Code that tells if the procedure call was successful and, if not, why.

CONDITION CODES

	Error No.	Status
CCE	0	The address was valid and an external FPMAP record was returned.
CCG	2	The address was out of range.
	3	The logical segment *segnum* did not exist.
CCL	10	The file *progfnum* did not contain FPMAP data.
	11	The file *progfnum* was not a program of SL file.
	12	A file system I/O error occurred on file *progfnum*.

ADDITIONAL DISCUSSION

Page 3-10a.

Searches the FPMAP in a program or SL file for a segment name or an entry point name.

SYNTAX

```
                    IV        BA        BA       IA
FINDPMAPNAME (progfnum,segname,entname,xpmaprec,

                     V          I
                xmapreclen,status);
```

FUNCTIONAL RETURN

This intrinsic returns the external FPMAP record specified. If unable to find the segment or entry point specified, the intrinsic returns a status code that indicates why the search failed.

PARAMETERS

progfnum *integer value*
The number of the program or SL file whose FPMAP is searched, as returned from the FOPEN intrinsic. The caller must have MULTI-RECORD and READ access to open the file.

segname *byte array*
Optional. The name of the segment to be found. Terminate with a blank. If the first character of *segname* is blank, *segname* is omitted from the calling sequence. A maximum of 15 characters can be specified for the search. All characters will be upshifted.

entname *byte array*
Optional. The name of the procedure or secondary entry point to be found. Terminate with a blank. If the first character of *entname* is blank, *entname* is omitted from the calling sequence. A maximum of 15 characters can be specified for the search. All characters will be upshifted.

xpmaprec *integer array*
The external FPMAP record corresponding to the entry point passed in *entname*. The format of the record is defined in figure 3-1. The record returned may be a segment, a procedure, or a secondary entry point (depending upon *segname* and *entname* specified).

xpmapreclen *value*
The number of words of the external FPMAP record returned in *xpmaprec*. If the call requests more words than are currently defined for an external FPMAP record, zeros are returned in the extra words.

status *integer*
A word that contains a Condition Code that tells if the procedure was successful and, if not, why.

CONDITION CODES

	Error No.	Meaning
CCE	0	No errors detected.
CCG	1	Entry point name not located.
	2	Address to be located outside the bounds of segment specified.
	3	Program/SL file did not contain segment specified, or, the SL segment requested was marked as deleted.
	4	External PMAP file was too small.
CCL	9	Option variable parameter list was illegal.
	10	Program/SL file did not contain a PMAP.
	11	Program/SL file code was not that of a program/SL file, or, SL loader ID was not compatible with this version of the PMAP intrinsics.
	12	File system error on the program/SL file.
	13	File system error on the external PMAP file.
	14	Internal PMAP file improperly FOPENed.
	15	Bad internal PMAP.

ADDITIONAL DISCUSSION

Page 3-10a.

Initializes a buffer corresponding to record 0 for a User Subprogram Library (USL) file to the empty state.

SYNTAX

```
                I                    IV        IA
errnum := INITUSLF (uslfnum, records);
```

FUNCTIONAL RETURN

This intrinsic returns an error number if the condition code is equal to CCL.

PARAMETERS

uslfnum *integer by value.*
A word identifier supplying the file number of the USL file.

records *integer array.*
A 128-word buffer, corresponding to the first record of the USL file (record 0), to be initialized to the empty state. This buffer should be set to all zeros. The intrinsic will set certain values in record 0 before returning to the calling program.

CONDITION CODES

CCE — Request granted.

CCG — Not returned by this intrinsic.

CCL — Request denied. The following error number is returned:

Error No.	Meaning
3	Your request attempted to exceed the maximum file size (32,768 records), or was smaller than the minimum file size (4 records).

ADDITIONAL DISCUSSION

Page 2-21.

SEGMENTER ERROR MESSAGES

SECTION V

Table 5-1. Segmenter Error Messages

#	ERROR MESSAGE	COMMENTS	ACTION
-	ILLEGAL COMMAND NAME	Entry unacceptable.	Check command entered. Check syntax and spelling.
-	ILLEGAL NUMBER OF PARAMETERS	Entry unacceptable.	Check number of parameters.
-	ILLEGAL PARAMETER	Entry unacceptable.	Review acceptable parameters. Check spelling and syntax.
-	COMMAND BUFFER OVERFLOW	The maximum size of the Segmenter command buffer is 160 characters.	Check command entered. Eliminate blanks where possible and re-enter.
-	INSUFFICIENT CAPABILITY	Must have Interactive Access (IA) and Batch Access (BA) to use Segmenter. Some commands require special capabilities.	Check requirements for individual commands. Check your capabilities. Have System Manager change if necessary.
-	ILLEGAL USE OF PARAMETERS	Entry unacceptable.	Check command specifications.
-	ILLEGAL DELIMITERS	Entry unacceptable.	Check command specifications.
0	ILLEGAL ENTRY	Bad USL generated by compiler. ***numericparm*** is the decimal address of the entry in the USL.	Re-compile.
1	ILLEGAL HEADER	Bad USL generated by compiler. ***numericparm*** is the decimal address of the entry in the USL.	Re-compile.
2	ATTEMPT TO EXCEED MAXIMUM DIRECTORY SPACE	USL file maximum directory size is %77777.	Find alternate means of storing some of the USL material. (RL file or SL file).

