HONEYWELL

LEVEL 68
INTRODUCTION TO PROGRAMMING ON MULTICS

SOFTWARE

LEVEL 68

INTRODUCTION TO PROGRAMMING ON MULTICS

SUBJECT

Introduction to Programming in the Multics Operating System Environment, Intended as a Guide for Applications Programmers

SPECIAL INSTRUCTIONS

This manual presupposes some basic knowledge of the Multics operating system. This information can be found in the 2-volume set, *New Users' Introduction to Multics* (Order Nos. CH24 and CH25).

This manual supersedes AG90, Revision 2, which was titled *Multics Programmer's Manual.* Together with the 2-volume set, *New Users' Introduction to Multics,* it supersedes AL40, Revision 1, which was titled *Multics Introductory Users' Guide*. The manual has been extensively revised and does not contain change bars.

SOFTWARE SUPPORTED

Multics Software Release 9.0

ORDER NUMBER

AG90-03

July 1981

Honeywell

PREFACE

The purpose of this manual is to introduce the Multics environment to applications programmers who have experience on another operating system but are new to Multics.

It is very important that you understand exactly who this manual is for, and what assumptions this manual makes about its audience, before you begin to use it.

The intended audience of this manual is applications programmers. It is assumed that you have programmed on some other system(s) and that you have some basic knowledge of at least one higher level language (COBOL, FORTRAN, PL/I, etc.). No attempt is made here to teach you how to program. This manual is only intended to show you how to do the things you know how to do on another system on Multics.

As an applications programmer, you look at an operating system from the viewpoint of some programming language. This manual does not attempt to discuss the use of any particular language on Multics, but rather, concerns itself with those practices which are appropriate no matter which language you use. For information on specific languages you should refer to the Language Users' Guides. The names of these guides are included in the list of useful manuals for new programmers given at the end of this preface.

This manual assumes that you are registered on Multics, and that you know how to log in and use a terminal. It also assumes that you have some general familiarity with the fundamental concepts and facilities of the Multics system. This information is available in the following publications:

New Users' Introduction to Multics - Part I	Order No. CH24
New Users' Introduction to Multics - Part II	Order No. CH25

You should feel comfortable with the use of segments, directories, text editors, access control, commands, and active functions. If you don't, you should review the manuals listed above, as no review of this material will be presented here.

File No.: 1L13

AG90-03

Section 1 of this manual offers an overview of the Multics operating system in general terms, to give you some idea of why programming on Multics may be different from working on other systems.

Section 2 offers a step-by-step approach to the essentials of programming on Multics. It shows you how to create, compile, execute, revise, and document your programs in this environment, how to manipulate your segments, and how to create storage system links. Sample terminal sessions are also included.

Section 3 takes you one step further by showing you the uses of dynamic linking on Multics.

Section 4 provides you with an introduction to Multics input/output processing, showing you how to use the terminal for I/O and how to begin using I/O commands.

Section 5 discusses the use of a Multics debugging tool.

Section 6 discusses the use of a Multics performance measurement tool.

Section 7 explains the Multics absentee facility, which offers capabilities similar to batch processing on other systems.

Section 8 offers a reference to all of the Multics commands by function, including a brief description of each command.

The appendixes of this manual contain material which is specific to a particular language, somewhat advanced, or useful only to certain users.

Appendix A shows you how to use Multics to best advantage in PL/I programming.

Appendix B offers a step-by-step explanation of a PL/I text editor program. (This is for people who are ready to begin systems programming work.)

Appendix C briefly introduces you to various Multics subsystems.

Appendix D shows you how to use the Edm text editor.

The information presented here is a subset of that contained in the primary Multics reference document, the Multics Programmers' Manual (MPM). The MPM should be used as a reference to Multics once you have become familiar with the concepts covered in this introductory guide. The MPM consists of the following individual manuals:

Reference Guide	Order No. AG91
Commands and Active Functions	Order No. AG92
Subroutines	Order No. AG93
Subsystem Writers' Guide	Order No. AK92
Peripheral Input/Output	Order No. AX49
Communications Input/Output	Order No. CC92

Throughout this manual, references are made to the MPM Reference Guide, the MPM Commands and Active Functions, the MPM Subroutines, and the MPM Subsystem Writers' Guide manuals. For convenience, these references are as follows:

MPM Reference Guide
MPM Commands
MPM Subroutines
MPM Subsystem Writers' Guide

Other Multics manuals of interest to new programmers are listed below.

- Languages:

Multics APL	Order No. AK95
Multics Basic	Order No. AM82
Multics COBOL Users' Guide	Order No. AS43
Multics COBOL Reference Manual	Order No. AS44
Multics FORTRAN Users' Guide	Order No. CC70
Multics FORTRAN Reference Manual	Order No. AT58
Multics PL/I Language Specification	Order No. AG94
Multics PL/I Reference Manual	Order No. AM83

- Subsystems:

Multics FAST Subsystem Users' Guide	Order No. AU25
Multics GCOS Environment Simulator	Order No. AN05
Multics Graphics System	Order No. AS40
Logical Inquiry and Update System Reference Manual	Order No. AZ49
Multics Relational Data Store (MRDS) Reference Manual	Order No. AW53
Multics Report Program Generator Reference Manual	Order No. CC69
Multics Sort/Merge	Order No. AW32
WORDPRO Reference Guide	Order No. AZ98

- Micellaneous:

Multics Pocket Guide - Commands and Active Functions	Order No. AW17
Index to Multics Manuals	Order No. AN50

The Multics operating system is referred to in this manual as either "Multics" or "the system". The Emacs, Qedx, Ted, and Edm text editors are referred to as "Emacs", "Qedx", "Ted", and "Edm" respectively.

CONTENTS

Page

ILLUSTRATIONS

SECTION 1

THE MULTICS APPROACH

The Multics approach is quite different from that of a traditional batch operating system. The intent of this section is to show you how Multics is different, by giving you a general overview of the system's "personality", then describing in more detail three of its major characteristics: segmented virtual memory, dynamic linking, and controlled sharing and security. As these characteristics are discussed, important concepts associated with each will be introduced and explained. Familiarity with these concepts will help you when you read later sections of this manual and begin to program on Multics.

Multics is a large, powerful, well-established system, which is constantly being refined, and provides a wide range of commands, languages, and subsystems. Despite its size and complexity, Multics is easy to learn and use. It has been designed to serve a wide variety and number of users, all cooperating and sharing resources. Multics offers its users the following advantages:

- support for online usage: Multics has been designed to support online processing as well as batch processing. You can accomplish all of your programming tasks as either an interactive (online) user or an absentee (batch) user. Applications, debugging tools, data base management facilities, administrative tools and utilities are all accessible online. In one terminal session, you can write, compile, execute and debug your program. (See "Sample Terminal Sessions" in Section 2, and "Probe" in Section 5.)

- consistent user interface: A great deal of thought has gone into making similar parts of Multics work in similar ways. For example, common control arguments such as -all and -brief are used with many different commands, and in each case, the control argument performs a similar function. In addition, all parts of the system have been designed to work together.

- uniformity of control language: Batch processing on Multics is supported by the <u>absentee facility</u> (described in Section 7). An absentee job is processed like an interactive terminal session; it's directed by the same language as that used for interactive jobs. In other words, no special job control language (JCL) is ever required on Multics. The system commands and routines provide the logical branching, conditional execution, input/output control, and file system specifications necessary to direct any job.

- ease of use: On Multics, users are not asked to give information or make decisions ahead of time. There are many examples of this. You don't have to know or specify either a segment's size or its location to use it. You don't have to make your need for tape drives and similar resources known in advance. Intelligent defaults mean that you need not create a correspondence between a file and an I/O name. Dynamic linking (described later in this section) means that you need not name or prefetch programs you want to execute. You can set up a temporary working array for your PL/I or FORTRAN program in its own segment, without specifying how much space you need or worrying that the array will get too big. You will find that this lack of required prespecification greatly simplifies your use of the system.

SEGMENTED VIRTUAL MEMORY

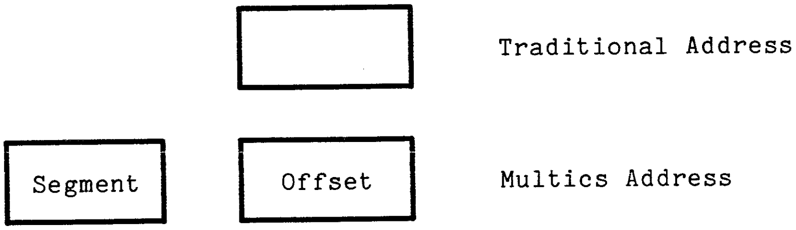
The most significant difference between the Multics programming environment and that of most other contemporary computer programming systems lies in its approach to addressing online storage. Most computer systems have two sharply distinct environments: a resident file storage system in which programs are created, and translated programs and data are stored; and an execution environment consisting of a processor and a "core image", which contains the instructions and data for the processor. Supervisor procedures provide subroutines for physically moving copies of programs and data back and forth between the two environments.

In Multics, there is one conceptual memory, which is known as the *virtual memory*. The traditional distinction between secondary storage and main memory has no meaning, because a single infinitely large memory is simulated by the software, with data stored in finite segments which appear to be in memory at all times. Figure 1-1 illustrates this difference between a traditional system and the Multics virtual memory.

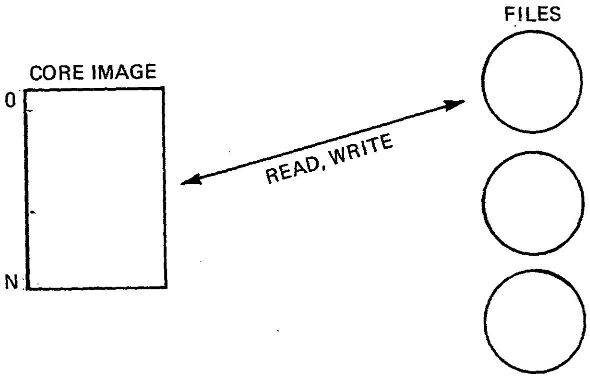
With the line between the two traditional environments deliberately blurred, program construction on Multics is simplified: most programs need only be cognizant of one environment instead of two. This blending of the two environments is accomplished by extending the processor/core image environment. In Multics, your share of the processor is termed a *process*, and your core image is abstracted into what is called an *address space*. In a sense, each segment is a core image, and your process can have lots of them.

The easiest way to think about the terms process and address space is to imagine your process as a private computer and your address space as a private memory for your process to work in. (See "Process, Address Space, and Execution Point" next in this section.)

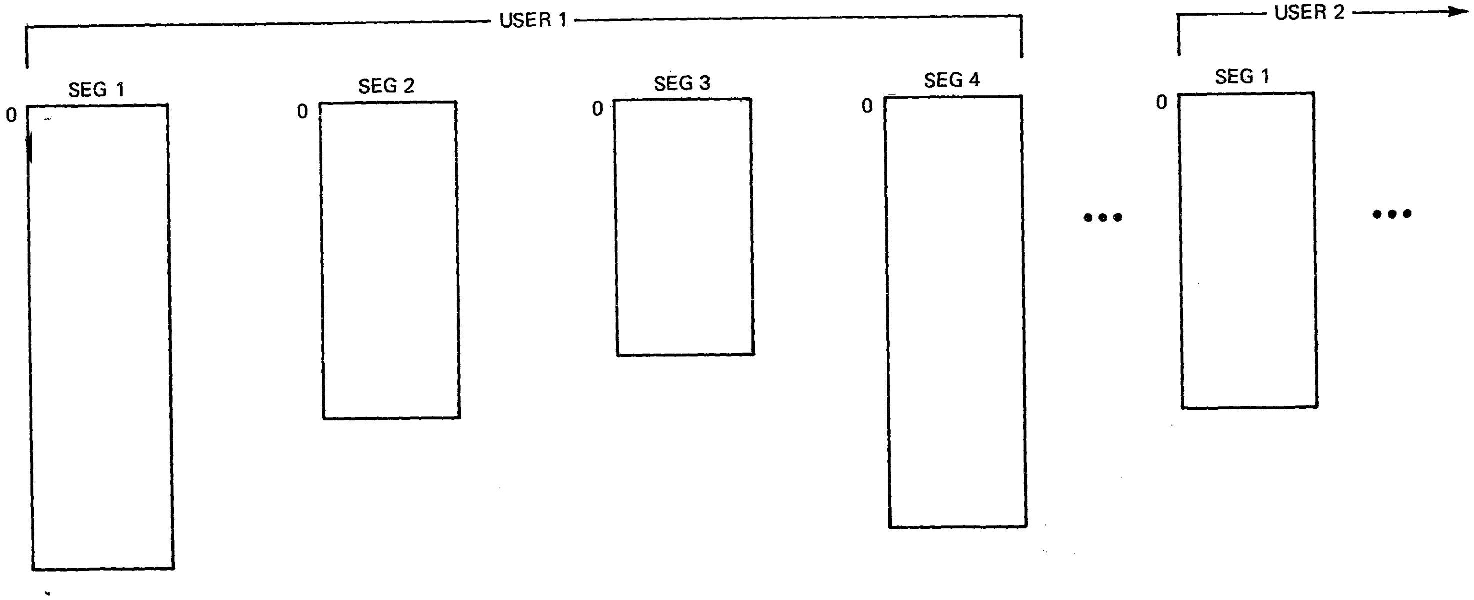
Another important difference between the Multics environment and that of most other systems is that an address in Multics has two parts: a segment identifier and a location, or *offset*, within the segment.



(See "Segments and Addressing" later in this section.)



TRADITIONAL SYSTEM



MULTICS VIRTUAL MEMORY

Figure 1-1. Traditional System vs. Multics Virtual Memory

Process, Address Space, and Execution Point

When you log in to the system, you are allocated system resources in an environment known as a process. A process consists of a collection of segments called an address space, over which a single execution point is free to roam (i.e., to fetch instructions and make data references).

A process executes programs on your behalf, either directly in response to your instructions or automatically as part of supporting the programs you invoke directly. The programs executed on your behalf and the data they reference make up your address space, and that address space combined with the action of executing those programs make up your process. Your execution point is whatever is executing at any moment.

Space within the virtual memory is dynamically assigned to your address space. Its contents are a function of the sequence of instructions that are processed between the time you log in and the time you log out, and thus it dynamically shrinks and grows as necessary. Your address space is different from the usual core image in that it is larger and it is segmented. A segment may be of any size from 0 to 255K, and an address space may have a large number of segments (typically about 200). Usually, each separately translated program resides in a different segment; collections of data which are large enough to be worthy of a separate name are placed in a segment by themselves. The system assigns attributes (access control and length, for example) to each of these segments based on their logical use. There is a distinct address space for each user who is logged in, even though many users may share the very same segments in their address spaces.

Your process is created when you log in, and destroyed when you log out, when you request a new process with the new_proc command, or when some kinds of errors occur. You may view your process as if all system resources are dedicated to it alone--as if you have a processor all to yourself--when in reality, all resources are being shared among many processes. Not only are there other interactive processes running, there are also absentee processes running as "background" to the interactive ones, and there are various daemon processes running, which are associated with the normal operation of the system and not connected to any user. All of these processes are continually cooperating and competing for processor time and main storage resources. The processor is multiplexed between processes according to rules defined for the system as a whole, with the object of sharing resources in an equitable manner.

Processes can share with each other, and this sharing is of two types. First, any references to a segment by more than one process are references to the same segment. Second, a large part of the address space in all processes is identical, because the parts of the system shared by all users are given segment numbers (described below) that are the same for all processes. Figure 1-2 illustrates this sharing of segments.

You should remember that each process's virtual memory is private to it. This means that changes made to one process's virtual memory assignments do not affect those of other processes. In addition, when a segment is being shared, it means that multiple users may not only read the segment, but also write it.

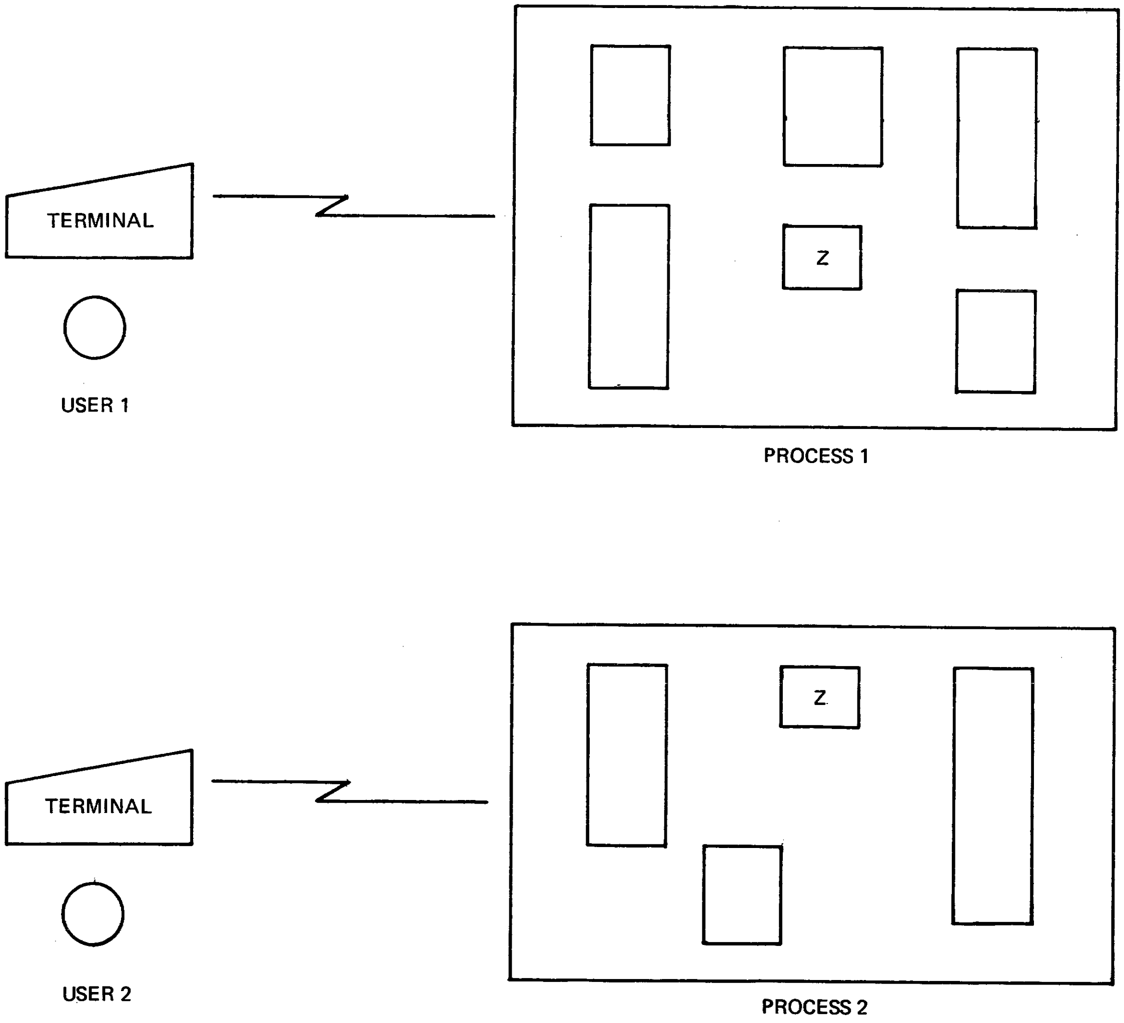


Figure 1-2. Processes Sharing a Segment

Segments and Addressing

It's important to understand that a Multics segment is not a file. A segment can be addressed directly, like memory. It doesn't have to be read or written record by record like a file on other systems. On Multics, everything is in a segment:

program source code
program object code
data files
mail boxes
work areas
temporary storage
exec_coms
.
.
.

There are two main reasons why segments are used in Multics. The first is that they make it possible for all your process's programs and data to be easily and directly addressable. The second is that they make it possible to protect and share programs and data by controlling access at the hardware level. (For more on this, see "Controlled Sharing and Security" later in this section.)

The segment is often described as the basic unit of storage in Multics because all locating (addressing) of data in the system is done in terms of segments. The physical movement of information between main memory and secondary storage is fully automatic in Multics (it is done by the paging mechanism). The usual complex combination of file access methods and job control language which you are probably used to is replaced by a simple two-dimensional addressing scheme. This scheme involves the user-assigned symbolic name of the segment (its pathname), and the address of the desired item within the segment. Even relative addresses are usually given in symbolic terms through the data description facilities of the language you're using. Thus, each segment appears to its user as independent memory, symbolically located. Segments don't have to be in specific storage locations. They can be relocated anywhere in memory and grow and shrink as need be.

References to any portion of your address space consist of a segment name and a location within the segment; all addresses are interpreted as offsets within segments. To increase the efficiency of a storage reference, a segment number becomes associated with a segment name when the segment is <u>initiated</u> (added to your process's address space). A segment is said to be <u>known to a process</u> when it has been uniquely associated with a segment number in that process. The segment number is a temporary alias for the segment name, which is more easily translated into a storage address by the hardware. When you write:

```
<symbolic_name> | [symbolic_offset]
```

the hardware uses:

```
<segment_number> | [offset_number]
```

The association between a segment name and a segment number is retained until the segment is <u>terminated</u> (removed from your process's address space). If it is terminated and initiated again, the number will be different. (See the discussion of initiating and terminating segments in Section 3.) Thus, every address or pointer is a pair of numbers: the segment number and the offset within the segment. This pair of numbers forming an address represents the coordinate of a location in the two-dimensional address space. See Figure 1-3 for a graphic representation of a two-dimensional address space. See Figure 1-4 for an illustration of the life of a segment.

A program can create a segment by issuing a call to the system specifying the symbolic name as an argument. Different users can incorporate the same segment into their programs just by specifying its name. (A program need not copy a segment to use it.) A program can address any item within a segment using "segment, 1" where segment is the symbolic name of the segment and 1 is the location of the desired item within the segment. The ALM (Multics assembly language) instruction shown below illustrates a symbolic reference to location "x" in segment "data":

```
lda data$x
```

For more information on the Multics virtual memory, see the MPM Reference Guide.

DYNAMIC LINKING

Many programs make calls to external subroutines or use external variables. On most systems, these external references are resolved during loading or linkage editing. When the program is loaded into memory, external subroutines are loaded from libraries or user data sets, and storage is allocated for external variables. On Multics, external references are resolved <u>when the program runs</u>; i.e., the point at which something is used is the point <u>at which it is found</u>. This means that a compiled program on Multics is directly executable. Segmentation is what makes this possible - it gives each segment a "zero" location, so no relocation is necessary.

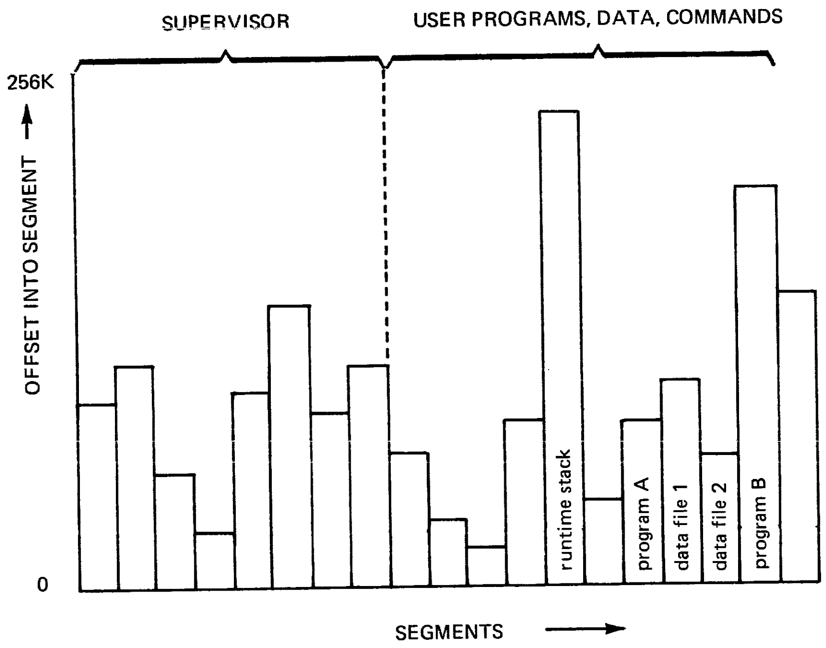


Figure 1-3. Two-Dimensional Address Space

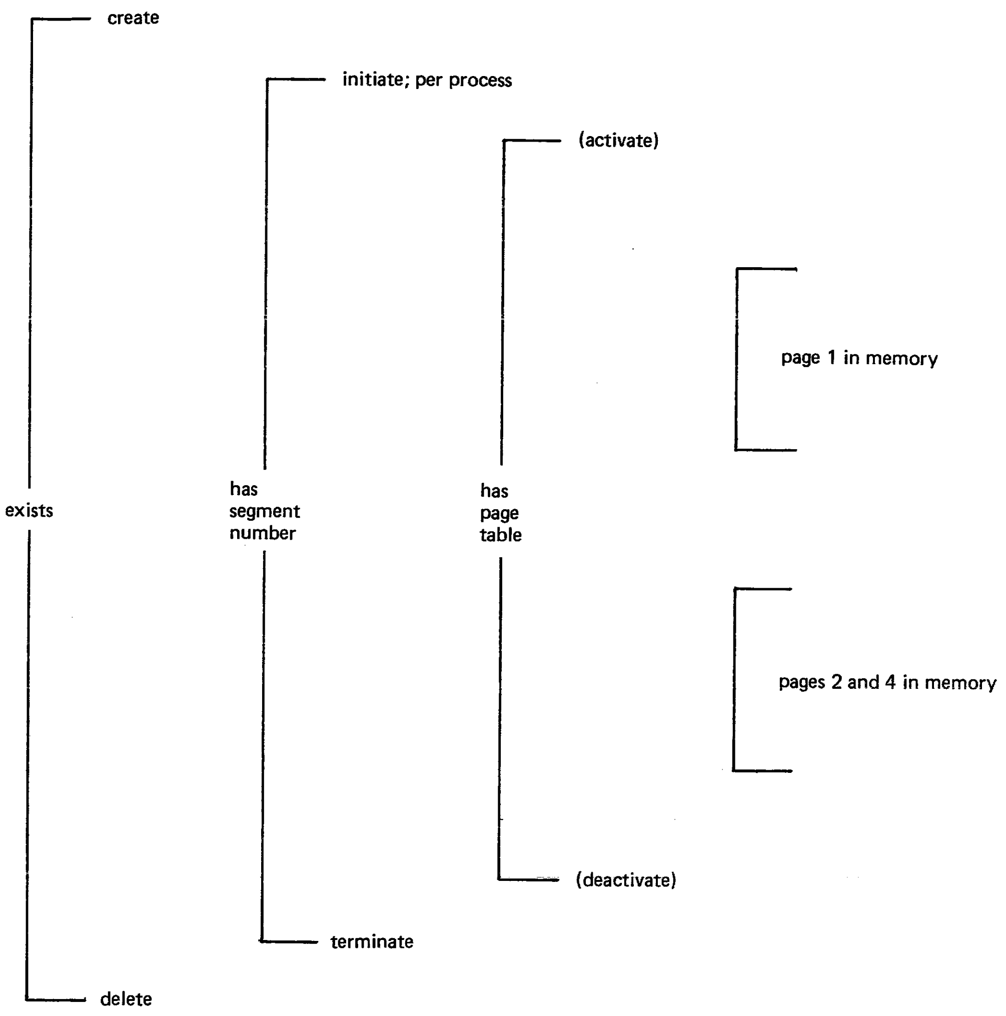


Figure 1-4. The Life of a Segment

Note 1. Events in parentheses are not user visible.

Note 2. Segments are automatically divided by the hardware into storage units known as pages, with a fixed size of 1024 words. (One word is equal to 36 bits or 4 9-bit bytes.)

Dynamic linking is accomplished by having the compiler leave in the object code of a compiled program an indirect word with a "fault tag" which, if used in an indirect address reference, causes a *linkage fault* to the dynamic linker. The linker inspects the location causing the fault, and from pointers found there, locates the symbolic name of the program being called or the data segment being referenced. It then locates the appropriate segment, maps it into the current address space, and replaces the indirect word with a new one containing the address of the program or data entry point, so that future references will not cause a linkage fault. When the system comes across an unresolved reference, it uses what are known as *search rules* (described in Section 3) to find the needed segment and establish the necessary link. This process is known as *snapping a link*. To see how the linkage fault caused by the ALM instruction mentioned previously would be resolved, refer to Figure 1-5.

With dynamic linking, you don't pay the cost of resolving references (for example, calls to error routines) unless they are actually needed. If a subroutine is never called, it doesn't even have to exist, and the main program will still run correctly. An item in the file system has to be in your address space for you to use it, but it doesn't have to be copied and brought into memory before execution. The virtual memory guarantees that any item you reference is where the processor can address it directly.

Dynamic linking simplifies your programming by totally eliminating the loading step. It also eliminates the need for a complicated job control language for retrieving, prelinking, and executing programs, and for defining and locating input/output files.

For more information on dynamic linking, see the MPM Reference Guide.

CONTROLLED SHARING AND SECURITY

Multics permits controlled sharing of the operating system software and libraries, the language compilers, the data bases, and all user code and data. You can create links to other programs and data, give and revoke access, directly access any information in the system to which you have access, and share a single copy in core.

