Honeywell

MULTICS PROGRAMMERS' MANUAL COMMANDS AND ACTIVE FUNCTIONS ADDENDUM C

SERIES 60 (LEVEL 68)

SOFTWARE

SUBJECT:

Additions and Changes to the Standard Multics Commands.

SPECIAL INSTRUCTIONS:

This is the third addendum to AG92, Revision 1, dated January 1975.

Insert the attached pages into the manual according to the collating instructions on the back of this cover. The following commands and preaccess requests are new and, therefore, do not contain change bars:

attach_lv
copy_file
detach_lv
repeat_query
hello
slave

The code command has been renamed encode and does not contain change bars. Throughout the rest of the manual, change bars in the margins indicate technical additions and changes; asterisks denote deletions. These changes will be incorporated into the next revision of the manual.

NOTE: Insert this cover after the manual cover to indicate the updating of the document with Addendum C.

SOFTWARE SUPPORTED:

Multics Software Release 4.0

DATE:

July 1976

ORDER NUMBER:

AG92C, Rev. 1

15787
1576
Printed in U.S.A.

COLLATING INSTRUCTIONS

To update this manual, remove old pages and insert new pages as follows:

Remove	Insert
v through viii	v through viii
1-1 through 1-7.1	1-1 through 1-7.1
3-7, 3-8	3-7, 3-8
3-11, 3-12	3-11, 3-12
3-24.1, 3-24.2	3-24.1 through 3-24.4
3-45, 3-46	3-45, 3-46
3-49, 3-50	blank, 3-50
3-54.1 through 3-56.2	3-54.1 through 3-56.2
3-87 through 3-96	3-87 through 3-96
3-99 through 3-104	3-99 through 3-104
3-111, 3-112	3-111 through 3-112
3-125, 3-126	3-125, 3-126
3-131, 3-132	3-131, 3-132
3-183 through 3-185.1	3-183 through 3-185.3
3-193 through 3-200	3-193 through 3-200
3-202.1, blank	3-202.1, 3-202.2
3-203 through 3-208.2	3-203 through 3-208.2
3-211 through 3-216	3-211 through 3-216
3-229, 3-230	3-229, 3-230
	3-274.1, blank
3-305 through 3-308	3-305 through 3-308
3-318.3 through 3-326	3-318.3, 3-326
3-333.1 through 3-334.2	3-333.1 through 3-334.2

File No.: 1L13

7/76

AG92C

3-337 through 3-340	3-337 through 3-340
4-5, 4-6	4-5, 4-6
4-9 through 4-11	4-9 through 4-13
A-1, A-2	

This page intentionally left blank.

Honeywell

MULTICS PROGRAMMERS' MANUAL COMMANDS AND ACTIVE FUNCTIONS ADDENDUM B

SERIES 60 (LEVEL 68)

SOFTWARE

SUBJECT:

Additions and Changes to the Standard Multics Commands.

SPECIAL INSTRUCTIONS:

This is the second addendum to AG92, Revision 1, dated January 1975.

Insert the attached pages into the manual according to the collating instructions on the back of this cover. The help and list commands have been totally revised and, therefore, do not contain change bars; the vfile_adjust and vfile_status commands are new and also do not contain change bars. Throughout the rest of the manual, change bars in the margins indicate technical additions and changes; asterisks denote deletions. These changes will be incorporated into the next revision of the manual.

NOTE: Insert this cover after the manual cover to indicate the updating of the document with Addendum B.

SOFTWARE SUPPORTED:

Multics Software Release 3.1

DATE:

March 1976

ORDER NUMBER:

AG92B, Rev. 1

15038
1276
Printed in U.S.A.

COLLATING INSTRUCTIONS

To update this manual, remove old pages and insert new pages as follows:

Remove	Insert
iii through viii	iii, iv v, vi vii, blank
1-3 through 1-6	1-3 through 1-6
1-8 through 1-18	
2-25, blank	2-25, 2-26
3-3 through 3-6	3-3 through 3-6
3-11, 3-12	3-11, 3-12
3-19 through 3-22	3-19 through 3-22
3-43, 3-44	3-43, 3-44
3-59, 3-60	3-59, 3-60
3-85, 3-86	3-85, 3-86
3-113, 3-114	3-113, 3-114
3-119 through 3-124	3-119 through 3-124
3-141, 3-142	3-141, 3-142
3-149, 3-150	3-149, 3-150
3-153 through 3-164	3-153 through 3-164
3-167 through 3-170	3-167, 3-168 3-169, 3-170 3-170.1, 3-170.2 3-170.3, 3-170.4
3-173 through 3-176	3-173 through 3-176
3-183, 3-184	3-183, 3-184
3-187 through 3-190	3-187, 3-188 3-189, 3-190 3-190.1, 3-190.2 3-190.3, 3-190.4 3-190.5, 3-190.6 3-190.7, 3-190.8
3-193, 3-194	3-193, 3-194
3-213 through 3-216	3-213 through 3-216
3-219 through 3-222	3-219 through 3-222

File No.: 1L13

3-225, 3-226	3-225, 3-226
3-301 through 3-304	3-301 through 3-304
3-318.1, 3-318.2	3-318.1, 3-318.2
	3-334.1, 3-334.2 3-334.3, blank
4-5 through 4-8	4-5 through 4-8

This page intentionally left blank.

Honeywell

MULTICS PROGRAMMERS' MANUAL COMMANDS AND ACTIVE FUNCTIONS ADDENDUM A

SERIES 60 (LEVEL 68)

SOFTWARE

SUBJECT:

Additions and Changes to the Standard Multics Commands, Including a New Section on Access to the System.

SPECIAL INSTRUCTIONS:

This is the first addendum to AG92, Revision 1, dated January 1975.

Insert the attached pages into the manual according to the collating instructions on the back of this cover. The prelogin requests in Section IV plus the following commands and active function reflect new information issued in this addendum and do not contain change bars.

assign_resource	print_auth_names
cancel_cobol_program	print_proc_auth
cobol	print_request_types
copy_cards	probe
cumulative_page_trace	run_cobol
dial	set_tty
display_cobol_run_unit	sort
display_pl1io_error	stop_cobol_run
format_cobol_source	trace
list_resources	unassign_resource
page_trace	have_mail (active function)

Throughout the rest of the manual, change bars in the margins indicate technical additions and changes; asterisks denote deletions. These changes will be incorporated into the next revision of the manual.

NOTE: Insert this cover after the manual cover to indicate the updating of the document with Addendum A.

SOFTWARE SUPPORTED:

Multics Software Release 3.0

DATE:

September 1975

ORDER NUMBER:

AG92A, Rev. 1

13935
1975
Printed in U.S.A.