Table 5-1. Segmenter Error Messages (Continued)

#	ERROR MESSAGE	COMMENTS	ACTION
3	AVAILABLE DIRECTORY SPACE EXHAUSTED	USL file is too small.	Use the Segmenter -COPYUSL command to expand USL; or use ADJUSTUSLF intrinsic to programatically expand the space in the USL directory.
4	AVAILABLE INFO SPACE EXHAUSTED	USL file too small or compiler error.	Use -COPYUSL to expand USL; or ADJUSTUSLF to expand the information space in the USL.
5	USL FILE NOT DESIGNATED	Tried to -USE, -CEASE, -PREPARE, -NEWSEG, -LISTUSL, -COPY, -ADDRL, -ADDSL, -HIDE, -REVEAL or -PURGERBM without specifying -USL.	Specify USL using the Segmenter -USL command.
6	ILLEGAL USL FILE SPECIFICATION	USL file length is less than 4 records or greater than 32768 records.	Check the file size of the USL file specified.
7	UNABLE TO OPEN USL FILE	Non-existent permanent file. FOPEN failure in -AUXUSL, -BUILDUSL, or -USL. ***numericparm*** is FCHECK value.	Check file name.
8	INVALID USL FILE	File code is not USL or USL file is bad.	Check file name of USL specified to be sure it is a USL.
9	UNABLE TO CLOSE USL	Filename already in use for some other permanent or temporary file. FCLOSE failure during -EXIT or other USL specification. ***numericparm*** is FCHECK value.	No action possible. File may be lost.
10	UNABLE TO CLOSE SL FILE	Filename already designated for some other permanent or temporary file. FCLOSE failure during -EXIT or on previous SL during -SL or -BUILDSL. ***numericparm*** is FCHECK value.	No action possible. File may be lost.

Table 5-1. Segmenter Error Messages (Continued)

#	ERROR MESSAGE	COMMENTS	ACTION
11	AVAILABLE FILE SPACE EXHAUSTED	RL or SL file is full.	For an SL, use the Segmenter -COPYSL command to move the material to a larger file. For an RL, if you have the source code available, build another, larger RL; recompile and re-prepare the code. If you have the USL available, use Segmenter commands to move the material into a new, larger RL or into an SL.
12	ENTRY POINT ALREADY DEFINED	Attempted to add duplicate procedure name to RL or SL. ***stringparm*** is entry name.	Re-name and recompile procedure or purge the already existing procedure.
13	SEGMENT CONTAINS PROGRAM UNITS OTHER THAN PROCEDURES	Attempted to put outer block in SL, where a segment resides which already has an outer block. Only a program file can contain an outer block. Outer block is present.	Re-compile subprogram or -CEASE outer block.
14	SEGMENT REQUIRES GLOBAL STORAGE	Tried to put global variables in SL. FORTRAN DATA, or SPL GLOBAL or OWN variables were specified. ***stringparm*** is entry name.	Use Segmenter commands to put the code in an RL; or leave the code in the USL and prepare it from there.
15	SEGMENT ALREADY DEFINED	-ADDSL ***segname*** and ***segname*** already exists. ***stringparm*** is ***segname***.	If your source code is available, rename and recompile the segment; then use -ADDSL. If source code is not available, purge the existing segment.
16	SL FILE NOT DESIGNATED	Tried to -PURGESL, -ADDSL, or -LISTSL with no SL file open.	Open the SL file using the Segmenter -SL command.
17	ILLEGAL SL FILE SPECIFICATION	SL file length is less than 4 records or greater than %77777 records (or greater than %177777 for SL.PUB.SYS).	Check file size of specified -SL.