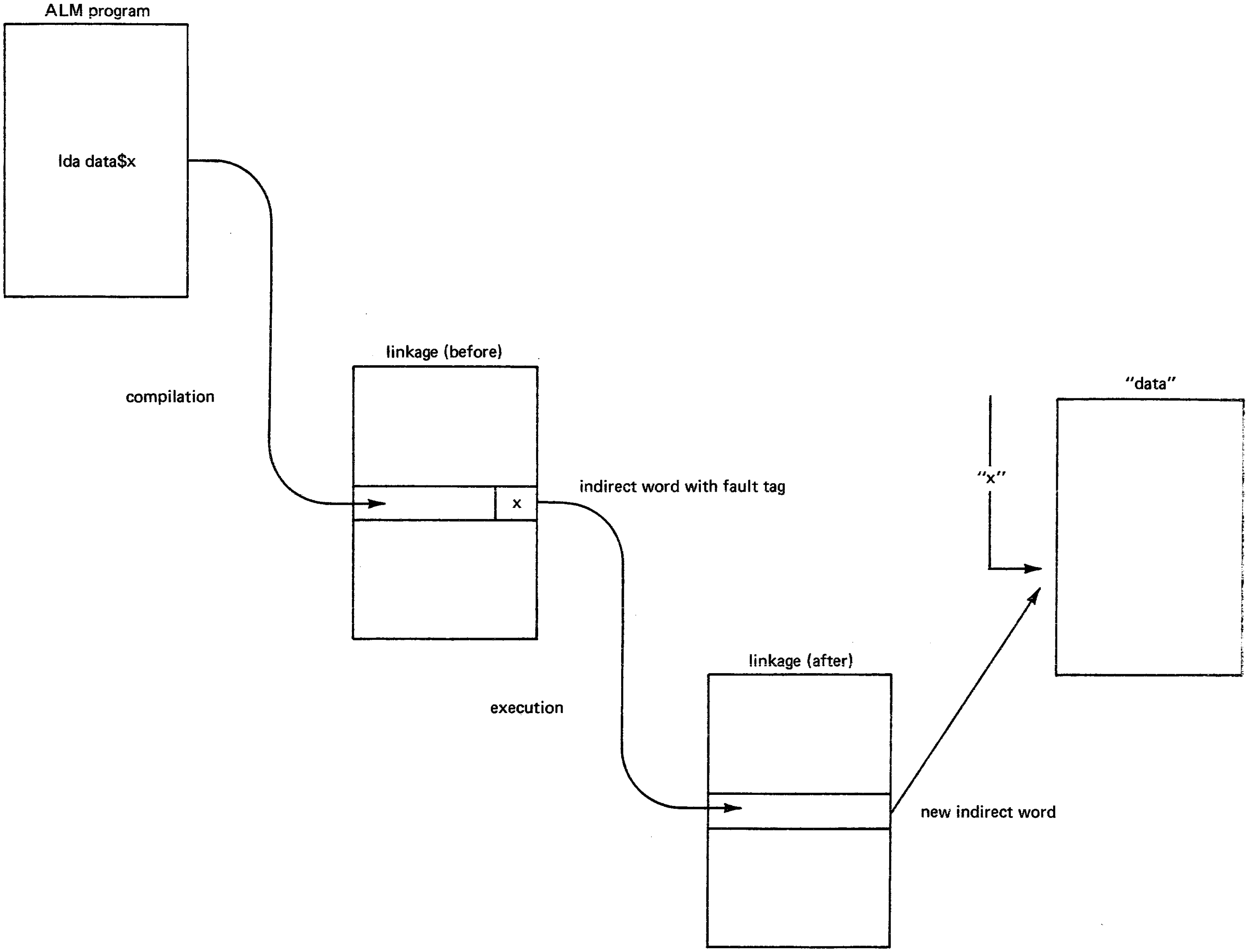


Figure 1-5. Resolving a Linkage Fault (Snapping a Link)

Access Control Lists

One way of controlling the sharing and security of information is by using access control lists. ACLs, as you have already learned in the New Users' Introduction to Multics, define the access rights for each segment and directory. You can grant permission to use your segments and directories by individual user, by project, by instance (interactive/absentee), or by combinations of these. You can also grant different access to different users of the same segment. A good example of using ACLs is a compiler which resides in a segment that can be executed but not written.

For more details on access control, see the MPM Reference Guide.

Administrative Control

Another kind of information control is administrative. Multics administration defines three levels of responsibility: system, project, and user. A system administrator allocates system resources among the projects on his system; a project administrator allocates project resources among the users on his project; a user can manage his own data through storage management and access controls.

Your project administrator can define the environment of the users under his project. He can give you complete control in creating your own process, or he can limit the requests and commands available to you. He can determine the dollar limit that you may incur in a single month (or other period of time), and arrange things so you'll be automatically logged out if you exceed this limit. You won't be able to log in again until the next month begins or the limit is changed. He can also determine several other items, including whether a user can preempt others, specify his own directory, or have primary or standby status when logging in.

You yourself also have flexibility in shaping your programming environment on Multics. A good example of this is the special command processor which allows you to make abbreviations for your frequently used commands (abbrev).

For more information on Multics administrative features, refer to one of the manuals in the Multics Administrators' Manual (MAM) set:

Project Administrator	Order No. AK51
Registration and Accounting Administrator	Order No. AS68
System Administrator	Order No. AK50

SECTION 2

PROGRAMMING ON MULTICS

Programming on Multics is very different from programming on other systems. Many of the constraints and restrictions you may be used to are simply removed. The system provides high-level terminal control, data base management, I/O interfaces, and data security. There is no need for overlays, chaining or partitions.

This section explains how to write, compile and execute programs in the Multics environment. It also offers advice on revising and documenting programs, manipulating segments, and creating storage system links.

DESIGNING AND WRITING PROGRAMS

Let's say you've been given specifications for a program which will compute the sum of three numbers. Obviously, this is not a realistic task for a computer, but it will provide us with a very simple example.

Of course, the first thing you need to do is to develop a design for your program, be it a flow chart, a functional diagram, a hierarchy, or whatever. Once you have a good design, the next step is to decide which language you will write your program in. The following programming languages are available on Multics:

- APL: A terse, powerful language, with strong data manipulation capabilities.
- BASIC: A simple language for beginners, which can perform string and arithmetic operations without much difficulty.
- COBOL: A business oriented, high-level, English-like language with many string and arithmetic capabilities.
- FORTRAN: A high-level, scientific language designed mostly for arithmetic applications, with very limited character manipulation capabilities.
- PL/I: A very powerful, high-level language that offers almost total control over the operations of the program, and has many capabilities to manipulate characters and perform arithmetic operations.

(ALM, the assembly language on Multics, is also available, but is not recommended for general use.) For this program, let's say you choose PL/I. The code for your program might look like this:

```
simple_sum: proc options (main);

/* this program computes the sum of three numbers set in the program,

   then prints the answer at the terminal */

declare
      sysprint  file,                    /* the terminal output */
      first_no  fixed binary (17),       /* the first number */
      second_no fixed binary (17),       /* the second number */
      third_no  fixed binary (17),       /* the third number */
      the_sum   fixed binary (17);       /* the answer */

/* set the three numbers */

      first_no = 123;
      second_no = 456;
      third_no = 789;

/* add them up */

      the_sum = first_no + second_no + third_no;

/* print the answer */

      put skip list ("The sum of the three numbers is:", the_sum);
      put skip;

   end simple_sum;
```

Notice the use of sysprint for the terminal output. For more information on this, see "Using the Terminal for I/O" in Section 4.

Source Segments

The next step is to create a segment containing your code. You can input your code by using any one of several text editors. Two editors you are already familiar with are Qedx and Emacs. Detailed information on these editors is available in the *Qedx Users' Guide* (Order No. CG40) and the *Emacs Users' Guide* (Order No. CH27) respectively. Of special interest to programmers are the programming language modes available in Emacs. The FORTRAN, PL/I and ALM modes provide editing environments which facilitate the creation, formatting and debugging of programs written in these languages.

Two more editors will be introduced here. One is Edm. This is the most basic Multics editor and is described in Appendix D of this manual. The other is Ted. Ted is a more advanced version of Qedx, which offers many advantages. These include more flexibility in addressing characters within a line, two types of input mode, regular and bulk, and more ways of manipulating buffers. Ted is a programmable editor, which means that you can write character manipulation programs in the Ted editor language. Other Ted features include sorting and tabbing capabilities, the ability to translate letters from upper to lower case and vice versa, and the ability to have lines fill and adjust. For more information on Ted, use the help command.

The segment that your source code is stored in is called a *source segment*. Once your source segment is created, you should give it an entryname which follows the Multics convention for such names. This convention is to add a dot suffix to the end of the name indicating which language the program is written in. Thus, the form for a source segment entryname is:

```
program_name.lang_name
```

A good name for your program would therefore be:

```
simple_sum.pl1
```

Some other examples of program names are:

```
ran_num_gen.basic
payroll.cobol
square_root.fortran
```

(Remember that upper and lower case characters are not interchangeable on Multics. Thus, "payroll.cobol" and "Payroll.cobol" are two different names. See the MPM Reference Guide for more information on naming conventions.)

You will probably find it useful to create several different directories for yourself, each containing a different sort of segment. For example, you could have one directory for the final (debugged) versions of your programs, one directory for the programs you are writing or revising, another directory for test data, etc. If you write programs in several different languages, you could also have directories for programs in each language. (Remember that your segments are not physically located in directories any more than you are physically in the phone book. When a segment is said to be "in" a directory, it means that the directory contains an entry for the segment.)

COMPILING PROGRAMS

Multics provides a compiler for each higher level language it supports. Compilers are system programs which translate source code into object code, machine level language that is executable by the hardware. The input to a compiler is a source segment. The output of a compiler is a corresponding object segment. (This discussion does not apply to APL, which is an interpreted language. There is no APL compiler and no APL object segment.) Your working directory is always assumed to be the location of the source segment you want to compile, and the intended location of the object segment you want to create, unless you say otherwise.

To execute a compiler, you invoke it as a command, with a command line which looks like this:

```
language_name path {-control_arguments}
```

where language_name is the name of the language your program is written in, path is the entry name of your source segment, and {-control_arguments} are any of a number of optional control arguments you can supply to the compiler. Several of these control arguments instruct the compiler to create a listing segment in your directory. (No compile listing is produced by default.) This segment has the same entryname as your source segment, but with a suffix of "list" instead of "pl1" or whatever. A listing segment contains a line-numbered list of your source program, plus information that is useful for understanding, debugging, and improving the performance of your program.

The control arguments which produce a listing segment are:

-list
    produces a complete source program listing including an assembly-like listing of the compiled program. Use of this control argument significantly increases compilation time and should be avoided whenever possible by using -map.

-map

produces a partial source program listing of the compiled program which should contain sufficient information for most online debugging needs.

Another useful control argument is:

-table

generates a full *symbol table* for use by symbolic debuggers. The symbol table is part of the *symbol section* of the object program (discussed later in this section) and consists of two parts: a statement table that gives the correspondence between source line numbers and object locations, and a name table that contains information about names actually referenced by the source program. This control argument usually causes the object segment to become significantly longer, so when the program is thoroughly debugged, it should be recompiled without -table.

See the MPM Commands under the specific compiler for detailed information on all of the control arguments and the information they provide. Also see the various Language Users' Guides.

So, your command line for compiling your program might look like this:

```
! pl1 simple_sum.pl1 -map
```

In this and all interactive examples in this manual, an exclamation point is used to indicate a line that you type at the terminal. You do not type the exclamation point, nor does Multics type it as a way of prompting you. It is strictly a typographical convention, to distinguish between typing done by you and typing done by Multics.

In reality, you don't have to type the dot suffix component of your entryname. The compiler assumes that the input is a source segment, and will search your working directory (or whatever directory you're using) for the segment with the appropriate suffix. Thus:

```
! pl1 simple_sum.pl1
```

means exactly the same to the compiler as:

```
! pl1 simple_sum
```

If your source code is clean and the compile is successful, an object segment is placed in the directory you're using, with the same entryname as your source segment, but stripped of the language name suffix:

```
ran_num_gen.basic       ---------->    ran_num_gen
payroll.cobol           ---------->    payroll
square_root.fortran     ---------->    square_root
```

So, if you execute this command line:

```
! pl1 simple_sum -map
```

then you list your working directory, you'll see:

```
simple_sum
simple_sum.pl1
simple_sum.list
```

Your listing segment, simple_sum.list, can be printed on your terminal with the print command, or printed on paper with the dprint command. Since listing segments take up a large amount of space, the sensible thing to do is to dprint the segment, then delete it:

```
! dprint -delete simple_sum.list
```

If there are problems with your source code, the compiler will produce error messages. The compiler can detect errors according to the definitions of the language involved. These include typing errors, syntax errors, and semantic errors. These messages are printed for you at your terminal. The format and details of error messages vary from compiler to compiler. The following is a sample PL/I error message:

```
ERROR 158, SEVERITY 2 ONLINE 30
A constant immediately follows the identifier "zilch"
SOURCE:  a = zilch 4;
```

If your compile is taking a long time, you can issue a QUIT signal and take a look at your ready message. Since a ready message contains the amount of CPU time used since the last ready message, if the CPU times on your last two messages are different, you know your compilation is working. To resume it, type start. You can also use the progress (pg) command to get information on how a command's execution is going. To check on your compile of simple_sum.pl1 with the -map control argument, you would type:

```
! progress pl1 simple_sum -map
```

The system would periodically type information about the pl1 command's progress in terms of CPU time, real time, and page faults. (A page fault occurs when a page of a referenced segment is not in memory.) See the MPM Commands for a detailed explanation.

Object Segments

As you may remember from the discussion of dynamic linking in Section 1, an object segment is an executable module. This is quite different from other systems, where the object module which is the output of the compiler cannot be executed until it has been through some kind of linkage editing to become a load module. On Multics, there is no such distinction between an object module and a load module. Thus, there is no need for you to determine in advance the absolute addresses of programs in memory, or give instructions for linking and calling programs or loading them. All compiled programs are ready to run.

Most higher level languages supported by Multics compile into Multics standard object segments. These are divided into several sections. The first section is called the text section and contains the binary machine instructions that were translated from the source code and are executed by the processor. The next section is the definition section, which defines the names and locations of entry points present in the segment, and the names of external entry points used by the segment. An entry point is a symbolic offset within a segment. (See "A Naming Convention" in Section 3.) After the definition section comes the linkage section, which serves as a template of all virtual addresses for all external entry points used by the program. It contains per-process information used by the dynamic linker to resolve these external references. The next section is the static section, which contains data items to be allocated on a per-process basis. (This section may be included in the linkage section, and not exist as a separate section.) Then there is the symbol section, which contains information on all the variables declared in the program. The symbol section is always present in the object segment. If -table is specified when the program is compiled, then a symbol table is included in this section. Some compilers (e.g., pl1) support the -brief_table control argument, which produces a shorter symbol section. Finally there is the object map, which contains the lengths and offsets for each section of the object segment. Details about the format of object segments and what each section contains may be found in the MPM Subsystem Writers' Guide.

Where the standards for the source language permit, all object segments produced by Multics are:

- pure: the object segment contains no code that modifies itself during execution. Information about calls outside the segment is copied into a special segment, and all modifications are made to the copy. The same segment can be executed by more than one user. No copies of object segments are made on a per-user basis; there is one shared segment in the address space of all who use it. For example, even when multiple users are simultaneously compiling COBOL programs, only one copy of the COBOL compiler is in use.
- recursive: the object segment can call itself.
- in standard format: the calling protocols for object segments are the same irrespective of the higher-level language of origin. This means that a program in one language can call a program in another language. Programs can also access any data or file which can be described by data types supported by the particular language.

EXECUTING PROGRAMS

Now that you have an object segment, you are ready to try executing your program. To do this, all you have to do is type the name of your program from command level. The entryname is understood as a command--the system is instructed to find your program and execute it, just as when you type the name of a command (like list), the system is instructed to find the program by that name and execute it. Source and object segments are both permanent (they don't have to be copied to a special directory to be saved), so your program can be run over and over until you choose to delete it.

Some Results of Execution

- The program runs to normal termination and you get a ready message, indicating that execution was successful.

```
r 10:29 3.0 350
```

- The program pauses for input from your terminal.
- The program halts because of a breakpoint you've put in it for debugging purposes.
- The program runs to normal termination, but the output you get is wrong.
- The program halts because you issue a QUIT signal, and the system responds with a ready message indicating a new command level:

```
! QUIT
  r 10:40 0.1 497 level 2
```

- The program halts because of an execution error. Examples of such errors are overflows, underflows, data conversions, and undefined references. The system prints an error message, then gives you a ready message indicating a new command level:

```
Error: Exponent overflow by >udd>ProjA>MacSissle>bad_pgm|143
       (line 33)
System handler for condition returns to command level
r 10:38 0.185 98 level 2
```

The new command level means that you are again in a position to invoke commands. There are some special commands that can be put to appropriate use here, such as the release, start, program_interrupt, or probe commands. The release command returns you to the original command level--the work you were doing at the time of the interrupt is simply discarded. The start command resumes execution where it left off. The program_interrupt command returns execution to a predetermined point from which to resume execution. For the use of the probe command see Section 5, "Debugging Tools."

Multics will provide you with as specific an error message as possible. One common error that happens to almost everyone at some time or other is the following:

```
Error: record_quota_overflow condition by <program_name>
```

This message means that you have run out of storage space in the system. The best way to fix this situation is to delete unneeded segments and type start. (For descriptions of other common error messages, see Multics Error Messages: Primer and Reference Manual, Order No. CH26.)

REVISING AND DOCUMENTING PROGRAMS

If you edit your program and recompile it, you may want to save the old object segment instead of replacing it with the new one. In the process of developing and testing new versions of a program, you may in fact end up with several versions, all of which you want to keep. Here are some ways you can do it:

- You can move the old object to another directory, using the move command:

```
! move simple_sum obsolete_pl1_obj>simple_sum
```

- You can copy the faulty source (should you wish to save it as well) and give a new name to the edited version using the copy and rename commands:

```
! copy simple_sum.pl1 obsolete_pl1_source>simple_sum.pl1
! rename simple_sum.pl1 new_simple_sum.pl1
```

- You can change the name of the old object:

```
! rename simple_sum old_simple_sum
```

You need to be aware of certain dangers involved in renaming segments which are already known to your process. Renaming a segment doesn't change the association between the segment name and the segment number. So, if pgma calls pgmb, then you rename pgmb as badb, create a new pgmb, and run pgma again, when pgma calls pgmb, it will end up with the old badb instead of the new pgmb. For more information on the association between segment names and segment numbers, see "A Note on Initiated Segments" in Section 3.

If you ever get confused as to which version of your source program is which, you can use the compare_ascii (cpa) command, which compares ASCII segments and prints any differences.

Remember that final versions of your programs should be correctly formatted to improve their readability. There are several Multics commands which can help you do this. For example, the indent (ind) command indents free-form PL/I source code according to a set of standard conventions. For another example, the format_cobol_source (fcs) command converts free-form COBOL source programs to a fixed format. These commands also detect and report certain types of syntax errors, and can be used for pre-compile examinations.

Your final versions should also be well-documented. There are two kinds of documentation for programs. One is internal, and consists of a step-by-step description of what the program does. This sort of documentation is best created by the generous use of comments throughout your code. The other kind of documentation is external, and consists of a more general description of the programs purpose, design, and use. Writing info segments is an excellent way of creating this sort of documentation. (Remember that the information in an info segment is printed using the help command).

Finally, all of your source and object segments should have the proper access set, so only the appropriate people can use them.

SAMPLE TERMINAL SESSIONS

Figure 2-1 displays the interaction between Multics and the user Karen MacSissle as she logs in and writes, compiles, and executes the simple_sum program. MacSissle uses the Qedx editor to put the program online, the pl1 command to compile it, and the program name (without the language suffix) to execute it. Note that MacSissle does not have the usual ready message. She sets her message to "Karen is here" by using the general_ready (gr) command in her start_up.ec, the special exec_com that runs each time she logs in. (See the MPM Commands for information on the use of general_ready.)

In Figure 2-2, user Tom Smith is shown writing a program called times_2, which accepts an integer and prints the value of 2 times that integer. Smith takes advantage of the terminal for both input to and output from his program.

A Note on Examples

Because Multics is written mainly in PL/I, you may find that its runtime environment is somewhat oriented towards the convenience of PL/I programmers. Ways to take advantage of this orientation are presented in Appendix A, "Using Multics to Best Advantage". However, as mentioned in the preface, this manual is intended to be useful for all programmers. Although the majority of the examples are given in PL/I, there is no need to be discouraged if you aren't familiar with this language. Most of the examples are extremely simple. To see how you could write the same program in either PL/I, FORTRAN, or COBOL, see Section 4, "Using the Terminal for I/O".

ARCHIVING SEGMENTS

Segments in Multics are assigned space in increments of pages (4096 characters). This can be very wasteful if you have many short files stored in the system. The archive (ac) command allows you to combine several segments into a single segment called an archive. Once in an archive, the individual segments are called components of the archive segment. Packing segments together in this way can produce significant savings in storage allocation and cost.

By invoking the archive command with different arguments, you can manipulate the archive segment in a variety of ways. For example, in addition to creating your archive, you can also get a table of contents that names each component in the archive, extract one or more components from the archive, update and replace one or more components, and delete individual components.

```
!  login MacSissle
   Password:
!
   MacSissle ProjA logged in 03/18/81 0921.4 mst Wed from VIP7801
    terminal "none".

   Last login 03/18/81 0726.2 mst Wed from VIP7801 terminal "none".
   Karen is here

!  qedx
!  a
!  simple_sum: proc options (main);
!
!  /* this program computes the sum of three numbers set in the program,
!     then prints the answer at the terminal */
!
!  declare
!        sysprint  file,                          /* the terminal output */
!        first_no  fixed binary (17),             /* the first number */
!        second_no fixed binary (17),             /* the second number */
!        third_no  fixed binary (17),             /* the third number */
!        the_sum   fixed binary (17);             /* the answer */
!
!  /* set the three numbers */
!
!       first_no = 123;
!       second_no = 456;
!       third_no = 789;
!
!  /* add them up */
!
!       the_sum = first_no + second_no + third_no;
!
!  /* print the answer */
!
!       put skip list ("The sum of the three numbers is:", the_sum);
!       put skip;
!
!    end simple_sum;
!  \f
!  w simple_sum.pl1
!  q
   Karen is here

!  pl1 simple_sum
   PL/I
   Karen is here

!  simple_sum
   The sum of the three numbers is:      1368
   Karen is here
```

Figure 2-1. Sample Terminal Session #1

```
!  login TSmith
   Password:
!
   TSmith ProjA logged in 06/07/79 0937.5 mst Tue from ASCII
    terminal "234".
   Last login 06/06/79 1359.8 mst Mon from ASCII terminal "234".

   A new PL/I compiler was installed; type: help pl1_new
   Rates for CPU usage have changed; type: help prices
   r 9:37 1.314 30

!  qedx
!  a
!  times_2:  proc;
!  declare (num,product) fixed bin(17);
!  declare (sysin input, sysprint output) file;
!  put list ("Enter integer");
!  put skip;
!  get list (num);
!  product = num*2;
!  put skip list ("2 times your integer is:", product);
!  put skip;
!  close file (sysin), file (sysprint);
!  end;
!  \f
!  w times_2.pl1
!  q
   r 9:40 4.875 62

!  pl1 times_2
   PL/I
   r 9:41 2.906 272

!  times_2
   Enter integer
!  19

   2 times your integer is:     38
   r 9:43 0.231 50
```

Figure 2-2. Sample Terminal Session #2

For more information about the archive command and its use, refer to the MPM Commands.

BINDING SEGMENTS

The Multics bind (bd) command is used to merge several separately compiled object segments into a single executable object segment called a bound segment. The binder is primarily an optimizer, which saves search time and link snapping. It resolves as many external references as it can in order to avoid the necessity of resolving them at run time. These references are resolved without recourse to the search rules--the binder looks only in the programs that are being bound, and rejects any programs in which there are ambiguous external references.

Binding offers the advantages of taking up less storage for the object code, decreasing execution time, and avoiding many linkage faults that would otherwise occur if the bound programs referenced each other from separate segments. Those programs that you call frequently and that are interrelated (ie, reference one another) should be bound to improve program efficiency. The segments must be archived before they are bound.

For more information about the bind command, refer to the MPM Commands. Also, the MPM Subsystem Writers' Guide provides information on the structure of bound segments.

LINKS

The word "link" is used for two separate things in Multics: an intersegment link and a storage system link. This can be confusing for beginners, but once you know the system, things are usually clear from their context.

An intersegment link is an interprocedure reference, resolved by the linker. This kind of link is described in Section 3, "Dynamic Linking".

A storage system link is essentially a "pointer" to a "target". This kind of link is described here. A storage system link is catalogued in a directory like a segment, but just gives the pathname of some other place in the directory hierarchy. The target of such a link is usually a segment, but it can also be a directory, or even another link. A storage system link enables you to access a segment located in some other portion of the directory hierarchy without actually making a copy of it, just as if it were catalogued in your own working directory. This is one of the ways in which Multics facilitates sharing.

Multics allows you to create a link anywhere in the storage system as long as you have the proper access to the directory in which the link is to be placed. You invoke the link (lk) command to create a link and the unlink (ul) command to delete a link. (Refer to the MPM Commands.) To see a list of the links you have in your working directory, you can use the list command with the -link control argument.

SECTION 3

DYNAMIC LINKING

As the discussion of dynamic linking in Section 1 indicated, external references on Multics are resolved when a program is executed. When the system comes across an unresolved reference, it uses what are known as search rules to find the necessary segment and establish the link. The purpose of this section is to explain how the search rules operate, then to show you some of the uses of dynamic linking.

A NAMING CONVENTION

Due to a Pl/I extension which is local to Multics, the "$" character is understood when it appears as part of an external name. a$b is interpreted to mean segment a, entry point b. (Remember that an entry point is a symbolic offset within the segment. Refer to the discussion of two-dimensional addressing in Section 1.) Thus, hcs_$initiate, which will be discussed later in this section, is interpreted to mean segment hcs_, entry point initiate.

SEARCH RULES

Let's suppose that you are writing a new version of the Qedx Text Editor, and have a segment in your working directory named "qedx". If you type "qedx" on your terminal, you are instructing Multics to find the program named qedx and execute it. But which qedx do you want--yours or the system's? To make the situation a little bit more complicated, let's suppose that one of your coworkers is also writing a new version of Qedx, and has a segment in one of his directories named "qedx", to which you have access. You might want to run his program sometimes instead of yours or the system's.

In each case, it's up to Multics to figure out which segment you want. The way Multics does this is by searching. To understand why Multics searches the way it does, you first need to know some of the assumptions it works under.

Once you have invoked some program or accessed some data base, Multics assumes there is a good chance you will do so again. If the item is in your address space, that cuts down on the system overhead required to make a complete search for it a second or third time. So Multics keeps track of all the work you do after you login. It records your movement through the file system, noting each item it has located for you and putting these items in your address space. Multics also assumes that any time you use a reference name which you have already used, you mean the same item you meant the first time. (A reference name is a name used to identify a segment that has been made known by the user.) The name of the item and the information the system needs to find it are recorded in a table called the reference name table. Segments in this table are referred to as initiated segments.

The search rules are a list of directories which are searched in order until the desired segment is found. The standard search rules are:

1. initiated_segments

   Reference names for segments that have already been made known to a specific process are maintained by the system. A reference name is associated with a segment in one of four ways:

   a. use in a dynamically linked external program reference.

   b. use in an invocation of the initiate command.

   c. a call to hcs_$initiate, hcs_$initiate_count, or hcs_$make_seg with a nonnull character string supplied as the ref_name argument. These hcs_ entry points are described in the MPM Subroutines.

   d. a call to hcs_$make_ptr or hcs_$make_entry (described in the MPM Subroutines).

2. referencing_dir

   The referencing directory contains the segment whose call or reference initiated the search. So, if pgma calls pgmb, and pgmb isn't in the reference name table, the system looks for pgmb in the directory where pgma resides.

3. working_dir

   The working directory is the one associated with you at the time of the search. This may be any directory established as the working directory by either the change_wdir command or the change_wdir_ subroutine (described in the MPM Commands and MPM Subroutines respectively). The initial working directory is your home directory.

4. system libraries
   The system libraries are searched in the following order:

   >system_library_standard
   This library contains standard system service modules, i.e., most system commands and subroutines.

   >system_library_unbundled
   This library contains Multics Separately Priced Software.

   >system_library_1
   This library contains a small set of subroutines that are reloaded each time the system is reinitialized.

   >system_library_tools
   This library contains software primarily of interest to system programmers.

   >system_library_auth_maintained
   This library contains user maintained and installation maintained programs.

You can see what your process's current search rules are by using the print_search_rules (psr) command:

```
!  psr
   initiated_segments
   referencing_dir
   working_dir
   >system_library_standard
   >system_library_unbundled
   >system_library_1
   >system_library_tools
   >system_library_auth_maint
```

Note that, according to these search rules, if you have in your working directory a program with the same name as a system command or subroutine, your program will be used rather than the system's. Don't give your programs the same names as those of system programs, unless you really are trying to replace them. Here is an example of the trouble you can get into when you duplicate the name of a system program. Suppose you have a program of your own which creates an output file and you name the file "list." If you run your program, then try to list your working directory using the list command, you will get a message like this:

```
command_processor_:  Linkage section not found.  list
```

The system thinks you are trying to run your output file, list, as a program!

