LEVEL 68 MULTICS INTRODUCTORY USERS' GUIDE

SUBJECT

Basic Introduction to Multics, Intended as a Guide for New Users

SPECIAL INSTRUCTIONS

For a more complete description on using the Multics System, refer to the *Multics Programmers' Manual* (MPM):

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

This manual supersedes AL40, Revision 0, which was titled, *Multics Users' Guide*. The Manual has been extensively revised and does not contain change bars.

SOFTWARE SUPPORTED

Multics Software Release 6.0

ORDER NUMBER

AL40, Rev. 1

July 1977

Honeywell

PREFACE

The purpose of this manual is to provide programmers and other users with a basic introduction to Multics use, a practice workbook that guides new users through their first sessions at a terminal. The facilities described have been chosen either because they are immediately useful to new users or because they are representative of the system as a whole.

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 the user has become familiar with this introductory guide. The MPM consists of the following individual documents:

Reference Guide
Commands and Active Functions
Subroutines
Subsystem Writers' Guide
Peripheral Input/Output

Throughout this manual, references are frequently made to portions of the MPM. For convenience, these references are as follows:

Document	Referred To In Text As
Reference Guide (Order No. AG91)	MPM Reference Guide
Commands and Active Functions (Order No. AG92)	MPM Commands
Subroutines (Order No. AG93)	MPM Subroutines
Subsystem Writers' Guide (Order No. AK92)	MPM Subsystem Writers' Guide
Peripheral Input/Output (Order No. AX49)	MPM I/O

File No.: 1L13

AL40

Other Multics manuals of interest to new users are listed below. For a complete, up-to-date list of Multics manuals, contact the Honeywell Distribution Center, 40 Guest Street, Brighton, MA 02135 and ask for the Honeywell Publications Catalog, Order No. AB81.

Multics Pocket Guide -- Commands and Active Functions	Order No. AW17
Multics PL/I Language Specification	Order No. AG94
Multics PL/I Reference Manual	Order No. AM83
Multics FORTRAN Reference Manual	Order No. AT58
Multics COBOL Users' Guide	Order No. AS43
Multics COBOL Reference Manual	Order No. AS44
Multics BASIC	Order No. AM82
Multics APL	Order No. AK95
Multics FAST Subsystem Users' Guide	Order No. AU25
Multics GCOS Environment Simulator	Order No. AN05
Multics Sort/Merge	Order No. AW32
Multics Graphics System	Order No. AS40
Multics Administrators' Manual (MAM)-- System Administrator	Order No. AK50
MAM -- Project Administrator	Order No. AK51
MAM -- Registration and Accounting Administrator	Order No. AS68

CONTENTS

ILLUSTRATIONS

SECTION I

INTRODUCTION

Multics is a general purpose computer system developed to serve a wide variety of users. It supports, simultaneously, end users of transaction-oriented applications, word processing, and application and system programming development. It supports a wide variety of languages, most notably PL/I, COBOL, FORTRAN, BASIC, and APL. The Multics system multiplexes a central computer among the jobs of many users, each of whom accesses Multics from a terminal. With Multics, users have facilities that allow them to edit, compile, debug, and run programs in one continuous, interactive session. Each Multics user can structure information, manipulate it, and simultaneously share it with other users.

But Multics is more than a mere time-sharing system. It supports diverse subsystems, such as GCOS and FAST (a facility based on the full Multics system, offering BASIC and FORTRAN compilers) and offers two data base managers (Multics Integrated Data Store and Multics Relational Data Store), a graphics system, and extensive word processing tools to mention just a few facilities.

This manual helps the new user become familiar with the Multics system. Basic Multics concepts, such as the storage system and access control, are described briefly. However, this manual offers a limited discussion of Multics concepts. Instead, it focuses on those facilities that all users need daily, regardless of the nature of their terminal work.

Thus, the first topic in this document is the procedure by which the new user enters and exits the system plus a brief explanation of some terminal and Multics conventions (Section II). Next, the user is introduced to the Multics environment (Section III), that is, the storage system, naming conventions, and appropriate terminology. Once the user is familiar with the environment, the Multics commands are described (Section IV). The Multics system supports user-defined commands as well as system commands. Both types of commands and various command conventions are included in the commands description. As a special aid for the new user, certain frequently called system commands are also described (Section V).

To get data on the system, modify the data, and save it, the user can call one of the Multics editors. One Multics context editor, qedx, gives the beginner the ability to manipulate data through many simple requests (Section VI) and also offers the user a powerful interpretive language through more advanced requests (not covered in this manual). (Another Multics context editor, edm, is described in Appendix D.)

Of particular interest to the programmer is the description of programming in the Multics environment. While this manual does not teach programming, it does describe the ease with which programming can be accomplished on Multics (Section VII). Programmers can write a source program, edit it, compile it, debug and run the object program--a portion at a time if they choose--and do it all online in one terminal session.

After describing the programming environment, the manual presents a brief discussion of the Multics access control concepts (Section VIII). On Multics, each user controls the access that other users have to programs and data he creates. Different access modes may be assigned to different users of the same program. The user who creates the program can set, change, and delete access rights just by invoking simple system commands. (Examples of these commands are also given in Section VIII.)

Another important, useful feature of Multics is online communication (Section IX). Through various commands, users can communicate with one another instantaneously, send mail, or even check to see who else is online at the moment.

The absentee use of Multics is briefly described in Section X. This facility is similar to batch processing on other systems. On Multics, the command language and syntax are the same for absentee and interactive usage.

The final section of this manual (Section XI) serves two purposes: it suggests further information to pursue for the novice, and it puts this manual in perspective with respect to the Multics system. The reader should understand that the material presented in this document represents only a fraction of the Multics system capabilities. However, this material should provide the new user with a challenge for two or three brief terminal sessions, after which Section XI and Appendices B and C will help to suggest further information to explore.

The new user will find Appendix A particularly helpful. It is a glossary of basic Multics terminology.

SECTION II

HOW TO ACCESS THE MULTICS SYSTEM

Before users can gain access to Multics, they must be registered on the system by the site system administrator and allowed access to a particular project by the administrator of that project. The system administrator assigns each user a unique two-part identification, consisting of a person identification (called a Person_id) and a project identification (called a Project_id). The Person_id is generally a variation of the user's surname; the Project_id is an arbitrary name for a project that is registered on the system for accounting purposes. In addition, the system administrator assigns a special password for each user. For example, if Tom Smith were a new user, the system administrator could assign TSmith as his Person_id and ProjA as his Project_id.

Notice that the Person_id (TSmith) contains no blanks but does contain capital letters. The Multics system distinguishes between uppercase and lowercase characters; the exact capitalization must be used or else the Person_id is not recognized by the system.

LOG-IN PROCEDURE

After the user has dialed the appropriate telephone number and a connection has been established between Multics and the user's terminal, Multics prints a message giving the number of the current system, the location of the system, the actual number of users logged in, and the number of users the system is currently accepting.

```
Multics XX-x: PCO, Phoenix, Az.
Load = 26.0 out of 100.0 units: users = 26
```

At this point, the user issues the login command and his Person_id, separated by a blank.

```
! login TSmith
Password:
```

NOTE: Throughout the interactive examples in this manual, an exclamation mark (!) precedes text typed by the user. This is done only to distinguish user text from system-generated text; users should <u>not</u> actually begin their text with an exclamation mark.

Also, a "carriage return" (moving the typing mechanism to the first column of the next line, called a newline on Multics) is implied at the end of every user-typed line. See the glossary under newline for details.

Multics then requests the user's password. Depending on the user's terminal, the printing of the password is either suppressed or hidden in a string of cover-up characters typed by the system. It is essential that users keep their own password secret to prevent unauthorized use of their programs and data and their account. If the user feels that his password has been compromised, he should notify his project administrator and also immediately change his password.

In fact, one of the first things a new user should do is change his password from the one assigned by the system administrator as part of the registration process (usually just the user's initials in lowercase characters) to one the user makes up. To change the password, the user logs in using the -change_password (or -cpw) control argument. For example, to change Tom Smith's password from tjs to mypass, he would type the following (passwords are shown here for clarity):

```
! login TSmith -cpw
  Password:
! tjs
  New Password:
! mypass
  New Password Again:
! mypass
  Password changed.
```

For future logins, mypass is Tom's password. He can repeat the above procedure and change his password any time he wishes.

If the user ever forgets his password, he must notify the system administrator and request a new password. (Once a password is registered on the system, it is encoded and cannot be decoded by anyone, including the system administrator.)

If the user makes an error during the log-in procedure, the system informs him of it and asks him to try again.

```
  Login incorrect.
  Please try again or type "help" for instructions.
! login TSmith
  Password:
!
```

Each Multics site sets its own limit on how many attempts a user may make to log in before the system automatically disconnects the line to the terminal.

```
Login incorrect.
hangup
```

Project administrators may interpose another authentication procedure after the user types his password. The format of this procedure is determined by the individual project administrator.

After the user has successfully typed his password, the system responds with information regarding the user's last login.

```
TSmith ProjA logged in 06/07/77  0937.5 mst Tue from ASCII terminal "234".
Last login 06/06/77  1359.8 mst Mon from ASCII terminal "234".
```

The log-in statistics can be used to detect unauthorized use of the user's name and password on previous logins, since the user knows when he was last logged in. In addition, the system informs the user of unsuccessful attempts to gain access to the system through his password. (Typing errors made by the user also count as unsuccessful attempts; see Section IV for how to correct typing errors.)

```
! login TSmith
  Password:
!
  Your password was given incorrectly at 06/07/77  0937.1 mst Tue from ASCII
    terminal "234".
  You are protected from preemption until 1037.
  TSmith ProjA logged in 06/07/77  0937.5 mst Tue from ASCII terminal "234".
  Last login 06/06/77  1359.8 mst Mon from ASCII terminal "234".
```

(The message from the system about an incorrect password is given in a slightly different format if the user has a mailbox and uses a start_up.ec segment. For information on mailboxes and start_up.ec segments refer to Sections IX and XI respectively.)

These statistics are followed by the "message of the day." This message is a convenient way to tell all system users important news, including schedules, information on new commands, and latest documentation.

```
A new PL/I compiler was installed; type: help pl1_new
Rates for CPU usage have changed; type: help prices
```

The last line of system-generated text in the log-in sequence is the ready message. This message is printed to indicate that Multics is at command level and ready to receive the next command. The ready message consists of the letter "r" followed by the time of day and three numbers that reflect system resource usage.

```
r 937 1.314 1.332 30
```

These usage numbers identify virtual CPU time, which is the actual CPU time consumed by the user since the last ready message minus some supervisor execution time; memory units, which is an approximation of the amount of memory he has used since his last ready message; and the actual number of pages of information brought into main memory from secondary storage since the last ready message. For more information about the ready message, refer to the ready command in the MPM Commands.

The complete log-in sequence for Tom Smith, assuming no one has attempted to use his password since his last login, would be:

```
! login TSmith
  Password:
!
  TSmith ProjA logged in 06/07/77  0937.5 mst Tue from ASCII terminal "234".
  Last login 06/06/77  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 937 1.314 1.332 30
```

Under certain circumstances, the user may be denied access to the system even though he has correctly logged in. For example, the system administrator may not yet have registered the user, the user may have exceeded the resource limits set for him by the project administrator, or the system may temporarily be full. In any case, if the user cannot get on the system, he generally receives a message from Multics telling him the reason he cannot log in and what steps, if any, he should take.

LOG-OUT PROCEDURE

When the user has completed his work, he breaks the connection between his terminal and the Multics system by issuing the logout command. The system responds by printing the identification of the user, the date and time of the logout, and the total CPU time and memory units used.

```
! logout
  TSmith ProjA logged out 06/07/77  1249.4 mst Tue
  CPU usage 17 sec, memory usage 103.1 units.
  hangup
```

Some projects have a log-in time limit after which a user is subject to an automatic logout. In such cases, Multics prints a warning several minutes before a session is automatically terminated. Unless the user is automatically logged out by the system, he should always log out before leaving the terminal, to avoid wasting computer time and preventing others from logging in.

SECTION III

MULTICS ENVIRONMENT

One major component of the Multics environment, the virtual memory, allows the user to forget about physical storage of information; he does not need to be concerned with, or even aware of, where his information is within the system or on what device it resides. However, the new user does need to have a basic grasp of another major component of the Multics environment, the storage system, before he can begin to understand the system environment.

STORAGE SYSTEM

One good way to visualize the storage system is to consider it a "tree-structured" hierarchy of directory segments. The basic unit of information within the storage system is the segment; it may contain a collection of program instructions or data, or it may be empty (a null segment). Some segments serve as catalogs of other segments beneath them in the tree structure, listing the attributes of the subordinate segments; these cataloging segments are called directory segments. All the other segments are called nondirectory segments. However, by convention directory segments are called simply "directories"; nondirectory segments, simply "segments."

At the beginning of the tree is the root directory; all other directories and segments emanate from the root directory. For example, Figure 3-1 shows user Tom Smith and his project, ProjA, in relation to the root. (Directories are represented by rectangles and segments by circles.) Notice the two directories immediately under the root (sss and udd). The sss directory is one of several library directories that catalog all the system commands and subroutines. The udd directory is a catalog of project directories. It contains one directory entry for each project on the system. Likewise, each project directory normally contains one directory for each user on that project.

NAMING CONVENTIONS

The actual name of any segment or directory reflects its position in the hierarchy in relation to the root directory. This name, called the pathname, shows the "path" from the root directory to the specific segment or directory. Each name between the root and the specific segment or directory indicates another level in the directory hierarchy. To refer to a particular segment or directory, the user must list these names in the proper order (i.e., beginning with the root and coming down) and must include a greater-than character between each name. The greater-than character (>) is used in Multics pathnames to denote hierarchy levels.