Table 5-1. Segmenter Error Messages (Continued)

#	ERROR MESSAGE	COMMENTS	ACTION
18	UNABLE TO OPEN SL FILE	FOPEN failure during -BUILDSL or -SL. ***numericparm*** is FCHECK value.	Check FCHECK value.
19	INVALID SL FILE	File code is not SL or SL file is bad.	Check file code.
20	ILLEGAL RL FILE SPECIFICATION	RL file length is less than 4 records or greater than %77777 records.	Check file size.
21	RL FILE NOT DESIGNATED	Tried to -ADDRL, -LISTRL, or -PURGERL without opening RL.	Open the RL file using the Segmenter -RL command.
22	INVALID RL FILE	File code is not RL or RL file is bad.	Check file code.
23	UNABLE TO CLOSE RL FILE	FCLOSE failure on RL file. ***numericparm*** is FCHECK value.	Check file system error message.
28	PROCEDURE HAS NO USABLE ENTRY POINT	RBM has no entry point.	Check RBMs in USL file.
30	UNABLE TO OPEN RL FILE	FOPEN failure. ***numericparm*** is FCHECK value.	Check file system error message.
32	INVALID PROGRAM FILE	File is not program file.	Check file code.
33	ILLEGAL CAPABILITY SPECIFICATION	Program capability specification is greater than the user's.	Request that System Manager change capabilities.

Table 5-1. Segmenter Error Messages (Continued)

#	ERROR MESSAGE	COMMENTS	ACTION
34	MORE THAN ONE EXTENT USED	Occurs when you have inherited a program file which is not large enough to hold additional material you have tried to PREPARE into it. Not all code is in one extent. Program will not run unless code is contiguous on disc.	Purge the old file and PREPARE your program again. The Segmenter will open a new program file of the proper size.
35	NO PROGRAM TO PREPARE	No program in USL.	Compile your program.
36	UNABLE TO CLOSE PROGRAM FILE	FCLOSE failure. ***numericparm*** is FCHECK value.	Check FCHECK value.
37	UNABLE TO OPEN PROGRAM FILE	FOPEN failure. ***numericparm*** is FCHECK value.	Check FCHECK value.
38	DATA SEGMENT OVERFLOW	Data is greater than %37777 words.	Examine program. Reduce requested DL size or rewrite program with less global data.
39	TOO MANY CODE SEGMENTS	More than 255 segments for programs or greater than 510 for SL.PUB.SYS, or greater than 254 for all other SLs.	Re-segment your code.
40	CODE SEGMENT OVERFLOW	Code segment length is greater than %37774.	Re-segment your code.
41	STT OVERFLOW	Too many PCAL instructions. ***stringparm*** is entry name where overflow occurred.	Rewrite your program, reducing the number of references and thus the number of inter- and intra-segment transfers.
42	SEGMENT HAS NO USABLE ENTRY POINT	You have attempted to add to an SL a segment which has had all entry points deactivated.	Select one and re-activate it.
43	UNABLE TO ACCESS PROCEDURE	Illegal P-Label or ***parameters*** not matching.	Make all calling sequences to one particular procedure compatible.

Table 5-1. Segmenter Error Messages (Continued)

#	ERROR MESSAGE	COMMENTS	ACTION
44	REQUIRES PRIVILEGED MODE CAPABILITY	User needs privileged mode capability to add privileged segment.	Check your capabilities; see System Manager.
45	ACTUAL PARAMETERS INCOMPATIBLE WITH FORMAL PARAMETERS	(FORTRAN program) ***stringparm*** is procedure name. Several calls to one external procedure have different parameter types.	Make all calls to a particular procedure compatible.
46	PROGRAM UNIT CONTAINS FATAL ERROR	***stringparm*** is unit name.	Re-compile.
47	PROGRAM UNIT CONTAINS NON-FATAL ERROR	Although error is non-fatal, it may cause problems during preparation or execution. ***stringparm*** is unit name.	Re-compile.
48	CODE SEGMENT TOO LARGE	The length of the code segment exceeds the configured system maximum size.	Re-segment or rewrite your code.
49	ACTUAL FUNCTION INCOMPATIBLE WITH FORMAL FUNCTION.	Several calls to one external procedure have different functions.	Make all calling sequences to one particular procedure compatible.
50	INCOMPATIBLE NUMBER OF PARAMETERS.	Several calls to one external procedure have different numbers of parameters.	Make all calling sequences to one particular procedure compatible.
60	NO OUTER BLOCK IS ACTIVE	The prepared program would have no outer block.	Activate outer block.
61	MORE THAN ONE OUTER BLOCK IS ACTIVE	The prepared program would have more than one outer block.	Choose one version of outer block and deactivate others.