You can modify your search rules by using the add_search_rules (asr), delete_search_rules (dsr), and set_search_rules (ssr) commands, described in the MPM Commands. In addition, your system administrator can modify the default search rules described above for all users at your site.

Thus, the first time you invoke a program after login, the system begins its search for the segment by looking in the reference name table. The search fails there, so it continues through the list of directories in the search rules until the segment is found or all the directories have been searched. Subsequent invocations of the same program are much faster, because the system finds the program right away in the reference name table.

A Note on Initiated Segments

If your program named x references a program named y by means of a call or function reference, a dynamic link is established between x and y so that all subsequent references to y by x are accomplished by using the segment number (the alias for the segment name discussed in Section 1). If you change to a new working directory, and execute a program named z that calls a program in this new directory named y, the system will establish a dynamic link to the original segment y because the reference name y is still associated with the original segment and segment number. The system maintains this association until the reference is terminated. See Figure 3-1 for an illustration of initiated segments working in this way.

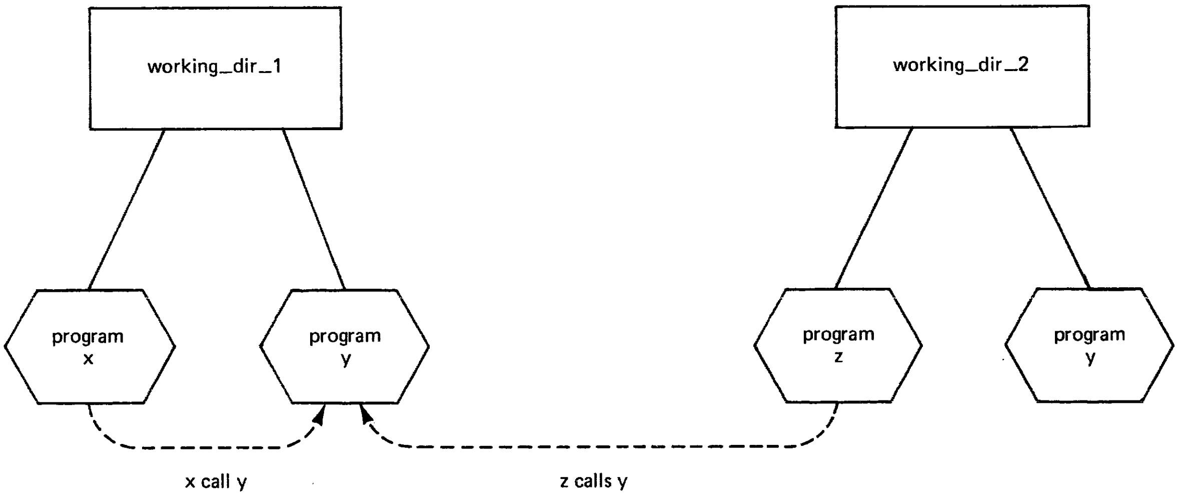


Figure 3-1. Initiated Segments

Segments can be made known to your process by using the initiate (in) command. You can list your initiated segments with the list_ref_names (lrn) command. References can be terminated by using one of the terminate commands, either terminate (tm), terminate_refname (tmr), terminate_segno (tms) or terminate_single_refname (tmsr), which allow you to remove segments from the list of segments known to your process. (The new_proc command also erases all previous association between segment names and segment numbers, by sweeping out your entire address space.) For more detailed information on these commands, see the MPM Commands.

Deleting a segment also terminates it. Recompiling a program unsnaps all links in the current process which point to the program, since the location of symbolic entry points may be changed by recompilation. Both of these actions affect only the process performing the operation. Recompiling or deleting a segment in one process may cause other processes using the segment to malfunction.

USES OF DYNAMIC LINKING

There are many ways in which dynamic linking can be used, but the following three are probably the most significant:

- to permit initial debugging of collections of programs before the entire collection is completely coded.
- to permit a program to include a conditional call to an elaborate error handling or other special-case handling program, without invoking a search for or mapping of that program unless the condition arises in which it is actually needed.
- to permit a group of programmers to work on a collection of related programs, such that each one obtains the latest copy of each subroutine as soon as it becomes available.

The use of dynamic linking in program development is shown by the following script. When the script starts, the program "k" and subprogram "y" have already been written and compiled by our user MacSissle.

```
k: procedure;

      declare (x, y, z)              entry;
      declare i                      fixed binary;
      declare (sysprint, sysin)        file;

      put list ("Which option?");
      get list (i);
      if i = 1 then call x;
      else if i = 2 then call y;
      else if i = 3 then call z;
      else put list ("Bad option ");
      return;
   end k;

y: procedure;
      declare sysprint               file;
      put list ("y has been called.");
      put skip;
   end y;
```

In this example and all others like it in this manual, comments on the script are distributed throughout the script itself.

```
! k
  Which option? ! 2
        y has been called.

  r 17:11 0.123 11
```

The program "k" is invoked by typing its name. MacSissle calls for option 2, and the program "y" is called. "k" runs successfully even though two of the three subroutines it could call do not exist, because the subroutine it does call is available. Since linking is done on demand, and no demand for "x" or "z" occurs, their nonexistence does not keep the program from running.

In the next use of "k", MacSissle asks for an option corresponding to the program "z," which doesn't exist.

```
! k
  Which option? ! 3
  Error:  Linkage error by >udd>ProjA>MacSissle>k|152 (line 11)
   referencing z|z
   Segment not found.
  r 17:11 0.283 90 level 2
```

The attempt to call the nonexistent subroutine "z" fails. The linkage error handler invokes a second command level, as indicated by the field "Level 2" in the ready message. The error message shows the full pathname of the program attempting to locate "z," and gives the name of the program that could not be found. The notation "z|z" means entry point "z" in segment "z." It is necessary to separate entry point name from segment name, since a PL/I program in a segment could have several entry points with different names.

Execution of "k" is suspended, since it cannot continue with the call. MacSissle has the choice of giving up, or creating "z." She invokes the qedx editor and creates the segment.

```
! qedx
! a
! z: procedure;
!       declare sysprint file;
!       put list ("This is Z")
!       put skip;
!    end z;
! \f
! w z.pl1
! q
  r 17:12 0.382 48 level 2
```

Now the segment must be compiled to create a callable object segment.

```
! pl1 z -table
  PL/I
  r 17:12 0.234 65 level 2
```

With the object segment "z" created, the call from "k" can be restarted. MacSissle does this with the start command.

```
! start
         This is Z
 r 17:12 0.166 27
```

The program finishes successfully. It can now be run with option 3 without any additional intervention.

```
! k
 Which option? ! 3
         This is Z

 r 17:13 0.075 18
```

For more information on the details of dynamic linking, see the MPM Reference Guide sections on object segments, system libraries and search rules. You might also want to learn about the resolve_linkage_error (rle) command, which can be used to satisfy the linkage fault after your process encounters a linkage error. This command is described in the MPM Commands.

SEARCH PATHS

Searching is something that Multics has to do all the time. So far we've only talked about searching for object segments--what Multics has to do when you type the name of a program you want to execute, or your program references an external procedure. Multics has to search for other things, too, notably input of some kind. For example, the help command requires as input an info segment. You can tell the system to look in specific places for the input by creating search paths. Search paths have the same basic function as search rules, but are used for things like subsystems and language compilers. A set of commands similar to those available for modifying search rules are available for modifying search paths. These commands are add_search_paths (asp), delete_search_paths (dsp), print_search_paths (psp), set_search_paths (ssp), and where_search_paths (wsp). All are documented in the MPM Commands.

SECTION 4

INPUT/OUTPUT PROCESSING

Input/output (I/O) processing on Multics can be handled in many different ways. The intent of this section is to show you how to do simple kinds of I/O on Multics, and to introduce you to the basics of doing more complex I/O.

The Multics I/O system handles logical rather than hardware I/O. This means that I/O on Multics is essentially device independent. In other words, you don't have to write your program with a specific device in mind. Most I/O operations refer only to logical properties (e.g., the next record, the number of characters in a line) rather than to particular device characteristics or file formats. To understand how I/O processing on Multics works, you must first be familiar with two important terms.

(1) I/O switch: a software construct through which the file name in your program is associated with an actual device. The I/O switch is like a channel, in that it controls the flow of data between your program and a device. It keeps track of the association between itself and the device and the I/O module.

(2) I/O module: a system or user-written program that controls a physical device and acts as an intermediary between it and your program. The I/O module knows what the attributes of the device are, and "hides" them from you so you don't have to worry about them. It processes the I/O requests that are directed to the switch attached to it. The Multics system offers the following I/O modules:

discard_
provides a "sink" for unwanted output.

rdisk_
supports I/O directly from/to removable disk packs. (These are packs which are allocated in their entirety to a process; they do not contain files in the Multics storage system.)

record_stream_
provides a means of doing record I/O on a stream file or vice-versa.

syn_
establishes one switch as a synonym of another.

tape_ansi_
supports I/O from/to magnetic tapes according to standards proposed by the American National Standards Institute (ANSI).

tape_ibm_
  supports I/O from/to magnetic tapes according to IBM standards.

tape_mult_
  supports I/O from/to magnetic tapes in Multics standard tape format.

tape_nstd_
  supports I/O from/to magnetic tapes in nonstandard or unknown format.

tty_
  supports I/O from/to terminals.

vfile_
  supports I/O from/to files in the storage system.

Figure 4-1 illustrates the flow of data between a program, an I/O switch, an I/O module, and a device.

THE FIVE BASIC STEPS OF INPUT/OUTPUT

For every input/output data stream you are using, you must follow the 5 basic steps of Multics I/O processing, which involve attaching an I/O switch to an I/O module, opening the switch, performing the data transfer, closing the switch, and detaching it from the I/O module. These steps may be accomplished outside of your program by means of commands input before and after your program runs, or inside your program by means of subroutine calls or language I/O statements. (Defaults are arranged so you can often appear to skip these steps, and they will be done correctly anyway.)

(1) Attach the Switch

This step associates your data with a file in your program. The switch is the program's name for each data stream. (In FORTRAN, switches are called file05, file10, etc.) An attachment statement in Multics is comparable to a JCL data definition (DD) statement in IBM systems. A switch remains attached until you detach it or you issue a new_proc or logout command.

A switch may be attached by:

- invoking the io_call command
- issuing a call to the iox_ subroutine
- using a language open statement (if the switch hasn't been previously attached)
- using the default attachments associated with PL/I gets and puts, FORTRAN reads and writes, or COBOL reads and writes

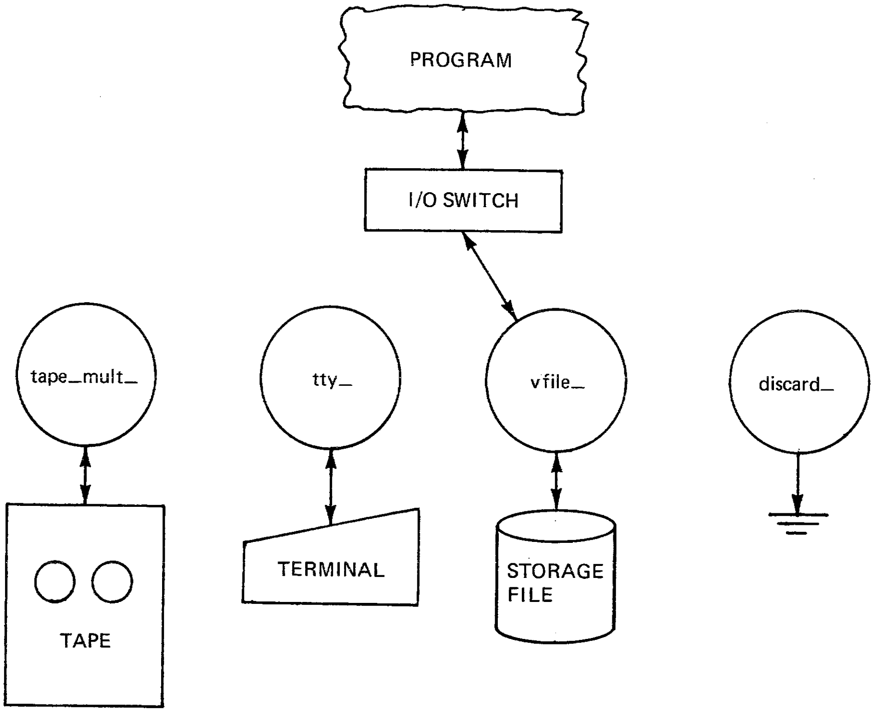


Figure 4-1. Flow of Data

(2) Open the Switch

This step describes the data you're going to use. It tells the system how the data is organized (its file type) and how it is to be accessed (its mode). Data sets can be organized in four fundamental ways: stream, sequential, blocked, and keyed. Only the first two ways will be discussed here.

A stream file is a collection of data that is like free-form text. The data is a continuous flow of information, with individual items separated by blanks, commas, or newline characters. A stream file can be created, examined, and updated via a text editor, and can be meaningfully printed on a terminal or line printer, because it contains only ASCII characters. It's size is arbitrary.

A sequential file is a collection of data that is broken into discrete units called records, which have a fixed form. A sequential file is created by a program, and is used for information which is meant to be read and processed by another program. The data are in the same coded form as data stored internally in the computer and can't be printed meaningfully.

Most tape files are sequential. Disk files may be either stream or sequential. Terminal I/O is stream-oriented.

Data sets can be operated on in three fundamental ways: input only, output only, or both input and output. Some of the opening modes of a switch are therefore:

si - stream input	sqi - sequential input
so - stream output	sqo - sequential output
sio - stream input/output	sqio - sequential input/output

A switch may be opened by:

- invoking the io_call command
- issuing a call to the iox_ subroutine
- using a language open statement
- using PL/I gets, puts, reads, and writes, FORTRAN reads and writes, or COBOL reads and writes--the switch is opened by default

(3) Perform I/O Operations

This step is where the data transfer actually occurs.

Data transfer may be performed by:

- invoking the io_call command
- issuing a call to the iox_ subroutine
- using language defined I/O statements (gets, puts, reads, writes, etc.)

(4) Close the Switch

This step tells the system you are through (at least temporarily) with the I/O switch. It prevents further access to the data through that switch, enables you to re-open the switch later with a different mode, and with output disk files and tapes, sets the length of the file.

A switch may be closed by:

- invoking the io_call command
- issuing a call to the iox_ subroutine
- using a language close statement
- default (on your program's return), if and only if the switch was opened by default

(5) Detach the Switch

This step disconnects your program from your data.

A switch may be detached by:

- invoking the io_call command
- issuing a call to the iox_ subroutine
- using a language close statement
- default (on your program's return), if and only if the switch was attached by default

USING THE TERMINAL FOR I/O

The simplest way to do I/O on Multics is to use the terminal. There are four standard switches which are attached when your process is created.

(1) user_i/o: this switch acts as a common collecting point for all terminal I/O. It's attached to your terminal through the I/O module tty_ and opened for stream input and output.

(2) user_input: this switch controls command and data input at the terminal. It's attached to user_i/o through the I/O module syn_, and through that to your terminal. It's opened for stream input.

(3) user_output: this switch controls command and data output at the terminal. It's attached to user_i/o through the I/O module syn_, and through that to your terminal. It's opened for stream output.

(4) error_output: this switch controls output of error messages at the terminal. It's attached to user_i/o through the I/O module syn_, and through that to your terminal. It's opened for stream output.

Figure 4-2 illustrates these standard attachments.

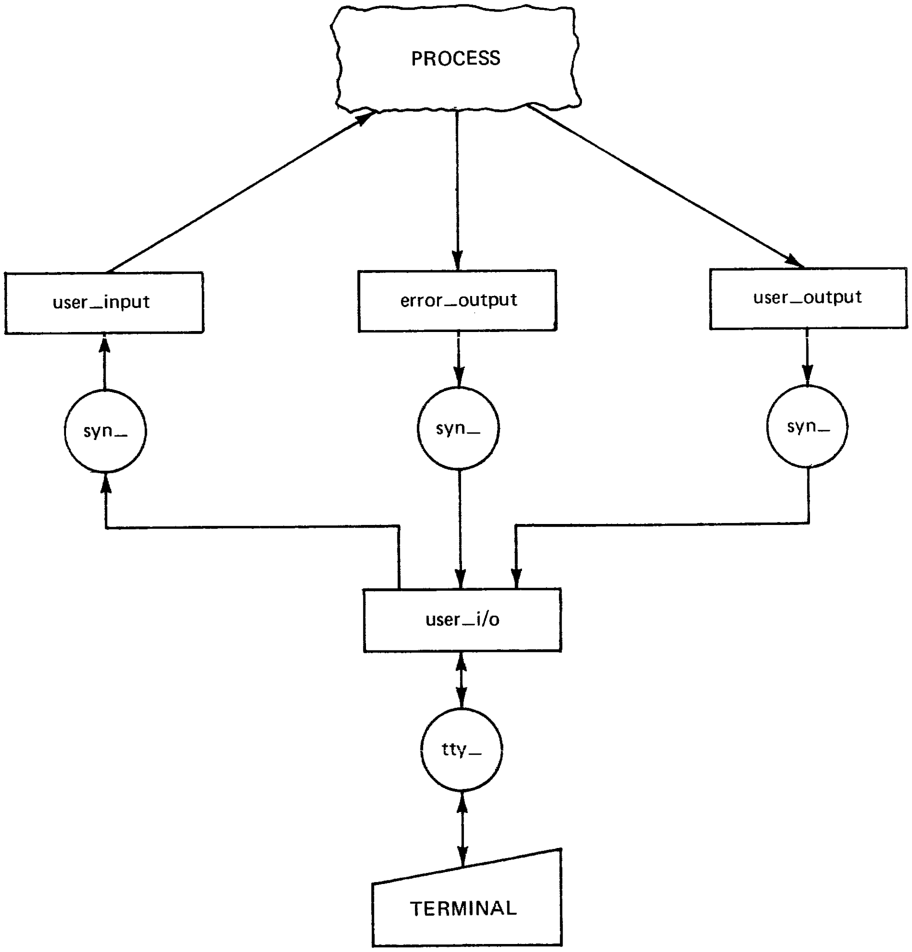


Figure 4-2. Standard Attachments

If you don't specify switch names and I/O modules when you run your program, the system uses these defaults. So, it's possible to write your program using the terminal for input and output and not worry about files. For example, here is a revised version of our sample program from Section 2, simple_sum. It has been renamed any_sum, and changed to accept input typed by the user at the terminal in response to a prompting message. The output is typed back on the terminal. Notice the use of sysin and sysprint for the terminal input and output.

```
any_sum: proc options (main);

/* this program computes the sum of any three 1 to 6 digit numbers typed
   at the terminal, then prints the answer at the terminal */

declare
       sysin     file,                      /* the terminal input */
       sysprint  file,                      /* the terminal output */
       first_no  fixed binary (20),         /* the first number */
       second_no fixed binary (20),         /* the second number */
       third_no  fixed binary (20),         /* the third number */
       the_sum   fixed binary (24);         /* the answer */

/* get the three numbers */

       put skip list ("please type three 1 to 6 digit numbers:");
       get list (first_no, second_no, third_no);

/* add them up */

       the_sum = first_no + second_no + third_no;

/* print the answer */

       put skip list ("the sum of the three numbers is:", the_sum);
       put skip;

     end any_sum;
```

Here are FORTRAN and COBOL versions of the same program.

```
c       This program computes the sum of any three numbers typed at the
c       terminal, then prints the answer at the terminal.

        integer first_no, second_no, third_no         ! the 3 numbers
        integer the_sum                               ! the answer

c       Get the three numbers

        print, "please type three numbers:"
        input, first_no, second_no, third_no

c       Add them up

        the_sum = first_no + second_no + third_no

c       Print the answer

        print, "the sum of the three numbers is:", the_sum

        stop
        end
```

Detailed information about how the command utility and active function error subroutines can be used from an active function procedure is provided in the MPM Subroutines and the MPM Subsystem Writers' Guide respectively.

The same procedure can be programmed to operate both as an active function and as a command procedure. Typically when such procedures are called as a command, they print on the user's terminal the value of the string they would return as an active function. These command/active function procedures are coded as active functions and should call cu_$af_return_arg instead of cu_$af_arg_count. If cu_$af_return_arg returns the error code error_table_$not_act_fnc, they operate as commands. If the code returned is zero, they use the returned pointer and length to base the return value. Any other nonzero error code should be fatal. Note that cu_$af_return_arg always returns a correct argument count even if the active function was invoked as a command, so the user can go on to use cu_$arg_ptr with no further checking.

ADDRESS SPACE MANAGEMENT

When a user logs in, he or she is assigned a newly created process. Associated with the process is a collection of segments that can be referenced directly by system hardware. This collection of segments, called the address space, expands and contracts during process execution, depending on which segments are used by the running programs.

Address space management consists of constructing and maintaining a correspondence between segments and segment numbers, segment numbers being the means by which the system hardware references segments. Segment numbers are assigned on a per-process basis (i.e., for the life of the process), by supplying the pathname of the segment to the supervisor. This assignment is referred to as "making a segment known." Segments are made known automatically by the dynamic linker when a program makes an external reference; making a segment known can also be accomplished by explicit calls to address management subroutines. In addition, when a segment is made known, a correspondence can be established between the segment and one or more reference names (used by the dynamic linker to resolve external references); this is referred to as "initiating a reference name." When dynamic linking is the means used to make a segment known, the initiation of at least one reference name is performed automatically. (For more information on reference names, see "Reference Names" in Section 3 and "Making a Segment Known" below.) A general overview of dynamic linking is given below.

Dynamic Linking

The primary responsibility of the dynamic linker is to transform a symbolic reference to a procedure or data into an actual address in some procedure or data segment. In general, this transformation involves the searching of selected directories in the Multics storage system and the use of other system resources to make the appropriate segment known. The search for a referenced segment is undertaken after program execution has begun and is generally required only the first time a program references the address.

The dynamic linker is activated by traps originally set by the translator in the linkage section of the object segment. These traps are used by instructions making external references. When such an instruction is encountered during execution, a fault (trap) occurs and the dynamic linker is invoked.

The dynamic linker uses information contained in the object segment's definition and linkage sections to find the symbolic reference name. (For a detailed description of these sections, see "Multics Standard Object Segment" in Section 1 in the MPM Subsystem Writers' Guide.) Using the search rules currently in effect, the dynamic linker determines the pathname of the segment being referenced and makes that segment known. The linkage trap is modified so that the fault does not occur on subsequent references; this is referred to as snapping the link.

```
identification division.
program-id. anysum.
author. KMacSissle.
date-written. February 1981.
date-compiled.
     remarks. This program computes the sum of any three 1 to 6 digit
     numbers typed at the terminal, then prints the answer at the
     terminal.

environment division.
configuration section.
source-computer.  Multics.
object-computer.  Multics.

data division.
working-storage section.

01  first-no     pic 9(6)  value zeroes.
01  second-no    pic 9(6)  value zeroes.
01  third-no     pic 9(6)  value zeroes.
01  the-sum      pic 9(7)  value zeroes.

procedure division.

100-get-three-numbers.
     display "please type three 1 to 6 digit numbers".
     display "(numbers less than 6 digits long must be zero-filled,".
     display " and each number must be typed on a new line):".

     accept first-no.
     accept second-no.
     accept third-no.

200-add-them-up.
     compute the-sum = first-no + second-no + third-no.

300-print-the-answer.
     display "the sum of the three numbers is: ", the-sum.
     stop run.
```

USING SEGMENTS AS STORAGE FILES

When your application requires the use of a storage file for I/O, the easiest thing to do is to use a segment in your working directory (or a segment in another directory to which you have created a link). In your program, you must do the following:

(1) Give the file a name and declare it as a file;

(2) Open it (connect it to your program, prepare it for processing, and position it at the beginning);

(3) Do data transfer via one or more get, put, read or write statements (depending on the language you're using);

(4) Close it (disconnect it from your program).

Here is a revised version of the any_sum program. It's been renamed compute_sum, and changed so that it gets its input from a segment in your working directory called in_file. The output goes to another segment in your working directory called out_file.

```
compute_sum: proc options (main);

/* this program computes the sum of three 1 to 6 digit numbers read from
   an input file, then writes the answer to an output file  */

declare
       in_file   stream file,                      /* the input file */
       out_file  stream file,                      /* the output file */
        first_no  fixed binary (20),               /* the first number */
       second_no fixed binary (20),                /* the second number */
       third_no  fixed binary (20),                /* the third number */
       the_sum   fixed binary (24);                /* the answer */

/* open the files */

       open file (in_file) input,
            file (out_file) output;

/* get the three numbers from the input file */

       get file (in_file) list (first_no, second_no, third_no);

/* add them up */

       the_sum = first_no + second_no + third_no;

/* put the answer in the output file */

       put file (out_file) list (the_sum);


/* close the files */

       close file (in_file),
            file (out_file);

    end compute_sum;
```

Doing I/O this way also takes advantage of the default switches and modules. The open statement attaches and opens the switch, the close statement closes and detaches the switch.

What if the files you need to use are not segments in your working directory? One thing you can do, if you're a PL/I programmer, is to use the title option on your open statement. For example:

```
open file (in_file) title
   ("vfile_ >udd>ProjA>MacSissle>data_files>test_file_1") input;
```

where

```
vfile_ >udd>ProjA>MacSissle>data_files>test_file_1
```

is an example of an *attach description*. An attach description is a string of characters which identify the name of an I/O module and options to control its operation. In this case, the only option given is the source/target of the attachment (i.e., the name of the device or file).

Other languages have constructs which are somewhat similar to the PL/I title option. In FORTRAN, there is the attach specifier, which is used on an open statement. In COBOL, there is the catalog-name clause. See the Language Users' Guides for information on how to use these constructs.

USING I/O COMMANDS AND SUBROUTINES

The use of I/O commands and subroutines is where I/O processing may become more complex. The following discussion is not intended to fully explain their use, but rather, to introduce the basic concepts involved. For more information, refer to the MPM Reference Guide, Section 5. Information is also available in the Language Users' Guides.

The command for performing operations on designated I/O switches is io_call (io). Its syntax is:

```
io opname switchname {args}
```

It is used as follows:

(1) To attach a switch:

```
syntax:  io attach switchname modulename {args}
example: io attach my_switch vfile_ >udd>ProjA>MacSissle>my_file
```

(vfile_ >udd>ProjA>MacSissle>my_file is another example of an attach description.)

(2) To open a switch:

```
syntax:  io open switchname mode
example: io open my_switch sequential_input
```

(3) To close a switch:

```
syntax:  io close switchname example: io close my_switch
```

(4) To detach a switch:

```
syntax:  io detach switchname example: io detach my_switch
```

The io_call command is used outside of your program. A typical sequence at command level would involve attaching and opening the switches, running your program, then closing and detaching the switches. (Switches that are attached and opened at command level should usually be closed and detached at command level. However, they can also be closed explicitly by the program using language close statements.)

Other I/O-related commands include:

close_file (cf)
: closes specified FORTRAN and PL/I files. This command is very useful if your program opens a file, then terminates unexpectedly before closing it. You must close the file before you run the program again, or you'll get an end-of-file error.

copy_cards (ccd)
copies specified card image segments from the system pool storage into your directory. The segments to be copied must have been created using the Multics card image facility.

copy_file (cpf)
copies records or lines from an input file to an output file.

display_pl1io_error (dpe)
describes the most recent file on which a PL/I I/O error was raised and displays diagnostic information associated with that error.

file_output (fo)
directs all subsequent output over user_output to a specified segment.

print_attach_table (pat)
prints information about I/O switch attachments.

revert_output (ro)
restores all subsequent output to the previous device.

stop_cobol_run (scr)
causes the termination of the current COBOL run unit.

terminal_output (to)
directs all subsequent output over user_output to a terminal.

Three of these commands can show you a little about how switches work. Type "pat" on your terminal and the system will print this:

```
user_i/o              tty_ -login_channel
        stream_input_output
user_input            syn_ user_i/o
user_output           syn_ user_i/o
error_output          syn_ user_i/o
```

You can see from this that user_i/o is attached via the module tty_ to the login channel, and user_input, user_output, and error_output are attached via the module syn_ to user_i/o.

Type "fo my_file; pat; ro; pr my_file" on your terminal and the system will print something like this:

```
         my_file           03/10/81           1124.0 est Mon

user_i/o              tty_ -login_channel
           stream_input_output
user_input            syn_ user_i/o
user_output           syn_ fo_!BBBJKqdcZHXHFf
error_output          syn_ user_i/o
fo_save_!BBBJKqdcZJXgxW
     syn_  user_i/o
fo_!BBBJKqdcZHXHFf  vfile_ >udd>ProjA>MacSissle>my_file -extend
           stream_output
```

You can see from this that user_output was attached via vfile_ instead of syn_. (Refer to Figure 4-3.) For complete information on all of these commands, see the MPM Commands.

The most important subroutine for doing I/O is iox_. It is called from within your program just like any other subroutine, and can be used to attach, open, close and detach switches, as well as to read and write records, and perform various other I/O operations. Another subroutine for doing I/O is ioa_, which is used for producing formatted output; it can be very handy. The use of these subroutines is beyond the scope of this manual. Detailed information is available in the MPM Subroutines.

CARD INPUT AND CONVERSION

You may have programs punched on cards that you would like to compile and run under Multics. The standard way of handling a card deck on Multics is to place the deck in a card reader and read it into a system pool. Once this is done, you log in on a terminal, and transfer the card file from the system pool to your working directory using the copy_cards command already mentioned.