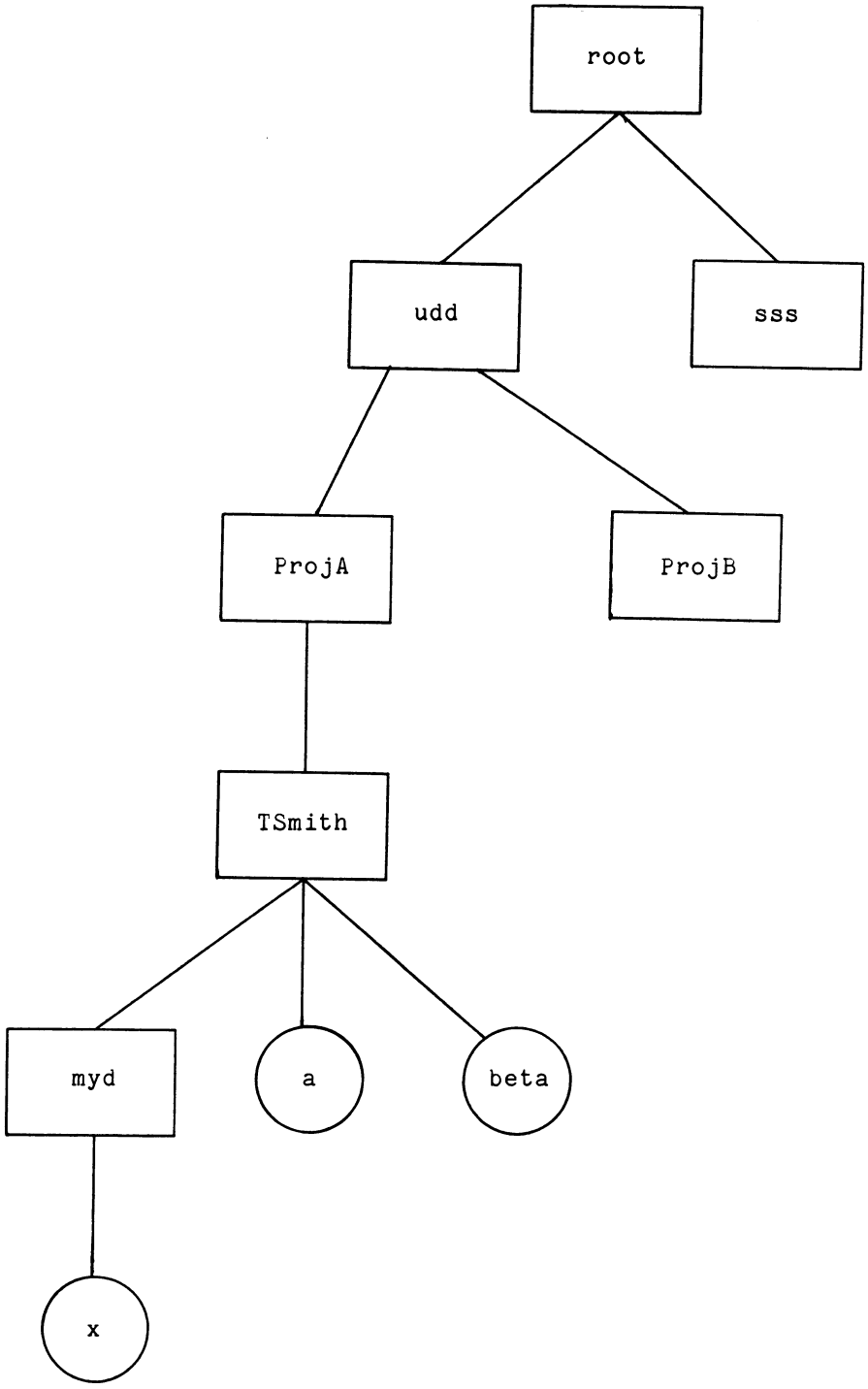

Figure 3-1. Hierarchical Storage System

The pathname for segment x in Tom Smith's myd directory is:

```
>udd>ProjA>TSmith>myd>x
```

which is read as "greater than udd, greater than ProjA, greater than TSmith, greater than myd, greater than x."

By convention, the word "root" is omitted since it would be the first entry in every pathname. The pathname in the above example is called an absolute pathname because it identifies the "absolute" or complete path between the root and the specific segment. For the user's convenience, Multics also accepts a shortened version of the pathname, called the relative pathname, that identifies the specific segment in relation to the current working directory.

The working directory is simply the directory in which the user is currently working; it identifies his current location within the storage system (see also "Working Directory Concept" below). Because the system keeps track of a user's working directory, the user needs to identify only the names between his working directory and the specific segment. Any pathname the user types that does not begin with the greater-than character is considered relative to his working directory. Thus, the relative pathname for segment x when TSmith is the working directory is:

```
myd>x
```

and the relative pathname for segment x when myd is the working directory is simply:

```
x
```

Entrynames

Each individual name in the pathname is called an entryname. Entrynames must be unique within any one directory. For example, Tom Smith can use "test" as an entryname in the TSmith directory and also in the myd directory; furthermore, he could also use "test" as a directory name and have a "test" segment in this directory. However, he cannot use "test" as the name for two different segments in the same directory.

An entryname is a user-assigned identifier, from one to 32 characters long, chosen from the full ASCII [1] character set (excluding the greater-than and less-than characters). The user should avoid the use of a space as part of an entryname; it is permitted but cumbersome because the command language uses spaces to delimit command names and arguments. By convention, the underscore (_) is used to simulate a space for readability, e.g., new_x. The use of certain special symbols, such as the asterisk (*) and equals (=) characters, is not recommended; these characters have special meanings in many Multics standard commands. (Refer to Appendix B for information on these special meanings.)

---

1. American Standard Code for Information Interchange. The full ASCII character set, including any special Multics interpretations is given in Appendix A of the MPM Reference Guide.

Component Names

The use of a period (.) in an entryname has a special meaning on Multics. It separates the name into logical parts called components. Some Multics commands require a specific last component; e.g., the mail command (described in Section IX) requires the user's mailbox entryname to have "mbx" as the last component. Several system conventions (e.g., the star convention and equal convention both described in Appendix B) operate on components. Also, compilers implemented on Multics expect the language name to be the last component of the name of a source segment to be compiled, such as, square_root.pl1 for the name of a PL/I source segment.

Multiple Names

It is permissible--and very convenient--to assign more than one entryname to any segment or directory. For example, the entryname "test" would be easier to type than "part1.test.new_compiler" any time Tom Smith wanted to work with that particular segment.

WORKING DIRECTORY CONCEPT

Each user on Multics functions as though he performs his work from a particular location within the Multics storage system--his working directory.

Working Directory

The working directory is just that--the directory in which the user is currently doing his work. The main purpose of the working directory is convenience; the user does not have to take the time to type in absolute pathnames because the system assumes that any name he types that does not begin with the greater-than character is relative to the current working directory. Thus, the user can type the shorter relative pathname, and the system, knowing the current working directory, supplies the rest of the pathname.

The user can change his working directory simply by invoking a standard Multics command (see Section V). Then he can enter pathnames relative to the new working directory, and the system again supplies the rest of the pathname.

Initial Working Directory

Whenever a user logs in, he logs into a particular directory within the storage system; that is, the system sets his working directory for him. This initial working directory is also known as the home directory. The system "remembers" the pathname of each user's home directory and automatically assigns the user to that directory when he logs in. Generally, the home directory is:

```
>udd>Project_id>Person_id
```

For example, Tom Smith's home directory would be:

```
>udd>ProjA>TSmith
```

Although the user can change his working directory several times during one terminal session, no matter what the identity of his working directory when he logs out, his working directory when he next logs in is always his home directory.

SECTION IV

COMMANDS

A command is something the user types to get the system to perform some action for him. When the user types a command, the system invokes a program (called a command procedure). Most commands have both a long and a short name (e.g., print and pr) and require one or more arguments to specify names of data segments containing information to be acted upon by the command or special kinds of information needed to execute the command. So, to issue a command, the user types the name of the command followed by one or more arguments. For example, the command:

```
print report
```

asks Multics to locate the segment named report within the user's working directory and print it on the terminal.

SYSTEM COMMANDS

System commands are kept in directories called system libraries and are supplied with the Multics system. These commands are generally available to all users of the full Multics system. System commands are well engineered and tested and are equipped to diagnose errors by printing self-explanatory error messages. If the user makes a typing error or types a command that does not exist, an explanatory message is typed on the terminal. For example, if the user wants to invoke the command, print report, and instead types:

```
primt report
```

the system responds with an error message and a ready message (the system returns to command level):

```
Segment primt not found.
r 937 .144 2.402 55
```

USER-WRITTEN COMMANDS

User-written commands are cataloged in directories of the user's choice. These commands may have the same names as system commands; they are distinguished from system commands by their location.

Multics searches in various locations (user directories, system libraries, etc.) in a particular order to find the requested command. The user can alter that order by issuing a system command to change the search rules (see "Setting Search Criteria" in Section XI). By redefining the search rules, the user can determine whether a user-written command or the system command is to be used where both have identical names. For example, when a user types a command at his terminal, like print report, the system first looks through the user's working directory to see if the "print" program exists. If so, that version is executed. If not, the system searches the system- and user-supplied directories (according to the rules specified by the user) for the "print" program. If the "print" program is still not found, the system types an error message and returns to command level.

For more information on what a command is and how to write one, see the MPM Reference Guide ("Command Language" in Section III and "Writing a Command" in Section IV).

STOPPING DURING COMMAND EXECUTION

If the user wants to halt program execution at any time, he can do so by issuing a quit signal. The user invokes the quit signal by pressing the proper key on his terminal (e.g., ATTN, BRK, INTRPT, INTERRUPT). As soon as the system receives this signal, Multics stops executing the program and prints QUIT and a ready message.

For example, when the user issues the print command, he may not need to see the entire segment. So as soon as the system prints the information he needs, he issues a quit signal. The quit signal causes Multics to stop printing the segment and instead print QUIT and a ready message. The ready message printed after a quit signal is slightly different from other ready messages because it contains additional information after the standard numbers:

```
r 938 1.213 3.107 62 level 2, 17
```

This level information indicates that a new command level is established and the interrupted work is being held (preserved as is). Since the system is at command level, therefore ready to accept more commands, the user can either continue the interrupted work or go on to something else.

If the user wishes to continue the work interrupted by the quit signal, he can issue either the start or the program_interrupt command. The start command resumes execution of the original program from the point of interruption. The program_interrupt command resumes execution of the original program from a known, predetermined reentry point.

If the user does not want to continue the interrupted work, he should issue the release command before he issues any other commands. The release command releases the work interrupted (and held) by the quit signal (and drops the level information from the ready message).

For more information on these commands and their relation to the quit signal, see the MPM Commands.

COMMAND CONVENTIONS

The general format of a Multics command is:

command_name argument1 argument2 ... argument*n*

Most Multics commands have various arguments that allow the user to modify command execution to suit his needs. However, the new user does not have to know any argument except a pathname for many Multics commands. By using the simplified command line format (i.e., command_name pathname) the new user can effectively use the Multics system.

For the user's convenience, a brief description of storage system and command conventions is given in Appendix B. Also, for a description of the full command language environment, refer to the MPM Reference Guide.

CORRECTING TYPING ERRORS

There are two special symbols for correcting typing errors, the character delete and the line delete. These symbols may vary, depending on the type of terminal; but generally the number sign (#) is the character-delete symbol, and the commercial at sign (@) is the line-delete symbol. (In fact, Multics allows users to define their own character-delete and line-delete symbols; see "Terminal Settings" in Section XI.)

The character-delete symbol "erases" one previously typed character when typed directly after the error. The line-delete symbol "erases" every character previously typed on the line. Examples of both symbols are given in the login command lines below. Each line is interpreted by Multics as--login TSmith.

```
login TSM#mith

logen T####in TSmith

logen TSmit@login TSmith

kigum T@loge#in TSmith
```

Notice that several successive number signs erase an equal number of typed characters preceding the number sign (see the second line above). However, a single number sign following "white space" (any combination of spaces and horizontal tabs) erases all the white space. This white space rule saves the user the trouble of remembering how many spaces or tabs he just typed and also reduces the number of keystrokes necessary to remove any white space.

For example, assume the user is typing a table in which any column containing information must end with a colon and he types:

col_1<tab><tab><tab>co

at this point the user realizes the colon is missing at the end of the first column. Making use of the white space rule, he types (as the rest of the same line):

###:<tab><tab><tab>col_3:

so that the line he typed:

col_1<tab><tab><tab>co###:<tab><tab><tab>col_3

produces:

col_1: col_3:

Because the white space rule is an exception to the general rule of each character-delete symbol erasing one previously typed character, new users frequently have trouble when attempting to erase one of many white space characters. For example, typing:

col_1<tab><tab><tab>col_3######col_3

produces:

col_1col_3

and not the "col_1<tab><tab>col_3" the user probably wanted.

SECTION V

SAMPLE COMMAND EXECUTION

This section illustrates certain, frequently issued Multics commands. Since qedx, access control, and the online communication commands are fully described elsewhere (Sections VI, VIII, and IX respectively), they are not included here.

Some of the commands shown below have many control arguments that allow the user to tailor the action of the command to his exact needs. No attempt is made to cover such options here; complete descriptions of the commands that follow are given in the MPM Commands.

PRINT WORKING DIRECTORY COMMAND (print_wdir)

The print working directory command (invoked by typing print_wdir or pwd) requests that the system print the name of the working directory. Multics responds by printing the absolute pathname of the user's current working directory.

```
! pwd
  >udd>ProjA>TSmith
  r 938 1.347 2.315 41
```

CHANGE WORKING DIRECTORY COMMAND (change_wdir)

The change working directory command (invoked by typing change_wdir or cwd) redefines the working directory. To change his working directory, the user types the cwd command followed by the pathname of the directory he wishes to redefine as his working directory. The following command changes the working directory to JDoe under the ProjB project.

```
! cwd >udd>ProjB>JDoe
  r 938 .972 1.731 25
```

To revert to the initial working directory (also called the home directory), the user types a cwd command without an argument.

```
! cwd
  r 939 1.024 1.378 35
```

The cwd command allows the user to manipulate his own position within the storage system hierarchy with respect to the segments he wishes to use. However, the user must remember that it will not help to change working directories if he does not have access permission to use segments cataloged in that directory.

The list command (invoked by typing list or ls) prints out a list of all the segments in a directory. If the user issues the list command with no arguments, the working directory is assumed. The command prints information about the number of segments in the directory, access attributes of the user, and the length of each segment. Segments most recently created are at the top of the list.

```
! ls

   Segments = 21, Lengths = 46.

   r w     4  report.runout
   r w     3  report.runoff
   rew     7  index
   r w     5  test.pl1
                .
                .
                .
   r w     2  exchange_rate.fortran

   r 939 1.866 2.084 41
```

The list command can also be used to print a list of only certain kinds of segments (i.e., those having a certain type of entryname). For example, if the user wants to see all the PL/I source segments in the directory, he types:

```
! ls **.pl1

   Segments = 2, Lengths = 12.

   r w     5  test.pl1
   r w     7  index.pl1

   r 939 1.371 1.936 37
```

The "**.pl1" argument is a common form of a star name that tells the list command to print all entrynames (in this directory) whose last component is pl1. For more information, see "Star Convention" in Appendix B.