Table 5-1. Segmenter Error Messages (Continued)

#	ERROR MESSAGE	COMMENTS	ACTION
62	MORE THAN ONE OUTER BLOCK HAS ACTIVE ENTRY POINTS	The prepared program would have more than one outer block.	Choose one version of outer block and deactivate all entry points in others.
63	EXTERNAL VARIABLE NOT DECLARED GLOBAL	***stringparm*** is variable name.	Declare your external variable.
64	EXTERNAL VARIABLE INCOMPATIBLE WITH GLOBAL VARIABLE	***stringparm*** is variable name.	Correct your variable declaration.
66	TOO MANY COMMON DATA LABELS	(BASICOMP or FORTRAN program) You have exceeded the maximum allowable number of common data labels.	Re-code your program, reducing the number of common data labels.
67	COMMON DECLARED WITH DIFFERENT SIZE	***stringparm*** is common name.	None necessary; warning only.
68	ATTEMPT TO USE BLOCK DATA ON NON-EXISTENT COMMON	(FORTRAN program) You are trying to access a common block that was never initialized.	Check your program
69	ATTEMPT TO USE BLOCK DATA ON INCOMPATIBLE COMMON	(FORTRAN program) Your common block is too small.	Check your program.
70	ILLEGAL STACK SIZE	Stack size less than 0.	Check specifications of the command you entered.
71	ILLEGAL DL SIZE	DL size less than 0.	Check specifications of the command you entered.
72	ILLEGAL MAXDATA SIZE	MAXDATA less than 0.	Check specifications of the command you entered.

Table 5-1. Segmenter Error Messages (Continued)

#	ERROR MESSAGE	COMMENTS	ACTION
73	DUPLICATE ACTIVE ENTRY POINT NAME	Tried to prepare program or -ADDSL using a procedure name already in use for an active procedure.	De-activate the existing procedure or rewrite your code, changing the name of the procedure you were attempting to introduce.
74	UNABLE TO PREPARE WITH SYMBOLIC DEBUG	If any problem exists which interferes with the operation of Debug, the Segmenter will still prepare your program but will not include debug information.	Check Symbolic Debug installation.
80	INSUFFICIENT STORAGE	Could not obtain enough DL area (system problem). Program is too large for the Segmenter.	SEGPROC must be prepared with larger DL.
81	ILLEGAL PATCH	Bad header generated by a compiler.	Re-compile.
82	UNABLE TO OPEN SCRATCH FILE	Scratch area used to prepare code from USL before moving to SL file. ***numericparm*** is FCHECK value.	Check file system error message.
83	UNABLE TO OPEN LIST FILE	FOPEN failure. ***numericparm*** is FCHECK value.	Check file system error message.
84	UNEXPECTED I/O ERROR	***numericparm*** is FCHECK value and ***stringparm*** is file type. Files opened are: USL, AUXUSL, SL, RL, RL LIBRARY, PROGRAM, LIST and SCRATCH files.	Check file system error message. If -PREPAREing into an existing program file, it may be too small. :PURGE the old program file and re-issue the command; or -PREPARE the USL into a new file. (You will probably want to save the new file after you exit the Segmenter.)

Table 5-1. Segmenter Error Messages (Continued)

#	ERROR MESSAGE	COMMENTS	ACTION
86	ITEM DIFFERENT FROM CLASS SPECIFICATION	One of the following occurred: 1. Tried to -COPY an item of a different class than specified. 2. Tried to -PURGERBM an item that differs in class specified. 3. Tried to -USE item with a class that differs from that specified.	Correct class specification.
87	ITEM NOT PRIMARY ENTRY POINT	Tried to -ADDRL an item that is not a primary entry point.	Use -ADDRL for primary entry points only.
88	INCOMPATIBLE ITEM TYPE	One of the following occurred: 1. Tried to -HIDE an item that is not an entry point. 2. Tried to -NEWSEG an item that is not a procedure. 3. Tried to -PURGERBM an item that is not a UNIT.	Correct type specification.
89	INVALID CLASS SPECIFICATION	One of the following occurred: 1. Tried to -COPY an ENTRY. 2. Tried to -PURGERL a SEGMENT. 3. Tried to -PURGESL a UNIT.	Correct class specification.

Table 5-1. Segmenter Error Messages (Continued)

#	ERROR MESSAGE	COMMENTS	ACTION
93	UNABLE TO LOCATE ITEM	One of the following occurred: 1. Item not in AUXUSL for -COPY. 2. Item not in USL for -ADDSL. 3. Item not in USL for -ADDRL. 4. Item not in USL for -HIDE or -REVEAL. 5. Item not in USL for -NEWSEG. 6. Item not in RL for -PURGERL. 7. Item not in USL for -USE or -CEASE. 8. Item not in USL for -PURGERBM.	Check the item name you have specified to be sure the item exists and is spelled correctly.
94	UNEXPECTED END-OF-FILE	Segmenter internal error.	Send supporting documentation with Service Request to your Hewlett-Packard Service Representative.
95	INVALID COPY FACTOR	Tried to use a negative value or too large a value with -COPY.	Review command specifications; check your entry.
96	ILLEGAL FILE ACCESS	You do not have the necessary file access capability.	Have the file owner release the file, or have the System Manager increase your capability.
97	UNABLE TO CLOSE SCRATCH FILE	File system message; indicates problem with internal file.	Check file system error message.
110	SEGMENT CURRENTLY LOADED	Segment referenced is in use and has been loaded by the loader.	Wait until program using segment has finished executing.