A minimum of three control cards must accompany your deck. These control cards identify you to the system, and specify the format of the card input you are submitting. There are two kinds of card input on Multics. One is bulk data input, which is usually a program or a data file. The format of a card deck for bulk data input is shown below:

```
++DATA  DECK_NAME  PERSON_ID  PROJECT_ID
++PASSWORD  PASSWORD
++CONTROL  OVERWRITE
++AIM  ACCESS CLASS OF DATA CARDS
++FORMAT  PUNCH_FORMAT  MODES
++INPUT
   .
   .
   .
(user data cards)
```

The three cards required as a minimum are the first, which is an identifier card, the second, which is a password card, and the last, which signals the end of control input.

The other kind of card input is remote job entry, which is a series of Multics commands to be run as an absentee job. For information on absentee jobs, and the format of a card deck for remote job entry, see Section 7. For a complete explanation of all the Multics control cards, see Appendix C of the MPM Reference Guide.

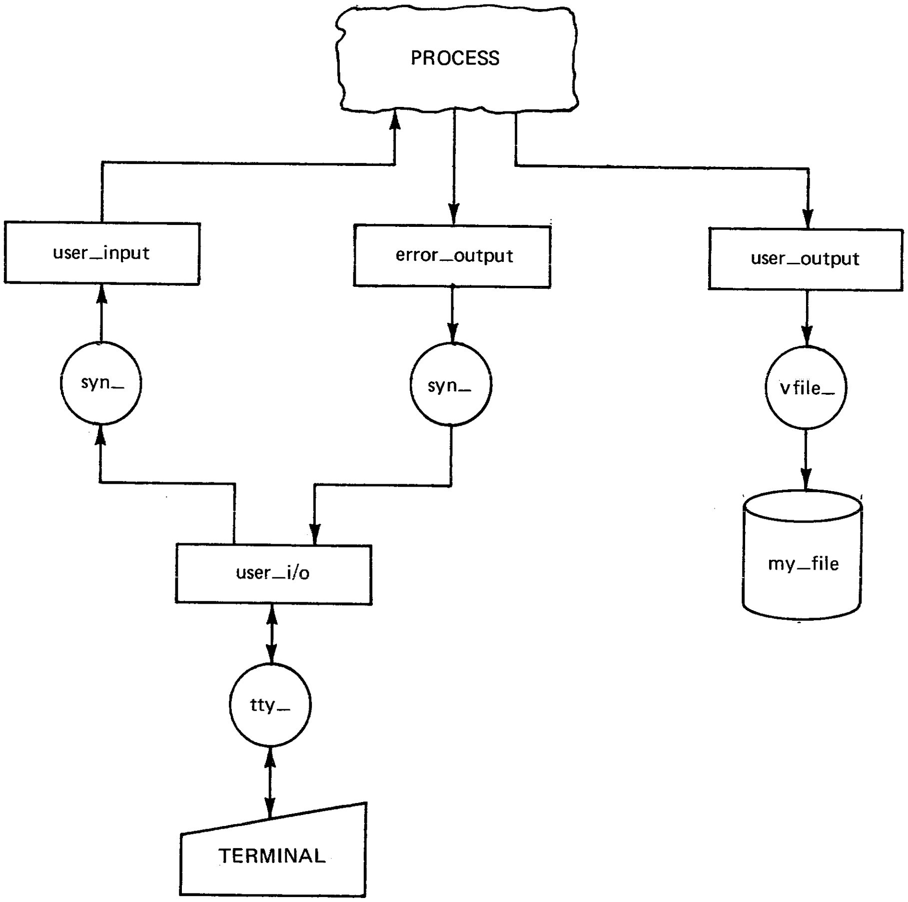


Figure 4-3. Attachments After Execution of file_output Command

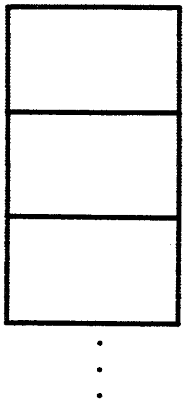
SECTION 5

A DEBUGGING TOOL

A variety of debugging tools are available on Multics. They allow you to look at your program piece by piece, in a way that is closer to the way the machine sees it. The most powerful of these tools is an interactive program named probe, which permits source-language breakpoint debugging of PL/I, FORTRAN, and COBOL programs. To understand the discussion of probe given later in this section, you must first know a little about the Multics stack.

THE STACK

Each process has associated with it a stack segment (called the stack) that contains a history of the environment. The stack is essentially a push down list which contains the return points from a series of outstanding interprocedure calls. It also holds storage for automatic variables. If you were to stop a running process and trace its stack, you would find, starting at the oldest entry in the stack, a record of the procedures used to initialize the process, followed by the command language processor, followed by the procedure most recently called at command level and any procedures it has called. Your stack can be visualized as follows:



The lines in the illustration above define stack frames. As control passes from program to program within the system, your stack "grows" new stack frames:

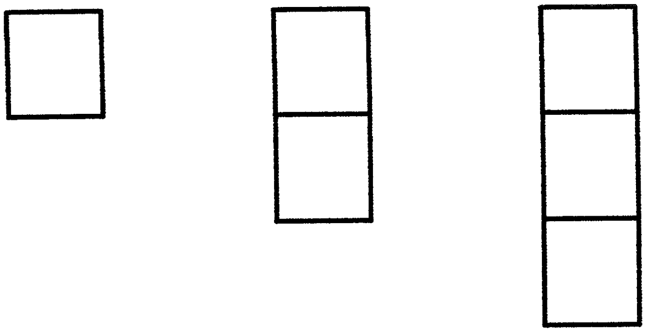


Figure 5-1 gives a pictorial view of what the stack might look like at different times during the execution of a program. In Figure 5-1a, the last frame of the stack is for the command level programs. From command level, you can type commands at the terminal. Once a command is typed, that program is called and a stack frame immediately allocated for it. (This is shown in Figure 5-1b). The stack remains in this state for the duration of execution of the program.

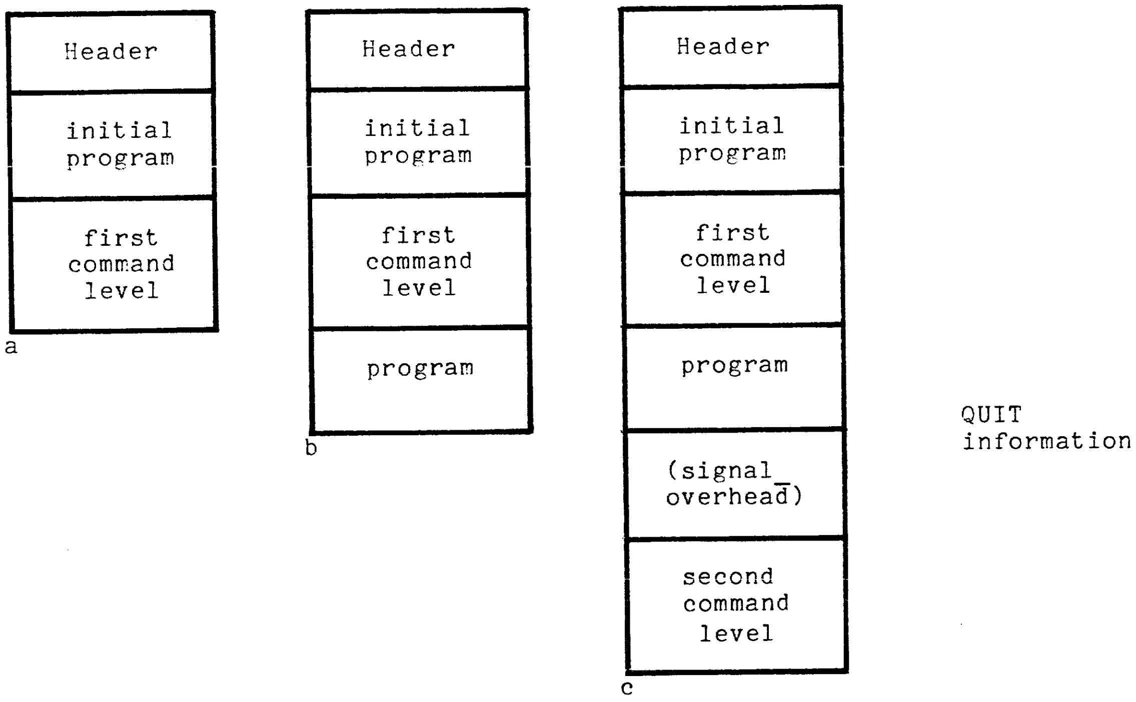


Figure 5-1. State of Stack

(a) State of Stack after Login
(b) State of Stack after Command is invoked
(c) State of Stack after QUIT

Figure 5-1c depicts the stack after a QUIT is signalled. Here a second command level is established. The first command level, and the program itself, have been suspended, but nothing has been thrown out.

At this point further commands could be issued. The start command would cause the program to resume execution, and the stack to revert to the state illustrated in Figure 5-1b. The release command would cause the stack frame (and hence the execution state) of the program to be discarded, and the stack to revert to the state depicted in 5-1a.

Note that it would be possible at the second command level (Figure 5-1c) to invoke the same program called at the first command level.

Figure 5-2 illustrates several of the states of the stack during execution of a program consisting of several subprograms. The call/return sequence depicted is:

Program A calls program B
Program B calls program C
Program C returns to B
Program B calls program D
Program D returns to B
Program B returns to A
Program A returns to command processor

These diagrams illustrate the behavior of four separately compiled programs, each allocated a new stack frame every time it is called:

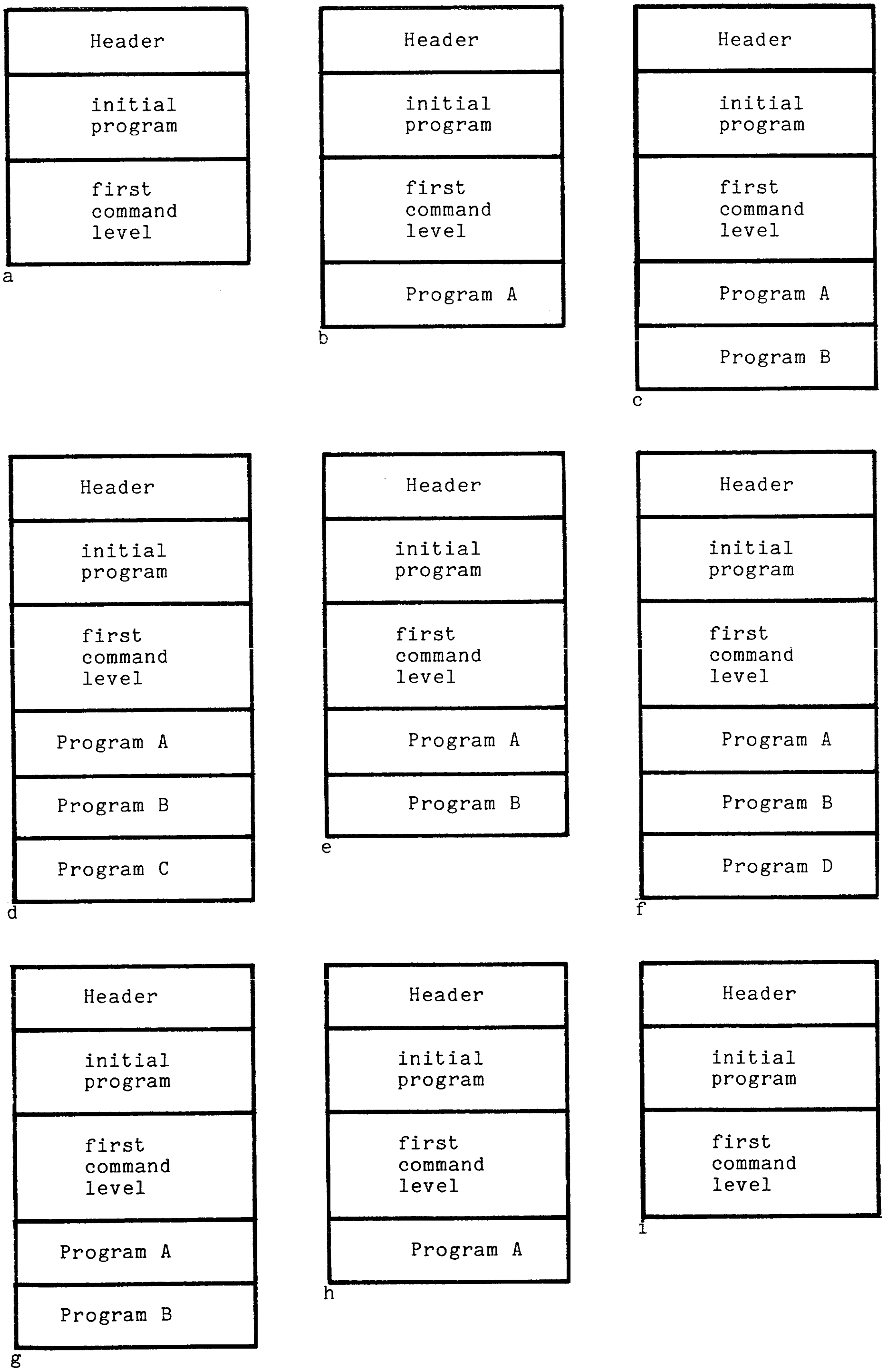


Figure 5-2. Allocation of Stack Frames

(a) User at command level.

(b) A is invoked and gets stack frame, in which automatic variables are allocated and initialized.

(c) A calls B. B gets stack frame, in which automatic variables are allocated and initialized.

(d) B calls C, C gets stack frame, in which automatic variables are allocated and initialized.

(e) C returns to B, the stack frame for C is discarded, and storage is released.

(f) B calls D, D gets stack frame, in which automatic variables are allocated and initialized.

(g) D returns to B, the stack frame for D is discarded, and storage is released.

(h) B returns to A, the stack frame for B is discarded, and storage is released.

(i) A returns to command level. All program-specific automatic storage has been released.

Automatic storage is storage which stays around only for the life of a program. Static storage is storage which stays around for the life of a process, or is retained across processes.

If an unexpected error occurs (or you press the QUIT button), the system will save the current environment, mark the stack at its current level, and push a frame onto the stack for a new activation of the command processor.

The new activation of the command processor accepts commands just as the original one did. It is possible to restart the suspended program, or to discard the saved environment, or to use one of the Multics debugging tools to examine the saved environment.

The release command causes the command processor to return to its own previous activation, and discard the intervening stack contents. The programs whose stack contents have been discarded cannot be resumed or examined after the stack has been released.

The start command causes the command processor to attempt to continue execution of the suspended program at the point of interruption. Depending on the nature of the error, and what has been done since the error occurred, the restart attempt may or may not succeed. Programs may always be restarted after a QUIT, but only seldom after an error. If the program cannot be restarted, the error message will usually be repeated. An unsuccessful attempt to restart a program is usually harmless.

If you would like to examine the stack history of your process in detail, try using the trace_stack (ts) command, described in the MPM Commands.

PROBE

The probe (pb) command can be used to examine the saved stack and the current state of suspended programs. (Remember that a program which makes a call to another program is suspended just as a program which makes an error is suspended, except that a program which makes a call can always be resumed.) Probe can print the values of program variables and arguments, as well as reporting the last program location to be executed.

The use of probe is shown here in a series of examples, which make use of the following program, blowup.pl1. This program has an illegal reference to the array "a", and the subscriptrange condition occurs when it is run. Since subscriptrange checking is disabled by default in PL/I, the error manifests itself as an out_of_bounds condition instead of a subscriptrange. (In practice, it is recommended that PL/I programmers' enable such conditions as subscriptrange.) Although this error is easy to spot, the behavior of the program is typical of other, harder to spot errors.

```
! print blowup.pl1

                    blowup.pl1          04/17/80  1332.0 mst Thu

  blowup: procedure;

          dcl     j                   fixed binary;
          dcl     a (10)              fixed binary;
          dcl     sum                 fixed binary;

          a (*) = 1;
          do j = -1 to -100000 by -1;
               sum = a (j);
          end;
     end blowup;


  r 13:32 0.110 20

! pl1 blowup -table
  PL/I
  r 13:32 0.675 174
```

The program is compiled with the -table control argument. This action causes a symbol table to be created, and stored with the program in the executable object segment. The information it contains can be used by the Multics debugging aids. A symbol table should always be created while debugging, so that errors may be found more easily.

```
! blowup

  Error:  out_of_bounds at >udd>ProjA>MacSissle>blowup|24 (line 9)
   referencing stack_4|777777 (in process dir)
   Attempt to access beyond end of segment.
  r 13:32 0.228 32 level 2
```

The program is invoked by typing its name. It takes an 'out_of_bounds' fault, because the subscript used in the reference to array "a" is invalid. The program does not use PL/I subscriptrange checking, so it attempts to calculate the address of the (nonexistent) element of "a" referenced. The resulting address does not exist, so the fault occurs.

This message shows the name of the error condition, the pathname of the program, the octal location in the object segment where the error occurred, the line number, and an additional message about the error. If blowup was a FORTRAN program, the pathname would look like this: >udd>ProjA>MacSissle>blowup$main_, blowup being the name of the segment and main_ the name of the program entry point. This is because every FORTRAN program has a "main" program entry point and Multics uses this as part of its name. If the program had not included a symbol table, the line number would not have been part of the message.

```
! probe
  Condition out_of_bounds raised at line 9 of blowup (level 7).
```

MacSissle invokes the probe command. Probe looks for the program which caused the trouble, and prints a message about the most recent error found in MacSissle's process. The word "level" here refers not to command processor level, but to the number of programs saved on the stack. The error occurred in blowup, which was the seventh program on the stack.

```
! stack

  13          read_list¦13400
  12          command_processor_¦10301
  11          abbrev_¦7507
  10          release_stack¦7355
   9          unclaimed_signal¦24512
   8          wall¦4410
   7          blowup (line 9)                    out_of_bounds
   6          read_list¦13400
   5          command_processor_¦10301
   4          abbrev_¦7507
   3          listen_¦7355
   2          process_overseer_¦35503
   1          user_init_admin_¦40100
```

The stack is displayed by the "stack" request. This request shows every program on the stack, in the order invoked. There will always be unfamiliar programs on your stack. You can just ignore them--they are for handling errors, processing command, etc. The numbers on the left show the order of activation. The entry for blowup shows the source line number corresponding to the last location executed, and the name of the error that occurred. The line number can be determined because blowup was compiled with a symbol table. The other programs have no symbol table, so the display shows the octal offset of the last instruction executed.

```
!  source
   sum = a (j);
```

Using the "source" request, the source statement for line 9 is displayed. This is the line that was being executed when the error occurred. More precisely, the error occurred executing the object code corresponding to this source line.

```
! value j
  j = -2689
! symbol a
  fixed bin (17) automatic  dimension (10)
  Declared in blowup
```

The value of the variable "j" is displayed with the "value" request. This request takes as its argument the name of a variable, and prints the value of the variable. (Note that a program must be suspended for you to look at its automatic variables.) Next, the "symbol" request is used, to show the attributes of "a."

```
! position 8

         do j = -1 to -100000 by -1;
```

The "position" request is used to examine different lines of the program, in this case the line before the one that caused the hang. This request can also be used to examine different programs on the stack. For example, to look at the abbrev program on level 4, MacSissle could type "position level 4". However, she would most likely get the answer "probe (position): Cannot get statement map for this procedure," which means that the program was not compiled with the -table option. (Most system commands have -table omitted, to save space.)

```
! quit
  r 13:33 1.080 129 level 2
```

The last probe request used is "quit," which exits probe, and returns to command level. MacSissle is still at command level two, and the program is still intact. The next command typed is the release command, which discards the saved frames, returning to level one.

```
! release
  r 13:33 0.057 16
```

Unlike interactive programs like read_mail, probe doesn't prompt you for requests. If you're not sure whether probe is listening, type a dot, and probe will respond with "probe 5.2" (or whatever the version number is) if it is there.

Probe has many more features than there is room to present here. It should still be useful to you even if you don't use the other features, but to learn about them you can use the "list_requests" request, which tells you the name of every probe request, and the "help" request, which tells you about probe requests and also about probe itself. For example, you can type "help value" to find out about the "value" request, or "help help" to find out about "help".

Another debugging tool which you may find useful is the trace command, which allows you to monitor all calls to a specified set of external procedures. Full descriptions of the probe and trace commands are available in the MPM Commands.

SECTION 6

A PERFORMANCE MEASUREMENT TOOL

After a program is written and debugged, it is often desirable to increase its efficiency. Multics provides performance measurement tools which identify the most expensive and most frequently executed programs in a given collection. Within these crucial programs, the most costly lines are found by using the profile facility.

To use the profile facility, the first thing you have to do is compile your program with the -profile control argument. This control argument causes the compiler to generate special code for each statement, recording the cost of execution on a statement-by-statement basis. Then, after executing your program many times, you can use the profile command to look at its performance statistics.

The example that follows shows the use of profile with a very small sample program to be used as a subroutine:

```
prime_: procedure (trial_prime) returns (bit (1) aligned);
          declare trial_prime         fixed binary (35) parameter;
          declare trial_factor        fixed binary,
                  last_factor         fixed binary;
          declare (mod, sqrt)         builtin;
          last_factor = sqrt (trial_prime);
          do trial_factor = 2 to last_factor;
               if mod (trial_prime, trial_factor) = 0
               then return ("0"b);
          end;
          return ("1"b);
     end prime_;
```

This subroutine cannot be called directly from command level, since only programs whose arguments are nonvarying character strings may be called directly. It is to be used with other programs. To test it, a simple command is written which accepts one argument, converts it to binary, and calls the prime_ subroutine. The testing command is called test_prime. It is not shown here.

```
! pl1 prime_ -profile
  PL/I
  r 17:44 0.699 140

! test_prime 3
   3 is a prime.
  r 17:44 .110 23
```

First, the prime_ subroutine is compiled using the -profile control argument. Next, the test_prime command is invoked with the argument "3". Test_prime converts the 3 to binary, and calls the prime_ subroutine with it.

```
! discard_output "test_prime ([index_set 500])"
  r 17:45 5.103 54
```

To evaluate the performance of the subroutine, several hundred calls to it should be made, over a wide range of values. The next command line invokes test_prime 500 times, with values from 1 to 500. The index_set active function returns the numbers from 1 to 500, and the parentheses invoke test_prime once for each value.

The output from the program is not interesting, so the discard_output (dco) command is used. This command causes output from the program to be discarded, instead of printed on the terminal.

```
! profile prime_

  Program: prime_
   LINE STMT   COUNT       COST STARS   OPERATORS

     6          1000      34000 ****    fx1_to_f12, dsqrt, f12_to_fx1
     7          1000       3000
     7          4418      13254 ***
     8          4218      59052 ****    mod_fx1
     9           800       8800 **      return
    10          3418       6836 **
    11           200       2600         return
  -------
  Totals:      15054     127542
  r 17:46 0.368 51
```

While the program was run, performance statistics were saved. Now the profile command is used to display those statistics. For each line, it displays the total times executed, an estimate of the cost, and the PL/I operators used.

Note that some statements (those in the loop) were executed more than others. The COST for a statement is the product of the number of instructions for the statement and the number of times the statement was executed. This cost does not take into account the fact that some instructions are faster than others, or the time spent waiting for missing pages (page faults). The STARS column gives a rough indication of the relative cost of each statement.

The names of the PL/I operators used are also given. The operator fx1_to_f12 is used to convert the fixed point number to float, so that its root may be taken. The dsqrt operator takes the square root. Finally, the operator f12_to_fx1 converts the result back to integer. The PL/I mod builtin is implemented by the mod_fx1 operator. These operators are the most expensive things in the program. Occasionally a program can be rewritten to not require expensive operators.

```
! profile prime_ -sort cost -first 5

  Program: prime_
   LINE STMT    COUNT        COST STARS    OPERATORS
      8          4218       59052 ****     mod_fx1
      6          1000       34000 ****     fx1_to_fl2, dsqrt, fl2_to_fx1
      7          4418       13254 ***
      9           800        8800 **       return
     10          3418        6836 **

  -------
  Totals:       15054      127542
  r 17:46 0.205 49
```

When profiling large programs, it is usually desirable to look only at the most expensive lines, since they are the only ones of interest. The profile command can be instructed to sort the lines by cost, and display the five most costly lines in order.

The profile command can also be instructed to produce a source language type of listing with performance statistics adjacent to each source line. Figure 6-1 shows MacSissle using the profile command with the -list control argument to produce such a listing for the compute_sum program. Note that when -list is used, the profile command produces a segment with the same name as the program, but with a suffix of "pfl". (Note also that MacSissle has again set her ready message to read "Karen is here".)

More detailed records of execution are available if you compile your program with the -long_profile control argument. When this is done, the program samples the Multics clock before every instruction, so the total time per statement is available to the profile command. The performance data from a program compiled with -long_profile is displayed with the profile command. For further information, see the MPM Commands description of profile.

```
! pl1 compute_sum -profile
  PL/I
  Karen is here

! compute_sum
  Karen is here

! profile compute_sum -list
  Karen is here

! print compute_sum.pfl

                compute_sum.pfl      05/01/81  1126.6 edt Fri


Profile listing of >udd>ProjA>MacSissle>compute_sum.pl1
Date: 05/01/81  1124.7 edt Fri
Total count: 7   Total cost: 197

  COUNT      COST STARS LINE SOURCE
                           1 compute_sum: proc options (main);
                           2
                           3 /* this program computes the sum of three 1 to 6 digit numbers read from an
                           4    input file, then writes the answer to an output file  */
                           5
      1        20 ***      6 declare
                           7          in_file stream file,                                /* the input file */
                           8          out_file stream file,                               /* the output file */
                           9          first_no fixed binary (20),                         /* the first number */
                          10          second_no fixed binary (20),                        /* the second number *
                          11          third_no fixed binary (20),                         /* the third number */
                          12          the_sum fixed binary (24);                          /* the answer */
                          13
                          14 /* open the files */
                          15
      1        35 ***     16           open file (in_file) input,
                          17                file (out_file) output;
                          18
                          19 /* get the three numbers from the input file */
                          20
      1        59 ****    21           get file (in_file) list (first_no, second_no, third_no);
                          22
                          23 /* add them up */
                          24
      1         4         25           the_sum = first_no + second_no + third_no;
                          26
                          27 /* put the answer in the output file */
                          28
      1        43 ****    29           put file (out_file) list (the_sum);
                          30
                          31 /* close the files */
                          32
      1        26 ***     33           close file (in_file),
                          34                file (out_file);
                          35
      1        10 **      36      end compute_sum;
                          37
```

Figure 6-1. Use of profile Command With -list Control Argument

SECTION 7

ABSENTEE FACILITY

A common programming pattern is to develop a program online, using debugging tools and trying a variety of test cases interactively to check on a program's correctness. After the program is working, you may wish to do a large "production" run. Since the production run may produce a large amount of output or take a long time, you may not wish to wait at your terminal for the results. Production runs on Multics are best done using *absentee jobs*, which are somewhat analogous to batch jobs on other systems.

An absentee job runs in an environment similar to that of an interactive user. In other words, an absentee job uses Multics in much the same way that a person does. It logs in to your home directory, and runs your start_up.ec, if any. This must be kept in mind, both when writing a start_up.ec and when submitting an absentee job. If you forget that your absentee job will run your start_up.ec, you may discover that it has stolen your messages or tried to read your mail. If you assume that your absentee job will log in to the directory from which you submitted it, you may discover that it has run the wrong version of your program.

A big difference between an absentee job and an interactive user is that an absentee job is not associated with a terminal. Its input comes from a file, and its output goes to a file. (In an absentee process, the I/O switches are attached to the input and output segments, instead of the terminal.)

An absentee input file, or *control file*, is a segment with the suffix "absin". At its simplest, it is just a collection of commands to be executed. The language used in an absentee job is the same as that used in exec_coms. It is a superset of the command language. You must anticipate any responses or commands you must give ahead of time, and put all of this data into your control file.

An absentee job is submitted by supplying the name of the absin file to the enter_abs_request (ear) command. The absin file is *not* copied. It stays absentee job. You must not, for example, edit a file it is using, or recompile a program it is running.

The absentee job is placed in a queue and run as "background" to the normal interactive work of the system. This technique allows the system to utilize its resources most effectively, by keeping a queue of jobs that can always be run, and delayed for serving interactive users. For these reasons, the charging rate for absentee jobs is normally substantially lower than for interactive work.

Output from an absentee job goes into a file whose name is the same as the absin segment, but with the suffix "absout" instead of "absin". When the job completes, you may print this absout segment. Figure 7-1 illustrates the differences between interactive usage and absentee usage.