COLLATING INSTRUCTIONS

To update this manual, remove old pages and insert new pages as follows:

Remove	Insert
title, preface	title, preface
iii through vii	iii through viii
1-1 through 1-8	1-1 through 1-6 1-7, 1-7.1 blank, 1-8
2-1, 2-2	2-1, 2-2
2-25	2-25
	3-24.1, 3-24.2
3-39 through 3-42	3-39, blank 3-39.1, 3-40 3-41, 3-42
	3-48.1, 3-48.2 3-48.3, blank
	3-54.1, blank
3-55, 3-56	3-55, 3-56 3-56.1 through 3-56.4
3-99, 3-100	3-99, 3-99.1 3-99.2, 3-100
3-105 through 3-110	3-105 through 3-110 3-110.1, blank
3-123 through 3-126	3-123 through 3-126
	3-138.1, blank
3-145, 3-146	3-145, 3-146
3-177 through 3-186	3-177 through 3-184 3-185, 3-185.1 blank, 3-186
3-195, 3-196	3-195, 3-196

File No.: 1L13

3-203 through 3-208	3-202.1, 3-203/3-204 3-205 through 3-208 3-208.1, 3-208.2
3-215 through 3-218	3-215, 3-216 3-216.1, blank 3-217, 3-217.1 3-217.2, 3-218
3-223, 3-224	3-223, 3-224
3-227, 3-228	3-227, blank 3-227.1, 3-228
	3-230.1, 3-230.2
	3-232.1 through 3-232.30
	3-278.1, 3-278.2
	3-318.1, 3-318.12 3-318.13, blank
3-323 through 3-326	3-323 through 3-326 3-326.1, blank
	3-328.1 through 3-328.10
3-333, 3-334	3-333, blank 3-333.1, 3-334
	4-1 through 4-11

This page intentionally left blank.

March 28, 1975

Massachusetts Institute of Technology
Information Processing Center
Cambridge, Massachusetts 02139
Publications
39-483

This manual (MPM Volume 3) is part of a rather extensive revision of the Honeywell MPM. It is meant to replace the following sections of Part 2 of the M.I.T. MPM:

1.5	Constructing and Interpreting Names
1.6	Command and Active Function Name Abbreviations
1.8 - 1.15	Active Functions
9	Commands

Note the following command writeups which were in revision 15 of the MPM have been removed for the reason given. The user may wish to retain these writeups until they are published elsewhere.

Command	Reason*
alm	SWG
alm_abs	SWG
archive_sort	SWG
basic_run	MIT-Dartmouth
decam	MIT
display_component_name	SWG
endfile	Obsolete (Use close_file)
hold	Obsolete
iomode	Obsolete
lisp	MIT-LISP
lisp_compiler	MIT-LISP
make_peruse_text	SWG
names	SWG
page_trace	SWG
peruse_text	SWG
print_bind_map	SWG
print_dartmouth_library	MIT-Dartmouth
print_entry_usage	Obsolete
print_link_info	SWG
print_linkage_usage	SWG
reorder_archive	SWG
set_dartmouth_library	MIT-Dartmouth
set_search_dirs	Obsolete (Use set_search_rules or add_search_rules)
sort_file	Obsolete (Use sort_seg)
v5basic	MIT-Dartmouth

*SWG means moved to the Subsystem Writers' Guide (Vol. 5); MIT means the documentation is for MIT users only.

Honeywell

MULTICS PROGRAMMERS' MANUAL COMMANDS AND ACTIVE FUNCTIONS

SOFTWARE

SERIES 60 (LEVEL 68)

SUBJECT:

Description of Standard Multics Commands, Including Details of Their Calling Sequence and Usage.

SPECIAL INSTRUCTIONS:

This manual is one of four manuals that constitute the _Multics Programmer's Manual_ (MPM).

Reference Guide	Order No. AG91
Commands and Active Functions	Order No. AG92
Subroutines	Order No. AG93
Subsystem Writers' Guide	Order No. AK92

This manual supersedes AG92, Rev. 0, and its Addendum A. The manual has been extensively revised; therefore, marginal change indicators have not been included in this edition.

SOFTWARE SUPPORTED:

Multics Software Release 2.0

DATE:

January 1975

ORDER NUMBER:

AG92, Rev. 1

PREFACE

Primary reference for user and subsystem programming on the Multics system is contained in four manuals. The manuals are collectively referred to as the Multics Programmers' Manual (MPM). Throughout this manual, references are frequently made to the MPM. For convenience, these references will be 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

The MPM Reference Guide contains general information about the Multics command and programming environments. It also defines items used throughout the rest of the MPM. And, in addition, describes such subjects as the command language, the storage system, and the input/output system.

The MPM Commands is organized into four sections. Section I contains a list of the Multics command repertoire, arranged functionally. It also contains a discussion on constructing and interpreting names. Section II describes the active functions. Section III contains descriptions of standard Multics commands, including the calling sequence and usage of each command. Section IV describes the requests used to gain access to the system.

The MPM Subroutines is organized into three sections. Section I contains a list of the subroutine repertoire, arranged functionally. Section II contains descriptions of the standard Multics subroutines, including the declare statement, the calling sequence, and usage of each. Section III contains the descriptions of the I/O modules.

The MPM Subsystem Writers' Guide is a reference of interest to compiler writers and writers of sophisticated subsystems. It documents user-accessible modules that allow the user to bypass standard Multics facilities. The interfaces thus documented are a level deeper into the system than those required by the majority of users.

Examples of specialized subsystems for which construction would require reference to the MPM Subsystem Writers' Guide are:

- A subsystem that precisely imitates the command environment of some system other than Multics.
- A subsystem intended to enforce restrictions on the services available to a set of users (e.g., an APL-only subsystem for use in an academic class).
- A subsystem that protects some kind of information in a way not easily expressible with ordinary access control lists (e.g., a proprietary linear programming system, or an administrative data base system that permits access only to program-defined, aggregated information such as averages and correlations).

Several cross-reference facilities help locate information:

- Each manual has a table of contents that identifies the material (either the name of the section and subsection or an alphabetically ordered list of command and subroutine names) by page number.
- Each manual contains an index that lists items by name and page number.

CONTENTS

Contents (cont)

SECTION I

MULTICS COMMAND ENVIRONMENT

The Multics command environment discussion, presented in this section, consists of two major parts. The first part of the discussion lists the Multics command repertoire arranged according to function and described briefly. The second part of this section establishes rules for constructing and interpreting the various types of names used on Multics and details the several standard conventions for using these names.

REFERENCE TO COMMANDS BY FUNCTION

The Multics command repertoire is divided according to the command function into the following 18 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
GCOS Environment
Miscellaneous Tools

Since many commands can perform more than one function, they are listed in more than one group.