Table 5-1. Segmenter Error Messages (Continued)

#	ERROR MESSAGE	COMMENTS	ACTION
111	SEGMENT CONTAINS EXTERNAL VARIABLE	FORTRAN. Problem occurs during -ADDSL. ***stringparm*** is entry name.	You cannot put this segment into an SL.
112	SEGMENT CONTAINS COMMON	FORTRAN. ***stringparm*** is entry name.	You cannot put this segment into an SL.
113	SEGMENT CONTAINS LOGICAL UNITS	FORTRAN. Problem occurs during -ADDSL. FORTRAN program contains reference to logical units.	You cannot put this segment into an SL.
120	AUXUSL FILE NOT DESIGNATED	Tried to -COPY with no AUXUSL open.	Open file with Segmenter -AUXUSL command.
121	UNABLE TO OPEN NEW USL FILE	FOPEN failure.	Check file system error message.
122	DUPLICATE FILE NAME	Tried to -COPY SL or USL to already-existing file (other than itself).	Purge existing file or re-build the newest file with a different name.

FORTRAN COMPILER CONVENTIONS

APPENDIX
A

- One RBM for:

- Main.
- Each subroutine.
- Each function.

Na me of main and of each subroutine or function can be up to 15 alphanumeric characters in length, starting with an alphabetic character. Names must be unique within 15 characters to meet the requirements of the Segmenter.

$CONTROL SEGMENT=*name*

- Used to assign an RBM to a specified segment.
- A *name* can be up to 15 alphanumeric characters in length, starting with an alphabetic character. Must be unique within 15 characters to meet the requirements of the Segmenter.
- In effect until the next $CONTROL SEGMENT command.
- Must precede each program unit that is to go into a specified segment.

If no $CONTROL SEGMENT=*name* command appears, the compiler uses SEG' as the default segment name of the current segment and places all RBMs into SEG'.

- Default name for MAIN is MAIN'.

- Can override with PROGRAM *name* statement.
- A *name* can be up to 15 alphanumeric characters in length, starting with an alphabetic. All *names* must be unique within 15 characters to meet the requirements of the Segmenter.

No dummy outer block (main) generated when subroutines/functions compiled separately.

$CONTROL CHECK=*n*

- Specifies the level of checking the Segmenter will perform on all calls to the subroutine function immediately preceded by the CHECK parameter. This parameter does not affect the level of checking of calls to other subroutines from this one.
- The level specified, *n*, is an integer in the range 0 to 3. It determines the amount of information placed in the USL.
- Level 3 is the default setting (check function type, number of procedure or function parameters, and the type of each parameter).
- Parameter can appear at the beginning of program units only.

SPL COMPILER CONVENTIONS

APPENDIX B

One RBM for:

- Outer block (initialization).
- Each procedure.

$C ONTROL SEGMENT=*segmentname*

- Used to assign an RBM to a specified segment.
- In effect until the next $CONTROL SEGMENT command.
- A *segname* may be up to 15 alphanumeric characters in length, starting with an alphabetic character. Must be unique within 15 characters to meet the requirements of the Segmenter.
- For procedures, the $CONTROL SEGMENT command must precede the procedure declaration of the first procedure in the segment.
- If a new segment is to be specified for the main program, the $CONTROL SEGMENT command follows the procedure and intrinsic declarations and precedes the global subroutines and main body.
- Global subroutines must be in the same segment as the main body.
- All program units having the same segment name are compiled into one segment.

If no $CONTROL SEGMENT command appears, the compiler uses SEG' as the default name of the current segment and places all RBMs into SEG'.

Outer block generated unless subprogram is specified.

$CONTROL SUBPROGRAM = *list_of_procedure_names*

- Permits independent compilation of specified procedures, allowing programmer to select parts of a large program for compilation (and subsequent preparation and execution).
- Minimizes the number of entries in the USL directory.
- If used, must be specified at the beginning of the program.
- Suppresses generation of the outer block.