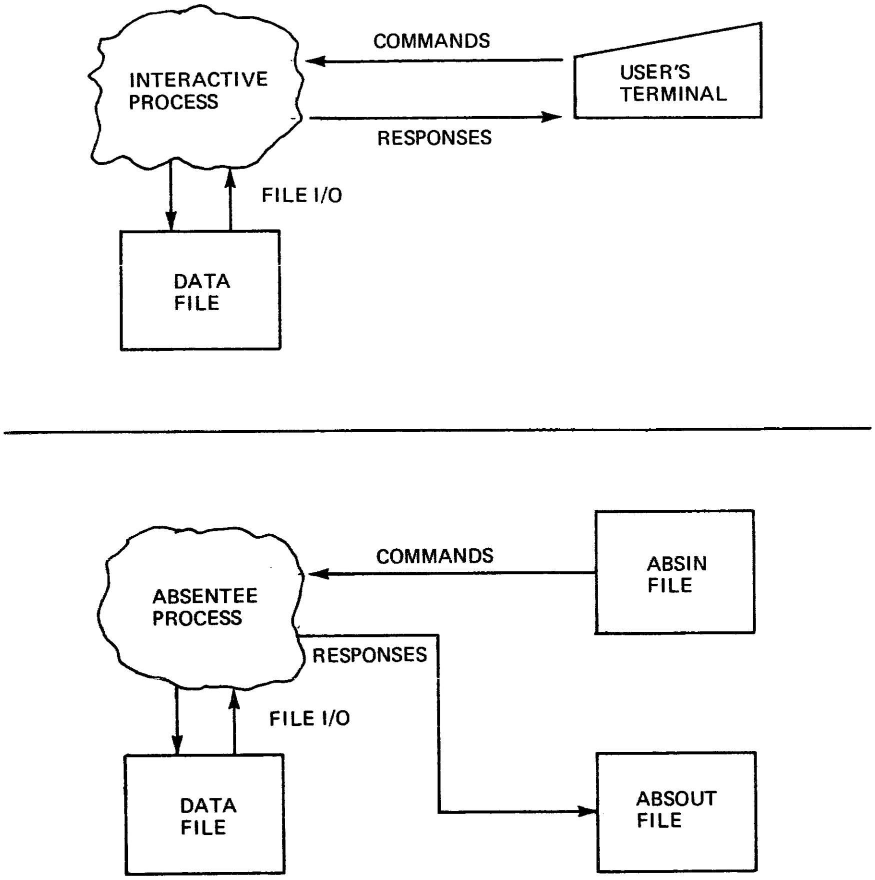


Figure 7-1. Interactive vs Absentee Usage

Suppose MacSissle has written a FORTRAN program which figures square roots. The program resides in her directory of FORTRAN programs, and she would like to compile and run it absentee. The first thing she does is create a segment called compile_run.absin.

```
cwd >udd>ProjA>MacSissle>fort_progs
fortran square_root.fortran -list
dprint -dl square_root.list
square_root
dprint file10
logout
```

Then she types this command line:

```
! enter_abs_request compile_run
```

Her absentee job is submitted. When it runs, it changes to the proper working directory, compiles the program and produces a listing segment, prints the listing segment on the line printer and deletes it, runs the program, prints the output file "file10" on the line printer, and finally, logs out.

To run this same absentee job via remote job entry, MacSissle would put the statements shown above on cards instead of in a segment. Then she would surround her cards with control cards and put the deck in a card reader. Her absentee job would be executed automatically.

The format of a card deck for remote job entry is shown below:

```
++RJE  DECK_NAME  PERSON_ID  PROJECT_ID
++PASSWORD  PASSWORD
++AIM  ACCESS CLASS OF ABSENTEE PROCESS
++RJECONTROL  CONTROL ARGS TO THE EAR COMMAND
++RJEARGS  ARGUMENTS FOR THE ABSENTEE PROCESS
++EPILOGUE  COMMAND
++FORMAT  PUNCH_FORMAT  MODES
++INPUT
    .
    .
    .
(user absentee file)
```

The three cards required as a minimum are the first, which is an identifier card, the second, which is a password card, and the last, which signals the end of control input

For another example, suppose MacSissle wants to use the prime_ subroutine discussed in Section 6 to check the prime-ness of the first five integers, and she wants to use the absentee facility to do it. Remember that prime_ is called by test_prime, and that the index_set active function can be used to return a set of numbers.

```
! qedx
! a
! test_prime ([index_set 5])
! \f
! w test5.absin
! q
  r 16:40 0.218 39
```

MacSissle uses the Qedx editor to create her absin file.

```
! enter_abs_request test5 -notify
  ID 210805.1; 5 already requested
  r 16:41 0.450 63
```

Multics confirms her submission, giving the request id and the number of previously submitted jobs in the absentee queue. Often, many of these jobs may be "deferred", which is to say, they will not be run until a later time. Thus, "5 already requested" doesn't necessarily mean that five jobs must be run before MacSissle's job will run.

```
From Initializer.SysDaemon (absentee) 04/21/80 1641.4 mst Mon:
Absentee job >udd>ProjA>MacSissle>test5.absin 210805.1
logged in.
```

MacSissle used the -notify control argument on her ear command, so the system sends her a message when her job logs in.

```
! who -absentee

  Absentee users 3/9
  JQUser.ProjB*
  TSmith.ProjA*
  MacSissle.ProjA*
  r 16:42 0.272 22
```

MacSissle uses the who command to print a list of all absentee jobs. It shows that there are three already running, and that a total of nine can run at one time. Absentee users are identified by the asterisk after their project.

```
From Initializer.SysDaemon (absentee) 04/21/80 1643.1 mst Mon:
Absentee job >udd>ProjA>MacSissle>test5.absin  210805.1
logged out.
```

The system also sends her a message when her job logs out.

```
! print test5.absout

                  test5.absout 04/21/80 1643.6 mst Mon

  Absentee user MacSissle ProjA logged in:  04/21/80 1641.4 mst Mon
  r 16:41 2.364 55

  test_prime ([index_set 5])
  1 is a prime
  2 is a prime
  3 is a prime
  4 is not a prime
  5 is a prime
  r 16:42 0.198 20

  abs_io_:  Input Stream exhausted.

  Absentee user MacSissle ProjA logged out 04/21/80 1643.1 mst Mon
  CPU usage 3 sec, memory usage 1.0 units
```

MacSissle's job is done, so she prints the absout segment.

With more advanced use of the absentee facility, you can also supply arguments to be substituted inside the absentee control segment, make absentee job steps conditional, delay absentee work until a chosen time, and develop a periodic absentee job which is run, say, once every two days.

The next example shows how absentee jobs can accept arguments.

```
! print prime.absin

                  prime.absin         04/21/80  1655.7 mst Mon

  test_prime ([index_set &1])

  r 16:55 .110 19
```

This absin segment accepts one argument. The character string "&1" is replaced by the argument wherever it occurs. MacSissle tests it by invoking it as an exec_com. In order to use the absin segment as an exec_com, it must have a name with the suffix "ec" added to it.

```
! add_name prime.absin prime.ec
  r 16:56 0.100 5

! exec_com prime.ec 2
  test_prime ([index_set 2])
    1 is  a prime
    2 is  a prime

  r 17:00 0.210 30
```

MacSissle invokes the exec_com with the argument 2. As it runs, it prints the commands in the file. The argument mechanism seems to work, so she submits an absentee job.

```
! enter_abs_request prime.absin -arguments 100
 ID: 221023.4; 6 already requested.
 r 17:05 0.273 50
```

Here, the argument 100 is passed to the absentee job. MacSissle goes about other business while the request runs.

A common problem for many users is an absentee job that blows up unexpectedly because it is asked an unanticipated question, and the user has not provided an appropriate answer. For example, a job may be asked, "Do you wish to quit?" It can try to use its next command for an answer, but it will be told to "Please answer yes or no." At this point, the job will probably die.

Suppose MacSissle has set up a daily absentee job that reads her mail. Her absin segment, called mail.absin, looks like this:

```
enter_abs_request mail -time  "07:00"  -notify
read_mail
print all
quit
dprint -delete mail.absout
```

MacSissle types the command line

```
! enter_abs_request mail -time  "07:00"  -notify
```

once. Her absentee job submits a request for the next absentee job, then reads her mail. Once in the read_mail request loop, it asks that all of her mail be printed, then quits out of the loop. Finally, it dprints her absout segment.

This job seems like it should work fine. But what will happen if MacSissle doesn't have any mail? The request to read her mail will return the answer, "You have no mail". Then the request to print all of her mail will return the answer, "Segment all not found". The request to quit will return a similar answer. So, the job may not die in this case, but it will give MacSissle some unexpected results. To avoid this problem, MacSissle can change her absin segment to look like this:

```
enter_abs_request mail -time  "07:00"  -notify
read_mail -request "print all; quit"
dprint -delete mail.absout
```

Now, if she has no mail, she'll just get the answer, "You have no mail", which is what she wants.

For further information on absentee jobs, see the MPM Commands manual descriptions of the enter_abs_request and exec_com commands. See also the descriptions of the pl1_abs, cobol_abs, and fortran_abs commands, which invoke language compilers in absentee jobs.

SECTION 8

REFERENCE TO COMMANDS BY FUNCTION

All of the Multics commands described in the MPM Commands are arranged here according to function and are briefly described. The Multics command repertoire is divided into the following 17 groups:

Access to the System
Storage System, Creating and Editing Segments
Storage System, Segment Manipulation
Storage System, Directory Manipulation
Storage System, Access Control
Storage System, Address Space Control
Formatted Output Facilities
Language Translators, Compilers, and Interpreters
Object Segment Manipulation
Debugging and Performance Monitoring Facilities
Input/Output System Control
Command Level Environment
Communication Among Users
Communication with the System
Accounting
Control of Absentee Computations
Miscellaneous Tools

Many commands can perform more than one function, so they are listed in more than one group.

Detailed descriptions of these commands, arranged alphabetically rather than functionally, are given in the MPM Commands. In addition, many of the commands have online descriptions, which you may obtain by invoking the help command.

ACCESS TO THE SYSTEM

dial	connects an additional terminal to an existing process
echo	sets terminal into echoplex mode before login
enter enterp	connects an anonymous user to the system (used at dialup only)
hangup	terminates communication between terminal and Multics
hello	repeats greeting message printed when terminal is first connected
login	connects registered user to the system (used at dialup only)
logout	disconnects user from the system
modes	sets terminal modes before login
slave	changes service type of channel from login to slave for duration of connection

terminal_type	sets terminal type before login
MAP	tells system user is attempting to gain access from terminal whose keyboard generates only uppercase characters
029 and 963	tells system whether user is attempting to gain access from device similar to EBCDIC or Correspondence code IBM Model 2741

STORAGE SYSTEM, CREATING AND EDITING SEGMENTS

adjust_bit_count	sets bit count of a segment to last nonzero word or character
canonicalize	ensures that contents of a segment are in canonical form
compare_ascii	compares ASCII segments, reporting differences
compose	composes formatted documents for production on various devices, including terminals and line printers
edm	allows inexpensive, easy editing of ASCII segments
emacs	enters the Emacs text editor, which has a large repertoire of requests for editing and formatting text and programs
indent	indents a PL/I source segment to make it more readable
merge_ascii	merges two or more related ASCII text segments
qedx	allows sophisticated editing, including macro capabilities
set_bit_count	sets the bit count of a segment to a specified value
ted	used to create and edit ASCII segments; can do many kinds of text processing

STORAGE SYSTEM, SEGMENT MANIPULATION

adjust_bit_count	sets bit count of a segment to last nonzero word or character
archive	packs segments together to save physical storage
archive_table	returns the names of specified archive components in specified archive segment
compare	compares segments word by word, reporting differences
compare_ascii	compares ASCII segments, reporting differences
copy	copies a segment or multisegment file and its storage system attributes
copy_file	copies records from an input file to an output file
create	creates an empty segment
damaged_sw_off	resets damaged switch off for segments
damaged_sw_on	sets damaged switch on for segments
delete	deletes a segment or multisegment file and questions user if it is protected
link	creates a storage system link to another segment, directory, link, or multisegment file
merge_ascii	merges two or more related ASCII text segments

move	moves segment or multisegment file and its storage system attributes to another directory
set_bit_count	sets the bit count of a segment to a specified value
sort_seg	sorts ASCII segments according to ASCII collating sequence
tape_archive	performs a variety of operations to create and maintain a set of files on magnetic tape
truncate	truncates a segment to a specified length
unlink	removes a storage system link
vfile_adjust	adjusts structured and unstructured files
volume_dump_switch_off	turns off the specified volume dump switch of a segment
volume_dump_switch_on	turns on the specified volume dump switch of a segment

STORAGE SYSTEM, DIRECTORY MANIPULATION

add_name	adds a name to a segment, directory, link, or multisegment file
cancel_retrieval_request	deletes request for a volume retrieval that is no longer needed
copy_dir	copies a directory and its subtree to another point in the hierarchy
create_dir	creates a directory
delete_dir	destroys a directory and its contents after questioning user
delete_name	removes a name from a segment, directory, link, or multisegment file
enter_retrieval_request	queues volume retrieval requests for specific segments, directories, multisegment files, and subtrees
link	creates a storage system link to another segment, directory, link, or multisegment file
list_retrieval_requests	lists retrieval requests in the retrieval daemon queues
list	prints directory contents
move_dir	moves a directory and its subtree to another point in the hierarchy
rename	renames a segment, directory, link, or multisegment file
safety_sw_off	turns safety switch off for a segment, directory, or multisegment file
safety_sw_on	turns safety switch on for a segment, directory, or multisegment file
status	prints all the attributes of an entry in a directory
tape_archive	performs a variety of operations to create and maintain a set of files on magnetic tape
unlink	removes a storage system link
vfile_status	prints the apparent type and length of storage system files
volume_dump_switch_off	turns off the specified volume dump switch of a segment
volume_dump_switch_on	turns on the specified volume dump switch of a segment

STORAGE SYSTEM, ACCESS CONTROL

check_iacl	compares segment ACLs with the initial ACL
copy_acl	copies ACL from segment or directory
copy_iacl_dir	copies a directory initial ACL
copy_iacl_seg	copies a segment initial ACL
delete_acl	removes an ACL entry
delete_iacl_dir	removes an initial ACL for new directories
delete_iacl_seg	removes an initial ACL for new segments
list_accessible	lists segments and directories with a given access condition
list_acl	prints an ACL entry
list_not_accessible	lists segments and directories to which user does not have a given access condition
list_iacl_dir	prints an initial ACL for new directories
list_iacl_seg	prints an initial ACL for new segments
print_auth_names	prints names of sensitivity levels and access categories for an installation
set_acl	adds (or changes) an ACL entry
set_iacl_dir	adds (or changes) an initial ACL for new directories
set_iacl_seg	adds (or changes) an initial ACL for new segments

STORAGE SYSTEM, ADDRESS SPACE CONTROL

add_search_paths	adds one or more search paths to the specified search list
add_search_rules	allows users to change (insert) search rules dynamically
attach_lv	calls the resource control package to attach a logical volume
change_default_wdir	sets the default working directory
change_wdir	changes the working directory
delete_search_paths	allows user to delete one or more search paths from specified search list
delete_search_rules	allows users to delete current search rules
detach_lv	detaches logical volumes attached by the resource control package
get_system_search_rules	prints definitions of site-defined search rule keywords
initiate	adds a segment to the address space of a process
list_ref_names	prints all names by which a segment is known to a process
new_proc	creates a new process with a new address space
print_default_wdir	prints name of default working directory
print_proc_auth	prints access authorization of the current process and current system privileges
print_search_paths	prints the search paths in the specified search list
print_search_rules	prints names of directories searched for segments referenced dynamically
print_wdir	prints name of current working directory
set_search_paths	allows user to replace search paths contained in specified search list
set_search_rules	allows users to modify search rules
terminate	removes a segment from the address space of a process
walk_subtree	executes a command line in all directories below a specified directory
where	uses current search rules to locate and print pathname of a segment

where_search_paths	returns absolute pathname(s) of entryname when search list name and entryname are specified

FORMATTED OUTPUT FACILITIES

cancel_daemon_request	cancels a previously submitted daemon request
compose	composes formatted documents for production on various devices, including terminals and line printers
dprint	queues a segment or multisegment file for printing on the high-speed printer
dpunch	queues a segment or multisegment file for card punching
dump_segment	prints segment contents in octal, ASCII, or EBCDIC
list_daemon_requests	prints list of print and punch requests currently queued
move_daemon_request	moves a request from one I/O daemon queue to another
overlay	reads several ASCII segments and writes on user_output I/O switch output that is the result of superimposing print positions from each segment
print	prints an ASCII segment

LANGUAGE TRANSLATORS, COMPILERS, AND INTERPRETERS

apl	invokes the APL interpreter
basic	compiles BASIC programs
bind	packs two or more object segments into a single executable segment
cancel_cobol_program	cancels one or more programs in the current COBOL run unit
cobol	compiles COBOL programs
cobol_abs	submits an absentee request to perform COBOL compilations
create_data_segment	translates a create_data_segment source program into an object segment
display_cobol_run_unit	displays the current state of a COBOL run unit
expand_cobol_source	translates COBOL source program containing COPY and REPLACE statements to equivalent source program not containing these statements
fast	allows user to enter FAST subsystem
format_cobol_source	converts free-form COBOL source to fixed-format COBOL source
fortran	invokes the site's "standard" FORTRAN compiler
fortran_abs	invokes the site's "standard" FORTRAN compiler in an absentee job
indent	indents a PL/I source segment to make it more readable
lisp	enters interactive Lisp subsystem, where Lisp forms can be typed at user's terminal and evaluated
pl1	compiles PL/I programs
pl1_abs	invokes the PL/I compiler in an absentee job
profile	prints information about execution of individual statements within program
run_cobol	executes a COBOL run unit in a main program

set_cc	sets carriage control transformation for FORTRAN files
set_fortran_common	initializes common storage for a FORTRAN run
stop_cobol_run	terminates the current COBOL run unit

OBJECT SEGMENT MANIPULATION

archive	packs segments together to save physical storage
archive_table	returns the names of specified archive components in specified archive segment
bind	packs two or more object segments into a single executable segment
date_compiled	prints date and time compiled and compiler identifier for object segments

DEBUGGING AND PERFORMANCE MONITORING FACILITIES

attach_audit	sets up specified I/O switch to be audited by the audit_ I/O module
change_error_mode	adjusts length and content of system condition messages
cumulative_page_trace	accumulates page trace data
debug	permits symbolic source language debugging
display_audit_file	displays the file produced by the audit_ I/O module
display_pl1io_error	displays diagnostic information about PL/I I/O errors
dump_segment	prints segment contents in octal, ASCII, or EBCDIC
general_ready	allows user to format ready messages
page_trace	prints a history of system events within calling process
probe	permits program debugging online
profile	prints information about execution of individual statements within program
progress	prints information about the progress of a command as it is being executed
ready	prints the ready message: a summary of CPU time, paging activity, and memory usage
ready_off	suppresses the printing of the ready message
ready_on	restores the printing of the ready message
reprint_error	reprints an earlier system condition message
resolve_linkage_error	satisfies linkage fault after a process encounters a linkage error
trace	permits the user to monitor all calls to a specified set of external procedures
trace_stack	prints stack history

INPUT/OUTPUT SYSTEM CONTROL

assign_resource	assigns peripheral equipment to user
cancel_resource	cancels reservations made with the reserve command
cancel_daemon_request	cancels a previously submitted print or punch request
close_file	closes open PL/I and FORTRAN files
copy_cards	copies card decks read by I/O Daemon
copy_file	copies records from an input file to an output file

discard_output	executes a command line while temporarily suppressing output on specified I/O switches
display_pl1io_error	displays diagnostic information about PL/I I/O errors
dprint	queues a segment or multisegment file for printing on the high-speed line printer
dpunch	queues a segment or multisegment file for card punching
file_output	directs terminal output to a file
io_call	allows direct calls to input/output system entries
line_length	allows users to control maximum length of output lines
list_daemon_requests	prints list of print and punch requests currently queued
list_resource_types	prints a list of all resource types described in a resource type description table (RTDT)
list_resources	lists peripheral equipment assigned to user
print	prints an ASCII segment
print_attach_table	prints list of current input/output system switch attachments
print_request_types	prints available I/O Daemon request types
reserve_resource	reserves resource(s) for use by the calling process
tape_archive	performs a variety of operations to create and maintain a set of files on magnetic tape
unassign_resource	unassigns peripheral equipment assigned to user
vfile_adjust	adjusts structured and unstructured files
vfile_status	prints the apparent type and length of storage system files

COMMAND LEVEL ENVIRONMENT

abbrev	allows user-specified abbreviations for command lines or parts of command lines
add_search_paths	adds one or more search paths to the specified search list
add_search_rules	allows users to change (insert) search rules dynamically
answer	answers questions normally asked of the user
attach_audit	sets up specified I/O switch to be audited by the audit_ I/O module
change_default_wdir	sets the default working directory
change_error_mode	adjusts length and content of system condition messages
change_wdir	changes the working directory
delete_search paths	allows user to delete one or more search paths from specified search list
delete_search_rules	allows users to delete current search rules
detach_audit	removes audit_ from specified switch
display_audit_file	displays the file produced by the audit_ I/O module
do	expands a command line with argument substitution
exec_com	allows a segment to be treated as a list of executable commands
fast	allows user to enter FAST subsystem
file_output	directs terminal output to a file

gcos	invokes GCOS environment simulator to run single GCOS job in user's process
general_ready	allows user to format ready messages
get_system_search_rules	prints definitions of site-defined search rule keywords
if	conditionally executes a command line
line_length	allows users to control maximum length of output lines
memo	allows users to set reminders for later printout
new_proc	creates a new process with a new address space
on	establishes handler for specified set of conditions, executes imbedded command line with handler in effect, reverts handler
print_default_wdir	prints name of default working directory
print_search_paths	prints the search paths in the specified search list
print_search_rules	prints names of directories searched for segments referenced dynamically
print_wdir	prints name of current working directory
program_interrupt	provides for command reentry following a quit or an unexpected signal
ready	prints the ready message: a summary of CPU time, paging activity, and memory usage
ready_off	suppresses the printing of the ready message
ready_on	restores the printing of the ready message
release	discards process history retained by a quit or an unexpected signal interruption
repeat_query	repeats the last query by the command_query_ subroutine
reprint_error	reprints an earlier system condition message
resolve_linkage_error	satisfies linkage fault after a process encounters a linkage error
run	provides user with temporary, somewhat isolated, environment for execution of programs
set_search_paths	allows user to replace search paths contained in specified search list
set_search_rules	allows users to modify search rules
set_tty	prints and sets modes associated with user's terminal
start	continues process at point of a quit or an unexpected signal interruption
stop_run	effects abnormal termination of run-unit created by run command
where_search_paths	returns absolute pathname(s) of entryname when search list name and entryname are specified

COMMUNICATION AMONG USERS

accept_messages	initializes the process to accept messages immediately
defer_messages	inhibits the normal printing of received messages
delete_message	deletes messages saved in user's mailbox
immediate_messages	restores immediate printing of messages
print_mail	prints all messages in a mailbox
print_messages	prints any pending messages
read_mail	provides a facility for examining and manipulating messages

send_mail	transmits a message to one or more recipients
send_message	sends message to specified user
send_message_acknowledge	sends message and acknowledges its receipt
send_message_express	sends message only if user will receive it immediately
send_message_silent	sends message but does not acknowledge its receipt
who	prints list of users and absentee jobs currently logged in

COMMUNICATION WITH THE SYSTEM

cancel_retrieval_request	deletes request for a volume retrieval that is no longer needed
check_info_segs	checks information (and other) segments for changes
damaged_sw_off	resets damaged switch off for segments
damaged_sw_on	sets damaged switch on for segments
help	prints special information segments
how_many_users	prints the number of logged-in users
enter_retrieval_request	queues volume retrieval requests for specific segments, directories, multisegment files, and subtrees
list_help	displays names of all info segments pertaining to a given topic
list_retrieval_requests	lists retrieval requests in the retrieval daemon queues
move_abs_request	moves a request from one absentee queue to another
no_save_on_disconnect	disables process preservation across hangups in user's process
print_motd	prints the portion of the message of the day that changed since last printed
save_on_disconnect	reverses effect of no_save_on_disconnect command
volume_dump_switch_off	turns off the specified volume dump switch of a segment
volume_dump_switch_on	turns on the specified volume dump switch of a segment
who	prints list of users and absentee jobs currently logged in

ACCOUNTING

get_quota	prints secondary storage quota and usage
move_quota	moves secondary storage quota to another directory
resource_usage	prints resource consumption for the month

CONTROL OF ABSENTEE COMPUTATIONS

cancel_abs_request	cancels a previously submitted absentee job request
cobol_abs	submits an absentee request to perform COBOL compilations
enter_abs_request	adds a request to the absentee job queue
fortran_abs	invokes the site's "standard" FORTRAN compiler in an absentee job
how_many_users	prints the number of logged-in users
list_abs_requests	prints list of absentee job requests currently queued
move_abs_request	moves a request from one absentee queue to another
pl1_abs	invokes the PL/I compiler in an absentee job
runoff_abs	invokes the runoff command in an absentee job
who	prints list of users and absentee jobs currently logged in

MISCELLANEOUS TOOLS

calc	performs specified calculations
calendar	prints a calendar page for one month
canonicalize	ensures that contents of a segment are in canonical form
decode	deciphers segment, given proper coding key
encode	enciphers segment, given a coding key
manage_volume_pool	allows users to regulate use of a predefined set of volumes
memo	allows users to set reminders for later printout
merge	provides generalized file merging capability
progress	prints information about the progress of a command as it is being executed
sort	provides generalized file sorting capability

APPENDIX A

USING MULTICS TO BEST ADVANTAGE

You may, if you wish, treat Multics as simply a PL/I, FORTRAN, APL, BASIC, or COBOL machine, and contain your activities to just the features provided in your preferred programming language. On the other hand, much of the richness of the Multics programming environment involves use of system facilities for which there are no available constructs in the usual languages. To use these features, it is generally necessary to call upon library and supervisor subroutines. Unfortunately, a simple description of how to call a subroutine may give little clue as to how it is intended to be used. The purpose of this appendix is to illustrate typical ways in which many of the properties of the Multics programming environment may be utilized.

When you choose a language for your implementation, you should carefully consider the extent to which you will want to go beyond your language and use system facilities of Multics which are missing from your language. As a well-known standard for completeness of that language (e.g., ANSI or IBM). However, in going beyond the standard languages, you will find that Multics supervisor and library routines are designed primarily for use from PL/I programs. This results from the fact that most of these routines are themselves implemented in PL/I. For example, if you plan to write programs which directly call the Multics storage system privacy and protection entries, in FORTRAN or BASIC, you have no convenient way to express such structures. Note that the situation is not hopeless, however. Programs which stay within the original language can be written with no trouble. Also, in many cases, a trivial PL/I interface subroutine can be constructed, which is callable from, say, a FORTRAN program, and goes on to reinterpret arguments and invoke the Multics facility desired. This is made possible by the Multics conventions which ensure that FORTRAN and PL/I programs can communicate. (For more information, see the MPM Subsystems Writers' Guide.) Using such techniques, almost any program a standard call is performed, the argument pointer is set to point at the originally prepared for another system can be moved into the Multics environment.

The examples which follow show that the effect of the mapping together of the main memory and secondary storage environments can range from the negligible (programs can be written as though there was a traditional two-environment system) to a significant simplification of programs which make extensive use of the storage system. Here are seven brief examples of programs which are generally simpler than those encountered in practice, but which illustrate ways in which online storage is accessed in Multics.

1. Internal Automatic Variables. The following program types the word "Hello" on four successive lines of terminal output:

```
a:    procedure;
      declare i fixed binary;
      do i = 1 to 4;
        put list ("Hello");
        put skip;
      end;
      return;
      end a;
```

The variable i is by default of PL/I storage class <u>internal automatic</u>: in Multics it is stored in the stack of the current process and is available by name only to program a and only until a returns to its caller. It is declared binary for clarity: although the default base for the representation of arithmetic data is binary according to the PL/I standard, as well as in Multics PL/I, some other popular implementations have a decimal default. There is no need for decimal arithmetic in this program, and binary arithmetic is faster.

2. Internal Static Variables. The following program, each time it is called, types out the number of times it has been called since its user has logged in:

```
b:    procedure;
      declare j fixed binary internal static intial(0);
      j = j + 1;
      put list (j, "calls to b.");
      put skip;
      return;
      end b;
```

The variable j is of PL/I storage class <u>internal static</u>; in Multics it is stored in b's static section (discussed in Section 2) and is available by name only to program b. Its value is preserved for the life of the process, or until b is terminated (by the terminate command, recompilation, etc.), whichever time is shorter. The "initial" declaration causes the value of j to be initialized at the time this procedure is first used in a process.

3-4. External Static. Suppose you wish to set a value in one program and have it printed by some other program in the same process:

```
c:    procedure;
      declare z fixed binary external static;
      z = 4;
      return;
      end c;


d:    procedure;
      declare z fixed binary external static;
      put list (z);
      put skip;
      return;
      end d;
```

In both programs, the variable z is of PL/I storage class *external static*; in Multics it is stored in a particular segment where *all such* variables are stored, and is available to all procedures in a particular process, until the process is destroyed. External static is analogous to common in FORTRAN, but with the important difference that data items are accessed by name rather than by relative position in a declaration. Program d above could be replaced by the following FORTRAN program:

```
integer n
common /z/ n
print, n
end
```

Multics calls such data items *external variables*. There are commands (for example, list_external_variables) to list, reinitialize, and otherwise deal with all the external variables used by a process. Each variable which is accessed in this form generates a linkage fault the first time it is used. Later references to the variable by the same procedure in that or subsequent calls do not generate the fault.

5. Direct Intersegment References. The following program prints the sum of the 1000 integers stored in the segment w:

```
e:    procedure;
      declare w$(1000) fixed binary external static;
      declare (i, sum) fixed binary;
           sum = 0;
      do i = 1 to hbound (w$,1);
           sum = sum + w$(i);
      end;
      put list (sum);
      put skip;
      return;
      end e;
```

The dollar sign in the PL/I identifier "w$" is recognized as a special symbol by the PL/I compiler, and code for statement 6 is constructed which anticipates dynamic linking to the segment named w. Upon first execution, a linkage fault is triggered, and a search undertaken for a segment named w. If one is found, the link is snapped, and all future references will occur with a single machine instruction. The storage for array "w$" is the segment w.