Detailed descriptions of the commands are given in Section III, arranged in alphabetical rather than functional order. Detailed descriptions of the requests used to gain access to the system are given in Section IV. Many of the commands in Section III also have online descriptions, which the user may obtain by invoking the help command.

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
basic_system	provides a standard source editor and run dispatcher for interactive use with BASIC
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

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_acl	prints an ACL entry
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
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_wdir	prints name of current working directory
set_search_rules	allows users to modify search rules
terminate terminate_refname terminate_segno terminate_single_refname	removes a segment from process address space
where	prints absolute 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
basic_system	provides a standard source editor and run dispatcher for interactive use with BASIC
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
format_cobol_source	converts free-form COBOL source to fixed-format COBOL source
fortran	compiles FORTRAN programs
fortran_abs	invokes the FORTRAN compiler in an absentee job
indent	indents a PL/I source segment to make it more readable
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
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 segment

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
basic_system	provides a standard source editor and run dispatcher for interactive use with BASIC
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
file_output	directs terminal output to a segment
get_com_line	prints the maximum length of the command line
line_length	allows users to control maximum length of output lines
memo	allows users to set reminders for later printout
new_proc	creates a new process with a new address space
print_default_wdir	prints name of default working directory
print_search_rules	prints names of directories searched for segments referenced dynamically
print_wdir	prints name of current working directory
program_interrupt	provides for command reentry following a quit or an unexpected signal
ready	prints the ready message: a summary of CPU time, paging activity, and memory usage
ready_off	suppresses the printing of the ready message
ready_on	restores the printing of the ready message
release	discards process history retained by a quit or an unexpected signal interruption
repeat_query	repeats the last query by the command_query_ subroutine

reprint_error	reprints an earlier system condition message
set_com_line	sets the maximum length of the command line
set_search_rules	allows users to modify 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
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
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 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

GCOS Environment

gcos	invokes GCOS environment simulator to run one GCOS job
gcos_card_utility	copies card image files, translating from GCOS format to ASCII or vice-versa
gcos_sysprint	converts GCOS BCD sysout print file to ASCII file suitable for use with the dprint command
gcos_syspunch	converts GCOS BCD sysout punch file to file suitable for use with the dpunch command

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

SECTION II

ACTIVE FUNCTIONS

GROUPING OF ACTIVE FUNCTIONS

This section describes the active functions of interest to most Multics users. The active functions have been divided into seven operational groupings: Logical Active Functions, Arithmetic Active Functions, Character String Active Functions, Segment Name Active Functions, Date and Time Active Functions, Question Asking Active Functions, and User Parameter Active Functions. "The Command Language" in Section I of the MPM Reference Guide describes the purpose of active functions and illustrates their use.

In the usage lines throughout this section, any argument preceded and followed by a minus sign (-) is an optional argument. For more information on usage formats used in this manual, see "Command Descriptions" in Section III.

The following alphabetical list of active functions indicates which grouping contains each description.

Active Function	Group	Page
and	logical	2-2
ceil	arithmetic	2-6
date	date and time	2-18
date_time	date and time	2-18
day	date and time	2-18
day_name	date and time	2-19
directories, dirs	segment name	2-12
directory	segment name	2-12
divide	arithmetic	2-6
entry	segment name	2-12
equal	logical	2-3
exists	logical	2-3
files	segment name	2-13
floor	arithmetic	2-6
format_line	character string	2-8
get_pathname, gpn	segment name	2-13
greater	logical	2-4
have_mail	user parameter	2-25
home_dir	segment name	2-13
hour	date and time	2-19
index	character string	2-9
index_set	character string	2-9
length	character string	2-10
less	logical	2-4
links	segment name	2-14
long_date	date and time	2-19
max	arithmetic	2-6
min	arithemtic	2-6
minus	arithemtic	2-7

Active Function	Group	Page
minute	date and time	2-19
mod	arithmetic	2-7
month	date and time	2-19
month_name	date and time	2-20
nequal	arithmetic	2-4
ngreater	arithmetic	2-4
nless	arithmetic	2-4
nondirectories, nondirs	segment name	2-14
nonlinks, branches	segment name	2-14
nonsegments, nonsegs	segment name	2-14
not	logical	2-5
or	logical	2-5
path	segment name	2-15
pd	segment name	2-15
plus	arithmetic	2-7
query	question asking	2-21
quotient	arithmetic	2-7
response	question asking	2-21
search	character string	2-10
segments, segs	segment name	2-15
string	character string	2-10
strip	segment name	2-16
strip_entry, spe	segment name	2-16
substr	character string	2-10
suffix	segment name	2-16
system	user parameter	2-22
time	date and time	2-20
times	arithmetic	2-7
trunc	arithmetic	2-8
unique	segment name	2-17
user	user parameter	2-24
verify	character string	2-11
wd	segment name	2-17
year	date and time	2-20

LOGICAL ACTIVE FUNCTIONS

The logical active functions described below return a character string value of either true or false. They are intended to be used with the &if control statement of the exec_com command.

In the descriptions of logical active functions, the term decimal number is used. This includes integers (such as 5 and 1024), real numbers (such as 1.37 and .5), and floating point numbers (such as -1.3e+4 and 12.556e+0).

Name: and

Usage

```
[and args]
```

where args are character strings that must have one of the values true or false (if not, an error diagnostic is issued and the value is undefined). If all the args = true, then true is returned; otherwise, false is returned.

Name: equal

Usage

    [equal A B]

where A and B are any character strings. The value true is returned if A = B; otherwise false is returned.

Name: exists

Usage

    [exists KEY arg]

The exists active function checks for the existence of various types of items depending on the value of KEY, where KEY may have one of the following values:

entry	returns true if an entry with pathname arg exists; otherwise it returns false.
branch	returns true if a branch with pathname arg exists; otherwise it returns false.
segment	returns true if a nondirectory segment with pathname arg exists; otherwise it returns false.
directory	returns true if a directory with pathname arg exists; otherwise it returns false.
link	returns true if a link with pathname arg exists; otherwise it returns false.
non_null_link	returns true if a link with pathname arg exists and points to an existing segment, directory, or multisegment file; otherwise it returns false.
argument	returns true if it has been passed an argument arg; otherwise it returns false.
msf	returns true if a multisegment file with pathname arg exists; otherwise it returns false.
file	returns true if a segment or multisegment file with pathname arg exists; otherwise it returns false.

Name: greater

Usage

    [greater A B]

where A and B are character strings. If A > B, then true is returned; otherwise, false is returned.

Name: less

Usage

    [less A B]

where A and B are character strings. If A < B, then true is returned; otherwise, false is returned.