OPTION EXTERNAL

- Specifies that the procedure body (or code) exists external to the program being compiled. The procedure is not included in the declaration and will be linked to the main program later by the operating system.
- When procedures are compiled separately to be called later with this option, they can use the EXTERNAL-GLOBAL mechanism to establish data linkages.

- EXTERNAL variable declarations link global variables in a separately compiled main program to variables in a procedure. The global variables must be declared with the GLOBAL attribute. See the SPL Programming Language Reference Manual (30000-90024) for two methods of using the EXTERNAL-GLOBAL mechanism.

- OPTION CHECK

- Specifies the level of checking the Segmenter will perform for OPTION EXTERNAL procedure declarations which will subsequently be called as externals by other programs.

- No checking performed if this option is not specified.

- If the procedure or function has a lower checking level, the Segmenter ignores the level indicated by OPTION CHECK and uses the lower level.

COBOL COMPILER CONVENTIONS

APPENDIX C

For information about COBOLII/3000 conventions, see the COBOLII/3000 Reference Manual (32233-90001).

One RBM for:

- Initialization (main).
- All consecutive sections with the same priority number (subprogram).

One segment for each RBM.

Can use priority numbers to assign sections to specific segments.

The outer block must appear at the top of the USL list (must be the first RBM prepared). That is, it must be the first module prepared that allocates (sets up) the secondary DB.

$CONTROL SUBPROGRAM generates non-dynamic subprogram.

$CONTROL DYNAMIC generates dynamic subprogram.

COBOL naming conventions: RBM

- Main and subprogram initialization program name: First 14 characters plus apostrophe.
- Main and subprogram section name: First 12 characters plus *nn*' (*nn*=priority number).
- Hyphens are not used in names.

COBOL naming conventions: SEGMENT

- Main initialization program name: First 15 characters.
- Subprogram initialization: First 14 characters plus apostrophe.
- Main and subprogram section name: First 12 characters plus *nn*' (*nn*=priority number).
- Hyphens are not used in names.

BASIC COMPILER CONVENTIONS

APPENDIX D

One RBM for:

- Outer block (initialization).
- Each compiled FAST-SAVE program.

All BASIC programs to be compiled into the same USL must have unique local *filenames*, since the compiler eliminates any qualifying lockword and group and account names.

- All *filename*s may be up to 15 alphanumeric characters in length, starting with an alphabetic, and may include apostrophe. These naming conventions also meet the requirements of the Segmenter.

$CONTROL SEGMENT=*segmentname*

- Used to assign an RBM to a specified segment.
- In effect until the next $CONTROL SEGMENT command.
- A *segmentname* may be up to 15 alphanumeric characters in length, starting with an alphabetic character, and may include an apostrophe.

If no $CONTROL SEGMENT command appears, the compiler uses SEG' as the default name of the current segment and places all RBMs into SEG'.

Outer block generated unless subprogram specified.

$CONTROL SUBPROGRAM

- Turns on the "subprogram" condition, which inhibits the compiler's generation of an outer block.

PASCAL COMPILER CONVENTIONS

APPENDIX E

When the option PRIVATE-PROC is ON (default):

- One RBM for the outer block and for each level 1 procedure or function and its nested non-level 1 procedures or functions.
- The names of the non-level 1 procedures or functions do not appear in the USL directory.
- The names of the level 1 procedures or functions do appear in the USL directory and must be unique within 15 characters to meet the requirements of the Segmenter.
- The name of the outer block and of each procedure or function can be 15 alphanumeric characters in length, starting with an alphabetic character.

When the option PRIVATE-PROC is OFF:

- One RBM for the outer block, for each level 1 procedure or function, and for each non-level 1 procedure or function.
- The names of the non-level 1 procedures or functions, as well as of the level 1 procedures or functions, appear in the USL directory and must be unique within 15 characters to meet the requirements of the Segmenter.
- The name of the outer block and of each procedure or function can be 15 alphanumeric characters in length, starting with an alphabetic character.

$SEGMENT *s*$

- A segment name (*s* = *segmentname*) may be up to 15 alphanumeric characters in length, starting with an alphabetic character. Each name must be unique within 15 characters.
- Used to assign an RBM to a specified segment.
- In effect until the next $SEGMENT *s*$ command.
- If a segment with the specified name exists, the compiler places the object code in it. Otherwise, it creates a new segment with the name indicated by *s*.
- $SEGMENT *s*$ command may appear anywhere in source code, but the compiler puts the object code for entire compilation block in the last-named segment. Not possible to place part of a compilation block in a particular segment.

If no $SEGMENT *s*$ command appears, the compiler uses SEG' as the default name of the current segment and places all RBMs into SEG'.