PRINT COMMAND (print)

The print command (invoked by typing print or pr) prints, in ASCII, the contents of a segment. The segment name may be either an absolute or relative pathname.

```
! pr circle.pl1

                        circle.pl1          06/07/77  0940.2 mst Tue


    circle:proc;
    declare (radius, area) float bin;
    declare (sysin input, sysprint output) file;
    put list ("enter radius");
    put skip;
    get list (radius);
    area = 3.14159*radius**2;
    put skip list ("The area is:", area);
    put skip;
    close file (sysin), file (sysprint);
    end;

    r 940 1.245 1.921 53
```

HELP COMMAND (help)

The help command (invoked by typing help) prints information about commands, subroutines, the current system, etc. When the help command is issued, it prints out the specified segment (called an info segment) a portion at a time, identifying the number of lines that follow and giving the user the option to continue. For example, if the user wants information about the help command, he types:

```
! help help
   (4 lines follow; 37 lines in segment)
   07/23/75 help

   Function:  prints out system information segments.
   For a list of useful info segments type
     help info_segs
   1 line titled "Syntax" follows. More help?
```

NOTE: Since info segments are updated continually and a new set of info segments is sent out with each system release, the system response to the "help help" command line above may differ from the response shown here.

At this point, the user can type "yes" if he wants to see the "Syntax" portion or "no" if he does not want any more information (return to command level).

There are many info segments that cover general information topics such as access control, new language compilers, and system conventions. New general information info segments (each one has a "gi" component in its entryname) are often added and old ones deleted when a new system release is issued. To get the names of all general information info segments currently on the system, type:

```
! help **.gi -he
```

(This command line is an example of the Multics star convention described in Appendix B.)

SECTION VI

MULTICS EDITOR

The qedx command, which is a Multics context editor, is used to create and edit ASCII segments. (Another editor, edm, is described in Appendix D. It is similar to editors on other time-sharing systems. Actually, the basic editing requests in edm and qedx are similar, but qedx offers much more powerful features once the "basics" are mastered. For this reason, new users are encouraged to use the qedx editor.) To invoke the qedx editor, the user types:

```
qedx
```

(or alternately, the short name, qx).

The qedx editor operates in one of two principal modes: edit or input. The user is automatically in edit mode when he invokes qedx. Input requests place the editor into input mode and allow the user to enter new ASCII text from the terminal until a special character sequence is typed to switch the editor back to edit mode. Edit requests perform various editing functions (e.g., substitution, deletion, printing) on ASCII data, permit the user to read in the contents of an existing segment, and allow the user to save the editing work he has done in a new or existing segment.

Input and editing operations are performed in a temporary workspace called a buffer. When a user edits an already existing segment, a copy of that segment is placed in a buffer. All of the changes the user makes are made on the copy, not the original segment. The edited version of the segment replaces the original only on the user's orders (issuing the write request described later in this section).

The qedx requests treat a segment as a series of numbered lines with a conceptual pointer to indicate the current line. Some requests explicitly or implicitly move the pointer; 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

The discussion of the qedx editor in this section describes only 10 requests. This arbitrary subset of the editor is sufficient for the majority of editing needs, particularly those of a new user. Once familiar with these requests, the user can build on this subset quite easily to make use of the more powerful facilities of qedx (e.g., global requests, special editing sequences called macros--all described in the MPM Commands).

Two kinds of editing requests are described in this section: input and basic edit requests. The invocation and a brief description of the requests are given below. More complete descriptions of the requests are given later in this section.

Invocation	Description
INPUT REQUESTS	
a	Enter input mode and append lines typed from the terminal after a specified line.
i	Enter input mode and insert lines typed from the terminal before a specified line.
BASIC EDIT REQUESTS	
r path	Read contents of segment specified by path into the buffer.
p	Print specified line or lines on the terminal.
=	Print line number of specified line.
d	Delete specified line or lines from the buffer.
/xxx/	Locate and print the line containing the xxx character string.
s/old/new/	Substitute every occurrence of the old character string with the new character string in specified line or lines.
w path	Write buffer contents into segment specified by path.
q	Quit using the editor and return to command level.

SAMPLE INVOCATION

Before getting into detailed descriptions of each of the 10 requests, read through the sample invocation below to see a typical editing session with qedx.

The user is creating a new segment so he invokes qedx and enters an input request immediately. The user types in the lines, making use of the character and line delete symbols, then leaves input mode (by typing "\f") and saves the information in a segment named circle.pl1.

```
! qedx
! a
! circle: proc;
! declre##are (radius,area float bin;
! declare (sysin input,syp#sprint output) file;
! put list ("enter radius);
! out ski@put skip
! get list (radius);
! area = 3.14159*radius**2;
! put skip list ("The area is:", area);
! put skip;
! close file (sysin), file (sysprint);
! end
! \f
! w circle.pl1
```

At this point the user has a new segment named circle.pl1 in his working directory. He is still in the qedx editor--in edit mode--so to check the material he has just input, he issues a print request:

```
! 1,$p
  circle: proc;
  declare (radius,area float bin;
  declare (sysin input,sysprint output) file;
  put list ("enter radius);
  put skip;
  get list (radius);
  area = 3.14159*radius**2;
  put skip list ("The area is:", area);
  put skip;
  close file (sysin), file (sysprint);
  end
```

The "1,$" preceding the request is a way of telling qedx to make the request operate on all the lines. (Refer to "Addressing" below for more information.) After looking at the input material, the user sees lines that need to be corrected. He corrects the lines using qedx edit requests and puts the corrected lines in the segment by issuing a w (write) request again.

```
! /float/
  declare (radius,area float bin;
! s/area/area)/p
  declare (radius,area) float bin;
! /enter/
  put list ("enter radius);
! s/)/")/p
  put list ("enter radius");
! +6
  end
! s/end/end;/p
  end;
! w
! q
  r 1026 0.608 5.510 261
```

Notice how the user combined s and p requests to have qedx print out the edited line for verification. This technique is highly recommended for new users. Also notice the editing request line "+6" to tell the editor to go six lines ahead of the current line; this is called addressing by relative line number. Addressing is an important concept in a line-oriented editor like qedx; it enables the user to tell qedx what lines the editing request should work on. (Refer to "Addressing" below for more information.) And finally, notice the second w (write) request. It is not necessary to give a path argument with the second, and any succeeding, w requests as long as the user intends it to be the same as the only other path argument given in this invocation of qedx.

ADDRESSING

As mentioned earlier, qedx is a line-oriented editor, meaning that requests operate on one or more lines. The editing requests can be preceded by an address to specify the line(s) on which the request is to operate. If no address is given, a default address is assumed. (The default address for each request is given in the request descriptions later in this section.) There are three ways of addressing lines:

- by absolute line number
- by relative line number
- by context

Absolute Line Number

All of the lines in the buffer are given line numbers by qedx; the first line is 1, second is 2, etc. The qedx editor renumbers these lines as soon as more lines are added or deleted. For example, if the user deletes line 5, line 6 becomes line 5, line 7 becomes line 6, etc.

For convenience, the last line in the buffer can also be addressed by using the dollar sign character ($). The user is thus saved the bother of keeping track of the absolute line number of the last line in the buffer.

Relative Line Number

The qedx editor keeps track of the "current" line, which is specified by the period character (.). Normally the current line is the last line addressed in an edit request or the last line entered in an input request.

Lines relative to the current line can be addressed by using a signed decimal integer. For example, +3 indicates the third line following the current line and -3 indicates the third line preceding the current line.

Context

Lines can be addressed by matching a "regular expression" to a string of characters in the line. The first line encountered that contains a match is the line addressed. The search for a match of the regular expression begins on the line following the current line, goes to the last line in the buffer, then goes to the first line in the buffer and continues down to the current line (i.e., from +1 to $, then from 1 to .). If no match is found, qedx prints a message telling the user the search has failed.

REGULAR EXPRESSION

In its simplest form, a regular expression is one or more characters delimited by the right slant character (/). For example, all of the following are valid regular expressions:

```
/one/
/one or/
/For/
/F/
/characters,/
/ters,/
/:/
```

Notice that spaces and punctuation characters can be part of a regular expression. In fact, a regular expression can consist of any character in the ASCII set except the newline character. However, certain characters have special meanings when used in regular expressions.

The use of one or more of the special characters in a regular expression makes it possible for the user to uniquely identify a particular character string with a minimum of typing. The characters with special meanings are:

/	delimits a regular expression
*	means any number (including none) of the preceding character
.	matches any character
^	as first character in a regular expression, means "character" preceding first character on a line
$	as last character in a regular expression, means "character" following last character on a line

Some simple examples show how these characters could be used. New users are often apprehensive about using these characters, but once they try a few (especially the .* combination), they see how much quicker and easier editing can be.

/a/	Matches the letter "a" anywhere on a line.
/abc/	Matches the string "abc" anywhere on a line.
/ab*c/	Matches "ac", "abc", "abbc", "abbbc", etc. anywhere on a line.
/in..to/	Matches "in" followed by any two characters followed by "to" anywhere on a line.
/in.*to/	Matches "in" followed by any number of any characters (including none) followed by "to" anywhere on a line.
/^abc/	Matches a line beginning with "abc".
/abc$/	Matches a line ending with "abc".
/^abc.*def$/	Matches a line beginning with "abc" and ending with "def" and having anything (including nothing) in between.
/.*/	Matches any line.
/^$/	Matches an empty line (a line containing only a newline character).

It is possible to use these characters without their special meaning; simply precede the special character with the \c escape sequence. For example, to match the string "/*" appearing anywhere on a line, the regular expression would be:

```
/\c/\c*/
```

The qedx editor "remembers" the last regular expression used. The user can reinvoke the last regular expression by using a null regular expression (i.e., //). This feature saves a lot of typing time when editing, especially if the regular expression is long or difficult to type. For example, if the user knows he wants to change the ABD string to ABC, he could use a regular expression to find the string and then use a null regular expression to change it:

```
/ABD/ s//ABC/
```

HELPFUL HINTS FOR NEW qedx USERS

The following list offers suggestions for users who are just beginning to work with the qedx editor.

1. The new user should get in the habit of issuing p (print) requests often, to verify changes.

2. Remember the escape sequence to terminate input, \f on most terminals.

   a. After issuing an input request (e.g., a for append), all lines until "\f" are considered input, including intended w and q requests. Without the \f sequence, the intended editor requests are simply more text, which must be located and deleted later.

   b. Often, users unknowingly put qedx in input mode by mistyping an editing request. If qedx does not respond to editing requests (e.g., user types "p" and nothing happens), chances are very good that qedx is in input mode. The user should type the \f sequence, print the current line and preceding lines to see if they need to be deleted, and then continue editing.

3. If the user has 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), the user loses only the work done since the last w request.

4. The qedx editor accepts more than one editing request on a single line. However, the following requests must be terminated by a newline character; therefore each one must be on a line by itself or at the end of a line containing other requests.

   ```
   r     read
   w     write
   q     quit
   ```

5. The qedx editor makes all changes on a copy of the segment, not on the original. Only when the user issues a w (write) request does the editor overwrite the original segment with the edited version.

6. Generally, the user should not issue a quit signal (press ATTN, BRK, INTERRUPT, etc.) while in the editor unless he is prepared to lose all of the work he has done since the last w (write) request.

   Occasionally, however, use of the quit signal is very handy. Suppose the user has read a segment containing several hundred lines into the buffer and types 1,$p by mistake. He should issue a quit signal, wait for the system to respond (QUIT and a ready message are printed on the terminal), and then issue the program_interrupt command. The program_interrupt command (with a short name of pi) returns the user to qedx where the editor waits for the next request, just as though no interruption has occurred.

If the user issues a quit signal accidentally while in the editor, he should wait for the system to respond, and then issue the start command. The start command (with a short name of sr) returns the user to qedx where the editor continues processing the interrupted request.

REQUEST DESCRIPTIONS

A request to qedx can generally take any one of the following three general formats:

```
<request>
ADR<request>
ADR1,ADR2<request>
```

where ADR, ADR1, and ADR2 are valid addresses (as described under "Addressing" above) and <request> is a valid qedx request. When addressing a series of lines (ADR1,ADR2<request>), any one of the three types of addressing can be used for either ADR1 or ADR2. For example, if line 1 is the current line and the buffer contains the following:

```
b:procedure;
a=r;
c=s;
k=t;
end b;
```

the user could print lines 2 through 4 by typing a p (print) request preceded by any one of several address combinations; a few of the possible print requests are given below:

```
2,4p
2,+3p
2,/^k/p
+1,/^k/p
/a=/,+3p
/a=/,/^k/p
```

In each of the request descriptions that follow, several "standard" headings are used:

Name	gives the invocation character followed by the request name
Format	shows the proper format to use when invoking the request
Default	explains what action qedx takes if the user does not specify an address in the request
Value of "."	identifies the position of the current line after the request operation is completed
Example	shows correct usage of the request, including buffer contents before and after the request is given

Other headings may be used in addition to these to explain material pertinent to a particular request.

The requests are presented in a somewhat functional order, i.e., input requests before edit requests and within the edit requests, read is first and quit is last. The exact order is:

Invocation Character	Name
a	append
i	insert
r	read
p	print
=	print line number
d	delete
(none)	locate
s	substitute
w	write
q	quit

NOTE: Users should remember when entering text that they must terminate an input request with the \f escape sequence. The qedx editor cannot respond to another request until it is in edit mode. All lines, including ones the user intends as requests, are regarded as input until the \f escape sequence is given.

a (append)

Name: a (append)

The append request is used to enter input lines from the terminal, appending these lines after the line addressed by the append request.

Format:
```
ADRa
TEXT
\f
```

Default: a means append after current line.

Value of ".": set to last line appended.

Example:

Buffer contents:
```
a:  procedure;
    x=y;
    end a;
```

Request sequence:
```
2a              /x=/a
q=r;     or     q=r;
\f              \f
```

Result:
```
a:  procedure;
    x=y;
"."-> q=r;
    end a;
```

Note: The request 0a can be used to insert text before line 1 of the buffer.

Name: i (insert)

The insert request is used to enter input lines from the terminal and insert the new text immediately before the addressed line.

Format:
```
ADRi
TEXT
\f
```

Default: i means insert before current line.

Value of ".": set to last line inserted.

Example: Buffer contents:
```
a:  procedure;
    x=y;
    end a;
```

Request sequence:
```
2i                    /x=/i
q=r;        or        q=r;
\f                    \f
```

Result:
```
        a:  procedure;
"."->       q=r;
            x=y;
            end a;
```


Name: r (read)

The read request is used to put the contents of an already existing segment into the buffer. This request actually appends the contents of a specified ASCII segment after the addressed line.