Name: nequal

Usage

    [nequal A B]

where A and B are the character string representation of decimal numbers. If A = B, the value true is returned; otherwise, false is returned.

Name: ngreater

Usage

    [ngreater A B]

where A and B are the character string representation of decimal numbers. If A > B, then true is returned; otherwise, false is returned.

Name: nless

Usage

    [nless A B]

where A and B are the character string representation of decimal numbers. If A < B, the value true is returned; otherwise, false is returned.

Name: not

Usage

```
     [not A]
```

where A is a character string. If A = true, then false is returned. If A = false, then true is returned; otherwise, an error diagnostic is issued.

Name: or

Usage

```
     [or args]
```

where args are character strings that must have one of the values true or false, (if not, an error diagnostic is issued and the value is undefined). If any arg = true, then true is returned; otherwise false is returned.

Example

The following example illustrates the use of one of the active functions discussed in this description. It involves the use of the &if control statement of the exec_com command. (See the description of the exec_com command.)

```
     &if [equal [wd] [home_dir]]

     &then &goto elsewhere

     &else change_wdir [home_dir]
```

This example compares the pathname of the working directory with the pathname of the home directory, and if they are not the same, changes the working directory to be the home directory.

ARITHMETIC ACTIVE FUNCTIONS

The following active functions all perform some arithmetic operation on their arguments and return the character string representation of the result.

In the descriptions of arithmetic active functions, the term decimal number is used. This includes integers (such as 5 and 1024), real numbers (such as 1.37 and .5), and floating point numbers (such as -1.3e+4 and 12.556e+0).

Name: ceil

Usage

```
[ceil A]
```

where A is the character string representation of a decimal number. The value returned is the smallest integer $\geq$ A.

Name: divide

Usage

```
[divide A B]
```

where A and B are the character representations of decimal numbers. The value returned is the integer part of the value of A/B.

Name: floor

Usage

```
[floor A]
```

where A is the character string representation of a decimal number. The value returned is the largest integer $\leq$ A.

Name: max

Usage

```
[max args]
```

where args are character strings representing decimal numbers. The value returned is the numerical maximum of args.

Name: min

Usage

```
[min args]
```

where args are character strings representing decimal numbers. The value returned is the numerical minimum of args.

Name: minus

Usage

```
[minus A B]
```

where A and B are character strings representing decimal numbers. The value returned is A - B.

Name: mod

Usage

```
[mod A B]
```

where A and B are character strings representing decimal numbers. The value returned is A modulo B.

Name: plus

Usage

```
[plus args]
```

where args are character strings representing decimal numbers. The value returned is the sum of args.

Name: quotient

Usage

```
[quotient A B]
```

where A and B are character strings representing decimal numbers. The value returned is A / B.

Name: times

Usage

```
[times A B]
```

where A and B are character strings representing decimal numbers. The value returned is A * B.

Name: trunc

Usage

```
     [trunc A]
```

where A is the character string representation of a decimal number. The value returned is the largest integer whose absolute value is $\leq$ absolute value of A.

Example

The following example illustrates the use of one of the active functions described above.

```
     set_bit_count my_seg [times 672 36]
```

This example sets the bit count of my_seg (assumed to contain 672 words of information) to 24,192 which is the product of 672 and 36.

CHARACTER STRING ACTIVE FUNCTIONS

The following active functions return the results of various operations on one or more character strings.

Name: format_line, fl

This active function returns a formatted character string that is constructed from a control string and other optional arguments. Quotes are placed around the return value so that the command processor treats it as a single argument. Any quotes contained in the return value itself are doubled when the value is placed in quotes, as required by the Multics command language convention for quoted strings.

Usage

```
     [format_line control_string -args-]
```

where:

1. control_string is an ioa_ control string that is used to format the return value of the active function. It can contain control characters within it. If no control characters occur, the string itself is returned as the value of the active function. If control characters exist, they govern the conversion of successive additional arguments that are expanded into the appropriate characters and inserted into the return value. The description of the ioa_ subroutine in the MPM Subroutines lists the control characters that can be used with ioa_. Of these, ^d, ^o, ^f, ^e, ^p, ^w, and ^A cannot be used with the format_line active function, because they control the conversion of argument types that cannot be processed by the command processor, and hence, cannot be input to an active function.

2. args are optional arguments that are substituted in the formatted return value, according to the ioa_ control string.

Name: index

Usage

```
     [index STRING STR]
```

where STRING and STR are character strings. The value returned is the character representation of the number returned by the PL/I builtin function index (STRING, STR).

Name: index_set

Usage

```
     [index_set N]
```

where N is a character representation of a decimal integer. The string returned is the sequence of numbers from 1 through N, separated by spaces; e.g., index_set 4 returns "1 2 3 4".

Name: length

Usage

```
     [length STRING]
```

where STRING is any character string. The value returned is the character representation of the number of characters in STRING.

Name: search

Usage

```
     [search STRING STR]
```

where STRING and STR are character strings. The value returned is the character representation of the number returned by the PL/I builtin function search (STRING, STR).

Name: string

Usage

```
     [string -args-]
```

where args are optional arguments that are to be returned as a single character string. If no arguments are present, then the active function string returns a null character string. If one or more arguments are present, then any quotes in these are doubled when the argument is placed in the quoted return string, as required by the Multics command language convention for quoted strings.

Name: substr

Usage

```
     [substr STRING A -B-]
```

where STRING is any character string. A and B are character strings (B is optional) representing decimal integers. The value returned is the value of the PL/I builtin function substr (STRING, A, B). "Stringrange" is not enabled.

Name: verify

Usage

```
     [verify STRING STR]
```

where STRING and STR are character strings. The value returned is the character string representation of the number returned by the PL/I builtin function verify (STRING, STR).

Examples

The following examples illustrate the use of two of the active functions discussed in this description. One of the examples involves the use of the &if control statement of the exec_com command. (See the description of the exec_com command.)

```
     delete seg([index_set 15])
```

This example is equivalent to the command line:

```
     delete seg(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
```

to delete the 15 segments seg1, seg2, ..., seg15.

```
     &if [query [format_line "Is ^a a good date?" [long_date]]]
     &then &print Beginning execution.
     &else &quit
```

This example might result in the following dialogue. The user's response has been underlined for the sake of clarity.

```
     Is November 22, 1975 a good date?  yes
     Beginning execution.
```

SEGMENT NAME ACTIVE FUNCTIONS

The following active functions return a pathname or entryname or some part thereof. Some of them perform manipulations on an input string to produce the output string. One returns a unique character string that is commonly used as the entryname of a segment.

Name: directories, dirs

This active function returns the names (separated by blanks) of all directories matching starname.

Usage

```
[directories starname]
```

where starname is a pathname for which the entryname portion (optionally) contains stars to be interpreted according to the star convention. (See "Constructing and Interpreting Names" in Section I.)