$SUBPROGRAM *s*$

- Permits compilation of a subset of level 1 procedures or functions.
- Allows programmer to select parts of a large program for compilation (and subsequent preparation and execution).
- Minimizes the number of entries in the USL directory.
- *s = list_of_procedure_names.*

$EXTERNAL$

- Used in conjunction with the option $GLOBAL$, this option permits the separate compilation of procedures and functions.
- When EXTERNAL appears in source code, the compiler generates information about variables declared in the outer block that will allow them to be matched up with variables of the same name and type in an outer block compiled with the GLOBAL option.
- The compiler generates object code only for procedures and functions, not for the statement part of the outer block.
- Global variables in a program compiled with the EXTERNAL option must be unique within 15 characters to meet the requirements of the Segmenter.

$GLOBAL$

- Used in conjunction with the option EXTERNAL, this option permits the separate compilation of procedures and functions.
- When GLOBAL is specified, the compiler prepares information about the variables declared in the outer block which will allow them to be matched with variables of the same name and type used in a procedure or function compiled with EXTERNAL.
- The compiler emits object code for the outer block as well as for all procedures or functions.
- Global variables in a program compiled with the GLOBAL option must be unique within 15 characters to meet the requirements of the Segmenter.

CHECK_ACTUAL_PARM *n*$

- This option specifies the level of checking the Segmenter will perform when a program calls a procedure or function. The level specified, *n*, is an integer in the range 0 to 3. It determines the amount of information placed in the USL file. The Segmenter uses this information to check the actual parameters against the formal parameters of the function or procedure.
- Level 3 is the default setting (check function type, number of procedure or function parameters, and the type of each parameter).

- If the procedure or function has a lower checking level, the Segmenter ignores the level indicated by CHECK_ACTUAL_PARM and uses the lower level.

CHECK_FORMAL_PARM *n*$

- This option specifies the level of checking the Segmenter will perform when a program calls a procedure or function. The level specified, *n*, is an integer in the range 0 to 3. It determines the amount of information placed in the USL. The Segmenter uses this information to check the formal parameters against the actual parameters of the function or procedure.

- Level 3 is the default setting (check function type, number of procedure or function parameters, and the type of each parameter).

- If the procedure or function has a lower checking level, the Segmenter ignores the level indicated by CHECK_FORMAL_PARM and uses the lower level.

RPG COMPILER CONVENTIONS

APPENDIX F

The structure of this language is significantly different from that of other programming languages. It is difficult to relate object code generated to original source statements. Thus, most RPG programmers do not interface with the system at the segment level, and the language provides less programmer control over segmentation than other languages.

The RPG compiler determines RBMs based on individual source program requirements.

The SEGMENT= parameter of the $CONTROL command, used by some compilers to assign different Segment names to program units, is not supported by RPG because of its limited application with this compiler.

$CONTROL SEG=*maxsegsize*

- Allows programmer to limit segment size to *n* x 1024 words (not to exceed 4096 words).

Outer block generated automatically.

- One RBM for the outer block.
- Programmer can specify name of outer block.
- Compiler assigns default name RPGOBJ.

PROGRAM LISTINGS

APPENDIX G

Two kinds of maps, or listings, are available to help you with your code management: the preparation, or PMAP; and the loader, or LMAP.

LISTING OF PREPARED PROGRAM: THE PMAP

If you want a map of your prepared program file, you can request a PMAP with the *PMAP* parameter of either the Segmenter -PREP command or the MPE :PREP command. You can also request a PMAP of an SL (which contains prepared code) by using the *PREP* parameter with the -ADDSL command. To print the PMAP on on the line printer, use a :FILE equation to define the file named SEGLIST as a line printer device file. (SEGLIST should not be used as the actual file designator, since it is the formal file designator.) By default, the listing is sent to $STDLIST.

Besides showing you what the prepared program file or SL looks like, the PMAP is useful when you are debugging a program, since it allows you to identify segments by number. Significant entries in figure G-1, which is a PMAP of a program file, are keyed with numbers and explained following the listing. A PMAP for an SL would have exactly the same format.

LISTING OF LOADED PROGRAM: THE LMAP

If you want a map of your loaded program, you can request an LMAP with the *LMAP* parameter of the :RUN command. To print the LMAP on the line printer, use a :FILE equation to define the file named LOADLIST as a line printer device file. (LOADLIST should not be used as the actual file designator, since it is the formal file designator.) By default, the listing is sent to $STDLIST.

The LMAP is useful as a debugging aid when you are developing a new program. It can also help you decide whether your segmentation is efficient (that is, whether you have achieved good localization), since it shows how many external references your program file makes, the logical number of each calling segment, the type of library each call was resolved from, and the logical segment number of the external procedure.