Format: ADRr path

where path is the pathname of the ASCII segment to be read into the buffer. The pathname can be preceded with any number of spaces and must be followed immediately by a newline character.

Default: r path is taken to mean $r path.

Value of ".": set to the last line read from the segment.

r (read)

Example:

Buffer contents:

```
a:  procedure;
    x=y;
    end a;
```

Request:

```
2r b.pl1   or   /x=/r b.pl1
```

where b.pl1 is the following text:

```
b:  procedure;
    c=d;
    end b;
```

Result:

```
a:  procedure;
    x=y;
b:  procedure;
    c=d;
"."-> end b;
    end a;
```

Note: The request "0r path" is used to insert the contents of a segment before line 1 of the buffer.

p (print)

Name: p (print)

The print request is used to print the addressed line or set of lines on the user's terminal.

Format: ADR1,ADR2p

Default: p means print the current line.

Value of ".": set to last line addressed by the print request (i.e., the last line to be printed).

p (print)

Examples:

Buffer contents:
```
a:  procedure;
    x=y;
    q=r;
    s=t;
    end a;
```

Request: 2,4p or /x=/,/s=/p

Result:
```
    x=y;
    q=r;
"."-> s=t;
```

Request: 1,$p

Result:
```
a:  procedure;
    x=y;
    q=r;
    s=t;
    end a;
```

= (print line number)

= (print line number)

Name: = (print line number)

This request is used to print the line number of the addressed line.

Format: ADR=

Default: = means print the line number of the current line.

Value of ".": set to line addressed by request.

Examples:

Buffer contents:
```
a:  procedure;
    x=y;
    p=q;
    end a;
```

Request: /q;/=

Result: 3

Request: $=

Result: 4

d (delete)

Name: d (delete)

The delete request is used to delete the addressed line or set of lines from the buffer.

Format: ADR1,ADR2d

Default: d means delete the current line.

Value of ".": set to line immediately following the last line deleted.

Example: Buffer contents:

```
a:  procedure;
    x=y;
    q=r;
    s=t;
    end a;
```

Request sequence: 3,4d or /q=/,/s=/d

Result:

```
a:  procedure;
    x=y;
"."->  end a;
```


(locate)

Name: (locate)

This request sets the value of "." to a specific line and prints the line. This request needs no letter to tell qedx what operation to perform; the user merely types a valid address (generally a context address although any type is permitted) followed by a newline character.

Format: ADR

Default: typing a period (.) prints the current line.

Value of ".": set to line addressed by request.

(locate)

Example:

```
Buffer contents:          aardvark
                    "."-> emu
                          gnu
                          kiwi
                          rhea

Request:                  /^k/  or +2  or 4

Result:                   kiwi
```


s (substitute)

Name: s (substitute)

The substitute request is used to modify the contents of the addressed line or set of lines by replacing all strings that match a given regular expression with a specified character string.

Format: ADR1,ADR2s/REGEXP/STRING/

(The first character after the "s" is taken to be the request delimiter and can be any character not appearing either in REGEXP or in STRING. It must be the same in all three instances.)

Default: s/REGEXP/STRING/ means substitute STRING for REGEXP in the current line.

Value of ".": set to last line addressed by request.

Operation: Each character string in the addressed line or lines that matches REGEXP is replaced with the character string STRING. If STRING contains the special character &, each & is replaced by the string matching REGEXP. The special meaning of & can be suppressed by preceding the & with the \c escape sequence.

Examples:

```
Buffer contents:    The quick brown sox

Request:            s/sox/fox/

Result:             The quick brown fox
```

s (substitute)

Buffer contents:	xyzindex=q;
Request:	s/index/(&)/
Result:	xyz(index)=q;
Buffer contents:	a=b c=d x=y
Request:	1,$s/$/;/

Result:	a=b;
	c=d;
	x=y;

w (write)

Name: w (write)

The write request is used to write the addressed line or set of lines into a specified segment.

Format: ADR1,ADR2w path

where path is the pathname of the segment whose contents are to be the addressed lines in the buffer. If the segment does not already exist, a new segment is created with the specified name. If the segment does already exist, the old contents are replaced by the addressed lines. The pathname can be preceded by any number of spaces and must be followed immediately by a newline character. If path is omitted and only one path argument (or more than one but always the same path) has been given previously in either a read or write request (in this invocation of qedx), this previously specified path argument is used. If path is omitted and no pathname has been given in this invocation of qedx, then the message "No pathname given" is printed and qedx awaits another request.

Default: w path is taken to mean 1,$w path.

Value of ".": unchanged.

w (write)

Example: Buffer contents:

```
a:  procedure;
    dcl a fixed bin(17);
        b char(*);
    end a;
      .
      .
      .
```

Request: 1,4w sam.pl1

Result: Either the first through fourth lines of the buffer replace the contents of the segment sam.pl1 in the user's working directory (assuming that the segment named sam.pl1 already exists) or sam.pl1 is created in the working directory and contains the first four lines of the buffer.

q (quit)

Name: q (quit)

The quit request is used to exit from the editor and does not itself save the results of any editing that might have been done. If the user wishes to save the modified contents of the buffer, he must explicitly issue a write request.

Format: q

Default: the quit request cannot have an address.

Note: The quit request must be followed immediately by a newline character.

EDITING EXAMPLES

Now that the reader is familiar with qedx, he should check the following examples. Using only the subset of qedx given in this section, the reader will see that there are several different ways to edit the same material. As familiarity with qedx increases, users tend to type less and less to accomplish the same tasks. The examples that follow illustrate this "type less" approach.

For the first example, consider the circle.pl1 segment given in the sample invocation at the beginning of this section. After the input request, the buffer contained:

```
circle: proc;
declare (radius,area float bin;
declare (sysin input,sysprint output) file;
put list ("enter radius);
put skip;
get list (radius);
area = 3.14159*radius**2;
put skip list ("The area is:", area);
put skip;
close file (sysin), file (sysprint);
end
```

The three errors in the program were corrected with these lines:

```
! /float/
  declare (radius,area float bin;
! s/area/area)/p
  declare (radius,area) float bin;
! /enter/
  put list ("enter radius);
! s/)/")/p
  put list ("enter radius");
! +6
  end
! s/end/end;/p
  end;
```

Other ways to accomplish the same corrections could be:

```
! 2s/area/&)/
! /enter/ s/ius/ius"/
! /end/ s//end;/
```

or:

```
! 2s/area/&)/ +2s/)/")/ $s/$/;/
```

As another example, assume the buffer contains the following lines:

```
Mondays child is far of face;
Tuesday's child is full of grace;
Wednesday's child is loving and givng;
Thursday's child works gard fir uts kuvubgl
```

These lines could be corrected in any of the following ways (with new users more apt to use a method similar to the first one):

```
! 1s/days/day's/ s/far/fair/p
  Monday's child is fair of face;
! /givng/ s//giving/p
  Wednesday's child is loving and giving;
! /work/ s/works.*$/works hard for its living;/p
  Thursday's child works hard for its living;
```

Some other ways to correct the lines:

```
! 1s/days/day's/ s/far/fair/
! /^W/ s/vng/ving/
! +1s/works.*$/works hard for its living;/
```

or:

```
! 1s/y/y'/ s/r/ir/
! /vng/ s//ving/
! $s/works.*$/works hard for its living;/
```

SECTION VII

PROGRAMMING ON MULTICS

This manual is not intended to offer instruction in programming; instead it describes programming within the Multics environment. On Multics, like many other systems, the basic steps are: write, compile, execute, and debug.

However, on Multics the user can do all four steps online in one terminal session. He has source language debugging capabilities that allow him to execute and test only certain portions of a program if he wishes. The Multics virtual memory and storage system eliminate the file input/output normally required to manage the transfer of information to and from secondary storage; physical movement of data from main memory to secondary storage and back is wholly automatic and of no concern to the programmer. In addition, dynamic linking and the organization of the storage system eliminate the need for extensive software management, since the latest copy of every program is immediately accessible by name from the terminal or from a program. This dynamic linking capability eliminates the need for a complicated job control language for retrieving, prelinking, and executing programs and for defining and locating input/output files.

WRITING A SOURCE PROGRAM

To write a source program, the user invokes an editor that allows him to input--and then edit--his work. On Multics, the user can write his source program in a variety of languages such as PL/I, FORTRAN, BASIC, COBOL, or APL, or even a language he himself has devised.

COMPILING A SOURCE PROGRAM

After a source program is online, the user compiles it by invoking the appropriate compiling command. The system then translates the source program into an object program, which can be executed directly by the processor.

The following list identifies the more common languages and shows their respective compiling commands and source segment suffixes. Source segment names must include the name of the language as the last component (e.g., tic_tac_toe.fortran, mileage.basic).

Language Translator	Translator Command	Source Suffix
PL/I compiler	pl1	pl1
FORTRAN compiler	fortran, ft	fortran
BASIC compiler	basic	basic
COBOL compiler	cobol	cobol

Although pl1 must be the last component in the PL/I source program name (e.g., circle.pl1 or new.circle.pl1), it need not be used in the pl1 command, which compiles the source program. The last component is understood as being implied by each of the language processors.

For example, to compile the PL/I source program, prog.pl1, the user invokes the pl1 command by typing:

```
pl1 prog

   or

pl1 prog -table
```

The optional argument added in the second command line, -table, is one of the many control arguments accepted by the pl1 command. The -table control argument produces a symbol table that is valuable for use with one of the Multics debugging facilities.

EXECUTING A PROGRAM

To run an object segment, the user simply types its name (either relative or absolute pathname). To run the compiled version of source program prog.pl1, the user types:

```
prog
```

DEBUGGING A PROGRAM

Multics compilers print a list of errors when they compile a source segment. For example, the list might indicate incorrect syntax in the source segment. This list of errors may be graded by severity; the user may judge whether he wishes to continue compilation or halt it by issuing a quit signal. Severe errors automatically cause compilation to cease. The compiler prints an error message, and the system returns to command level and prints a ready message.

Multics permits users to run a part of a program, temporarily halt its running, debug that portion of the program needing changes, and resume running the program.

Multics provides extensive interactive program debugging facilities through the two commands, probe and debug. To perform symbolic debugging with these commands, the user must have compiled his program with a symbol table (i.e., specifying the -table control argument). The two commands provide similar services, but the probe command is designed with the high-level language programmer in mind while the debug command is oriented more toward the needs of a machine language programmer. Multics also provides a trace command that traces the flow of control through program execution and a trace_stack command that traces the list of programs active on the program stack. (The probe, debug, trace, and trace_stack commands are described in the MPM Commands.)

A central feature of both probe and debug is the facility for setting breakpoints at specified program locations. The program is then executed. When a preset breakpoint is reached, execution is interrupted and the current state of variables preserved. The user can then perform other debugging operations such as examining the values of data items, inserting test values, executing other programs, and so on. He may then continue execution from the point at which execution was suspended.

A simple illustration of the use of probe, including setting a breakpoint, is shown in the second programming example in "Sample Programs" below.

SAMPLE PROGRAMS

Two simple programming examples are shown below. The first example shows the terminal interaction as a user logs in and writes, compiles, and executes a short program. The second example shows the terminal interaction as a user compiles, attempts to execute, debugs using probe, and finally executes successfully a short program.

In the first example, a program, named times_2, is written in PL/I and put online using the qedx editor. The times_2 program accepts an integer (online) and prints the value of 2 times that integer on the terminal.

```
! login TSmith
  Password:
!
  TSmith ProjA logged in 06/07/77  0937.5 mst Tue from ASCII terminal "234".
  Last login 06/06/77  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 937 1.314 1.332 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 940 4.875 7.621 62
```

After typing in the source program, going to edit mode to write it, and quitting qedx, the user is ready to compile his program. Notice that the program name (times_2.pl1) includes the language name as the last component. The language name also identifies the proper command to invoke for the compilation. Thus to compile the times_2.pl1 program, the user types:

```
! pl1 times_2
  PL/I
  r 941 2.906 8.022 272
```

Once the program is compiled, the user, and any other users to whom he gives proper access, can execute the program by typing "times_2"; for example:

```
! times_2
  Enter integer
! 19

  2 times your integer is:                38
  r 943 0.231 2.272 50
```

For the second example, assume that the user types in another source program, named cubes.pl1, containing the following lines:

```
cubes: proc;
declare (i,j) fixed bin init (-1);
declare k float bin;
declare (sysin input, sysprint output) file;
do i = 1 by 1 while (j ^=0);
k = i**3;
end;
put skip list (i, j, k);
put skip list ("cubes done");
put skip;
close file (sysin), file (sysprint);
return;
end;
```

The user compiles the program, using the -table control argument:

```
! pl1 cubes -table
  PL/I
  r 1018 2.798 37.848 270
```

Then the user executes the program and after waiting for a response (the program is supposed to print the values of i, j, and k and then say it is done), he decides something is wrong and issues a quit signal:

```
! cubes
  <user presses BRK or appropriate key>
  QUIT
  r 1018 5.854 1.560 39 level 2, 10
```

Next, the user invokes the probe command and asks to see the source line being executed when the quit signal occurred and also the value of the variables:

```
! probe
  Condition quit raised at line 6 of cubes.
! source
  k = i**3;
! value i
      355131
! value j
      -1
```

When he sees the value of j is still -1, the user realizes he is in an endless loop. To test his theory that j is the problem, the user decides to set a break; the break halts execution of the program at that point, calls probe to execute the probe request (the text inside the parentheses), and continues executing the program.

```
! position 6
  k = i**3;
! before: (if i = 6: let j = 0)
  Break set before line 6 of cubes.
! quit
  r 1020 0.714 10.864 205 level 2, 10
```

The user releases the work being held (because he issued a quit signal to interrupt the execution of the cubes program) and then invokes the cubes program again:

```
! release
  r 1020 0.038 0.756 27

! cubes

          7                    0          2.16000000e+002
  cubes done
  r 1020 0.161 2.100 47
```

The user's theory about j was correct. He can now go back and edit his source program accordingly. (Since the incrementing of i is done before the while option is evaluated for the last time, the value of i is 7 rather than 6 when the program finishes.)

SECTION VIII

ACCESS CONTROL

On the Multics system, the user is able to share as much or as little of his work with as many other users as he desires. The checking done by the hardware on each memory reference ensures that the access privileges described by the user for each of his segments are enforced. This kind of privacy and security gives Multics users great flexibility in the kinds of data they may put on the system. For example, if Tom Smith were the head of a personnel department, he could put the names and addresses, salaries, education, etc. of all the company employees online. He could then set different access rights on each of the segments. For example, he could assign read and write access to only himself on the segment containing salary information. He would not allow anyone else in the department to have any access to the salary segment. On the segment containing the names and addresses of all personnel, he could assign read and write access to himself and his assistant and only read access to the rest of his department. Multics allows the user to give different access rights to different users of the same segment.