Name: directory

This active function returns the directory portion of the absolute pathname of the specified segment.

Usage

```
[directory arg]
```

where arg is the segment's pathname.

Name: entry

This active function returns the entryname portion of the absolute pathname of the specified segment.

Usage

```
[entry arg]
```

where arg is the segment's pathname.

Name: files

This active function returns the names (separated by blanks) of all segments, directories, links, and multisegment files matching a given starname.

Usage

```
[files starname]
```

where starname is as described above for the directories active function.

Name: get_pathname, gpn

This active function returns the absolute pathname of the segment that is designated by the reference name or segment number specified. Reference names are discussed in the "Constructing and Interpreting Names" in Section I.

Usage

```
[get_pathname -control_arg- arg]
```

where:

1.	control_arg	if present, can be -name, in that case the following argument (which looks like an octal segment number) is to be interpreted as a segment name.
2.	arg	is a reference name or segment number (octal) known to this process.

Name: home_dir

This active function returns the pathname of the user's home directory (usually of the form >user_dir_dir>Project_id>Person_id).

Usage

```
[home_dir]
```

Name: links

This active function returns the names (separated by blanks) of all links matching a given starname.

Usage

```
[links starname]
```

where starname is as described above for the directories active function.

Name: nondirectories, nondirs

This active function returns the names (separated by blanks) of all segments, links, and multisegment files matching a given starname.

Usage

```
[nondirectories starname]
```

where starname is as described above for the directories active function.

Name: nonlinks, branches

This active function returns the names (separated by blanks) of all segments, directories, and multisegment files matching a given starname.

Usage

```
[nonlinks starname]
```

where starname is as described above for the directories active function.

Name: nonsegments, nonsegs

This active function returns the names (separated by blanks) of all directories, links, and multisegment files matching a given starname.

Usage

```
[nonsegments starname]
```

where starname is as described above for the directories active function.

Name: path

This active function returns the absolute pathname of the specified segment.

Usage

```
[path arg]
```

where arg is the segment´s pathname.

Name: pd

This active function returns the pathname of the process directory (see "The Storage Systems Directory Hierarchy" in Section II of the MPM Reference Guide) of the process in which it is invoked.

Usage

```
[pd]
```

Name: segments, segs

This active function returns the names (separated by blanks) of all segments matching a given starname.

Usage

```
[segments starname]
```

where starname is as described above for the directories active function.

Name: strip

This active function forms the absolute pathname of the specified entry. If the entryname portion has more than one component, then the last component is removed and the resulting pathname is returned. If the last component matches a specified string, then it is removed and the resulting pathname is returned. If the entryname has only one component or if the last component does not match the specified character string, then the absolute pathname is returned.

Usage

```
     [strip arg1 -arg2-]
```

where:

1.	arg1	is a pathname (absolute or relative).
2.	arg2	is an optional character string that, if present and if it matches the last component of the entryname portion of arg1, is removed from that entryname. If arg2 is not given, any last component is removed from the entryname portion of arg1, assuming arg1 has more than one component in its entryname. If arg2 is not matched, then the full pathname of arg1 is returned.

Name: strip_entry, spe

This active function returns the entryname portion of the absolute pathname returned by the strip active function.

Usage

```
     [strip_entry arg1 -arg2-]
```

where:

1.	arg1	is the segment's pathname.
2.	arg2	is as described above under the strip active function.

Name: suffix

This active function returns the last component of the entryname portion of the specified segment. If that entryname has only one component, it returns the null string.

Usage

```
     [suffix arg]
```

where arg is the segment's pathname.

Name: unique

This active function returns a unique character string as generated by unique_chars_ (see the unique_chars_ description in the MPM Subroutines manual).

Usage

```
[unique]
```

Name: wd

This active function returns the pathname of the working directory of the process in which it is invoked.

Usage

```
[wd]
```

Examples

The following examples illustrate the use of three of the active functions discussed in this description.

```
status [get_pathname refname]
```

This example invokes the status command with the pathname of the segment that is known to the process by the reference name, refname.

```
list *.pl1 -p [directory [wd]]
```

This example lists PL/I source segments in the directory immediately superior to the working directory. The same list could also be obtained by the command list *.pl1 -p <.

```
copy ([segs *.pl1]) sub_dir>==
```

This example invokes the copy command to make a copy of all two-component PL/I source segments in the working directory and place it in the sub_dir directory.

DATE AND TIME ACTIVE FUNCTIONS

The following active functions return information about dates and times. All these active functions return their answers with quotes around the returned information.

Name: date

This active function returns a date abbreviation in the form "mm/dd/yy"; e.g., "02/20/74".

Usage

```
[date -args-]
```

where args are optional arguments that determine the date and time for which information is returned. These arguments must be in a form acceptable to the convert_date_to_binary_ subroutine (see the description of this subroutine in the MPM Subroutines manual). If no arguments are specified, information about the current date and time is returned.

Name: date_time

This active function returns a date abbreviation, a time from 0000.0 to 2359.9, a time zone abbreviation, and a day of the week abbreviation in the form: "mm/dd/yy hhmm.m zzz www"; e.g., "08/07/74 0945.7 est Mon.

Usage

```
[date_time -args-]
```

where args are as described above under the date active function.

Name: day

This active function returns the one- or two-digit number of a day of the month, from 1 to 31; e.g., "7".

Usage

```
[day -args-]
```

where args are as described above under the date active function.

Name: day_name

This active function returns the name of a day of the week; e.g., "Monday".

Usage

```
[day_name -args-]
```

where args are as described above under the date active function.

Name: hour

This active function returns the one- or two-digit number of an hour of the day, from 0 to 23; e.g., "9".

Usage

```
[hour -args-]
```

where args are as described above under the date active function.

Name: long_date

This active function returns a month name, a day number, and a year in the form: "month day, year". e.g., "August 7, 1974".

Usage

```
[long_date -args-]
```

where args are as described above under the date active function.

Name: minute

This active function returns the one- or two-digit number of a minute of the hour, from 0 to 59; e.g., "45".

Usage

```
[minute -args-]
```

where args are as described above under the date active function.

Name: month

This active function returns the one- or two-digit number of a month of the year, from 1 to 12; e.g., "8".

Usage

```
[month -args-]
```

where args are as described above under the date active function.

Name: month_name

This active function returns the name of a month of the year; e.g., "August".

Usage

```
[month_name -args-]
```

where args are as described above under the date active function.

Name: time

This active function returns a four-digit time of day in the form "hh:mm" where $00 \leq hh \leq 23$ and $00 \leq mm \leq 59$; e.g., "09:45".

Usage

```
[time -args-]
```

where args are as described above under the date active function.