Note that the load map may not be displayed if you are running the program simultaneously with another user. In this case, you are sharing code and other system resources, and the load map is displayed for the first user to request execution of the program.

Significant entries in the following sample listing are keyed with numbers and explained following the listing. The information in the first four columns of the LMAP (figure G-2) following the external procedure name refers to the program that called the procedure (SCR4.MPE.SYS). The information in the last four columns refers to the called external procedures.

```
PROGRAM FILE SCR4.MPE.SYS    **1**

SEG40 **2**  **3** 0
   NAME              STT  CODE  ENTRY  SEG
   LISTRL' **4**       1     0     32
                      **5**  **6**  **7**

   MAKEROOMINDL       30                10  **8**
   FGETINFO           31                 ?
   FREADADMR''        32                10
   NTOA               33                10
   BLANKLINE          34                10
   TESTBIT            35                10
   DNTOA              36                10
                      .
                      .
                      .
   OPENRL             27  2523  2523
   FOPEN              70                 ?
   FLOCK              71                 ?
   FREADMR'           72                10
   SEGMENT LENGTH         3130 **9**
SEG30              1
   NAME              STT  CODE  ENTRY  SEG
   LISTSL'             1     0    106
   FGETINFO           33                 ?
   CLEANUPRTBUF        2   545    545
                      .
                      .
                      .
   SEGMENT LENGTH         2030


PRIMARY DB         3350   INITIAL STACK     1440     CAPABILITY     101
             **10**                  **14**                **17**
SECONDARY DB       1216   INITIAL DL           0     TOTAL CODE   27320
             **11**                  **15**                **18**
TOTAL DB           1566   MAXIMUM DATA     40000     TOTAL RECORDS  155
             **12**                  **16**                **19**
ELAPSED TIME   00:14:29.887                    PROCESSOR TIME     00:24.156
             **13**                                          **20**
```

Figure G-1. The PMAP

ITEM NO.	MEANING
1	The name of the program file (*filename.groupname.accountname*).
2	The segment name.
3	The (logical) segment number.
4	The program unit entry point name or external procedure name.
5	The assigned entry number in the Segment Transfer Table (STT).
6	The beginning location of the procedure code in the segment.
7	The location of the entry point in this segment.
8	The (logical) segment number of the segment containing this external procedure. If this entry is a number, then the procedure is external to the segment but internal to the program file; if this entry contains a question mark (?), then the procedure is external to the segment and external to the program file.
9	The segment length (in words).
10	The primary DB area size
11	The secondary DB area size.
12	The total DB area size.
13	The time elapsed during the preparation process.
14	The initial stack size.
15	The initial DL size.
16	The maximum area available for data (maximum Z-DL size).
17	Capability of program file.
18	Total code in file.
19	Total records in file.
20	Total central processor time used during preparation process.

Figure G-1. The PMAP (Continued)

```
PROGRAM FILE SCR4.MPE.SYS  **1**
                          **3**     **5**    **7**    **9**
TERMINATE'  **2**         PROG  0   50   11   SSL  0   2   37
                              **4**    **6**       **8**   **10**

SENDMAIL                  PROG  0   46   11   SSL  0   2   40
DEBUG                     PROG  0   44   11   PSL  0   1   52
RECEIVEMAIL               PROG  0    6   10   GSL  0   5   40
AWAKE                     PROG  0    5   11   SSL  0   4   45
WHO                       PROG  0   45   10   GSL  0   3   33
SETSYSDB                  PROG  0   33    4   SSL  0   1   33
                                    25    2
GETUSERMODE               PROG  0   10   11   SSL  0   5   44
                                    35    4
                                    27    2
                                    55    1
                                .
                                .
                                .
301 302 303 304 305 306 307 310 311 312     **11**
```

Figure G-2. The LMAP

ITEM NO.	MEANING
1	The name of the program file (*filename.groupname.accountname*).
2	The name of the external procedure.
3	The type of the segment referencing the external procedure, where: PROG = program segment. GSL = group segmented library segment. PSL = public segmented library segment.
4	External parameter checking level.
5	External segment transfer table number (STT).
6	External (logical) segment number of calling segment.
7	Entry point segment type, where: GSL = group segmented library segment. PSL = public segmented library segment. SSL = system segmented library segment.
8	Entry point parameter checking level.
9	Entry point segment transfer table (STT) number.
10	Entry point (logical) segment number.
11	A list of the code segment table (CST) numbers to which the program file segments were assigned. The list is ordered by logical segment number.

Figure G-2. The LMAP (Continued)

INDEX

G

H

I

L

M

N

O

P

R

S

U

V

Part No. 30000-90011
Printed in U.S.A. 11/82
U0886

HEWLETT PACKARD