ACCESS CONTROL LIST

The access rights for each segment are described in an access control list (ACL). Each segment has its own ACL; it contains the identification of users permitted (or specifically denied) access to the segment plus a description of the type of access allowed.

The user identification (known as a User_id) in the ACL consists of a three-component name: Person_id, Project_id, and an instance tag, separated by periods. (The system assigns the instance tag when the user logs in.) Whenever anyone tries to access a segment on the Multics system, his User_id must match one of the entries on the ACL of that particular segment; if not, he has no access to that segment. (See "Ordering and Matching of ACLs" and "Selectively Identifying ACL Entries" below.)

ACCESS MODES

The type of access allowed is defined by access modes: four modes for segments and four modes for directories.

Access modes for segments are:

read	(r)	data in the segment can be read
write	(w)	data in the segment can be modified (written)
execute	(e)	an executing process can transfer to, and execute instructions in, this segment
null	(n)	access to the segment is denied

Access modes for directories are:

status	(s)	the attributes of segments, directories, and links contained in the directory can be obtained
modify	(m)	the attributes of existing segments, directories, and links contained in the directory can be changed or deleted; segments, directories, or links can be deleted
append	(a)	new segments, directories, and links can be created in the directory
null	(n)	access to the directory is denied; access may still be granted to specific segments within a directory according to the ACL on each segment

The user generally assigns combinations of access modes to his segments and directories. Useful access mode assignments for segments and directories are:

Segments	Directories
r	s
w	sm
re	sa
rw	sma
rew	null
null	

The user specifies one of the above access mode assignments for the persons and/or projects he wishes; he uses one command in specifying access to his directories and/or segments. Once specified, the access is not "frozen"; the user may change it at will just by issuing the command again, specifying different modes, persons, or projects.

ORDERING AND MATCHING OF ACLS

As noted earlier, each segment has its own ACL, which consists of a list of entries. Each entry specifies certain access modes and the users associated with these modes. Assume that a segment named test has the following ACL:

```
rew   JDoe.ProjB.*
rw    PFrank.Manuals.*
r     TSmith.*.*
r     *.Demo.*
null  *.ProjZ.*
r     *.*.*
```

The asterisk in the ACL entries means that any string can occupy that component position. For example, *.ProjZ.* designates all the users on the ProjZ project, and in the last entry line, *.*.* designates any Multics user. The order of the ACL entries, from JDoe.ProjB.* to *.*.*, is very important. Consider the two ACL entries:

```
r     TSmith.*.*
null  *.ProjZ.*
```

If TSmith is logged in under the ProjZ project, what access does he have to the test segment?

The answer is determined by the ACL ordering rules, which stated simply are: specifically named components come before asterisks; leftmost component comes first, then middle, then right. The ACL entries are arranged according to these rules and the system starts at the top of the ACL and works it way down trying to find a match for a User_id. Based on these rules, TSmith logged in under the ProjZ project has read access to test since the ACL entry "r TSmith.*.*" is encountered before the "null *.ProjZ.*" entry. JDoe logged in under the ProjZ project would have null access (JDoe.ProjB.* is not a match, the first match is *.ProjZ.*). KJones logged in under the ProjB project would have read access (the only match is in the last entry, *.*.*).

SELECTIVELY IDENTIFYING ACL ENTRIES

In many of the access commands, the user gives a User_id as part of the command line. The system interprets this User_id and then tries to match it to one of the User_ids in the ACL. Based on this match, the access commands select an ACL entry to modify. The following rules apply to the User_id the user specifies.

1. A specifically named component (including "*") matches only a component of the same name.

2. A missing component delimited by a period matches any component in that position.

3. Components to the right of a specifically named component may be omitted (if so, each is treated the same as a literal "*"); those to the left must be given (by a period if nothing else).

These rules are illustrated in the examples below, showing a User_id and the associated access command matching strategy.

TSmith..	matches any User_id in an ACL entry whose first component is TSmith
TSmith.*.*	matches only TSmith.*.*
TSmith	same as above (using the missing components rule, this User_id is treated as though it were TSmith.*.*)
ProjZ	matches only ProjZ.*.* (absence of a leading period makes ProjZ the first component)
.ProjZ.	matches any entry whose second component is ProjZ
..*	matches only *.*.*
..	matches any entry

SETTING ACCESS

The command the user invokes to set the ACL, set_acl, either adds an entry to the ACL or modifies an existing entry. The set_acl command, whose short name is sa, has the general format:

sa pathname accessmode(s) User_id

For example, Tom Smith has text in segment xsolve of his myd directory that Jane Doe wants to use. To give her access so she can read the segment, he types (if myd is his current working directory):

```
sa xsolve r JDoe.*.*
```

If this User_id is already part of the ACL for xsolve, the access is changed to r (for read). If this User_id is not on the ACL, the entry "r JDoe.*.*" is added to the ACL.

If he instead decides that his segment should not be available to Jane and wants to make sure she cannot read it, he types:

```
sa xsolve null JDoe..
```

The periods following Jane's Person_id (JDoe) in the above command line tell the system that the requested access applies to Jane no matter what project she may be on, no matter what instance tag may be associated with her work. For example, the User_id, JDoe.., matches the User_ids in all of the following ACL entries:

```
rew   JDoe.ProjB.*
rw    JDoe.Manuals.*
r     JDoe.Demo.*
r     JDoe.*.*
```

so the access in all of the above would be changed to null access. However, if Tom had given JDoe.*.* as the User_id, the only match would have been "r JDoe.*.*" and only that entry would have the access changed to null access.

LISTING ACCESS

To check the ACL of a segment, the user invokes the command that lists the ACL, list_acl. The list_acl command, whose short name is la, has the general format:

```
la pathname
```

As explained earlier, the system assumes that any pathname that does not begin with the greater-than character is relative to the working directory. Thus, if Tom Smith wants to list the ACL of xsolve, he types:

```
! la xsolve
   rw    TSmith.ProjA.*
   null  JDoe.*.*
   rw    *.SysDaemon.*
   r     *.ProjA.*
```

The User_id in the third ACL entry in the example, *.SysDaemon.*, identifies various system processes that control such things as printing and making copies of segments on "backup" tapes. The system normally places appropriate ACL entries on every segment the user creates so the system processes will have the necessary access to perform the various backup, metering, and input/output functions.

If Tom is interested in checking the access he has given only Jane on xsolve, he types:

```
! la xsolve JDoe..
  null  JDoe.*.*
```

or to check the access rights of only ProjA, he types:

```
! la xsolve .ProjA.
  rw    TSmith.ProjA.*
  r     *.ProjA.*
```

Notice that when specifying the User_ids in the above command lines, Tom uses periods to show missing components that may contain any value. As explained earlier (see "Ordering and Matching of ACLs"), if a user does not use periods for the missing components, a * is assumed for each missing component. Thus the command line "la xsolve JDoe" lists the ACL entry for JDoe.*.* but not for JDoe.ProjB.* (if such an entry exists).

DELETING ACCESS

A third access control command, delete_acl, allows the user to delete ACL entries. This command, whose short name is da, has the same general format and rules as the list_acl command.

For example, if Tom Smith has changed segment new_report, he might want to also change its ACL. First, he lists the ACL entries to see who currently has access to new_report:

```
! la new_report
  rw    TSmith.ProjA.*
  re    Gray.Merlin.*
  rw    Butler.Merlin.*
  rw    Jones.*.*
  re    JDoe.*.*
  rw    *.SysDaemon.*
  r     *.*.*
```

Tom decides that he no longer wants user Jones, anyone on the Merlin project, or the entire user community (represented by *.*.*) to have access to new_report. Therefore, he invokes the delete_acl command in the following manner:

```
da new_report Jones *.*.* .Merlin.
```

Here Jones becomes Jones.*.* by default. If an ACL entry had existed for Jones.ProjA.*, that entry would not have been deleted.

If Tom now again invokes list_acl, he will see that the requested change has already taken place.

```
! la new_report
  rw    TSmith.ProjA.*
  re    JDoe.*.*
  rw    *.SysDaemon.*
```

On Multics, changes in access rights occur instantaneously. If both Tom and Jane are online at the same time and she tries to access one of his segments and finds that she does not have the proper access, she can send him a message (see Section IX concerning online communication), asking him to give her proper access to the segment. He can then issue the appropriate arguments to the set_acl command, and she immediately has access to the segment. Access rights are revoked just as rapidly. If Jane has access to a segment of Tom's, and he changes the access while she is using the segment, the system prints out a message telling her that she has incorrect access to the segment and returns her to command level.

SECTION IX

ONLINE COMMUNICATION WITH OTHER USERS

The Multics system offers several commands that enable users to communicate with one another online. Such commands are extremely useful; for example, a user may require immediate access to another user's data, or a user may need to request an increase in his quota from his project administrator.

mail COMMAND

The mail command (short name, ml) is used to send mail to another user or print mail sent by another user. The mail is stored in the user's home directory in a mailbox segment with the absolute pathname:

```
>udd>Project_id>Person_id>Person_id.mbx
```

In order to receive mail, the user must have a mailbox segment. The mailbox is created automatically the first time a user invokes the mail command (or print_messages or accept_messages commands, described later in this section).

To read his mail, the user types:

```
mail
```

The system first tells the user how many messages are in his mailbox. Then the system prints all of the messages and asks the user if he wants to delete these messages. If the user answers yes, the messages are deleted; if he answers no, they are saved.

To send mail, the user types the mail command with the proper arguments.

```
mail message Person_id.Project_id
```

The message argument may be either the pathname of a segment or an asterisk (*). Generally, the user types an asterisk for the message argument. The system responds by printing "Input." The user then types his message, ending it by a line containing only a period (.). For example, if Tom Smith wants to send Jane Doe mail, he types:

```
! mail * JDoe.ProjB
  Input
! Dear Jane,
! Thank you for the draft copies of
! the new manual.  I promise to return
! them next week.
!                        Tom
! .
  r 1004 .741 3.386 99
```

If the information the user wishes to send is contained in a segment, he types the pathname of the segment for the message argument. The content of the segment is then placed in the receiving user's mailbox.

When the system sends the mail, it supplies a header that identifies the sender, the date and time the message was sent, and the number of lines in the message. The system then copies the mail with the header into the proper mailbox segment--in this case, >udd>ProjB>JDoe>JDoe.mbx. Thus, when Jane checks her mailbox, by issuing the mail command, Tom's message would appear on her terminal as:

```
! mail
  1 message.
  1) From: TSmith.ProjA  06/07/77  1004.6 mst Tue (6 lines)

  Dear Jane,
  Thank you for the draft copies of
  the new manual.  I promise to return
  them next week.
                        Tom

  mail: Delete?
```

MESSAGE FACILITY

The message facility permits online communication between users on different terminals. The system puts messages from other users in a mailbox segment (the same one used by the mail command) and prints them out at the option of the user, i.e., either immediately or on command. Even if the user who is to receive the message is not logged in or is deferring messages, messages can still be sent. If this happens, the user who is sending the message is notified that the message cannot be received online at this time, and the message is stored in the receiving user's mailbox segment.

If the user wants to receive messages, he must have a mailbox. As explained earlier, the system automatically creates a mailbox segment, if one does not already exist, having the absolute pathname:

```
>udd>Project_id>Person_id>Person_id.mbx
```

the first time the user invokes the mail, accept_messages, or print_messages commands.

Once the accept_messages command is invoked, the user receives--instantaneously--any messages sent to him during that terminal session. If a message is sent after he logs out, it is saved in the mailbox segment. In order to receive the messages saved in the mailbox, the user must issue the print_messages command (or its short name, pm) or the mail command.

NOTE: The mail, accept_messages, and print_messages commands operate on a per-terminal-session basis. Even though a user has a mailbox, he must issue the accept_messages command before messages are printed on his terminal (rather than stored in his mailbox) and he must issue the mail and print_messages commands to see the mail and messages currently in his mailbox. To avoid the necessity of issuing these commands at the beginning of each terminal session, refer to the discussion of the start_up.ec segment under the "exec_com command" description in Section XI.

When the user wants to send a message, he must invoke the send_message command (or its short name, sm) with the proper arguments.

```
send_message Person_id.Project_id message
```

If Tom Smith wants to send a message to Jane Doe, he types:

```
send_message JDoe.ProjB you now have access to xsolve
```

If the message argument is missing, the system types the word "Input" and the user then types his message. Each line of his message is sent as soon as the user types the carriage return. In this way one user can have a dialogue with another user without having to issue a send_message command to send each line. To indicate that the message is complete, the sending user types a period (.) on a separate line.

Whenever a message is sent, the sending user is identified by his Person_id and his Project_id. Tom's message would appear on Jane's terminal as:

```
From TSmith.ProjA 06/07/77  1047.3 mst Tue: you now have access to xsolve
```

who COMMAND

Although the who command is not an online communication command in the same sense as mail, accept_messages, print_messages, and send_message, it is still an important communications tool. By invoking this command, the user learns the number, identification, and status of all users currently on the system. (It is possible for a user to prevent his name from being listed; to do this, the user should first see his project administrator.) Often, the user issues the who command to see if a particular user is online before he issues the send_message command.

The who command first prints out a header line, listing the system name, the total number of users, the current system load, and the maximum load. After this header, the command lists the name and project of each user.

To invoke the who command, the user types:

```
! who

Multics XX-x, load 12.0/90.0; 12 users
Absentee users 0/3

IO.SysDaemon
Backup.SysDaemon
Metering.SysDaemon
Dumper.SysDaemon
JDoe.ProjB
Rolf.Maint
RSmith.North
TSmith.ProjA
Green.Manuals
Richards.ABC
Camp.Demo
Warren.XYZ

r 1048 .668 1.036 74
```

The list of users in the above example is sorted according to log-in time. The user can specify certain options when he calls the who command and change the sort key, suppress the header, list only those users of a particular project, and various other things. For information about the control arguments and other arguments that can be used with the who command, refer to the MPM Commands.

SUMMARY

This section shows the new user enough information about five communication commands to enable him to use them. However, he should realize that the mail, accept_messages, print_messages, send_message, and who commands offer a variety of options to make online communication even more meaningful to any one particular user; that is, each user can choose those options that are most suitable to the type of online work he does.

For example, a user who is printing out a final draft of a document would certainly not want to receive a message from another user in the middle of his printout. Therefore, he may wish to defer messages (by issuing the defer_messages command) while he is printing the draft; he still "receives" the messages, but they are saved in his mailbox until he asks for them.