Name: year

This active function returns the two-digit number of a year of the century; e.g., "74".

Usage

```
[year -args-]
```

where args are as described above under the date active function.

Example

The following example illustrates the use of one of the active functions discussed in this description.

```
     enter_abs_request abs_seg -time [date [month 1 month]/1]
```

This example enters an absentee request for deferred execution to start at the beginning of the next month. The arguments to the month active function indicate that "1 month" should be added to the current date to get the date from which the month is to be calculated. The "/1" when concatenated with the calculated month forms the date string "2/1".

QUESTION ASKING ACTIVE FUNCTIONS

The following active functions return the answer given by a user in response to a specified question.

Name: query

This active function asks the user a specified question and returns the value true if the user's answer was "yes" or the value false if the user's answer was "no". query continues to ask questions until a "yes" or "no" answer is given by the user.

Usage

```
     [query arg]
```

where arg is a question to be asked. If arg contains blanks, it must be enclosed in quotes. It should be worded so as to require a "yes" or "no" answer.

Name: response

This active function asks the user a specified question and returns the answer typed by the user in response.

Usage

```
     [response arg]
```

where arg is a question to be asked the user.

Example

The following example illustrates the use of one of the active functions discussed in this description. It involves the use of the &if control statement of the exec_com command. (See the description of the exec_com command.)

```
&if [query "Do you wish to continue?  "]
&then
&else &quit
```

This example causes the exec_com to continue or quit depending on the user´s answer.

USER PARAMETER ACTIVE FUNCTIONS

The following active functions return user parameters obtained from system data bases.

Name: system

This active function returns various installation-dependent system parameters.

Usage

```
[system arg]
```

where arg may have one of the following values:

company	returns the per-system parameter company name.
date_up	returns the date that the system was brought up, in the form "mm/dd/yy".
department	returns the per-system parameter computer center department name.
down_until_date	returns the date that the system will next be brought up, if specified by operator, in the form "mm/dd/yy".
down_until_time	returns the time that the system will next be brought up, if specified by operator, in the form "hhmm.t".
ds_company	returns the per-system parameter company name, with the characters of the name double spaced.
ds_department	returns the per-system parameter computer center department name, with the characters of the name double spaced.
installation_id	returns the per-system parameter installation identification.
last_down_date	returns the date that service was last interrupted, whether by shutdown or crash.

last_down_reason	returns the reason for the last system service interruption, if known. The reason may be: shutdown if the system shutdown normally crash if the system crashed a number if the system crashed Not all crashes can be assigned a crash number.
last_down_time	returns the time that service was last interrupted.
max_units	returns the current maximum number of load units, in the form "nnn.n".
max_users	returns the current maximum number of users.
n_units	returns the current number of logged-in load units including daemon and absentee, in the form "nnn.n".
n_users	returns the current number of logged-in users including daemon and absentee.
next_down_date	returns the date that system will next be shut down, if specified by operator.
next_down_time	returns the time that system will next be shut down, if specified by operator.
next_shift	returns the next shift number.
reason_down	returns the reason for next shutdown, if specified by operator.
shift	returns the current shift number.
shift_change_date	returns the date on which current shift number will change to next_shift.
shift_change_time	returns the time at which current shift number will change to next_shift.
sysid	returns the version number of the hardcore system tape currently running.
time_up	returns the time that system was brought up, in the form "hhmm.t".

Example

The following example illustrates the use of one of the arguments described in the above discussion.

```
ioa_ [system sysid]
```

This example causes the current system version number of the supervisor to be printed on the user's terminal.

Name: user

This active function returns various user parameters.

Usage

[user arg]

where arg can have one of the following values:

name	returns the user's User_id at login time.
project	returns the user's Project_id.
login_date	returns the date at login time. The date is of the form "mm/dd/yy".
login_time	returns the time of login. The time is of the form "hhmm.t".
anonymous	returns true if the user is an anonymous user; otherwise it returns false.
secondary	returns true if the user is currently subject to preemption; otherwise it returns false.
absentee	returns true if the user is an absentee user; otherwise it returns false.
term_id	returns the user's terminal ID code. It is "none" if the user's terminal does not have the answerback feature.
term_type	returns the user's terminal type. It can have one of the following values: "Absentee" "Network" "1050" "2741" "IBM2741" "TTY33" "TTY37" "TN300" "ARDS" "ASCII" The terminal types "2741" and "IBM2741" differ in that "IBM2741" designates a standard IBM 2741 terminal, and "2741" designates a 2741 terminal that has been modified to prevent keyboard locking after a carriage return. A "2741" terminal can be made to look exactly like an "IBM2741" terminal by placing its INHIBIT AUTO EOT switch in the _off_ position.
cpu_secs	returns the user's CPU usage, in seconds, since login. The usage is of the form "sss.t" with leading zeros suppressed.
log_time	returns the user connect time, in minutes, since login. The time is of the form "mmm.t".

preemption_time	if the user is a primary user, returns the time at which he becomes eligible for group preemption. The time is of the form "hhmm.t".
brief_bit	returns true if the user specified the -brief control argument in his login line; otherwise, returns false.
protected	if the user is currently a primary user and protected from preemption, returns true; otherwise, returns false.
absin	if the user is an absentee user, this returns the absolute pathname of his absentee input segment including the absin suffix; otherwise returns a null string.
absout	if the user is an absentee user, returns the absolute pathname of his absentee output segment; otherwise, returns a null string.
outer_module	returns the initial outer module for the terminal channel.
process_id	returns the user's process identification in octal.
auth	returns the short string for the authorization of the user's process or system_low.
auth_long	returns the long string (in quotes) for the authorization of the user's process or "system_low".
max_auth	returns the short string for the max authorization of the user's process or system_low.
max_auth_long	returns the long string (in quotes) for the max authorization of the user's process or "system_low".

Example

The following example illustrates the use of one of the active functions described above.

```
ioa_ [user login_time]
```

This example causes the time the user logged in to be printed at the user's terminal.

Name: have_mail

The have_mail active function returns the value true if there is mail in the user's current default mailbox or in a specified mailbox; otherwise, false is returned.

Usage

```
     have_mail -path-
```

where path is the pathname of a mailbox. If path is not specified, the have_mail active function looks at the user's ring 1 default mailbox. If this mailbox does not exist, the active function looks at the user's ring 4 default mailbox. If neither mailbox exists, the active function returns the value false.

SECTION III

COMMANDS

COMMAND DESCRIPTIONS

This section contains descriptions of the Multics commands, presented in alphabetical order. Each description contains the name of the command (including the abbreviated form, if any), discusses the purpose of the command, and shows the correct usage. Notes and examples are included when deemed necessary for clarity. The discussion below briefly describes the content of the various divisions of the command descriptions.