If no segment named w is found, the dynamic linker will report an error to the user and return to command level. At this point, it is possible to create an appropriate segment named w, and then continue execution of the interrupted program, if such action is appropriate.

6. Reference to Named Offsets. The following procedure calculates the sum of 1000 integers stored in segment x starting at the named offset u:

```
f:   procedure;
     declare x$u(1000) fixed binary external static;
     declare (i, sum) fixed binary;
     sum = 0;
     do i = 1 to 1000;
          sum = sum + x$u(i);
     end;
     put list (sum);
     put skip;
     return;
     end f;
```

The difference between this example and the previous one is that segment x is presumed to have some substructure, with named internal locations (entry points). To initially create a segment with such a substructure, the compilers and assemblers are used, since information must be placed in the segment to indicate where within it the entry points may be found. Unfortunately, the PL/I language permits specification of such structured segments only for procedures, not for data. The create_data_segment_ subroutine can be used in conjunction with the create_data_segment (cds) command to create such data segments from PL/I data structures passed to it as parameters. The create_data_segment command translates a CDS source program into a data segment (actually a standard object segment). A sample CDS source program, x.cds, is shown below:

```
x:   procedure;
     declare 1 x aligned,
               2 u(1000) fixed binary;
     declare create_data_segment_ entry (ptr, fixed binary (35));
     .
     .  (overhead required to utilize create_data_segment_)
     .
     call create_data_segment_ <cds_args>;
     return;
     end x;
```

The ALM assembler can also be used to create a structured data segment, as shown by x.alm below:

```
     name x
     segdef u
u:   bss 1000
     end
```

7. External Reference Starting With a Character String. In many cases, a segment must be accessed whose name has been supplied as a character string. In those cases, a call to the Multics storage system is required in order to map the segment into the virtual memory and to obtain a pointer to it. The following program uses the supervisor entry hcs_$make_ptr to perform a search for a segment of a given name, identical to that undertaken by the linker in the previous examples.

```
g:   procedure(string);
     declare string character(*) parameter;
     declare hcs_$make_ptr entry (pointer, character(*),
         character(*), pointer, fixed binary(35));
     declare null builtin;
     declare p pointer;
     declare ec fixed binary (35);
     declare hcs_$terminate_seg entry (ptr, fixed binary (1),
        fixed binary (35));
     declare com_err_ entry options (variable);
     declare (i, sum) fixed binary;
     declare v(1000) fixed binary based(p);
     call hcs_$make_ptr (null (), string, "", p, ec);
     if p= null then do;
        call com_err_ (ec, "g", "^a", string);
        return;
     end;
     sum = 0;
     do i = 1 to 1000;
        sum = sum + p  v(i);
     end;
     /* The segment should be terminated, since it was
        initiated */
     call hcs_$terminate_seg (p, 0, (0));
     return;
     end g;
```

The PL/I null string value ("") indicates that it is not a named entry point in the segment to which a pointer is wanted, but a pointer to its base. Perhaps the segment does not even have named entry points. The PL/I null pointer value (null()) and the zero passed by value ((0)) in the call to hcs_$make_ptr are relevant to its handling of error conditions and some of the parameters of the search for the segment. See the MPM Subroutines for a full description of the hcs_$make_ptr subroutine.

8. Reference to Segment Via Pathname. The following procedure finds a segment specified by an absolute or relative pathname given as an argument. Most Multics commands accept pathnames and find the segments they are to operate on in this fashion. This procedure also adds all the numbers in the segment, obtaining the number of entries in the array by using the bit count of the segment.

```
h:   procedure(string);
     declare string char(*);
     declare expand_pathname_ entry (char(*), char(*), char(*), fixed
        binary(35));
     declare dn char(168), en char(32), ec fixed binary(35);
     declare com_err_ entry() options(variable);
     declare hcs_$initiate_count entry char(*), char(*), char(*), fixed
        binary(24), fixed binary(2), ptr, fixed binary(35));
     declare null builtin;
     declare bc fixed binary(24);
     declare p ptr;
     declare nwords fixed binary;
     declare i fixed binary;
     declare sum fixed binary (35);
     declare w (nwords) fixed binary(35) based (p);
     declare hcs_$terminate_noname entry (ptr, fixed binary (35));
     declare sysprint file;
     call expand_pathname_ (string,dn,en,ec);
     if ec ^= 0 then do;
err: call com_err_ (ec,"h","^a",string);
        return;
     end;
     call hcs$initiate_count (dn,en,"",bc,0,p,ec);
     if p = null then goto err;
     nwords = divide (bc,36,17,0);
     sum = 0;
     do i = 1 to nwords;
        sum = sum + w(i);
     end;
     call hcs_$terminate_noname (p,(0));
     put list (sum);
     put skip;
end h;
```

The expand_pathname_ procedure is a library subroutine which accepts a relative or absolute pathname and returns the directory name and entryname ready for use by supervisor entries such as hcs_$initiate_count. No search for the segment specified is undertaken in this case. Since the segment was initiated with a null reference name (third argument to hcs_$initiate_count), the procedure is responsible for terminating it as well.

Further improvements to this procedure are possible. It lacks the ability to handle several common error cases; if no argument is supplied, for example, the program will malfunction. Code to handle this possibility should be included, as well as code to handle the possibility of a zero-length input segment, or the possibility of a fixed point overflow.

APPENDIX B

A SIMPLE TEXT EDITOR

The sample program discussed in this appendix is a printing-terminal text editor similar to, but simpler than, Edm. (See Appendix D for a description of Edm.) It is a typical example of an interactive program which makes use of the Multics storage system via the virtual memory. In overview, the editor creates two temporary storage areas, each large enough to hold the entire text segment being edited; copies the segment into one of these areas, so as not to harm the original; and then, as the user supplies successive editing requests, constructs in the other area an edited version of the segment. When the user finishes a pass through the segment, the editor interchanges the roles of the two storage areas for the next editing pass. When the user is done with the editor, the appropriate temporary storage area is then copied back over the original segment. This example is not intended to be a model for designing or implementing text editors, but rather, an illustration of the techniques used in interactive Multics PL/I programs, particularly commands.

For this example, a program listing as produced by the PL/I compiler is used. The program itself is derived from the edm command of Multics, and it exhibits several different styles of coding and commenting, since it has had many different maintainers.

The program listing is preceded by several pages of comments on the program. The comments appear in the same order as the item(s) in the program that they comment on. Where possible, they refer to line numbers in the program listing. Unfortunately, programs do not always invoke features in the best order for understanding, so the following strategy may be useful: as you read each comment, if its implications are clear and you feel you understand it, check it off. If you encounter one which does not fit into your mental image of what is going on, skip it for the moment. Later comments may shed some light on the situation, as will later reference to other Multics documentation. Finally, a hard core of obscure points may remain unexplained, in which case the advice of an experienced Multics programmer is probably needed. Be warned that the range of comments is very wide, from trivial to significant, from simple to sophisticated, and from obvious to extremely subtle.

Finally, some comments provide suggestions for "good programming practice". Such suggestions are usually subjective, and often controversial. Nonetheless, the concept of choosing among various possible implementation methods one which has clarity, is consistent, and minimizes side effects is valuable, so the suggestions are provided as a starting point for the reader who may wish to develop his own style of good programming practice.

You will also notice that some comments appear to be critical of the program style or of interfaces to the Multics supervisor. These comments should be taken in a spirit of illumination of the mechanisms involved. Often they refer to points which could easily be repaired, but which have not been in order to provide a more interesting illustration. Most of the points criticized are minor in impact.

The program listing appears after the commentary.

Line number

fifth unnumbered line

The command "pl1 eds -map -optimize" was typed at the terminal. This line records the fact that the map and optimize options were used. The map line option caused a listing and variable storage map to be produced. A source segment named eds.pl1 was used as input; the compiler constructed output segments named eds.list (containing the listing) and eds (containing the compiled binary program.)

1 No explicit arguments are declared here, even though eds should be called with one argument. Instead, the keyword "options (variable)" appears, which indicates that this program can be called with a variable number of arguments. This is a Multics extension to ANSI PL/I. Since eds is used as a command, it is a good human engineering practice to check explicitly for missing arguments; the PL/I language has no feature to accomplish this check gracefully. Library subroutines are available to determine the number and type of arguments supplied (see lines 102-121). All Multics commands are declared and process their arguments in this way.

3,4,5 It is common practice to include a short comment at the beginning of every program which briefly describes it. This should be followed by a comment or series of comments identifying the date of writing and original author, and the date, author and purpose of any subsequent modifications. This history, or "journalization" as it is called, is very helpful to others who may wish to modify the program in the future.

9 To avoid errors when program maintenance is performed by someone other than the original coder, all variables are explicitly declared. This practice not only avoids surprises, but also gives an opportunity for a comment to indicate how each variable is used.

9 One default which is used here (and is subject to some debate) is that the precision of fixed binary integers is not specified, leading to use of fixed binary(17). This practice has grown up in an attempt to allow the compiler to choose a hardware-supported precision, and in fear that an exact precision specification might cause generated code to check and enforce the specified precision at (presumably) great cost. In fact, the PL/I language does not require such checks by default (although they can be specified). Thus, it is usually wise to specify data precision exactly. In some cases (for instance, all of the fixed binary (21) variables used to hold string lengths), the compiler might attempt to hold these values in half-length registers were this precision not specified.

However, a large class of variables which will contain "small or reasonable size integers" can still be conveniently declared with the implementation's default precision.

12 All character strings in this program are declared unaligned, by the defaults of the language. Given the fact that the Multics hardware has extremely powerful and general string manipulation instructions, no advantage is to be gained in speed or length of object code by declaring strings (when they are over two words, or eight characters, long) with the aligned attribute.

Therefore, almost all supervisor and library subroutines which accept character string arguments require unaligned strings. By the rules of PL/I, aligned and unaligned strings may not be interchanged as parameters, and thus, there is incentive to avoid aligned character strings in all cases.

All line buffers are designed to hold one long typed line (132 characters for input terminals with the widest lines), plus a moderate number of backspace/overstrike characters. To support memorandum typing, the buffers permit a 70-character line which is completely underlined.

By use of temporary segments as work areas (see line 149), an almost unlimited number of nearly infinite work-variables can be constructed, virtually avoiding the "fixed length buffer" problem. However, the acquisition and maintenance of such segments are not as cheap as PL/I automatic variables, and judgement should be exercised as to where traditional "fixed length" variables are appropriate.

14 The variable named "code" has precision 35 bits, since it is used as an output argument for several supervisor entries which return a fixed binary(35) value. Almost all supervisor and library subroutine entries return an "error code" value, which indicates the degree of success of the operation requested. The values of system error codes require 35 bits. It would seem appropriate, on a 36-bit machine, to use fixed binary(35) declarations everywhere. However, use of fixed binary(35) variables for routine arithmetic should be avoided since, for example, addition of two such variables results in a fixed binary(36) result, forcing the compiler to generate code for double precision operations from that point on. We must be careful of the PL/I language rule which requires the compiler to maintain full implicit precision on intermediate results.

15 Legal PL/I overlay defining can be an extremely powerful tool for increasing the readability and maintainability of code. The variable "commands" is declared here as occupying the same storage as the variable "buffer", but only being as long as that part of it which contains valid characters, as defined by the value of "count". Thus, we need only write "commands" when we want the portion of "buffer" that has valid data in it, instead of "the substring of 'buffer' starting at the first character for 'count' characters."

23,24 All editing is done by direct reference to virtual memory locations. The variable "from_ptr" is set to point to a source of text, and the based variable "from_seg" is used for all reference to that text. The number 1048576 (two to the twentieth power) is the largest possible number of characters in a segment.

24,50 The general operation of the editor is to copy the text from one storage area to another, editing on the way. The names "from_seg" and "to_seg" are used for the two storage areas.

43 One set of supervisor interfaces calls for 24 bit integers; this declaration guarantees that no precision conversion is necessary when calling these interfaces. (See line 133.)

56 The PL/I language provides no direct way to express literal control characters. The technique used here, while it clutters the program listing, at least works. The string is typed as a quote, a newline, a tab, a space, and a quote. This order is used because it produces the least ambiguous printed representation; for instance, had the tab and space been reversed, it would not be possible to distinguish by observation between the space, tab sequence and a single tab.

PL/I does not provide any "named constant" facility, either. The Multics PL/I implementation allows the "options (constant)" attribute for internal static variables, which instructs the compiler to allocate the variable in the pure (unmodifiable) portion of the object segment. This is advantageous for three reasons: first, if an attempt is made to modify such a variable, the hardware will detect an error, thus checking and enforcing its "constant" use; second, it allows the variable to be shared between processes, conserving storage; third, it is an indication to others reading the program that a "named constant" is intended. These "constants" are customarily given all uppercase names, as an additional hint to the reader of their constant nature; this is a standard Multics PL/I convention.

64,77 Subroutines com_err_ and ioa_ are called with a different number of arguments each time, a feature not normally permitted in PL/I. The Multics implementation, however, has a feature to permit such calls. The "options" clause warns the compiler that the feature is used for this external subroutine.

65 All subroutines other than com_err_ and ioa_ are completely declared in order to guarantee that the compiler can check that arguments being passed agree in attribute with those expected by the subroutine. Warning diagnostics are printed if the compiler finds argument conversions necessary. (All of the subroutines used by this program are described in the MPM Subroutines Manual.

65 The procedure cu_ (short for command utility) has many different entry points. The Multics PL/I compiler specially handles names of external objects which contain the dollar sign character. The dollar sign is taken to be a separator between a segment name and an entry point name in the compiled external linkage. Thus, this line declares the entry point name arg_ptr in the segment named cu_.

67 For many procedures, the segment name and entry point name are identical, so the compiler also permits the briefer form cv_dec_, which is handled identically to cv_dec_$cv_dec_.

70 The hardcore (ring zero) supervisor entries (hardcore gates) are all easily identifiable since they are entered through a single interface segment named hcs_. Segment hcs_ consists of just a set of transfers to the subroutine wanted. A transfer vector is used to isolate, in one easily available location, all gates into the Multics supervisor. (There are in fact hardcore gate segments other than hcs_, but you will probably not have occasion to deal with them.) For a discussion of the ring structure and hardcore gates, see the MPM Reference Guide.

90 The program will need to know what I/O switches will be used in order to perform certain I/O operations. I/O switches are the general source/sink I/O facility of Multics. Multics PL/I programs manipulate I/O switches as PL/I pointer values. The two external variables declared on this line contain the pointer values identifying the standard terminal input and terminal output switches.

92 As mentioned above, system error codes are returned by most supervisor and library subroutine entries. In one case, we will need to know if a specific error (see line 142) was returned by a supervisor entry. A segment (error_table_) exists which has entry point definitions for external static variables (see Appendix A) containing all the possible values that can be returned as errors by system routines. The variable error_table_$noentry contains the value returned as an error code by system routines to indicate that "the entry you specified in the directory you specified does not exist".

102 The first order of business is to determine how many arguments were supplied to the command, and also to find out whether the command was called properly. This is done by calling a library subroutine.

103 If the error code from cu_$arg_count is nonzero, it means that the program which called cu_$arg_count was not invoked as a command. This usually indicates attempted use as an active function, which is invalid for eds.

104 The library subroutine com_err_ is called to print out the error message describing the invalid call. It produces an English explanation associated with the error code, which is obtained from a system-wide table (the error_table_). It also causes terminal output to be produced even if the user is temporarily diverting output to a file. In general, com_err_ should be called to report all command usage and operation errors. The output from such a call looks like this:

```
eds:  This command cannot be invoked as an active function.
```

105 A Multics command exits simply by returning to its caller. (See also line 437). It should, however, clean up allocated storage, terminate segments, and return temporary segments if it needs to. In general, a program should do exactly the same things when it exits normally as its cleanup handler does. These actions are omitted for this return (and the next) because the program has yet to do anything which would require cleaning up, and because the variables which would inform the cleanup handler of its job have not yet been set. (See lines 133-134.)

109 The eds editor must be invoked with exactly one argument. If it is not, we wish to print a message describing what was wrong, and suggesting the proper usage. This message is produced by picking an appropriate standard error_table_ code to describe the error, and assigning it to code. All the standard error_table_ codes are listed in the MPM Reference Guide, Section 7.

113 The com_err_ subroutine, as well as the ioa_ subroutine (see line 162), allows substitution of parameters in its message. The "^a" string here is used to get the command name into the error message. It is done this way, rather than simply putting "eds" in the message, to make it possible to change the name of the program by changing only the declaration of MYNAME.

117 After verifying that the right number of arguments (one) was supplied, we access the argument. As pointed out above, this is done via library subroutine rather than PL/I parameter passing. Since the command argument is nominally unlimited in length, cu_$arg_ptr returns a pointer to the argument as stored by the command processor, and its length. The based variable "sname" will describe the argument once this pointer and length are obtained. The last argument is a zero, passed by value, because it is known that there is exactly one argument, and there is therefore no reason to receive or check the error code. This should only be done when it is guaranteed that no error can arise from the call, since it will otherwise result in faults.

125 We must now convert the argument to a standard (directory name, entry name) pair. The subroutine expand_pathname_ implements the system-wide standard practice of interpreting the typed argument as either a pathname relative to the current working directory, or an absolute pathname from the root, as appropriate.

134 The program will soon acquire (on line 149) a process resource, namely two temporary segments from the process's pool of temporary segments. When the program is finished executing, it will return them (line 589) to the pool. However, the program may be interrupted (perhaps by a QUIT, or a record quota overflow), and the user may abandon its stack frame (perhaps via the "release" command). In this case, it would seem that the program would not get a chance to return its "borrowed" resources. However, Multics defines the "cleanup" condition, which is signalled in all procedures when their stack frame is about to be irrevocably abandoned. (Refer back to Figures 5-1 and 5-2.) The handler for the cleanup condition invokes the procedure "cleanup", which relinquishes these resources.

The array "temp_segs" is initialized to null pointer values before establishing the cleanup handler, so that the content of the array is well defined at all times. (The release_temp_segments_ subroutine checks for null pointer values, and performs no action if it encounters them.) Otherwise, if the cleanup handler were invoked before the temporary segments were acquired, the pointer array would have undefined, probably invalid values, and the call to release the temporary segments would have unpredictable results.

The cleanup handler is established before the temporary segments are reserved. This sequence guarantees that there will be no "window" in which the program can be abandoned between the time that the segments are acquired and the time that the cleanup handler is set up.

139 The supervisor entry point hcs_$initiate_count is invoked to map the segment specified by the (directory name, entry name) pair into the process's virtual memory. It returns a pointer to the segment, which it constructs from the segment number by which the segment was mapped into the virtual memory of the process (made known). If the segment was already "known", i.e., in the process's address space, the segment number from the existing mapping will be used to create a pointer to return. Refer to the MPM Reference Guide, Section 4, for details.

The PL/I null string ("") is a special signal that no (possibly additional) reference name is to be initiated for the segment.

141 Unfortunately, the zero/nonzero value of the return code from hcs_$initiate_count cannot be used to check whether the initiation (mapping into the address space) succeeded. In the particular case of this subroutine and hcs_$initiate, a nonzero error code is returned in the ostensibly successful case of the segment having already been in the address space or the process, a case which is rarely an error.

These two subroutines are defined to return a nonnull pointer value if and only if the segment has been successfully mapped into the address space, whether by prior act or anew. Thus, testing the return pointer for the PL/I null pointer value is the appropriate test for success.

142 The editor (eds) will create a new segment (see line 496) if an attempt is made to edit a segment which does not exist. By comparing the value of the error code returned from hcs_$initiate_count with the system error code stored in the variable error_table_$noentry, we can differentiate the case of failure to initiate simply because the segment did not exist from all other cases (e.g., incorrect access to the segment specified).

143 The pathname_ subroutine is used here to return a string, which is then substituted into the message produced by com_err_, which is the representation of the pathname. This cannot be done by simply concatenating the dir_name, a ">", and the entry_name, since if the dir_name were ">" (the root directory), this would result in an invalid pathname containing the sequence ">>".

149 A pool of segments in a process directory is maintained by the get_temp_segments_ and release_temp_segments_ subroutines. These segments are doled out to commands and subsystems which request them (via get_temp_segments_) and it is expected that they will be returned to the pool when there is no further use for them. This facility avoids the need for user programs to create and delete (or attempt to manage or share) segments needed on a "scratch" or "temporary" basis (for work areas, buffers, etc). Segments obtained from this facility are guaranteed to contain all zeros (truncated) when obtained.

The number of segments to be obtained is determined by get_temp_segments_ from the extent of the pointer array parameter. The name of the subsystem is passed to get_temp_segments_ both to facilitate additional checking by release_temp_segments, and to support the list_temp_segments command, which describes which subsystems in a process are using temporary segments.

161 If the segment specified on the command line does not exist, the editor is to assume that it is creating a new segment, and go into input mode. The value of the variable "source_ptr" will be null if this is the case.

162 The ioa_ subroutine is a handy library output package. It provides a format facility similar to PL/I and FORTRAN "format" statements, and it automatically writes onto the I/O stream named user_output, which is normally attached to the interactive user's terminal. When used as shown, it appends a newline character to the end of the string given. Programmers who are more concerned about speed and convenience than about compatibility with other operating systems use ioa_ in preference to PL/I "put" statements, because ioa_ is cheaper, easier to use, and far more powerful.

The formatting facilities of ioa_ are used in a simple way in this example. The circumflex ("^") in the format string indicates where a converted variable is to be inserted; the character following the circumflex indicates the form (in this case, a character string) to which the variable should be converted. The first argument is the format string, remaining arguments are variables to be converted and inserted in the output line.

165 The storage system provides for every segment a variable named the "bit count". For a text segment, by convention, the bit count contains the number of information bits currently stored in the segment. The bit count of the segment being edited was returned by hcs_$initiate_count (hence its name) on line 139.

This statement converts the bit count to a character count. Note that we have here embedded knowledge of the number of hardware bits per character in this program.

165 The PL/I language specifies that the result of a divide operation using the division sign is to be a scaled fixed point number. To get integer division, the divide builtin function is used instead. Note that the precision of the quotient is specified to match its size.

166 Here, we invoke some of the most powerful features of the Multics virtual memory. This simple assignment statement copies the entire source segment to be edited into the temporary buffer named "from_seg". A single hardware string-copy instruction is generated for this code, copying data at processor speed. The string-copy instruction may be interrupted by page faults on either "source_seg" or "from_seg" several times; after allocating or reading the required page, the instruction is restarted where it left off. Note that we are regarding the entire text segment as a simple character string of length "size". We may regard it this way because the storage representation for permanent text segments is, by convention, identical to that of a PL/I nonvarying character string.

167 Be sure to read the comments embedded in the program, too.

175 The standard I/O system is being invoked to read a line from the user's terminal. The line is read from the I/O switch identified by the external pointer iox_$user_input. Although passing the buffer to be used as a character string would be more convenient, this set of interfaces was designed with maximal efficiency in mind, and this form of call is more efficient. Note that it would also be safer than passing a pointer to the character string, since that would allow PL/I to check that an appropriate character string was being passed, as opposed to a pointer, which can point to any data type. This design demonstrates the frequent tradeoff between efficiency and convenience.

175 Subroutine iox_$get_line is often used for input rather than the PL/I statement "read file (sysin) into ...", again because of efficiency and error-handling considerations. The PL/I facility ultimately calls on the Multics iox_ package anyway. (Again, if you wished to write a program which would also work on other PL/I systems, you would be better advised to use the PL/I I/O statements instead.)

176 It is highly unlikely that a call to read a line from the terminal will fail. Nevertheless, in cases of people debugging their own extensions to the Multics I/O system (a practice intended by the designers of the I/O system), it can occur. It is reasonable to abort the entire editor in this unlikely case rather than repeating the call: presumably that would repeat the error too.

180 For the sake of human engineering, the editor ignores blank command lines. Since complete input lines from the typewriter end with a new line character, the length of a blank line is one, not zero.

182 The code to isolate a string of characters on the typed input line is needed in four places, so an internal subroutine is used. This subroutine is not recursive, which makes it possible for the compiler to construct a one-instruction calling sequence to the internal procedure. Certain constructs (e.g., variables of adjustable size declared within the subroutine) will force a more complex calling sequence. For details, you should review the documentation on the Multics PL/I implementation, contained in the Multics Pl/I Language Specification, Order No. AG94.

184 Although the dispatching technique used here appears costly, it is really compiled into very quick and effective code -- 2 machine instructions for each line of PL/I. For such a short dispatching table, there is really no point in developing anything more elaborate. If the table were larger, one might use subscripted label constants for greater dispatching speed.

189 Human engineering: the typist is forced to type out the full name of the one "powerful" editing request which, if typed by mistake, could cause overwriting of the original segment before that overwriting was intended.

200 Whenever a message is typed which the typist is probably not expecting, it is good practice to discard any type-ahead, so that he may examine the error message, and redo the typed lines in the light of this new information.

207 The general strategy of the editor is as follows: lines from the typewriter go into the variable named "buffer" (accessed as "commands") until they can be examined. Another buffer, named "line_buffer" (accessed as "line") holds the current line being "pointed at" by the eds conceptual pointer. Subroutine "put" copies the current line onto the end of to_seg, while subroutine "get" copies the next line in from_seg into the current line buffer.

225 The procedure get_num sets up the variable "n" to contain the value of the next typed integer on the request line. Such side-effect communication is not an especially good programming practice.

226 The delete request is accomplished by reading lines from from_seg, but failing to copy them into to_seg. If deletion were a common operation, it might be worthwhile to use more complex code to directly push ahead the pointer in from_seg, and thus avoid a wasted copy operation.

237 More side-effect communication: the variable "edct" is always pointing at the last character so far examined in the typed request line.

254,265 All movement of parts of the material being edited is accomplished by a simple string substitution, using appropriate indexes.

284 The locate request is accomplished by use of the index builtin function, used on whatever is still unedited in from_seg.

422 A negative number in the next request results in moving the conceptual pointer backward. The resulting code is quite complex because the eds editing strategy requires interchanging the input and output segments before backward scanning, so that the backward scan is with regard to the latest edited version of the segment.

427 This code to search a character string backward is recognized by the compiler as such. Extremely efficient object code to search the substring backward is generated, using a single hardware instruction. No copies are made in this fairly expensive-looking statement: it is, in fact, cheap. Combinations of reverse, index, substr, search, verify, etc. that seem like they ought to generate efficient code in fact usually do. The -profile control argument and the profile command are useful tools for discovering where inefficient code is causing performance problems.

456 Before exiting from the editor, the temporary segments should be returned to the temporary segment manager, and the segment that was initiated terminated.

468 Another human engineering point: since the user may have typed several lines ahead, the error message includes the offending request, so that he can tell which one ran into trouble and where to start retyping.

469 Note a small "window" in this sequence of code. If the editor is delayed (by "time-sharing") between lines 468 and 469, it is possible that the message on line 468 will be completed, and the user will have responded by typing one or more revised input lines, all before line 469 discards all pending input. Although in principle fixable by a reset option on the write call, Multics currently provides no way to cover this timing window. Fortunately, the window is small enough that most interactive users will go literally for years without encountering an example of a timing failure on input read reset.

500 Note the practice of copying data into the original segment, setting its bit count, and truncating it in that order. This provides for maximal data being saved should there be a system failure between any two lines. Common sense seems to indicate this order as "maximally safe", and analysis of the data involved will demonstrate this as well.

538-540 The input and output editing buffer areas are interchanged by these three statements. Here is an example of localizing the use of pointer variables to make clear that they are being used as escapes to allow interchange of the meaning of PL/I identifiers.

551 The I/O system provides this entry point to perform control operations (e.g., "resetread") upon the objects represented by I/O switches.

563 This editor considers typed-in tab characters to be just as suitable for token delimiters as are blanks. Ideally, tab characters would never reach the editor, having been replaced by blanks by the typewriter input routines. Such complete canonicalization of the input stream would result in some greater simplicity, but would require a more sophisticated strategy to handle editing of text typed in columns.

563, 566 The PL/I search and verify builtins, which are quite useful in circumstances like this (parsing lines), are compiled into very efficient single-instruction hardware operations by the Multics PL/I compiler.

580 The cv_dec_ library routine is used here rather than a PL/I language feature, because cv_dec_ will always return a value, even if the number to be converted is ill-formed (in which case it returns zero). Thus, the editor chooses not to handle ill-formed numbers. Had it wished to check for them, it could have used the cv_dec_check_ subroutine. PL/I language conversion would cause an error signal which must be caught and interpreted lest PL/I's runtime diagnostic appear on the user's console. Thus, eds retains complete control over the error comments and messages which will be presented to the user. Such control is essential if one is to construct a well-engineered interface which uses consistent and relevant error messages.

589 The cleanup procedure calls the release_temp_segments_ subroutine to release the temporary segments acquired earlier. A binary zero is passed to release_temp_segments_ by value (by enclosing it in parentheses) because the cleanup handler has no use for an error code. Cleanup procedures should never print messages, even error messages, because they are only invoked when exiting a procedure. There is no corrective action the user can take.