To learn more about the various control arguments and options available with these communication commands, refer to the MPM Commands.

SECTION X

ABSENTEE AND I/O DAEMON USAGE

Although this manual deals exclusively with the interactive usage of Multics, the new user should be aware that there is another type of Multics usage--absentee.

Absentee usage (batch processing on Multics) gives users the ability to execute large production runs without waiting at the terminal while the run is in progress. The user merely creates an absentee job and submits it for execution.

Absentee jobs are placed in a queue and run as background to the normal interactive work of the system. Because absentee jobs are usually deferred until the interactive load is light, the charges for absentee usage are substantially lower than the charges for interactive usage. That is, the charge for the highest priority absentee queue is generally much lower than the charge for the most expensive interactive shift. Often however, the cost of the cheapest interactive shift may be less than the highest absentee queue. (Since pricing is site dependent, the user should check the charges at his site to determine the most cost-effective method of performing certain tasks.)

To create an absentee job, the user creates an absentee input segment that contains those commands he wants executed. The job control language for absentee usage is identical to the command language for interactive usage. Basically, an absentee job is merely a "planned" interactive terminal session; that is, the user anticipates any responses or commands he must give and puts all of this data into his absentee input segment.

For example, suppose a user (TSmith on the ProjA project) has written a prime number generator program in FORTRAN, which resides in his directory of FORTRAN programs, and he wants to compile and run it. He could create a segment called compile.absin containing the following lines:

```
cwd >udd>ProjA>TSmith>fort_progs
fortran primes -list
dprint -dl primes.list
primes
dprint file10
logout
```

All the user has to do to compile and run his prime number generator program as an absentee job is type the following:

```
enter_abs_request compile.absin
```

This command line executes an absentee job that changes to the proper working directory, compiles the program and produces a listing segment, prints the listing segment on the line printer and then deletes it, runs the program, prints the results contained in file10 on a line printer, and finally logs out.

For more information about absentee usage and a complete description of how to request an absentee job, see the enter_abs_request command in the MPM Commands.

The I/O daemon is used to manage the central system unit record equipment (card readers, card punches, and high-speed printers) as well as remote devices (such as the IBM 2780) containing unit record equipment. Users can request printing and punching of files from their interactive terminals and absentee jobs; they can specify at which of several locations the processing is to be done and select special forms or printer attributes for these requests. (See the dprint and dpunch commands in the MPM Commands.)

Users can also bring card decks to the card readers run by the I/O daemon and have them read into the system. These card decks can contain absentee control files; the user can request, via control cards, that his job be scheduled and run as an absentee job and have printed output returned to the printer associated with the card reader. For example, the user could put the contents of the absentee segment shown above (compile.absin) on cards and surround these cards with appropriate control cards. Then he merely reads this deck into the system on a remote device in order to run the absentee job and get the output printed on a remote printer. The results are the same whether he reads the information into the system using a remote device or he gets online and types:

```
enter_abs_request compile.absin
```

SECTION XI

MULTICS FEATURES FOR ADVANCED USERS

Once the user becomes familiar with most of the basics, as described in earlier sections, he begins to take "shortcuts" by learning certain system features that he finds extremely useful in his particular type of work at the terminal. The purpose of this section is to enumerate and briefly describe several such features. These descriptions also identify appropriate reference materials so that interested users can easily add these features to their Multics repertoire. The fact that these features are part of the Multics System does not mean that they all need to be learned by any one user; each user learns the features he feels will be most helpful to him.

ABBREVIATION PROCESSOR

The abbreviation processor is a special command processor that is invoked for each command line only after the user invokes the abbrev command. The user defines his own abbreviations for frequently used command lines or other strings. Then, after invoking the abbrev command, the abbreviation processor checks each command line for abbreviations; any abbreviations the user has defined are expanded by the abbreviation processor and then passed on to the Multics command processor.

Use of the abbrev command (refer to the MPM Commands) greatly simplifies the user's terminal work. For a special use of the abbrev command, see the exec_com command description in this section.

ACTIVE FUNCTIONS

An active function is a program that is invoked as part of a command line; the character-string result of the active function replaces its invocation in the command line. In other words, the active function part of the command line is executed immediately, the result placed in the command line, and this expanded command line is passed on to the command processor for execution.

The active functions fall into nine operational groupings: logical, arithmetic, character string, pathname manipulation, storage system names, storage system attributes, date and time, question asking, and user parameter. Refer to Section II of the MPM Commands for descriptions of the active functions.

By using active functions, the user is able to make many standard Multics commands conditional commands. For example, the user may want to set the length of his command line to x only if he is working on a particular type of terminal; or he may want to enter a certain absentee request only if today's date equals a particular number.

ADMINISTRATIVE FEATURES

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

All of the administrative operations can be performed while the system is running; desired actions take place immediately. Multics administrative operations cover the following areas:

Resource distribution
Accounting and billing operations
Usage control
Environment shaping
Access control and security

If Tom Smith is the project administrator for ProjA, he can determine the dollar limit that a particular ProjA user may incur in a single month. If this limit is exceeded, the user is automatically logged out; he cannot log in again until either the next month begins or until the limit is changed.

As project administrator, Tom can also determine several other items, including whether a user can preempt others, specify his home directory, or have primary or standby status. In fact, the project administrator can so control each user's environment that he can deny a user access to the full Multics System and instead provide the user with access to only those commands that he (as the project administrator) specifies. Such a user is said to have access to a limited service system.

For more information on the 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

ARCHIVE SEGMENT

An archive segment is a single segment consisting of the contents of many different segments packed together. Once in an archive, the individual segments are called components of the archive segment. This packing, performed by invoking the archive command, reduces the user's storage load.

By invoking the archive command with different arguments, the user can manipulate the archive segment in a variety of ways. For example, he not only creates his archive; he also can 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. For more information about the archive command and its use, refer to the MPM Commands.

BOUND SEGMENT

A bound segment is a single executable procedure segment ("object segment") made up of one or more separately compiled, executable procedure segments. Again, as with an archive segment, the user reduces his storage load by combining several segments. However, a bound segment also automatically prelinks all the internal intersegment references thereby reducing execution time.

The user creates a bound segment by invoking the bind command. The bind command also allows the user to update, list, and map the bound segment as well as manipulate the manner in which the various segments in the bound segment are to be called.

Those programs that the user calls frequently and that are interrelated (i.e., reference one another) should be bound to improve program efficiency. By putting such programs in a bound segment, the user saves money through decreased computing time and storage space and, at the same time, decreases his execution time.

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

exec_com COMMAND

The exec_com command permits the user to execute a series of commands specified in a segment. This segment must contain only command lines and control lines; also, it must have the letters ec as the last component of its name (e.g., test.ec or weekly.report.ec). The exec_com command also allows the user to substitute special strings in the segment by giving certain arguments when he invokes the command.

The command lines in the segment can use any Multics command. The control lines are defined in the exec_com command description (refer to the MPM Commands).

Multics permits users to create a special exec_com segment that contains commands to be executed when the user logs in, before his process attempts to read from his terminal. In other words, he does not even need to invoke the exec_com command for this segment; it is automatically invoked for him as part of his log-in procedure. This segment must be named start_up.ec and must reside in the user's initial working directory. Some commands that a user typically includes in the start_up.ec are:

abbrev	so the user does not need to remember to invoke it during each terminal session in order to use personal abbreviations
accept_messages	so the user can receive online messages from other users
print_messages	so the user can receive messages saved in the mailbox
mail	so the system automatically prints the mail that has been sent to the user since his last terminal session
print_motd	so the system keeps a record of the message of the day and prints only those portions that differ from the last message the user saw

Some other commands that users often put in their start_up.ec segments show the latest changes in info segments (check_info_segs), print reminders of certain events (memo), or set up certain terminal modes or delay timings according to the terminal currently in use (set_tty used with the "user term_type" active function).

Also, a user generally writes his start_up.ec so that certain lines of the segment are executed when he logs in as an interactive user and other lines are executed when he is logged in as an absentee user. For example, the accept_messages, print_messages, mail, and print_motd commands would not be used in a start_up.ec for an absentee user.

A simple start_up.ec is shown below. It invokes the abbrev command, checks for messages and mail, and prints the message of the day. For an absentee user, it only invokes the abbrev command and then quits. Also, commands are not printed on the terminal as they are invoked by the start_up.ec segment. New users may want to put this start_up.ec online (with the qedx editor) and use it until they need or want a more elaborate one.

```
&command_line off
abbrev
&if [user absentee] &then &quit
accept_messages -print
mail
print_motd
&quit
```

GRAPHICS SYSTEM

The Multics Graphics System provides a general purpose interface through which user or application programs can create, edit, store, display, and animate graphic material.

The Multics Graphics System is a terminal-independent system. Thus a program written for one type of graphic terminal is operable on another terminal having similar capabilities without modification. Also, users can use programs developed on different graphic terminals by other users.

For more information see Multics Graphics System, Order No. AS40.

INPUT/OUTPUT SYSTEM

The Multics system contains a device-independent input/output system interface that programs can use. This interface allows interchangeable reading and writing via tapes, terminals, cards, printers, and storage system segments.

This generalized input/output means that segments and input/output devices are interchangeable since both are referenced by symbolic name. A user can place a series of commands in a segment, attach the segment to an input switch, and the system will process the user computation indicated by the commands as if input were from the terminal. An interface command that assigns input/output switches (the io_call command) is available to all users. Output also can be directed to a segment by issuing the file_output command. Moreover, the user can switch from input device to input segment and from output device to output segment in the same way that he can switch from device to device. In addition, Multics provides system commands for user output that automatically queue the specified segments for printing (the dprint command) or punching (the dpunch command).

The Multics input/output system was designed for flexibility. In fact, users can write their own input/output routines, which can be "plugged in" to this system.

For information about general input/output commands, see the MPM Commands; for descriptions of specific commands such as those related to magnetic tapes and for descriptions of input/output subroutines and modules, see the MPM I/O. A detailed description of the Multics input/output system is given in the MPM Reference Guide.

LINKING SEGMENTS

Multics allows a user to create a link to a segment anywhere in the storage system as long as he has the proper access to the directory in which the link is to be placed. The user invokes the link command to create a link (refer to the MPM Commands).

By creating a link, the user is able to reference another segment as though it were in the directory containing the link. In short, he has the use of this particular segment without actually having to make a copy of it. This linking feature allows users to share information easily and inexpensively.

PRODUCING MANUSCRIPT FORMAT

Multics has two commands for producing text segments in manuscript form: runoff and compose. Control arguments for either command allow the user to completely regulate the processing of his text. For example, he can convert the output to be suitable to a particular kind of type ball or device, start or end the printing at a particular page, have source line numbers printed in the left margin, have the system wait for a carriage return before beginning and after each page of output, or direct the output to a special segment so he can print the material on a high-speed printer using the dprint command.

The input segments for either runoff or compose contain control lines as well as text lines. There are over 50 types of control lines. They allow users to do such things as:

Specify up to 20 headers and 20 footers per page
Set the line length and page length
Have either roman or arabic page numbers
Skip a specified number of lines for an illustration
Center lines
Format equations
Control the size and number of margins
Set the spacing (single, double, or multiple)
Justify the margins
Translate a specified character to be printed as a different specified character in the output

This last feature is especially useful when the user wants a single blank between two character strings; the translate control line prevents splitting the strings between two lines or inserting padding spaces between the two strings.

More features are available with the compose command, which is the newer and more versatile of the two formatting commands. Probably the most important features that are available only with compose are user control of widow line processing (i.e., user determines the minimum number of lines at the bottom of one page and top of the next when a block of text is split) and direct support of active functions.

For complete information on all the control lines and the other capabilities of the runoff and compose commands, see the MPM Commands.

qedx EDITOR

The qedx context editor (described in Section VI) can be used to create and edit ASCII segments in a manner similar to the edm editor (described in Appendix D). However, qedx is a more powerful editor than edm. It honors global editing requests and supports a virtually unlimited number of buffers. In addition, the macro capabilities of qedx make it almost a programming language in itself.

At any one time, one of the qedx buffers is designated as the current buffer and all others are auxiliary buffers. The user can move information from one buffer to another, designate any buffer as the current buffer, and check the status of all the buffers. The user can place a frequently used editing sequence in one buffer and then through a special escape sequence invoke the contents of this buffer whenever necessary.

The user can place elaborate editor request sequences (called macros) into auxiliary buffers and then use the editor as an interpretive language. In a sense the macro is a subroutine, and the escape sequence is a call statement. The qedx editor also allows the user to invoke a qedx macro from command level. To do this, the user merely places his macro in a segment that has the letters qedx as the last component of its name (e.g., which_check.qedx or caps.cmds.qedx). He can then invoke the macro by issuing the qedx command followed by the appropriate segment name. For example, to invoke the caps.cmds.qedx macro, the user types:

```
qedx caps.cmds
```

A complete description of the qedx editor, including its buffer and macro capabilities, is given in the MPM Commands.

RESOURCE MEASURING

Through various commands, each Multics user can check his secondary storage quota usage; get information (including size, names, and access modes) about his segments, directories, multisegment files, and links; and print a month-to-date report of his resource consumption.

There are two commands directly related to secondary storage quotas. The get_quota command allows the user to get information about the amount of secondary storage he may use. The move_quota command moves all or part of a quota between two directories, one of which must be immediately inferior to the other.

Several commands print information about segments, directories, multisegment files and links. One of these commands, list, is described briefly in Section V. When the user invokes the list command with different control arguments, he can get specific kinds of information (e.g., only names, names and access, only a total count) for all the entries in a particular directory or for only one kind of entry in the directory (e.g., only segments) or for only entries having a particular type of entryname (e.g., **.basic for all BASIC source segments). Another command, status, prints detailed information about a specific entry (the name of this entry must be supplied as an argument to the status command). Like the list command, status has a variety of control arguments to enable the user to get only the information he requires (e.g., author, bit count, ring brackets).

The user can issue the resource_usage command to print a month-to-date usage report for his own resources. He cannot issue this command to get information about any resource usage except his own. Through this command, the user can check his dollar charges according to shift and queue and also type of usage (interactive, absentee, or input/output daemon).

All of these commands are fully described in the MPM Commands.

RING STRUCTURE

As a further refinement of access control (see Section VIII), the system uses a special capability called the ring structure. This additional degree of protection, implemented by special hardware, is unique to Multics. Most users need not be concerned about the rings in which they will be working. Advanced users who require the use of ring protection for special data bases should see their project and system administrators.