Name

The "Name" heading lists the full command name and its abbreviated form. The name is usually followed by a discussion of the purpose and function of the command and the expected results from the invocation.

Usage

This part of the command description first shows a single line that demonstrates the proper format to use when invoking the command and then explains each element in the line. The single line contains the full command name (or its abbreviated form) followed by the valid arguments. Some commands have required arguments; some commands have optional arguments. Most commands have both required and optional arguments; in general, the required arguments precede the optional arguments.

Any argument preceded and followed by a minus sign (-) is an optional argument. Any other argument is a required argument. Anything specifically identified as "control_arg" in the usage line must be preceded by a minus sign in the actual invocation of the command. For example, the usage line:

```
commandname path -control_arg- -xxx-
```

means that the command has one required argument and two optional arguments. Therefore, any of the following command lines are valid:

```
commandname path
commandname path -control_arg
commandname path xxx
commandname path -control_arg xxx
```

If a command accepts more than one of a specific type of argument, an "s" is added to the argument name. For example, the usage line:

```
commandname paths -control_args-
```

means that the user must specify at least one pathname and may specify none, one, or several control arguments.

If a command accepts multiple arguments that must be in a specific order, the usage line is as follows:

```
commandname xxx1 yyy1 ... xxxn yyyn
```

to show that although several xxx and yyy arguments can be given, they must be given in pairs.

Notes

Comments or clarifications that relate to the command as a whole are given under the "Notes" heading. Also, where applicable, the required access modes, default condition (invoking the command without any arguments), and any special case information are included.

Examples

The examples show different valid invocations of the command. The results of each example command line are either shown or explained.

Other Headings

Additional headings are used in some descriptions, particularly the more lengthy ones, to introduce specific subject matter. These additional headings may appear in place of, or in addition to, the notes.

Name: abbrev, ab

The abbrev command provides the user with a mechanism for abbreviating parts of (or whole) command lines in the normal command environment.

When it is invoked, the abbrev command sets up a special command processor that is called for each command line input to the system. The abbrev command processor checks each input line to see if it is an abbrev request line (recognized by a period (.) as the first nonblank character of the line) and, if so, acts on that request. (Requests are described below under "Control Requests.") If the input line is not an abbrev request line and abbreviations are included in the line, the abbreviations are expanded once and the expanded string is passed on to the normal Multics command processor. The abbrev command processor is, therefore, spliced in between the listener and the normal command processor. Note that abbreviations are expanded only once; i.e., abbreviations cannot be nested.

Usage

```
     abbrev
```

Notes

The abbrev command is driven by a user profile segment that contains the user's abbreviations and other information pertinent to execution on his behalf. The profile segment resides (by default) in the user's home directory. If the profile segment is not found, it is created and initialized with the name Person_id.profile where Person_id is the login name of the user. For example, if the user Washington logged in under the States project, the default profile segment would be:

```
     >user_dir_dir>States>Washington>Washington.profile
```

The profile segment being used by abbrev can be changed at any time with the .u control request (see below) to any profile segment in the storage system hierarchy to which the user has appropriate access. The entryname of a profile segment must have the suffix profile. A new profile segment can be created by specifying a nonexistent segment to the .u control request. The segment is then created and initialized as a profile segment, assuming the user has the necessary access. The user must be careful not to delete or terminate the segment that is currently being used as his profile unless he first quits out of abbrev by issuing the .q control request (see below).

The user can suppress expansion of a particular string in a command line by enclosing it within quotes ("). To suppress expansion of an entire command line, see the .<space> control request.

A user might want to include the invocation of the abbrev command in a start_up.ec segment so that he is automatically able to abbreviate whenever he is logged in. See Section I of the MPM Reference Guide for a definition of start_up.ec.

Control Requests

An abbrev request line has a period (.) as the first nonblank character of the line. An abbrev request line, with the exception of the .s and .<space> requests, is neither checked for embedded abbreviations nor (even in part) passed on to the command processor. If the command line is not an abbrev request line, abbrev expands it and passes it on to the current command processor.

The character immediately after the period of an abbrev request line is the name of the request. The following requests are recognized:

.a <abbr> <rest of line>	add the abbreviation <abbr> to the current profile segment. It is an abbreviation for <rest of line>. Note that the <rest of line> string can contain any characters. If the abbreviation already exists, the user is asked if he wishes to redefine it. The user must respond with "yes" or "no". The abbreviation must be no longer than eight characters and must not contain break characters. *
.ab <abbr> <rest of line>	add an abbreviation that is expanded only if found at the beginning of a line or directly following a semicolon (;) in the expanded line. In other words, this is an abbreviation for a command name.
.af <abbr> <rest of line>	add an abbreviation to the profile segment and force it to overwrite any previous abbreviation with the same name. The user is not asked if he wants the abbreviation redefined.
.abf <abbr> <rest of line>	add an abbreviation that is expanded only at the beginning of a line and force it to replace any previous abbreviation with the same name. The user is not asked if he wants the abbreviation redefined.
.d <abbr1> ... <abbrn>	delete the specified abbreviations from the current profile segment.
.f	enter a mode (the default mode) that forgets each command line after executing it. See the .r and .s requests.
.l <abbr1> ... <abbrn>	list the specified abbreviations and the strings they stand for. If no abbreviations are specified, all abbreviations in the current profile segment are listed.

.la <letter1> ... <lettern>	list all abbreviations starting with the specified letters. <letteri> is expected to be a single character. If no letters are specified, all abbreviations in the current profile segment are listed.
.q	quit using the abbrev command processor. This request resets the command processor to the one in use before invoking abbrev and, hence, prevents any subsequent action on the part of abbrev until it is explicitly invoked again.
.r	enter a mode that remembers the last line expanded by abbrev. See the .f and .s requests.
.s <rest of line>	show the user how <rest of line> would be expanded but do not execute it. The .s request with no arguments shows the user the last line expanded by abbrev and is valid only if abbrev is remembering lines. See the .f and .r requests.
.u <profile>	specify to abbrev the pathname of a profile segment to use. <profile segment> becomes the current working profile segment. The user needs "r" access to use the profile segment and "w" access to add and delete abbreviations.
.p	print the name of the profile segment being used.
.<space> <rest of line>	pass <rest of line> on to the current command processor without expanding it. Using this request, the user can issue a command line that contains abbreviations that are not to be expanded.

Break Characters

When abbrev expands a command line, it treats certain characters as special or break characters. Any character string that is less than or equal to eight characters long and is bounded by break characters is a candidate for expansion. The string is looked up in the current profile segment and, if it is found, the expanded form is placed in (a copy of) the command line to be passed on to the normal command processor.

