MPE SEGMENTER
Reference Manual

HP 3000 Computer Systems

HP Part No. 30000-90011
Printed in U.S.A. 19860801

U0886
DRAFT 2/11/100 10:13

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 dates on the title page change only when a new edition or a new update is published. No information is incorporated into a reprinting unless it appears as a prior update; the edition does not change when an update is incorporated.

The software code printed alongside the data indicates the version level of the software product at the time the manual or update was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual updates.

First Edition	Jun 1976
Second Edition	Feb 1978
Third Edition	Nov 1982
Update 1	Aug 1986

MPE V MANUAL PLAN

Programmer's Series

MPE XL
Documentation
Guide & Glossary
Of Terms
(5958-9511)

GENERAL REFERENCE

Compiler LIBRARY/XL Reference Manual (32650-90029)	Getting Started as an MPE XL Programmer (32650-90008)	HP Symbolic Debugger Quick Reference Guide (92435-90002)	HP Symbolic Debugger User's Guide (92435-90001)
LINK EDITOR/XL Reference Manual (32650-90030)	MPE XL Intrinsics Reference Manual (32650-90028)	Scientific LIBRARY/XL Reference Manual (31510-90001)	System Debug Reference Manual (32650-90013)

PROGRAMMING TECHNIQUES

Accessing Files Programmer's Guide (32650-90017)	Command Interpreter Access & Variables Programmer's Guide (32650-90011)	Data Types Conversion Programmer's Guide (32650-90015)	Getting System Information Programmer's Guide (32650-90018)
Interprocess Communication Programmer's Guide (32650-90019)	Message Catalog Programmer's Guide (32650-90021)	Native Language Programmer's Guide (32650-90022)	Process Management Programmer's Guide (32650-90023)
Resource Management Programmer's Guide (32650-90024)	SORT-MERGE/XL Programmer's Guide (32650-90080)	Trap Handling Programmer's Guide (32650-90026)	User Logging Programmer's Guide (32650-90027)

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.

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)*

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.

1

Introduction To The Segmenter

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 execution of a code segment and another user's execution needs the same segment, that segment can be used, is 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 FIGURE11 here.

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 FIGURE12 here.

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 FIGURE13 here.

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 ***<xref FIGURE14>: undefined***, 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.

Insert artwork here.

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 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 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
;	SL.PUB.SYS
LIB=S (or no LIB parameter)	

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.

2

Using The Segmenter

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:

```
text
-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

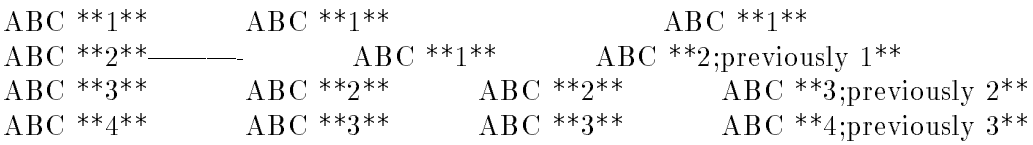
Study the -CEASE and -USE commands in Section IV of this manual for more information about why the index works differently with them than with the other Segmenter commands.

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 "Listing The USL" 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:



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.

<xref FIGURE2-1>: undefined 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 ***<xref FIGURE2-2>: undefined***, 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, Chapter 4.)

```
: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 `***<xref FIGURE2-3>: undefined***`, 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 ***<xref FIGURE2-4>: undefined***, 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)
```