Many system segments on Multics execute as part of a process, making calls and returns unknown to the user. These system segments must be protected from unauthorized modification by user segments. This protection is achieved by grouping segments into rings. Multics operation is controlled in such a way that procedure segments execute in a number of mutually exclusive subsets. These subsets may be considered concentric rings of privilege, representing different levels of memory access rights. The innermost or supervisor ring is made up of those segments essential to all users. This innermost ring, designated as ring 0, represents the highest level of privilege. The outermost ring, designated as ring 7, has the lowest level of privilege.

A procedure segment in an outer ring can call or pass data to, but cannot modify, a procedure segment in an inner ring. Normally, a procedure segment in an outer ring cannot access data in an inner ring. Within user-imposed limitations, an inner-ring segment can modify a segment in an outer ring. Every attempted access of one segment by another is checked for proper user access and for the ring of the referencing procedure segment to prevent invalid modification. The Multics ring-handling mechanism is enforced at the hardware level.

The ring structure capability permits the ready construction of protected data bases. In such a data base, privileged users could be given the ability to read detailed information while nonprivileged users could receive only summarizations of the information. For example, a management information system could be developed that would allow management to designate which data and procedures could be accessible to different levels of personnel.

For more information about the ring structure, refer to Section VI of the MPM Reference Guide under "Intraprocess Access Control."

SETTING SEARCH CRITERIA

Whenever the user issues a command or references a program, the system must search through directories to find the specified command or program. The search is regulated; that is, certain specified search rules are followed by the system.

The default search rules (those automatically used by the system) may be changed and/or supplemented by the user. The set_search_rules command allows the user to change the default search rules, and the add_search_rules and delete_search_rules commands allow the user to add or delete search directories in the default search rules. To check the current search rules, the user can invoke the print_search_rules command.

Adding another directory to be searched after the working directory is a convenient way for an entire project to share a group of special programs peculiar to the work of that project. After a user on the project adds this special directory to his search rules, he can execute any of the programs in that directory as easily as he executes system commands. This addition to the search rules means that each user on the project saves himself the time and cost of either copying each one of the programs or linking to each one.

The user can determine whether a system command or a user-written command with the same name is to be used by setting his search rules (refer to Section IV).

All four of the commands governing the search rules are documented in the MPM Commands.

TERMINAL SETTINGS

A wide variety of terminals can be used to access the Multics system. As an example of the diversity of terminal types that can be connected to Multics, consider the following terminals:

Tektronix Model 4012
Trendata Model 1000
GE TermiNet 1200
DTC 300 Series

A complete list of supported terminals would be inaccurate before it could be published since new ones are constantly being added to the group. Basically, Multics supports any Teletype model, the IBM 2741 and all its imitations, any general ASCII device, and display types of terminals.

Once a terminal is connected to the Multics system, the user is free to modify the terminal type associated with this terminal and/or the modes associated with terminal input/output by using the set_tty command. With this command the user can set various modes to effect certain terminal actions. Some examples are to make the terminal do any of the following: "echo" a carriage return when a linefeed is typed or vice versa; use full duplex (allows terminal to transmit and receive simultaneously); reprint the input line interrupted by output, such as a message from another user; hold output, such as a message from another user, until the user finishes the input line being typed. In addition, the user can set the delay timings (e.g., the number of delay characters that are output following a vertical tab or formfeed), specify the character-delete and line-delete symbols, and print the modes or delays currently in effect.

For a complete description of these and other functions of the set_tty command, refer to the MPM Commands.

walk_subtree COMMAND

The walk_subtree command is used to execute a specified command line in a specified directory and in all directories inferior to the specified directory. Through various options, the user can state the first and last levels at which the command line should be executed, or he can have the command line executed in the lowest level directory first.

The walk_subtree command is an especially convenient way to list certain segments in a group of directories (for example, to list all of a user's runoff segments in all of his directories). Refer to the MPM Commands for a complete description of this command.

WORD PROCESSING FACILITIES

Multics has a number of word processing tools that are collectively called WORDPRO. The WORDPRO subsystem assists users in the preparation and maintenance of online documents, gives users a list processing tool, and provides users with electronic mail (the mail and send_message commands described in Section IX).

In the document preparation area, WORDPRO offers several facilities including shorthand, so users need less keystrokes to accomplish tasks, and dictionaries, so spelling and consistency are checked automatically. Editing and text formatting facilities are also included (see the qedx description in Section VI and "Producing Manuscript Format" earlier in this section).

With the list processing tool, users first put lists of information online and then manipulate this information with various commands. For example, users can sort all the information according to any of the fields (such as address, last name, geographic area, job category, last order date, etc.), or select certain fields for printing, or combine lists of information. In addition, the list processing facility can be used to produce reports and form letters and distribute information.

APPENDIX A

GLOSSARY

access attributes
See access modes below.

access modes
Identify the kinds of access which may be set for a segment or directory. The access modes for segments are read (r), write (w), execute (e), and null (n). Those for directories are status (s), modify (m), append (a), and null (n). See Section VIII for more information.

access control list (ACL)
Describes the access modes associated with a particular segment or directory. The ACL is a list of user identifications and respective access modes. It is kept in the directory that catalogs the segment or directory.

"carriage return"
A term meaning that the typing mechanism moves to the first column of the next line. See newline below.

command level
A term used to indicate that lines input from a user's terminal are interpreted by Multics as a command (i.e., the line is sent to the command processor). A user is at command level when he logs in, when a command completes or encounters an error, or when the user stops command execution by issuing a quit signal. Command level is indicated by a ready message.

component
A logical part of an entryname. Entryname components are separated by a period (e.g., data_base is the second component of the entryname random.data_base.pl1).

daemon
One of several system processes that perform such tasks as process creation, backup, and printing segments on a line printer.

directory
A segment that contains information about other segments such as access attributes, names, and bit count.

directory (home)
The directory under which the user logs in. Usually this directory is named:

```
>udd>Project_id>Person_id
```

This directory is also known as the initial working directory.

directory (working)
The directory under which the user is doing his work. Often the working directory is also the home directory. (This is always true at log-in time.) The user may redefine his working directory by use of the change_wdir command.

entryname
The name by which a segment is cataloged in a directory. The entryname may contain more than one component.

equal convention
A method used by many commands to specify one or more characters in an entryname or, more often, a group of entrynames, by using a name that contains an equal sign (=). See Appendix B for more information.

info segments
The segments whose contents are printed by invoking the help command. These segments give information about various commands and subroutines as well as general information about the system.

link
A name in a directory that points to an entry in another directory. A link enables a user to access a segment without using the normal search rules; i.e., given proper access permission, he may specify the segment by entryname (as though it were cataloged in the working directory) without actually having to make a copy of the segment. This is one of the ways in which Multics facilitates sharing.

newline
A term used to indicate that the typing mechanism moves to the leftmost column of the next line. On Multics, this action is the result of the ASCII linefeed character (octal code 012). The terminal type determines which key(s) the user presses to perform the equivalent action (e.g., RETURN, LINE SPACE, or NL).

page (also known as record)
A unit of storage in Multics. A page contains 1024 36-bit words (4096 characters).

pathname
A character string that specifies a segment by its position in the storage system hierarchy. The pathname can be absolute or relative (see below).

pathname (absolute)
A pathname that identifies a specific segment with a name beginning at the root and continuing "down" to the specific segment.

pathname (relative)
A pathname that identifies a specific segment with a name beginning at the current working directory rather than the root.

Person_id
An identification code under which a particular user is registered on the system. It is usually some form of the user's name and contains both uppercase and lowercase characters. It may not contain blank characters.

Project_id
An identification code under which a particular project is registered on the system.

quit signal
The means by which users may interrupt Multics from processing a program or command lines. The quit signal is invoked by pressing the ATTN, INTERRUPT, BRK, or QUIT key on the terminal; Multics responds with a ready message.

ready message
A message that is printed each time the user is at command level, indicating the system is "ready" to accept another command.

segment
The basic unit of information within the Multics storage system. Each segment has access attributes, at least one name, and may contain data, programs, or be null.

star convention
A method used by many commands to specify a group of segments and/or directories by using one name that contains an asterisk (*). See Appendix B for more information.

suffix
The last component of an entryname; it usually specifies the segment type (e.g., xxx.basic is a segment containing BASIC source code).

User_id (user identification)
An access control name used to identify a user or group of users. It is a three-component name of the form Person_id.Project_id.tag; since the tag portion is rarely given, the term User_id is often defined as a Person_id.Project_id pair.

APPENDIX B

STORAGE SYSTEM AND COMMAND CONVENTIONS

This appendix briefly covers several Multics conventions that have been established for command names, command lines, arguments, and segment names. (Many of these conventions are described in Section III. Complete descriptions of these conventions are given in the MPM Reference Guide.) These conventions apply to most commands. However, some commands do not accept some conventions. Deviations are noted in the individual command descriptions in the MPM Commands.

COMMAND NAME CONVENTIONS

Command names never contain blanks. Commands that incorporate two or more words use an underscore (_) to separate words. Commands never have trailing underscores. (However, the majority of Multics subroutines do.) Most commands have an abbreviated name; the user may invoke the command by typing either the abbreviation or the full name. (This abbreviated name is independent of the abbreviation facility described in Section XI.)

COMMAND LINE CONVENTIONS

A command line consists of at least one command name and is terminated by a newline, i.e., the ASCII character whose octal code is 012. (Depending on the terminal, the proper key to use could be LINE SPACE, RETURN, or NL for newline.) The newline is a signal to Multics to begin action on the typed command. Two or more commands (with or without arguments) separated by semicolons may be typed on a single line.

ARGUMENT CONVENTIONS

Commands do not always require arguments. When they do, arguments are separated from the command name (and from each other) by one or more spaces. An argument may contain blanks if it is enclosed in quotes (e.g., "Tom Smith").

If a command requires a specific number of arguments, failure to provide the proper number may result in incomplete or incorrect action. In such cases, an error message is printed followed by a ready message. For many commands, the order in which arguments are typed is significant.

Control arguments for a command start with a hyphen (-) in order to differentiate between other arguments and to avoid ambiguity. A control argument specifies some modification to the type of action performed. In most cases the order of these arguments is unimportant.

PATHNAMES

Use of the Absolute Pathname

Most commands require an argument in the form of a pathname to specify the segment on which the command will act. The name that uniquely identifies a segment among all other segments in Multics is called an absolute pathname. The absolute pathname is used when a relative pathname is awkward to use; e.g., the specified segment is not located "near" the user's working directory.

Use of the Relative Pathname

Relative pathnames are often used instead of absolute pathnames because they are shorter and thus more convenient to type. They are relative to the user's working directory rather than to the storage system root. Relative pathnames do not begin with the greater-than character, although they can contain the greater-than character.

Often it is easier to redefine the working directory than to type absolute pathnames as arguments. Most users redefine their working directories (using the change_wdir command) in order to be able to use relative pathnames. They need to be sure that they have access to segments in these directories, however.

ENTRYNAME

The simplest form of relative pathname is called an entryname. For example,

```
alg.pl1
```

is a relative pathname for >udd>ProjA>TSmith>alg.pl1 if the working directory is >udd>ProjA>TSmith.

Each segment and directory may have more than one entryname. Usually, the user assigns a short entryname for typing convenience. For example, the entryname shown above, alg.pl1, could be the short name for an entryname like algorithm_test.pl1.

LONGER RELATIVE PATHNAMES

A longer relative pathname might be:

```
games>tic_tac_toe
```

where games is cataloged in the user's working directory, and tic_tac_toe is cataloged in games directory.

NAMING CONVENTIONS FOR MULTIPLE COMPONENT ENTRYNAMES

An entryname, the name by which a segment is cataloged in a directory, may not contain greater-than or less-than characters and should not contain other special symbols such as asterisk or equals characters. Also, entrynames containing blanks should be avoided since the command language uses blanks to delimit command names and arguments. Many entrynames have several components, separated by periods. By convention, entrynames for source programs in Multics have as the last component the name of the language in which they are written.

SPECIAL SYMBOLS

Less-Than Character

Less-than characters may take the place of directory names in a relative pathname. Each less-than character, like each greater-than character, denotes a hierarchy level; however, each less-than character indicates one level (one directory) back up the hierarchy, starting at the working directory and going up toward the root.

For example, in Figure B-1, if the working directory is TSmith, the absolute pathname for segment "a" would be:

```
>udd>ProjA>Rolf>a
```

A much briefer name, using the less-than character is:

```
<Rolf>a
```

where "<" represents the ProjA directory (one level back up the hierarchy from the TSmith directory), and the Rolf directory "catalogs" segment a.

Star Convention

The star convention is used by standard Multics commands to reference groups of segments and/or directories. If an asterisk is used for one component of an entryname, the command matches any name in that particular component position. For example,

```
*.pl1
```

matches any two-component entryname whose second component is pl1. And,

```
*.*.red
```

matches any three-component entryname whose third component is red.