590 If the segment edited was not known before editing it, it should be unknown after the editor finishes as well. The supervisor maintains a reference count for each segment in the process. This count is incremented by the call to hcs_$initiate and decremented by the call to hcs_$terminate_noname. If the count goes to zero (i.e. the segment was made known by the editor), then the segment is made unknown.

```
COMPILATION LISTING OF SEGMENT eds
Compiled by: Experimental PL/I Compiler of Thursday, February 26, 1981 at 18:23
Compiled at: Honeywell LISD Phoenix, System M
Compiled on: 06/01/81  1640.1 mst Mon
    Options: optimize map

eds:      procedure options (variable);

/*         Simple text editor -- example program */
/*         Written July, 1979, by Someone U. Know */
/*         Modified May, 1981, for MR9.0 subroutines, by Someone Else */

/*         internal variable declarations.  */

declare  arg_count                fixed binary;              /*  Number of command line arguments  */
declare  break                    character (1);             /*  Holds break char for change */
declare  crkl                     fixed binary;
declare  buffer                   character (210);           /*  Typewriter input buffer.  */
declare  changes_occurred         bit (1);
declare  code                     fixed binary (35);
declare  commands                 character (count) based (addr (buffer));
                                                             /*  Valid portion of buffer */
declare  count                    fixed binary (21);         /*  Valid length of data in "buffer " */
declare  csize                    fixed binary (21);
declare  edct                     fixed binary;
declare  dir_name                 character (168);           /*  Directory containing segment */
declare  entry_name               character (32);
declare  exptr                    pointer;                   /*  Temporary pointer holder.  */
declare  from_ptr                 pointer;                   /*  Pointer to current from_seg.  */
declare  from_seg                 character (1048576) based (from_ptr);
                                                             /*  Editing is from this segment.  */
declare  globsw                   bit (1);
declare  i                        fixed binary (21);
declare  ii                       fixed binary (21);
declare  indf                     fixed binary (21);
declare  inpt                     fixed binary (21);
declare  j                        fixed binary (21);
declare  k                        fixed binary (21);
declare  l                        fixed binary (21);
declare  line                     character (linel) based (addr (line_buffer));
declare  line_buffer              character (210);           /*  Holds line currently being edited.  */
declare  linel                    fixed binary;              /* length of "line" */
declare  located                  fixed binary;
declare  m                        fixed binary (21);
declare  n                        fixed binary (21);
declare  sname                    character (sname_lth) based (sname_ptr);   /* Source name */
declare  sname_lth                fixed binary (21);         /*  Length of source segment name.  */
declare  sname_ptr                pointer;                   /*  Pointer to source segment name.  */
declare  source_count             fixed binary (24);         /*  Holds segment bit length.  */
declare  source_ptr               pointer;                   /*  Pointer to source seg.  */
declare  source_seg               character (1048576) based (source_ptr);
                                                             /*  Outside segment for read or write.  */
declare  temp_segs                dimension (2) pointer;
declare  tlin                     character (210);           /*  Buffer to hold output of change.  */
declare  tkn                      character (8);             /*  Holds next item on typed line */
declare  to_seg                   character (1048576) based (to_ptr);
                                                             /*  Editing is to this segment.  */
declare  to_ptr                   pointer;                   /*  Pointer to to_seg.  */

/*  Constants */
```

```

declare  NL                     character (1) static options (constant) initial ("
");
declare  WHITESPACE             character (3) static options (constant) initial ("
          "); /* NL TAB SPACE */
declare  MYNAME                 character (3) static options (constant) initial ("eds");

/*        external subroutine declarations.  */

declare  com_err_               entry options (variable);
declare  cu_$arg_count          entry (fixed binary, fixed binary (35));
declare  cu_$arg_ptr            entry (fixed binary, pointer, fixed binary (21), fixed binary (35));
declare  cv_dec_                entry (character (*)) returns (fixed binary(35));
declare  expand_pathname_       entry (character (*), character (*), character (*), fixed binary (35));
declare  get_temp_segments_     entry (character (*), pointer dimension (*), fixed binary (35));
declare  hcs_$initiate_count    entry (character (*), character (*), character (*), fixed binary (24),
                                fixed binary, pointer, fixed binary (35));
declare  hcs_$make_seg          entry (character (*), character (*), character (*),
                                fixed bin (5), ptr, fixed binary (35));
declare  hcs_$set_bc_seg        entry (pointer, fixed binary (24), fixed binary(35));
declare  hcs_$terminate_noname  entry (pointer, fixed binary (35));
declare  hcs_$truncate_seg      entry (pointer, fixed binary (19), fixed binary(35));
declare  ioa_                   entry options (variable);
declare  iox_$control           entry (pointer, character (*), pointer, fixed binary (35));
declare  iox_$get_line          entry (pointer, pointer, fixed binary (21), fixed binary (21), fixed binary (35))
declare  iox_$put_chars         entry (pointer, pointer, fixed binary (21), fixed binary (35));
declare  pathname_              entry (character (*), character (*)) returns (character (168));
declare  release_temp_segments_ entry (character (*), pointer dimension (*), fixed binary (35));

declare  cleanup condition;
declare  (addr, divide, index, length, null, reverse, search, substr, verify)
                                builtin;

/*  External data */

declare (iox_$user_output, iox_$user_input)       pointer external static;
declare  error_table_$noarg                       fixed binary (35) external static;
declare  error_table_$noentry                     fixed binary (35) external static;
declare  error_table_$too_many_args               fixed binary (35) external static;

```

```


/*        .         .         .         P R O G R A M       .         .         .         */


/* Check to see if an input argument was given */

          call cu_$arg_count (arg_count, code);
          if code ^= 0 then do;                             /* Not called as a command */
               call com_err_ (code, MYNAME);
               return;
          end;

          if arg_count = 1 then code = 0;                   /* This is correct */
          else if arg_count = 0 then code = error_table_$noarg; /* Argument is missing */
          else code = error_table_$too_many_args;           /* Otherwise, there were too many */

          if code ^= 0 then do;                             /* If not called correctly, complain */
               call com_err_ (code, MYNAME, "^/Usage:^-^a <PATH>", MYNAME);
               return;
          end;

          call cu_$arg_ptr (1, sname_ptr, sname_lth, (0));
          if code ^= 0 then do;
               call com_err_ (code, MYNAME, "Usage: ^a <PATH>", MYNAME);
               return;
          end;

/* Now get a pointer to the segment to be edited */

          call expand_pathname_ (sname, dir_name, entry_name, code);
          if code ^= 0 then do;                             /* Bad pathname */
               call com_err_ (code, MYNAME, "^a", sname);
               return;
          end;

/*  Set up a cleanup handler in case the program is aborted */

          source_ptr = null ();
          temp_segs (*) = null ();                          /* Make sure handler has valid data */
          on condition (cleanup) call clean_up;

/* Initiate the source segment. */

          call hcs_$initiate_count (dir_name, entry_name, "", source_count, 0, source_ptr, code);
                                                            /* Initiate the segment */
          if source_ptr = null ()
               then if code ^= error_table_$noentry then do;/* Problem or just new seg? */
                    call com_err_ (code, MYNAME, "Cannot access ^a", pathname_ (dir_name, entry_name));
                    return;
               end;

/*  Set up buffer segments.  */

          call get_temp_segments_ (MYNAME, temp_segs, code);
          if code ^= 0 then do;
               call com_err_ (code, MYNAME, "Cannot get temporary segments.");
               call clean_up;
               return;
          end;
```

```
          from_ptr = temp_segs (1);
          to_ptr = temp_segs (2);

/* Check to see that the segment is there */

          csize, inpf, inpt = 0;                            /* Initialize buffer control vars. */
          if source_ptr = null then do;
               call ioa_ ("Segment ^a not found.", entry_name);
               go to pinput;
          end;
          csize = divide (source_count, 9, 21, 0);          /* change bit count to char count */
          substr (from_seg, 1, csize) = substr (source_seg, 1, csize);
                                                            /*  Move source segment into buffer.  */

/*  Main editing loop . . . . .  */


nedit:
          call ioa_ ("Edit.");
next:
          call iox_$get_line (iox_$user_input, addr (buffer), length (buffer), count, code);
          if code ^= 0 then do;
               call com_err_ (code, MYNAME, "Error reading input line");
               go to finish;
          end;
          if count = 1 then go to next;                     /* if null line then get another line, don't print error */
          scnt = 1;                                         /*  Set up counter to scan this line.  */
          call get_token;                                   /*  Identify next token.  */

          if tkn = "i" then go to insert;
          if tkn = "r" then go to retype;
          if tkn = "l" then go to locate;
          if tkn = "p" then go to print;
          if tkn = "n" then go to nexlin;
          if tkn = "save" then go to file;
          if tkn = "c" then go to change;
          if tkn = "d" then go to dellin;
          if tkn = "w" then go to wsave;
          if tkn = "t" then go to top;
          if tkn = "b" then go to bottom;
          if tkn = "." then go to pinput;

 /* If none of the above then not a request */

          call ioa_ ("'^a' Not an edit Request", substr (commands, 1, length (commands) - 1));
          call resetread;
          go to next;

/* ********* input mode ********* */

pinput:
          call ioa_ ("Input.");                             /* print word input */
input:
          call iox_$get_line (iox_$user_input, addr (buffer), length (buffer), count, code);
          if code ^= 0 then do;
               call com_err_ (code, MYNAME, "Error reading input-mode line.");
               go to finish;
          end;

          if substr (commands, 1, 1) = "." & count = 2
```

```
               then go to nedit;                              /* check for mode change */
          call put;
          linel = length (commands);
          line = commands;                                    /* move line inputted into intermediate storage */
          go to input;                                        /* repeat 'til "." */


/* ********* delete ********* */

dellin:
          call get_num;
          do i = 1 to n - 1;                                  /* do for each line to be deleted */
               call get;
          end;
          linel = 0;                                          /* nullify last line */
          go to next;

/* ********* insert ********* */

insert:
          call put;                                           /*  Add current line to output segment */
retype:                                                       /*  This is also the retype request.  */
          linel = length (commands) - edct;
          line = substr (commands, edct + 1);                 /* add replaced line */
          go to next;

/* ********* next ********* */

nexlin:   call get_num;
          if n < 0 then go to backup;
          m, j = indf;                                        /* save where you are */
          call put;
          do i = 1 to n;                                      /* once for each nl */
               if j >= csize then go to n_eof;                /* check for eof */
               k = index (substr (from_seg, j + 1, csize - j), NL);
                                                              /* locate end of line */
               if k = 0 then do;                              /* no nl (eof) print eof */
n_eof:              if indf >= csize then go to eof;
                    linel = 0;                                /* set to no line */
                    substr (to_seg, indt + 1, csize - m) = substr (from_seg, m + 1, csize - m);
                                                              /* move in top of file */
                    indf = csize;
                    indt = indt + csize - m;                  /* set pointers */
                    go to eof;
               end;
               j = j + k;                                     /* increment j by length of line */
          end;
          indf = j;                                           /* set pointers and move in top of file */
          linel = k;
          line = substr (from_seg, j - k + 1, linel);         /* put working line in line */
          substr (to_seg, indt + 1, indf - linel - m) = substr (from_seg, m + 1, indf - linel - m);
                                                              /* fill rest of file */
          indt = indt + indf - linel - m;
          go to next;

/* ********* locate ********* */

locate:   if edct = length (commands) then go to bad_syntax;      /* check for plain "l NL" */
          edct = edct + 1;                                    /*  Skip delimiter.  */
          l = indt;                                           /* initialize pointers for index type search */
```

```
          m = indf;
          n = csize - indf;
          call put;
          if (csize = 0) | (n <= 0) then do;
               call switch;
               if i > 0 then n = i - 1;
               else n = 0;
               m, i = 0;
          end;
          i = index (substr (from_seg, indf + 1, n), substr (commands, edct, length (commands) - edct));
          if i ^= 0 then do;                                     /* if found then do */
               k = index (reverse (substr (from_seg, 1, indf + i)), NL);
               if k ^= 0 then k = indf + i - k + 1;              /* k = index of NL */
               j = index (substr (from_seg, k + 1, csize - k), NL); /* find end of line */
               if j = 0 then indf = csize;
               else indf = j + k;
               substr (to_seg, indt + 1, k - m) = substr (from_seg, m + 1, k - m);
                                                         /* move in top of file */
               linel = indf - k;
               indt = indt + k - m;
               line = substr (from_seg, k + 1, linel);   /* put found line in line */
               n = 1;
               go to print1;                             /* print found line if wanted */
          end;
          call copy;
          call switch;
          go to next;                                    /* get next command */

/* ********* print ********* */

print:    call get_num;
          if linel = 0 then do;                          /* print indication of no lines */
               call ioa_ ("No line.");
               go to noline;
          end;

print1:   call iox_$put_chars (iox_$user_output, addr (line), length (line), code);
          if code ^= 0 then do;
               call com_err_ (code, MYNAME, "Problem writing editor output");
               go to finish;
          end;
                                                         /* write the line */
noline:   n = n - 1;
          if n = 0 then go to next;                      /* any more to be printed? */
          call put;
          call get;
          go to print1;

/* ********* change ********* */

change:   located = 0;
          if count = 2 then do;
bad_syntax:
               count = count - 1;                        /* Strip NL off "commands " */
               call ioa_ ("Improper:  ^a", commands);
               call resetread;
               go to next;
          end;
          brk1 = edct + 2;
          break = substr (commands, edct + 1, 1);        /*  Pick up the delimiting character.  */
```

```
                i = index (substr (commands, brk1), break);
                if i = 0 then go to bad_syntax;
                j = index (substr (commands, i + brk1), break);
                if j = 0 then j = length (commands) - i - brk1 + 1;
                edct = edct + i + j + 1;                            /*  Continue scanning edit line.  */
                globsw = "0"b;                                      /*  Assume only one change.  */
                n = 1;                                              /*  Assume only one line changed.  */
nxarg:
                call get_token;
                if tkn ^= " " then do;                              /*  If token there, process it.  */
                     if tkn = "g" then globsw = "1"b;               /*  Change all occurrances.  */
                     else call cv_num;
                     go to nxarg;                                   /*  Try for another argument.  */
                end;
                if linel = 0 then go to skipch;                     /*  Skip changing empty line.  */

ch1:            changes_occurred = "0"b;
                m, ij, l = 1;                                       /* indexes to strings */
                if i = 1 then do;                                   /* add to beginning of line */
                     changes_occurred = "1"b;
                     located = 1;
                     substr (tlin, 1, j - 1) = substr (commands, brk1 + i, j - 1);
                                                                    /* copy part to be added */
                     substr (tlin, j, length (line)) = line;        /* copy old line */
                     ij = j + linel - 1;
                     l = j + linel + 1;
                     go to cont;
                end;
ch2:            k = index (substr (line, m), substr (commands, brk1, i - 1));
                                                                    /* locate what is to be changed */
                if k ^= 0 then do;
                     substr (tlin, ij, k - 1) = substr (line, m, k - 1);
                                                                    /* copy line up to change */
                     substr (tlin, ij + k - 1, j - 1) = substr (commands, brk1 + i, j - 1);
                                                                    /* put in change */
                     m = m + k + i - 2;                             /* increment indexes */
                     ij = ij + k + j - 2;
                     l = l + k + j - 2;
                     changes_occurred = "1"b;                       /* indicate that you did someting */
                     located = 1;
                     if globsw then go to ch2;
                end;
                substr (tlin, ij, length (line) - m + 1) = substr (line, m);
                                                                    /* copy rest of line */
                ij = ij + length (line) - m;
                l = l + length (line) - m;
cont:
                if changes_occurred then do;                        /* Write changes */
                     call iox_$put_chars (iox_$user_output, addr (tlin), l, code);
                     if code ^= 0 then do;
                          call com_err_ (code, MYNAME, "Error writing change line");
                          go to finish;
                     end;
                end;
                linel = ij;
                line = substr (tlin, 1, ij);

skipch:         if n <= 1 then do;                                  /* finished */
                     if located = 0 then do;
                          count = count - 1;                        /* Get rid of NL i "commands" */
```

```
                    call ioa_ ("Nothing changed by:  ^a", commands);
                                                            /* if not located */
                    call resetread;
               end;
               go to next;
          end;
          n = n - 1;
          call put;
          call get;
          go to chl;



/* ********* top ********* */

top:      call copy;
          call switch;
          go to next;

/* ********* bottom ********* */

bottom:   call copy;
          linel = 0;                                        /* No line buffer */
          go to ginput;

/* ********* backup ********* */

backup:   i = indt;                                         /* save ptrs */
          call copy;
          call switch;
          indf = i + 1;                                     /* restore ptrs */
          do n = n to 0;                                    /*  Note that "n" starts negative.  */
               j = index (reverse (substr (from_seg, 1, indf - 1)), NL);
               if j ^= 0 then indf = indf - j;
               else if n = 0 then indf = 0;                 /*  First line case */
               else do;
                    linel = 0;                              /* went off top of file */
                    n = 1;
                    indt, indf = 0;
                    go to eof;
               end;
          end;
          indt = indf;                                      /* line starts as indt */
          substr (to_seg, 1, indt) = substr (from_seg, 1, indt);
                                                            /* move in top of file */
          do indf = indt + 1 by 1 to csize;                 /* find end of line */
               substr (line, indf - indt, 1) = substr (from_seg, indf, 1);
                                                            /* move into line */
               if substr (from_seg, indf, 1) = NL
               then go to line_end;                         /* search for end of line */
          end;
          indf = csize;
line_end:
          linel = indf - indt;
          n = 1;
          go to printl;

/* **********  "file" request ********** */

file:     call copy;                                        /*  Finish copy.  */
```

```
          call save;                                        /*  Save it.  */
finish:   call clean_up;                                    /*  Terminate source and release temp segs */
          return;                                           /*  Return to command processor */

/* ********** write save ********** */

wsave:    call copy;                                        /*  Finish copy.  */
          call save;                                        /*  Save it.  */
          go to next;                                       /*  Continue accepting requests.  */

/* ********* eof ********* */

eof:      count = count - 1;                                /* Remove NL */
          call ioa_ ("End of File reached by:^/^a", commands);
          call resetread;
          go to next;


/* ********* I N T E R N A L   P R O C E D U R E S ********* */


copy: procedure;                                            /* copy rest of file into to file */
          substr (to_seg, indt + 1, length (line)) = line;
                                                            /*  Copy current line.  */
          indt = indt + length (line);
          linel = 0;                                        /* No more line */
          if csize = 0
          then return;                                      /*  If new input, then no copy needed.  */
          ij = csize - indf,                                /* do rest of file */
          if ij > 0
          then substr (to_seg, indt + 1, ij) = substr (from_seg, indf + 1, ij);
          indt = indt + ij;                                 /* set counters */
          indf = csize;
          return;

     end copy;


save: procedure;                                            /*  Procedure to write out all or part of "to" buffer.  */
          if source_ptr = null then do;                     /* Must be a new segment */
               call hcs_$make_seg (dir_name, entry_name, "", 01010b, source_ptr, code);
               if code ^= 0 then do;
                    call com_err_ (code, MYNAME, "Cannot create ^a", pathname_ (dir_name, entry_name));
                    return;
               end;
          end,
          substr (source_seg, 1, indt) = substr (to_seg, 1, indt);
          call hcs_$set_bc_seg (source_ptr, indt * 9, code);
          if code = 0
          then call hcs_$truncate_seg (source_ptr, divide (indt + 3, 4, 19, 0), code);
          if code ^= 0 then do;
               call com_err_ (code, MYNAME, "Cannot truncate/set bit count (^d) on ^a",
                    indt * 9, pathname_ (dir_name, entry_name));
          end;
          return;

     end save;
```

```
put:
     procedure;
          substr (to_seg, indt + 1, length (line)) = line;  /* do move */
          indt = indt + length (line);                      /* set counters */
          linel = 0;                                        /*  Discard old line.  */
          return;
     end;


get:
     procedure;
          linel = 0;                                        /*  Reset current line length.  */
          if indf >= csize then go to eof;                                      /*  If no input left, give up.
          linel = index (substr (from_seg, indf + 1, csize - indf), NL);
                                                            /*  Find the next new line.  */
          if linel = 0 then linel = csize - indf;           /*  If no nl found, treat end of segment as one.  */
          line = substr (from_seg, indf + 1, linel);        /*  Return the line to caller.  */
          indf = linel + indf;                              /*  Move the "from" pointer ahead one line.  */
          return;
     end;

switch:
     procedure;                                             /* make from-file to file, and v.v. */
          exptr = from_ptr;
          from_ptr = to_ptr;
          to_ptr = exptr;
          csize = indt;
          indt, indf = 0;
          linel = 0;
          return;

     end switch;

resetread:
     procedure;                                             /*  Call i/o system reset read entry.  */
                                                            /* In one place to centralize error handling */
          call iox_$control (iox_$user_input, "resetread", null (), code);
          if code ^= 0 then call com_err_ (code, MYNAME, "Cannot resetread user_input");
          return;

     end resetread;

get_token:
     procedure;

declare (token_lth, white_lth) fixed binary (21);

          tkn = " ";                                        /* Set for easy failure */
          white_lth = verify (substr (commands, edct), WHITESPACE) - 1;
          if white_lth < 0 then return;                     /* Only whitespace left */
          edct = edct + white_lth;
          token_lth = search (substr (commands, edct), WHITESPACE) - 1;
          if token_lth < 0 then token_lth = length (commands)- edct; -
          tkn = substr (commands, edct, token_lth);         /* Extract token */
          edct = edct + token_lth;
          return;

     end get_token;
```

```
get_num:
     procedure;                                                 /*  Routine to convert token to binary integer.  */
          call get_token;                                       /*  Delimit the token.  */
cv_num:
     entry;                                                     /*  Enter here if token already available.  */
          n = cv_dec_ (tkn);                                    /*  Convert it.  */
          if n = 0 then n = 1;                                  /*  Default count is 1.  */
          return;

     end get_num;

clean_up:
     procedure;

         call release_temp_segments_ (MYNAME, temp_segs, (0));
         if source_ptr ^= null then call hcs_$terminate_noname (source_ptr, (0));

     end clean_up;

     end eds;
```

```
          SOURCE FILES USED IN THIS COMPILATION.

LINE      NUMBER  DATE MODIFIED     NAME                                  PATHNAME
             0    06/01/81  1643.1  eds.pl1                               >udd>Pubs>userd>AG90-02>eds.pl1
```

```
          NAMES DECLARED IN THIS COMPILATION.

IDENTIFIER              OFFSET  LOC STORAGE CLASS   DATA TYPE              ATTRIBUTES AND REFERENCES
                                                                           (* indicates a set context)

NAMES DECLARED BY DECLARE STATEMENT.
MYNAME                       000000 constant        char(3)                initial unaligned dcl 60 set ref 104* 113* 113* 119*
                                                                             119* 127* 143* 149* 151* 177* 210* 313* 385* 498*
                                                                             507* 552* 589*
NL                           004102 constant        char(1)                initial unaligned dcl 56 ref 249 286 288 427 443 528
WHITESPACE                   000001 constant        char(3)                initial unaligned dcl 58 ref 563 566
addr                                                builtin function       dcl 85 ref 174 174 199 199 199 199 207 207 214 217
                                                                             218 218 236 238 238 264 272 284 284 295 311 311
                                                                             311 311 311 311 329 334 335 337 338 356 358 358
                                                                             363 363 366 368 377 377 379 380 383 383 390 395
                                                                             441 468 478 478 480 517 517 518 531 563 566 567
                                                                             568
arg_count                    000100 automatic       fixed bin(17,0)        dcl 9 set ref 102* 108 109
break                        000101 automatic       char(1)                unaligned dcl 10 set ref 334* 335 337
brk1                         000102 automatic       fixed bin(17,0)        dcl 11 set ref 333* 335 337 338 356 363 368
buffer                       000103 automatic       char(210)              unaligned dcl 12 set ref 174 174 174 174 199 199 199
                                                                             199 207 207 207 207 214 217 218 236 238 272 284
                                                                             284 329 334 335 337 338 356 363 368 395 468 563
                                                                             566 567 568
changes_occurred             000170 automatic       bit(1)                 unaligned dcl 13 set ref 351* 354* 373* 381
   anup                      000472 stack reference condition              dcl 84 ref 135
  de                         000171 automatic       fixed bin(35,0)        dcl 14 set ref 102* 103 104* 108* 109* 110* 112 113*
                                                                             118 119* 125* 126 127* 139* 141 143* 149* 150 151*
                                                                             174* 176 177* 207* 209 210* 311* 312 313* 383* 384
                                                                             385* 496* 497 498* 503* 504 504* 506 507* 551* 552
                                                                             552*
com_err_                     000010 constant        entry                  external dcl 64 ref 104 113 119 127 143 151 177 210
                                                                             313 385 498 507 552
commands                            based           char                   unaligned dcl 15 set ref 199 199 199 199 214 217 218
                                                                             236 238 272 284 284 329* 334 335 337 338 356 363
                                                                             368 395* 468* 563 566 567 568
count                        000172 automatic       fixed bin(21,0)        dcl 17 set ref 174* 180 199 199 199 199 207* 214 214
                                                                             217 218 236 238 272 284 284 326 327* 327 329 329
                                                                             334 335 337 338 356 363 368 394* 394 395 395 467*
                                                                             467 468 468 563 566 567 568
csize                        000173 automatic       fixed bin(21,0)        dcl 18 set ref 160* 165* 166 166 248 249 252 254 254
                                                                             256 257 276 278 288 289 440 446 482 484 488 527
                                                                             528 530 541*
cu_$arg_count                000012 constant        entry                  external dcl 65 ref 102
cu_$arg_ptr                  000014 constant        entry                  external dcl 66 ref 117
cv_dec_                      000016 constant        entry                  external dcl 67 ref 580
dir_name                     000175 automatic       char(168)              unaligned dcl 20 set ref 125* 139* 143* 143* 496*
                                                                             498* 498* 507* 507*
divide                                              builtin function       dcl 85 ref 165 504 504
edct                         000174 automatic       fixed bin(17,0)        dcl 19 set ref 181* 236 238 272 273* 273 284 284 333
                                                                             334 339* 339 563 565* 565 566 567 568 569* 569
entry_name                   000247 automatic       char(32)               unaligned dcl 21 set ref 125* 139* 143* 143* 162*
                                                                             496* 498* 498* 507* 507*
error_table_$noarg           000056 external static fixed bin(35,0)        dcl 91 ref 109
error_table_$noentry         000060 external static fixed bin(35,0)        dcl 92 ref 141
error_table_$too_many_args   000062 external static fixed bin(35,0)        dcl 93 ref 110
expand_pathname_             000020 constant        entry                  external dcl 68 ref 125
exptr                        000260 automatic       pointer                dcl 22 set ref 538* 540
from_ptr                     000262 automatic       pointer                dcl 23 set ref 155* 166 249 254 264 265 284 286 288
                                                                             291 295 427 438 441 443 485 528 531 538 539*
```

```
from_seg                                       based          char(1048576)          unaligned dcl 24 set ref 166* 249 254 264 265 284
                                                                                       286 288 291 295 427 438 441 443 485 528 531
get_temp_segments_                      000022 constant       entry                  external dcl 69 ref 149
globsw                                  000264 automatic      bit(1)                 unaligned dcl 26 set ref 340* 345* 375
hcs_$initiate_count                     000024 constant       entry                  external dcl 70 ref 139
hcs_$make_seg                           000026 constant       entry                  external dcl 72 ref 496
hcs_$set_bc_seg                         000030 constant       entry                  external dcl 74 ref 503
hcs_$terminate_noname                   000032 constant       entry                  external dcl 75 ref 590
hcs_$truncate_seg                       000034 constant       entry                  external dcl 76 ref 504
i                                       000265 automatic      fixed bin(21,0)        dcl 27 set ref 226* 247* 284* 285 286 287 335* 336
                                                                                       337 338 339 353 356 363 368 370 422* 425
ij                                      000266 automatic      fixed bin(21,0)        dcl 28 set ref 352* 359* 366 368 371* 371 377 379*
                                                                                       379 389 390 484* 485 485 485 487
index                                                         builtin function       dcl 85 ref 249 284 286 288 335 337 363 427 528
indf                                    000267 automatic      fixed bin(21,0)        dcl 29 set ref 160* 245 252 256* 262* 265 265 267
                                                                                       275 276 284 286 287 289* 290* 293 425* 427 428*
                                                                                       428 429* 433* 437 440* 441 441 443* 446* 447 484
                                                                                       485 488* 527 528 528 530 531 532* 532 542*
indt                                    000270 automatic      fixed bin(21,0)        dcl 30 set ref 160* 254 257* 257 265 267* 267 274
                                                                                       291 294* 294 422 433* 437* 438 438 440 441 447 478
                                                                                       480* 480 485 487* 487 502 502 503 504 504 507 517
                                                                                       518* 518 541 542*
ioa_                                    000036 constant       entry                  external dcl 77 ref 162 172 199 205 307 329 395 468
iox_$control                            000040 constant       entry                  external dcl 78 ref 551
iox_$get_line                           000042 constant       entry                  external dcl 79 ref 174 207
iox_$put_chars                          000044 constant       entry                  external dcl 80 ref 311 383
iox_$user_input                         000054 external static pointer               dcl 90 set ref 174* 207* 551*
iox_$user_output                        000052 external static pointer               dcl 90 set ref 311* 383*
j                                       000271 automatic      fixed bin(21,0)        dcl 31 set ref 245* 248 249 249 260* 260 262 264
                                                                                       274* 280 280 282* 288* 289 290 337* 338 338* 339
                                                                                       356 356 358 359 360 368 368 371 372 427* 428 428
k                                       000272 automatic      fixed bin(21,0)        dcl 32 set ref 249* 251 260 263 264 286* 287 287*
                                                                                       287 288 288 290 291 291 293 294 295 363* 365 366
                                                                                       366 368 370 371 372
l                                       000273 automatic      fixed bin(21,0)        dcl 33 set ref 352* 360* 372* 372 380* 380 383*
length                                                        builtin function       dcl 85 ref 174 174 199 199 207 207 217 236 272 284
                                                                                       311 311 338 358 377 379 380 478 480 517 518 567
line                                           based          char                   unaligned dcl 34 set ref 218* 238* 264* 295* 311 311
                                                                                       311 311 358 358 363 366 377 377 379 380 390* 441*
                                                                                       478 478 480 517 517 518 531*
line_buffer                             000274 automatic      char(210)              unaligned dcl 35 set ref 218 238 264 295 311 311 311
                                                                                       311 358 358 363 366 377 377 379 380 390 441 478
                                                                                       478 480 517 517 518 531
linel                                   000361 automatic      fixed bin(17,0)        dcl 36 set ref 217* 218 229* 236* 238 253* 263* 264
                                                                                       264 265 265 267 293* 295 295 306 311 311 311 311
                                                                                       349 358 358 359 360 363 366 377 377 379 380 389*
                                                                                       390 417* 431* 441 447* 478 478 480 481* 517 517
                                                                                       518 519* 526* 528* 530 530* 531 531 532 543*
located                                 000362 automatic      fixed bin(17,0)        dcl 37 set ref 325* 355* 374* 393
m                                       000363 automatic      fixed bin(21,0)        dcl 38 set ref 245* 254 254 254 257 265 265 265 267
                                                                                       275* 282* 291 291 291 294 352* 363 366 370* 370
                                                                                       377 377 379 380
n                                       000364 automatic      fixed bin(21,0)        dcl 39 set ref 226 244 247 276* 278 280* 281* 284
                                                                                       296* 317* 317 318 341* 392 401* 401 426* 426* 429
                                                                                       432* 449* 580* 581 581*
null                                                          builtin function       dcl 85 ref 133 134 141 161 495 551 551 590
pathname_                               000046 constant       entry                  external dcl 81 ref 143 143 498 498 507 507
release_temp_segments_                  000050 constant       entry                  external dcl 82 ref 589
reverse                                                       builtin function       dcl 85 ref 286 427
search                                                        builtin function       dcl 85 ref 566
```

```
sname                                     based            char                 unaligned dcl 40 set ref 125* 127*
sname_lth                          000365 automatic        fixed bin(21,0)      dcl 41 set ref 117* 125 125 127 127
sname_ptr                          000366 automatic        pointer              dcl 42 set ref 117* 125 127
source_count                       000370 automatic        fixed bin(24,0)      dcl 43 set ref 139* 165
source_ptr                         000372 automatic        pointer              dcl 44 set ref 133* 139* 141 161 166 495 496* 502
                                                                                  503* 504* 590 590*
source_seg                                based            char(1048576)        unaligned dcl 45 set ref 166 502*
substr                                                     builtin function     dcl 85 set ref 166* 166 199 199 214 238 249 254* 254
                                                                                  264 265* 265 284 284 286 288 291* 291 295 334 335
                                                                                  337 356* 356 358* 363 363 366* 366 368* 368 377*
                                                                                  377 390 427 438* 438 441* 441 443 478* 485* 485
                                                                                  502* 502 517* 528 531 563 566 568
temp_segs                          000374 automatic        pointer              array dcl 47 set ref 134* 149* 155 156 589*
tkn                                000466 automatic        char(8)              unaligned dcl 49 set ref 184 185 186 187 188 189 190
                                                                                  191 192 193 194 195 344 345 562* 568* 580*
tlin                               000400 automatic        char(210)            unaligned dcl 48 set ref 356* 358* 366* 368* 377*
                                                                                  383 383 390
to_ptr                             000470 automatic        pointer              dcl 52 set ref 156* 254 265 291 438 478 485 502 517
                                                                                  539 540*
to_seg                                    based            char(1048576)        unaligned dcl 50 set ref 254* 265* 291* 438* 478*
                                                                                  485* 502 517*
token_lth                          000556 automatic        fixed bin(21,0)      dcl 560 set ref 566* 567 567* 568 569
verify                                                     builtin function     dcl 85 ref 563
white_lth                          000557 automatic        fixed bin(21,0)      dcl 560 set ref 563* 564 565

MES DECLARED BY EXPLICIT CONTEXT.
ackup                              002474 constant         label                dcl 422 ref 244
ad_syntax                          001775 constant         label                dcl 327 ref 272 336
ottom                              002471 constant         label                dcl 416 ref 194
h1                                 002136 constant         label                dcl 351 ref 404
h2                                 002203 constant         label                dcl 363 ref 375
hange                              001771 constant         label                dcl 325 ref 190
lean_up                            003433 constant         entry                internal dcl 586 ref 135 152 456
opy                                002650 constant         entry                internal dcl 477 ref 299 410 416 423 454 461
prt                                002345 constant         label                dcl 381 ref 361
v_num                              003406 constant         entry                internal dcl 578 ref 346
ellin                              001341 constant         label                dcl 224 ref 191
ds                                 000230 constant         entry                external dcl 1
of                                 002621 constant         label                dcl 467 ref 252 258 434 527
ile                                002607 constant         label                dcl 454 ref 189
inish                              002611 constant         label                dcl 456 ref 178 211 314 386
et                                 003172 constant         entry                internal dcl 524 ref 227 320 403
et_num                             003403 constant         entry                internal dcl 575 ref 224 243 305
et_token                           003324 constant         entry                internal dcl 557 ref 182 342 577
nput                               001251 constant         label                dcl 207 ref 219
nsert                              001357 constant         label                dcl 234 ref 184
ine_end                            002601 constant         label                dcl 447 ref 443
ocate                              001521 constant         label                dcl 272 ref 186
_eof                               001432 constant         label                dcl 252 ref 248
exlin                              001373 constant         label                dcl 243 ref 188
ext                                001020 constant         label                dcl 174 ref 180 201 230 239 268 301 318 331 399 412
                                                                                  463 470
oline                              001762 constant         label                dcl 317 ref 308
xarg                               002114 constant         label                dcl 342 ref 347
edit                               001005 constant         label                dcl 172 ref 214
input                              001236 constant         label                dcl 205 ref 163 195 418
rint                               001673 constant         label                dcl 305 ref 187
rint1                              001712 constant         label                dcl 311 ref 297 321 450
ut                                 003157 constant         entry                internal dcl 515 ref 216 234 246 277 319 402
esetread                           003242 constant         entry                internal dcl 548 ref 200 330 397 469
```

```
retype                         001360 constant      label          dcl 236 ref 185
save                           002706 constant      entry          internal dcl 494 ref 455 462
skipch                         002422 constant      label          dcl 392 ref 349
switch                         003225 constant      entry          internal dcl 536 ref 279 300 411 42
top                            002466 constant      label          dcl 410 ref 193
wsave                          002616 constant      label          dcl 461 set ref 192

THERE WERE NO NAMES DECLARED BY CONTEXT OR IMPLICATION.

STORAGE REQUIREMENTS FOR THIS PROGRAM.

          Object    Text      Link      Symbol    Defs      Static
Start          0       0      4364        4450    4103        4374
Length      4674    4103        64         210     261           0

BLOCK NAME                          STACK SIZE     TYPE                WHY NONQUICK/WHO SHARES STACK FRAME
eds                                        642 external procedure  is an external procedure.
on unit on line 135                         64 on unit
copy                                           internal procedure  shares stack frame of external procedure eds.
save                                           internal procedure  shares stack frame of external procedure eds.
put                                            internal procedure  shares stack frame of external procedure eds.
get                                            internal procedure  shares stack frame of external procedure eds.
switch                                         internal procedure  shares stack frame of external procedure eds.
resetread                                      internal procedure  shares stack frame of external procedure eds.
get_token                                      internal procedure  shares stack frame of external procedure eds.
get_num                                        internal procedure  shares stack frame of external procedure eds.
clean_up                                    60 internal procedure  is called by several nonquick procedures.

STORAGE FOR AUTOMATIC VARIABLES.

STACK FRAME                     LOC IDENTIFIER                   BLOCK NAME
eds                          000100 arg_count                    eds
                             000101 break                        eds
                             000102 brk1                         eds
                             000103 buffer                       eds
                             000170 changes_occurred             eds
                             000171 code                         eds
                             000172 count                        eds
                             000173 csize                        eds
                             000174 edct                         eds
                             000175 dir_name                     eds
                             000247 entry_name                   eds
                             000260 exptr                        eds
                             000262 from_ptr                     eds
                             000264 globsw                       eds
                             000265 i                            eds
                             000266 ij                           eds
                             000267 indf                         eds
                             000270 indt                         eds
                             000271 j                            eds
                             000272 k                            eds
                             000273 l                            eds
                             000274 line_buffer                  eds
                             000361 line1                        eds
                             000362 located                      eds
                             000363 m                            eds
                             000364 n                            eds
                             000365 sname_lth                    eds
                             000366 sname_ptr                    eds
                             000370 source_count                 eds
```

```
                          000372 source_ptr                  eds
                          000374 temp_segs                   eds
                          000400 tlin                        eds
                          000466 tkn                         eds
                          000470 to_ptr                      eds
                          000556 token_lth                   get_token
                          000557 white_lth                   get_token

THE FOLLOWING EXTERNAL OPERATORS ARE USED BY THIS PROGRAM.
alloc_cs            call_ext_out_desc   call_ext_out        call_int_this       call_int_other      return
enable              shorten_stack       ext_entry           int_entry           set_cs_eis          index_cs_eis

THE FOLLOWING EXTERNAL ENTRIES ARE CALLED BY THIS PROGRAM.
com_err_                      cu_$arg_count                 cu_$arg_ptr                   cv_dec_
expand_pathname_              get_temp_segments_            hcs_$initiate_count           hcs_$make_seg
hcs_$set_bc_seg               hcs_$terminate_noname         hcs_$truncate_seg             ioa_
iox_$control                  iox_$get_line                 iox_$put_chars                pathname_
release_temp_segments_

THE FOLLOWING EXTERNAL VARIABLES ARE USED BY THIS PROGRAM.
error_table_$noarg            error_table_$noentry          error_table_$too_many_args    iox_$user_input
iox_$user_output


   LINE    LOC       LINE    LOC       LINE    LOC       LINE    LOC       LINE    LOC       LINE    LOC       LINE    LOC
      1 000227        102 000235        103 000245        104 000247        105 000264        108 000265        109 000272
    110 000300        112 000303        113 000305        114 000334        117 000335        118 000355        119 000357
    120 000411        125 000412        126 000442        127 000444        128 000476        133 000477        134 000501
    135 000514        139 000536        141 000600        143 000610        144 000662        149 000663        150 000704
    151 000706        152 000732        153 000736        155 000737        156 000741        160 000743        161 000746
    162 000752        163 000775        165 000776        166 001001        172 001005        174 001020        176 001043
    177 001045        178 001074        180 001075        181 001100        182 001102        184 001103        185 001110
    186 001115        187 001122        188 001127        189 001134        190 001141        191 001146        192 001153
    193 001160        194 001165        195 001172        199 001177        200 001233        201 001235        205 001236
    207 001251        209 001274        210 001276        211 001322        214 001323        216 001332        217 001333
    218 001335        219 001340        224 001341        226 001342        227 001352        228 001353        229 001355
    230 001356        234 001357        236 001360        238 001363        239 001372        243 001373        244 001374
    245 001376        246 001401        247 001402        248 001411        249 001414        251 001431        252 001432
    253 001435        254 001436        256 001453        257 001455        258 001460        260 001461        261 001462
    262 001464        263 001466        264 001470        265 001477        267 001514        268 001520        272 001521
    273 001524        274 001525        275 001527        276 001531        277 001534        278 001535        279 001541
    280 001542        281 001547        282 001550        284 001552        285 001571        286 001572        287 001606
    288 001613        289 001627        290 001633        291 001635        293 001651        294 001654        295 001660
    296 001665        297 001667        299 001670        300 001671        301 001672        305 001673        306 001674
    307 001676        308 001711        311 001712        312 001733        313 001735        314 001761        317 001762
    318 001764        319 001766        320 001767        321 001770        325 001771        326 001772        327 001775
    329 001777        330 002025        331 002026        333 002027        334 002032        335 002037        336 002055
    337 002056        338 002076        339 002104        340 002111        341 002112        342 002114        344 002115
    345 002122        346 002132        347 002133        349 002134        351 002136        352 002137        353 002143
    354 002146        355 002150        356 002152        358 002166        359 002173        360 002177        361 002202
    363 002203        365 002227        366 002230        368 002246        370 002265        371 002272        372 002277
    373 002304        374 002306        375 002310        377 002312        379 002335        380 002341        381 002345
    383 002347        384 002366        385 002370        386 002414        389 002415        390 002417        392 002422
    393 002425        394 002427        395 002431        397 002457        399 002460        401 002461        402 002463
    403 002464        404 002465        410 002466        411 002467        412 002470        416 002471        417 002472
    418 002473        422 002474        423 002476        424 002477        425 002500        426 002503        427 002507
    428 002524        429 002530        431 002534        432 002535        433 002537        434 002541        436 002542
    437 002544        438 002546        440 002553        441 002563        443 002571        445 002575        446 002577
```

```
447 002601    449 002604    450 002606    454 002607    455 002610    456 002611    457 002615
461 002616    462 002617    463 002620    467 002621    468 002623    469 002646    470 002647
477 002650    478 002651    480 002657    481 002661    482 002662    484 002665    485 002667
487 002701    488 002703    489 002705    494 002706    495 002707    496 002713    497 002752
498 002754    499 003027    502 003030    503 003036    504 003054    506 003075    507 003077
510 003156    515 003157    517 003160    518 003166    519 003170    520 003171    524 003172
526 003173    527 003174    528 003177    530 003214    531 003220    532 003223    533 003224
536 003225    538 003226    539 003230    540 003232    541 003234    542 003236    543 003240
544 003241    548 003242    551 003243    552 003275    553 003323    557 003324    562 003325
563 003327    564 003346    565 003352    566 003353    567 003372    568 003376    569 003401
570 003402    575 003403    577 003404    576 003405    580 003407    581 003426    582 003431
586 003432    589 003440    590 003462    592 003501
```