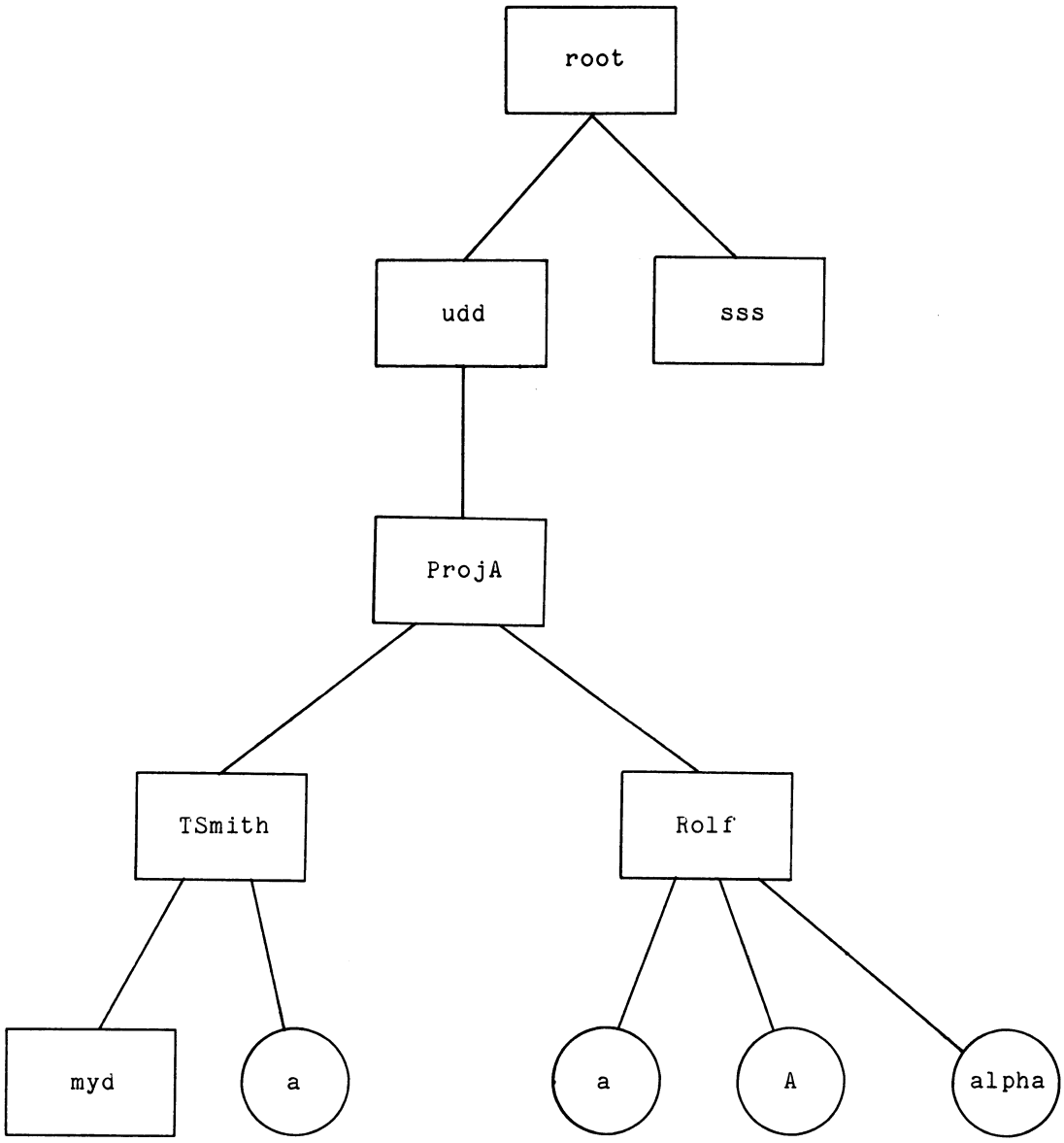

Figure B-1. Sample Hierarchy

A double asterisk may be used to match any number of components (including none). For example, if the user wants a list of all the segments in his working directory with "blue" as the first component, he types:

```
list blue.**
```

then, no matter how many other components exist in the entryname (including none), the command considers them a "match":

```
blue
blue.red
blue.x.y
blue.x.y.z.n
```

Another more common example of the use of the double asterisk with the list command is to list all entries of a certain type, for example all FORTRAN source program segments or all qedx macro segments. To get a list of all FORTRAN source program segments in the current directory, the user types:

```
list **.fortran
```

for the qedx macro segments, the user types:

```
list **.qedx
```

The double asterisk may also be used to designate all entries in the specified directory. For example, to give read permission to Jane Doe for every segment in his current working directory, the user types:

```
set_acl ** r JDoe
```

Not all commands implement the star convention. The user should consult the MPM Commands under the proper command before using the star convention.

Equal Convention

The equal sign (=) is used by standard Multics commands in the second of a pair of arguments to indicate equivalency with the same component position in the first argument. For example, if the user types:

```
add_name add_index_info.pl1 aii.=
```

the command interprets the second argument as aii.pl1. Thus, the segment may now be referenced as either add_index_info.pl1 or aii.pl1.

Many users find that they use the double equal sign (==) more often than the single equal sign. The double equal sign component of an entryname represents all components in the first argument that have no corresponding component in the second argument (the one containing the ==). For example, if the user types:

```
add_name random_figures.data_base.pl1 random.==
```

the command interprets the second argument as random.data_base.pl1 and adds that name to the segment.

The real power of the equal convention is most evident when it is used with the star convention. For example, by typing:

```
add_name **.basic ==old_basic
```

the user adds names to all segments in his current working directory that have a last component of basic. The added name is the same as the original except that the basic suffix becomes old_basic (e.g., compacts.gas_data.basic would also have the name compacts.gas_data.old_basic). As another example, typing:

```
rename **.data_base.pl1 ==.db.=
```

changes the names of all segments in the current working directory that end in data_base.pl1 so they now end in db.pl1; everything else in the names is the same (e.g., random.data_base.pl1 would become random.db.pl1).

Not all commands implement the equal convention. The user should consult the MPM Commands under the proper command before using the equal convention.

APPENDIX C

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 in terms of their function. The Multics command repertoire is divided, according to the command function, 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, Assemblers, 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

Since many commands can perform more than one function, 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 the user may obtain by invoking the help command as described in Section V.

Access to the System

(preaccess requests)	used to inform system of special terminal attributes
dial	connects an additional terminal to an existing process
enter, enterp	connects an anonymous user to the system (used at dialup only)
login	connects registered user to the system (used at dialup only)
logout	disconnects user from the system

Storage System, Creating and Editing Segments

adjust_bit_count	sets bit count of a segment to last nonzero word or character
compare_ascii	compares ASCII segments, reporting differences
edm	allows inexpensive, easy editing of ASCII segments
indent	indents a PL/I source segment to make it more readable
program_interrupt	provides for command reentry following a quit or an unexpected signal
qedx	allows sophisticated editing, including macro capabilities
runoff	formats a text segment according to internal control words
runoff_abs	invokes the runoff command in an absentee job
set_bit_count	sets the bit count of a segment to a specified value
sort_seg	sorts ASCII segments according to ASCII collating sequence

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
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
delete	deletes a segment or multisegment file and questions user if it is protected
delete_force	deletes a segment or multisegment file without question
link	creates a storage system link to another segment, directory, link, or multisegment file
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
truncate	truncates a segment to a specified length
unlink	removes a storage system link
vfile_adjust	adjusts structured and unstructured files

Storage System, Directory Manipulation

add_name	adds a name to a segment, directory, link, or multisegment file
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
fs_chname	renames a segment, directory, link, or multisegment file, bypassing naming conventions
link	creates a storage system link to another segment, directory, link, or multisegment file
list	prints directory contents
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
unlink	removes a storage system link
vfile_status	prints the apparent type and length of storage system files

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
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_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_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_rules	prints names of directories searched for segments referenced dynamically
print_translator_search_rules	prints current translator search rules
print_wdir	prints name of current working directory
set_search_rules	allows users to modify search rules
set_translator_search_rules	sets translator search rules
terminate terminate_refname terminate_segno terminate_single_refname	removes a segment from process address space
where	uses current search rules to locate and print pathname of a segment

Formatted Output Facilities

cancel_daemon_request	cancels a previously submitted daemon request
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 BCD
list_daemon_requests	prints list of print and punch requests currently queued
print	prints an ASCII segment
runoff	formats a text segment according to internal control words
runoff_abs	invokes the runoff command in an absentee job

Language Translators, Compilers, Assemblers, 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
display_cobol_run_unit	displays the current state of a COBOL run unit
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
new_fortran	invokes the new FORTRAN compiler
old_fortran	invokes the old FORTRAN compiler
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
qedx	allows sophisticated editing, including macro capabilities
run_cobol	executes a COBOL run unit in a main program
runoff	formats a text segment according to internal control words
runoff_abs	invokes the runoff command in an absentee job
set_cc	sets the carriage control transformation for FORTRAN files
stop_cobol_run	terminates the current COBOL run unit

Object Segment Manipulation

archive	packs segments together to save physical storage
bind	packs two or more object segments into a single executable segment

Debugging and Performance Monitoring Facilities

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_pl1io_error	displays diagnostic information about PL/I I/O errors
dump_segment	prints segment contents in octal, ASCII, or BCD
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
repeat_query	repeats the last query by the command_query_ subroutine
reprint_error	reprints an earlier system condition message
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_daemon_request	cancels a previously submitted print or punch request
close_file	closes open PL/I and FORTRAN files
console_output	restores terminal output to the terminal
copy_cards	copies card decks read by I/O Daemon
copy_file	copies records from an input file to an output file

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_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
set_cc	sets the carriage control transformation for FORTRAN files
set_tty	prints and sets modes associated with user's terminal
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_rules	allows users to change (insert) search rules dynamically
answer	answers questions normally asked of the user
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
console_output	restores terminal output to the terminal
delete_search_rules	allows users to delete current search rules
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
if	conditionally executes a command line
general_ready	allows user to format ready messages
get_com_line	prints the maximum length of the command line
get_system_search_rules	prints definitions of site-defined search rule keywords
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
print_default_wdir	prints name of default working directory
print_search_rules	prints names of directories searched for segments referenced dynamically
print_translator_search_rules	prints current translator search rules
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
set_com_line	sets the maximum length of the command line
set_search_rules	allows users to modify search rules
set_translator_search_rules	sets translator search rules
start	continues process at point of a quit or an unexpected signal interruption

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
mail	prints or sends mail
print_auth_names	prints names of sensitivity levels and access categories for an installation
print_messages	prints any pending messages
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

check_info_segs	checks information (and other) segments for changes
help	prints special information segments
how_many_users	prints the number of logged-in users
print_motd	prints the portion of the message of the day that changed since last printed
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
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
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
decode	deciphers segment, given proper coding key
encode	enciphers segment, given a coding key
memo	allows users to set reminders for later printout
progress	prints information about the progress of a command as it is being executed
walk_subtree	executes a command line in all directories below a specified directory

APPENDIX D

MULTICS edm EDITOR

The edm command, which is a simple Multics context editor, is used for creating and editing ASCII segments. To invoke edm, the user types:

```
edm pathname
```

where 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. The user can change from one mode to the other by issuing the mode change character: a period (followed by a "carriage return") when this 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; for these requests only the single character is accepted by edm to initiate the corresponding action.

REQUESTS

Various edm requests and their indicators 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 the new user will need as he begins using Multics. 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 for the new user.

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

2. The user should not issue a quit signal (press ATTN, BRK, INTERRUPT, etc.) while in the editor unless he is prepared to lose all of the work he has done since the last w (write) request. However, if a quit signal is issued, the user may return to edm request level without losing his work by issuing the program_interrupt command.

3. If the user has 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), the user loses only the work done since the last w request.

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

5. As the user becomes more familiar with the use of edm, he may conclude that it provides verification responses more often than necessary, thus slowing him down. He may use the k request to "kill" the verification response. However, once the user feels confident enough to use the k request, he is probably ready to begin using the more sophisticated editor, qedx. The qedx editor provides the user with a repertoire of more concise and powerful requests, permitting more rapid work.

REQUEST DESCRIPTIONS

The following edm requests are the ones that the new user will find most useful as he begins working on Multics. Examples are included to help the new user see the practical use of the requests.

Backup (-) Request

The backup request moves the pointer backward (toward the top of the segment) the number of lines specified by the user 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 the user wants the pointer at the line beginning with the word "sum," he types:

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

If the user does not 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 the user 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 the user wants to check the implicit line number of the current line, he issues this request and edm responds with a line number.

```
! =
   143
```

Comment Mode (,) Request

When the user invokes 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 the user types the mode change character (a period) as the only entry on a line.

To print the lines in comment mode means that edm prints the line without the carriage return, switches to input mode, and waits for the user's comment entry for that line. When the user gives his comment line and a carriage return, edm repeats the process with the next line.

If the user has no comment for a particular line, he types only a carriage return and edm prints the next line in comment mode. When the user wants to leave comment mode and return to edit mode, he types--as his 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 the user 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 the user leaves the comment mode request and returns to edit mode.

For example, when a user finishes typing information into a segment, he 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 the user has a lot of information to type in input mode; if he sees some mistakes in data previously typed, he can switch to edit mode, correct the error, then issue the bottom request and continue typing his 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 by the user. Deletion begins at the current line and continues according to the user's request. For example, to delete the current line plus the next five lines, the user types:

```
d6
```

If the user issues the delete request without specifying a number, only the current line is deleted. (That is, the user 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 designated by the user. 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 the user is 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 the user
```

When the user types the string, he 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 the user issues 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 the user to place a new line of information after the current line.

If the user invokes the insert request without specifying any new text, a blank line is inserted after the current line. If the user types text after the insert request, he 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 the user issued the following insert request:

```
i put skip;
```

the result would be:

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

If the user wants to insert a new line at the beginning of the segment, he first issues 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), or substitute (s) requests. To restore responses to these requests, the user issues the verbose (v) request.

It is recommended that the new user not use the kill request until he is thoroughly familiar with edm. The responses given in verbose mode are helpful; they offer an immediate check for the user by allowing him to see the results of his request.

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 by the user. If the user invokes the next request without specifying a number, the pointer is moved down one line. When the user does specify the number of lines he wants the pointer to move, the pointer is set to the specified line. For example, if the user types:

```
n4
```

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

Print (p) Request

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

If the user wants to see the current line and the next three lines, he types:

```
! 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 the user prints the entire segment, these lines are identified as "No line" and "EOF" respectively.

Quit (q) Request

The quit request is invoked by the user when he wants to exit from edm and return to command level.

For the user's convenience and protection, edm prints a warning message if the user does not issue a write (w) request to save his latest editing changes before he issues the quit request. The message reminds the user that his changes will be lost and asks if he still wishes to quit.

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

If the user answers by typing no, he is still in edit mode and can then issue a write request to save his work. If he instead answers by typing yes, he exits from edm and returns to command level.

Retype (r) Request

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

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, the user types the retype request and a carriage return.

Substitute (s) Request

The substitute request allows the user to change every occurrence of a particular character string with a new character string in the number of lines he indicates. If the user is 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 the user 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 the user wants to search the next three lines and change the word "sum" to "total," he types:

```
! 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 the user does not specify the number of lines he wants searched, edm only searches the current line. If the user does 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. The user 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 the user 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), or substitute (s) requests.

Actually, the user does not need to issue the verbose request to cause edm to print the responses; when he invokes edm, the verbose request is in effect. The only time the user needs 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 specified by the user. (The pathname can be either absolute or relative.)

If the user does not specify a pathname, the segment is saved under the name used in the invocation of edm. 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 the user does not specify a pathname and he did not use a pathname when he invoked edm, an error message is printed and edm waits for another request. If this happens, the user should reissue the write request, specifying a pathname.

MISCELLANEOUS

A

Q

R

S

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

CUT ALONG LINE

TITLE: LEVEL 68 MULTICS INTRODUCTORY USERS' GUIDE

ORDER NO. AL40, REV. 1

DATED JULY 1977

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be promptly investigated by appropriate technical personnel and action will be taken as required. If you require a written reply, check here and furnish complete mailing address below. ☐

FROM: NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

CUT ALONG LINE

FOLD ALONG LINE

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

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

Business Reply Mail
Postage Stamp Not Necessary if Mailed in the United States

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

FOLD ALONG LINE

Honeywell