APPENDIX C

MULTICS SUBSYSTEMS

The Multics system offers many special subsystems, designed to serve a particular set of users or perform a particular set of tasks. Some of these subsystems are already familiar to you--the Qedx and Emacs text editor systems, the input/output system. Various other subsystems are described briefly here. For detailed information on any of them, see individual manuals.

DATA BASE MANAGER

The Multics Data Base Manager (MDBM) supports the description and processing of data bases of widely varying sizes and organizations, and provides a large measure of data independence. It consists of an integrated set of functions which offer a full range of data base retrieval and update capabilities, and it is written to interface with any programming language that supports a call statement. The MDBM offers a powerful, extremely flexible method of structuring and manipulating data bases: the Multics Relational Data Store (MRDS).

MRDS supports the relational model of data base organization, in which data relationships are represented by means of formal algebraic entities. It allows you to structure and access data without concern for how or where it is actually stored. A special MDBM query language called LINUS (described later in this section) provides comprehensive query capabilities for MRDS data base users.

Data bases reside within the Multics storage system and are protected by all of the security features inherent in the Multics virtual memory environment.

FAST

The Multics FAST subsystem is a simple-to-use, low-cost user interface for creating and running BASIC and FORTRAN programs. The Multics FAST command language is a subset of Multics commands with additional commands for manipulating line-numbered text.

GCOS ENVIRONMENT SIMULATOR

The GCOS environment simulator, together with several Multics facilities, permits GCOS batch-processing jobs to be run under the control of Multics and provides some job-scheduling facilities. Invoked via the Multics gcos command, the simulator immediately runs one GCOS job in your process. Your terminal is treated as if it were the GCOS operator's console.

It's also possible to simulate GCOS time-sharing usage, by invoking the Multics gcos_tss (gtss) command.

GRAPHICS

The Multics Graphics System provides a general purpose interface through which user or application programs can create, edit, store, display, and animate graphic material. It is a terminal-independent system, which means that a program written for one type of graphic terminal is operable without modification on another terminal having similar capabilities.

LOGICAL INQUIRY AND UPDATE

The Logical Inquiry and Update System (LINUS) is a facility for accessing MRDS data bases. The complete data base management capability provided by LINUS includes both retrieval and update operations.

LINUS makes use of a high-level nonprocedural language called LILA (LINUS Language) that can be understood by individuals who aren't necessarily computer specialists.

REPORT PROGRAM GENERATOR

The Multics Report Program Generator (MRPG) is a language translator used to generate a PL/I source program from an MRPG source program, with the purpose of generating formatted reports.

SORT/MERGE

The Sort/Merge subsystem provides generalized file sorting and merging capabilities, specialized for execution by user-supplied parameters. Sort orders an unranked file according to the values of one or more specified key fields in the records you are using. Merge collates the contents of up to ten ordered files according to the value of one or more key fields. Input and output files associated with the Sort/Merge subsystem can have any file organization and be on any storage medium. Records can be either fixed or variable length.

WORDPRO

The Multics word processing system, WORDPRO, consists of a set of commands that assist you in the input, update, and maintenance of documents. The commands provide tools for text editing and formatting, Speedtype, dictionaries for hyphenation and spelling, list processing, and electronic mail.

An important part of the WORDPRO system is the compose command, which is used for formatting manuscripts, and has programmable requests that make it a minor programming language.

APPENDIX D

THE EDM EDITOR

The Edm editor is a simple Multics context editor which is used for creating and editing ASCII segments. Edm is less sophisticated than Qedx, and far less sophisticated than Emacs, so if you are already comfortable with one of these editors, this appendix will not be very useful to you. However, if you would like to learn how to use a simpler editor, this appendix will help.

To invoke the Edm editor, you type:

```
edm pathname
```

when pathname identifies the segment to be either edited or created.

The Edm editor operates in one of two principal modes: edit or input. If pathname identifies a segment that is already in existence, Edm begins in edit mode. If pathname identifies a segment that does not exist, or if pathname is not given, Edm begins in input mode. You can change from one mode to the other by issuing the mode change character: a period (followed by a carriage return) which is the only character on a line. For verification, Edm announces its mode by responding "Edit." or "Input." when the mode is entered.

The Edm requests assume that the segment consists of a series of lines and has a conceptual pointer to indicate the current line. (The "top" and "bottom" lines of the segment are also meaningful.) Some requests explicitly or implicitly cause the pointer to be moved; other requests manipulate the line currently pointed to. Most requests are indicated by a single character, generally the first letter of the name of the request.

REQUESTS

Various Edm requests and their functions are listed below. Detailed descriptions of these requests are given later in this section. This list does not include all of the Edm requests; it identifies only those requests that you will need as you begin using this editor. For a complete listing and description of all the Edm requests, see the MPM Commands.

-	backup
=	print current line number
,	comment mode
.	mode change
b	bottom
d	delete

f	find
i	insert
k	kill
l	locate
n	next
p	print
q	quit
r	retype
s	substitute
t	top
v	verbose
w	write

GUIDELINES

The following list offers helpful suggestions about the use of Edm.

1. It is useful to remember that the editor makes all changes on a copy of the segment, not on the original. Only when you issue a w (write) request does the editor overwrite the original segment with the edited version. If you type a q (quit) without a preceding w, the editor warns you that editing will be lost and the original segment will be unchanged, and gives you the option of aborting the request.

2. You should not issue a QUIT signal (press ATTN, BRK, INTERRUPT, etc.) while in the editor unless you are prepared to lose all of the work you have done since the last w request. However, if a QUIT signal is issued, you may return to Edm request level without losing your work by issuing the program_interrupt command.

3. If you have a lot of typing or editing to do, it is wisest to occasionally issue the w request to ensure that all the work up to that time is permanently recorded. Then, if some problem should occur (with the system, the telephone line, or the terminal), you only lose the work done since your last w request.

4. You should be sure that you have switched from input mode to edit mode before typing editing requests, including the w and q requests. If you forget, the editing requests are stored in the segment, instead of being acted upon. You then have to locate and delete them.

5. As you become more familiar with the use of Edm, you may conclude that it provides verification responses more often than necessary, thus slowing you down. You may use the k (kill) request to "kill" the verification response. However, once you feel confident enough to use the k request, you are probably ready to begin using the more sophisticated editor, Qedx. The Qedx editor provides you with a repertoire of more concise and powerful requests, permitting more rapid work.

REQUEST DESCRIPTIONS

The following Edm requests are the ones that you will find most useful as you begin working with this editor. Examples are included to help you see the practical use of each request.

Backup (-) Request

The backup request moves the pointer backward (toward the top of the segment) the number of lines specified, and prints the line to show the location of the pointer. For example, if the pointer is currently at the bottom line of the following:

```
get list (n1, n2);
sum = n1 + n2;
put skip;
put list ("The sum is:", sum);
```

and you want the pointer at the line beginning with the word "sum," you type:

```
!  -2
   sum = n1 + n2;
```

If you don't specify a number of lines with the backup request, the pointer is moved up one line. (Typing a space between the backup request and the integer is optional.)

Print Current Line Number (=) Request

The print current line number request tells you the number of the line the pointer is currently pointing to (all the lines in a segment are implicitly numbered by the system--1, 2, 3,..., n).

Whenever you want to check the implicit line number of the current line, you issue this request and Edm responds with a line number.

```
!  =
   143
```

Comment Mode (,) Request

When you invoke the comment mode request, Edm starts printing at the current line and continues printing all the lines in the segment in comment mode until it reaches the end of the segment, or until you type the mode change character (a period) as the only entry on a line.

To print the lines in comment mode means that Edm prints a line without the carriage return, switches to input mode, and waits for your comment entry for that line. When you give your comment line and a carriage return, Edm repeats the process with the next line.

If you have no comment for a particular line, you type only a carriage return and Edm prints the next line in comment mode. When you want to leave comment mode and return to edit mode, you type--as your comment--the mode change character (a period).

Programmers will find that the comment mode request gives them a fast and easy way to put comments in their programs.

Mode Change (.) Request

The mode change request allows you to go from input mode to edit mode or vice versa simply by typing a period as the only character on a line. This request is also the means by which you leave the comment mode request and return to edit mode.

For example, when you finish typing information into a segment, you must leave input mode and go to edit mode in order to issue the write (w) request and save the information.

```
! last line of segment
! .
  Edit.
! w
```

Bottom (b) Request

The bottom request moves the pointer to the end of the segment (actually sets the pointer _after_ the last line in the segment) and switches to input mode. This request is particularly helpful when you have a lot of information to type in input mode; if you see some mistakes in data previously typed, you can switch to edit mode, correct the error, then issue the bottom request and continue typing your information.

```
! red
! oramge
! yellow
! green
! .
  Edit.
! -2
  oramge
! s/m/n/
  orange
! b
  Input.
! blue
```

Delete (d) Request

This request deletes the number of lines specified. Deletion begins at the current line and continues according to your request. For example, to delete the current line plus the next five lines, you type:

```
! d6
```

If you issue the delete request without specifying a number, only the current line is deleted. (That is, you may type either d or d1 to delete the current line.)

After a deletion, the pointer is set to an imaginary line following the last deleted line but preceding the next nondeleted line. Thus, a change to input mode would take effect before the next nondeleted line.

Find (f) Request

The find request searches the segment for a line beginning with the character string you designate. The search begins at the line following the current line and continues, wrapping around the segment from bottom to top, until the string is found or until the pointer returns to the current line; however, the current line itself is not searched. If the string is not found, Edm responds with the following error message:

```
     Edm:  Search failed.
```

If the string is found and you are in verbose mode, Edm responds by printing the first line it finds that begins with the specified string.

```
   !  f If
      If the string is found and you are in verbose mode, Edm responds by
```

When you type the string, you must be careful with the spacing. A single space following the find request is not significant; however, further leading and embedded spaces are considered part of the specified string and are used in the search.

In the find request, the pointer is either set to the line found in the search or remains at the current line if the search fails. Also, if you issue the find request without specifying a character string, Edm searches for the string requested by the last find or locate (l) request.

Insert (i) Request

The insert request allows you to place a new line of information _after_ the current line.

If you invoke the insert request without specifying any new text, a blank line is inserted after the current line. If you type text after the insert request, you must be careful with the spacing. One space following the insert request is not significant, but all other leading and embedded spaces become part of the text of the new line.

For example, if the pointer is at the top line of the following:

```
     sum = n1 + n2;
     put list ("The sum is:", sum);
```

and you issue the following insert request:

```
   !  i put skip;
```

the result is:

```
     sum = n1 + n2;
     put skip;
     put list ("The sum is:",sum);
```

If you want to insert a new line at the beginning of the segment, you first issue a top (t) request and then an insert request.

Kill (k) Request

The kill request suppresses the Edm responses following the change (c), find (f), locate (l), next (n), and substitute (s) requests. To restore responses to these requests, you issue the verbose (v) request.

It is recommended that as a new user you not use the kill request until you are thoroughly familiar with Edm. The responses given in verbose mode are helpful; they offer an immediate check for you by allowing you to see the results of your requests.

Locate (l) Request

The locate request searches the segment for a line containing a user-specified string. The locate and find (f) requests are used in a similar manner and follow the same conventions. (Refer to the find request description for details.) With the find request, Edm searches for a line beginning with a specified string; with the locate request, Edm searches for a line containing--anywhere--the specified string.

Next (n) Request

The next request moves the pointer toward the bottom of the segment the number of lines specified. If you invoke the next request without specifying a number, the pointer is moved down one line. When you do specify the number of lines you want the pointer to move, the pointer is set to the specified line. For example, if you type:

```
! n4
```

the pointer is set to the fourth line after the current line. The Edm editor responds, when in verbose mode, by typing you-specified line.

Print (p) Request

The print request prints the number of lines specified, beginning with the current line, and sets the pointer to the last printed line. If you do not specify a number of lines, only the current line is printed.

If you want to see the current line and the next three lines, you type:

```
! p4
  current line
  first line after current line
  second
  third
```

In Edm, every segment has two imaginary null lines, one before the first text line and one after the last text line. When you print the entire segment, these lines are identified as "No line" and "EOF" respectively.

Quit (q) Request

The quit request is invoked when you want to exit from Edm and return to command level.

For your convenience and protection, Edm prints a warning message if you do not issue a write (w) request to save your latest editing changes before you issue the quit request. The message reminds you that your changes will be lost and asks if you still wish to quit.

```
! q
  Edm: Changes to text since last "w" request will be lost if you quit;
  do you wish to quit?
```

If you answer by typing no, you are still in edit mode and can then issue a write (w) request to save your work. If you instead answer by typing yes, you exit from Edm and return to command level.

Retype (r) Request

The retype request replaces the current line with a different line typed by you.

One space between the retype request and the beginning of the new line is not significant; any other leading and embedded spaces become part of the new line. To replace the current line with a blank line, you type the retype request and a carriage return.

Substitute (s) Request

The substitute request allows you to change every occurrence of a particular character string with a new character string in the number of lines you indicate. If you are in verbose mode (in which Edm prints responses to certain requests), Edm responds by printing each changed line. If the original character string is not found in the lines you asked Edm to search, Edm responds:

```
Edm:  Substitution failed.
```

For example, if the pointer is at the top line of the following:

```
get list (n1, n2);
sum = n1 + n2;
put skip;
put list ("The sum is:", sum);
```

and you want to search the next three lines and change the word "sum" to "total," you type:

```
! s4/sum/total/
  total = n1 + n2;
  put list ("The total is:", total);
```

The four lines searched by the editor are the current line plus the next three. (The search always begins at the current line.) If you do not specify the number of lines you want searched, Edm only searches the current line. If you do not specify an original string, the new string is inserted at the beginning of the specified line(s).

Notice in the example that a slash (/) was used to delimit the strings. You may designate as the delimiter any character that does not appear in either the original or the new string.

Top (t) Request

The top request moves the pointer to an imaginary null line immediately above the first text line in the segment. (See the print request description concerning imaginary null lines in Edm.)

An insert (i) request immediately following a top request allows you to put a new text line above the "original" first text line of the segment.

Verbose (v) Request

The verbose request causes Edm to print responses to the change (c), find (f), locate (l), next (n), and substitute (s) requests.

Actually, you do not need to issue the verbose request to cause Edm to print the responses; when you invoke Edm, the verbose request is in effect. The only time you need to issue the verbose request is to cancel a previously issued kill (k) request.

Write (w) Request

The write request saves the most recent copy of a segment in a pathname you specify. (The pathname can be either absolute or relative.)

If you do not specify a pathname, the segment is saved under the name used in the invocation of the edm command. When saving an edited segment without specifying a pathname, the original segment is overwritten (the previous contents are discarded) and the edited segment is saved under the original name.

If you do not specify a pathname _and_ you did not use a pathname when you invoked the edm command, an error message is printed and Edm waits for another request. If this happens, you should reissue the write request, specifying a pathname.

INDEX

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	LEVEL 68 INTRODUCTION TO PROGRAMMING ON MULTICS	**ORDER NO.**	AG90-03
		DATED	JULY 1981

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME ________________________ **DATE** ____________

TITLE ________________________

COMPANY ________________________

ADDRESS ________________________

____

PLEASE FOLD AND TAPE–
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 155 Gordon Baker Road, Willowdale, Ontario M2H 3N7
In the U.K.: Great West Road, Brentford, Middlesex TW8 9DH
In Australia: 124 Walker Street, North Sydney, N.S.W. 2060
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

32397, 5C981, Printed in U.S.A.