The characters that abbrev treats as break characters are:

tab	
newline	
space	
quote	"
dollar sign	$
apostrophe	'
grave accent	`
period	.
semicolon	;
vertical bar	|
parentheses	( )
less than	<
greater than	>
brackets	[ ]
braces	{ }

Example

Suppose that a user notices that he is typing the segment name suffixes fortran and incl.fortran often as he edits his FORTRAN source segments (e.g., alpha.fortran and alpha.incl.fortran). He might wish to abbreviate them to "ft" and "ift" respectively. He then types the lines specified to accomplish the following objectives:

1. Invoke the abbrev command:

   ```
   abbrev
   ```

2. Define the two abbreviations:

   ```
   .a ft fortran
   .a ift incl.fortran
   ```

3. Now that "ft" and "ift" are defined, invoke the text editor, edm, to create or edit his source segments:

   ```
   edm sample.ft
   edm insert.ift
   ```

4. Print the include file:

   ```
   print insert.ift
   ```

If the user chooses to write out one of the segments from edm by a different name, he must type the expanded name since the edm command (and not the abbrev command processor) is intercepting all terminal input. For example, after editing sample.fortran he might wish to write out the changed version as example.fortran. He would type to edm:

```
w example.fortran
```

If he instead types:

```
w example.ft
```

he creates a segment by exactly that name (example.ft). In this case, if the user tries to print the segment while at command level (by typing "print example.ft"), the abbrev processor expands the command line and the print command looks for a segment named example.fortran; since no such segment exists, the print command responds with an error message.

Name: accept_messages, am

The accept_messages command initializes or reinitializes the user's process for accepting messages sent by the send_message command. If the mailbox:

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

does not exist, the accept_messages command creates it. A channel is created to receive wakeups from send_message so that when a message is received, it is printed on the user's terminal immediately. Messages sent when the user is not logged in or when he is deferring messages (see the defer_messages command) are saved in the mailbox and can be read later by invoking the print_messages command. The mail command stores mail in the same mailbox. See "Extended Access" in the mail command description for an explanation of mailbox access.

Usage

```
accept_messages -control_args-
```

where control_args can be chosen from the following list:

-brief, -bf	prevents accept_messages from informing the user that it is creating a mailbox. This control argument also causes messages to print in short format (see the -short control argument below).
-long, -lg	precedes every message printed by the sender's Person_id and Project_id. This is the default mode.
-print, -pr	prints all messages that were received since the last time the user was accepting messages.
-short, -sh	precedes consecutive messages from the same sender by "=:" instead of the Person_id and Project_id.

Notes

The user should not give conflicting control arguments in the same invocation of the command (i.e., -long and -short or -long and -brief).

* Channel and process identifiers are stored in the user's mailbox. Since only one process can receive a wakeup when a message is placed in the mailbox, it is not advisable for several users to share the same mailbox.

Name: add_name, an

The add_name command adds an alternate name to the existing name(s) of a segment, multisegment file, directory, or link. See also the descriptions of the delete_name and rename commands.

Usage

```
add_name path names
```

where:

1. path is the pathname of the segment, multisegment file, directory, or link to which an additional name is to be added.

2. names are additional names to be added to the segment, multisegment file, directory, or link.

Notes

The user must have modify permission on the directory that contains the entry receiving the additional name.

The equal and star conventions can be used. See "Constructing and Interpreting Names" in Section I.

Two entries in a directory cannot have the same entryname; therefore, special action is taken by this command if the added name already exists in the directory that contains the path argument. If the added name is an alternate name of another entry, the name is removed from this entry, added to the entry specified by path, and the user is informed of this action. If the added name is the only name of another entry, the user is asked if he wishes to delete this entry. If he answers "yes", the entry is deleted and the name is added to the entry specified by path; if he answers "no", no action is taken.

Example

```
add_name >my_dir>example.pl1 sample.pl1
```

adds the name sample.pl1 to the segment example.pl1 in the directory >my_dir.

```
add_name >udd>**.private ==.public
```

adds to every entry having a name with private as the last component a similar name with public, rather than private, as the last component.

Name: add_search_rules, asr

The add_search_rules command allows the user to change his search rules dynamically. The search rules to be added may be inserted at any point in the current search rules.

Usage

```
    add_search_rules path11 -control_arg path21- ... path1n -control_arg path2n-
```

where:

1. path1i — is usually a pathname (relative or absolute) representing a directory to be added to the current search rules. It may be a keyword (see "Notes" below).

2. control_arg — may be one of the following:

   -before — place path1i before the current search rule identified by path2i.

   -after — place path1i after the current search rule identified by path2i.

3. path2i — is usually a pathname (relative or absolute) representing a current search rule. It may be a keyword (see "Notes" below).

Notes

If the add_search_rules command is invoked without the control_arg and path2i arguments, the pathname or keyword specified by path1i is appended to the end of the user's current search rules.

Any representation of a current search rule is acceptable for the path2i argument. It is not necessary to use the same name that appears when the print_search_rules command is invoked.

In addition to pathnames, both the path1 and path2 arguments accept the keywords initiated_segments, referencing_dir, and working_dir. The path2 argument also accepts the keywords home_dir, process_dir, and system_libraries.

Name: adjust_bit_count, abc

The adjust_bit_count command is used to set the bit count of a segment that for some reason does not have its bit count set properly (e.g., the program that was writing the segment got a fault before the bit count was set, or the process terminated without the bit count being set). The adjust_bit_count command looks for the last nonzero 36-bit word or (if specified) the last nonzero character in the segment and sets the bit count to indicate that the word or character is the last meaningful data in the segment.

Usage

```
     adjust_bit_count paths -control_args-
```

where:

1. paths	are the pathnames of segments whose bit counts are to be adjusted. The star convention is allowed.
2. control_args	are as follows and apply to all path arguments:
-character, -ch	set the bit count to the last nonzero character.
-long, -lg	print a message when the bit count of a segment is changed, giving the old and new values.

Notes

If the bit count of a segment can be computed but cannot be set (e.g., the user has improper access to the segment), the computed value is printed so that the user can use the set_bit_count command after resetting access or performing other necessary corrective measures. See the description of the set_bit_count command.

The user must have write access on the segment whose bit count is being adjusted. He need not have modify permission on the directory containing that segment.

The adjust_bit_count command should not be used on segments in structured files.

Name: answer

The answer command provides a preset answer to a question asked by another command. It does this by establishing an on unit for the condition command_question, and then executing the designated command. If the designated command calls the command_query_ subroutine (described in the MPM Subroutines) to ask a question, the on unit is invoked to supply the answer. The on unit is reverted when the answer command returns to command level. See "List of System Conditions and Default Handlers" in Section VI of the MPM Reference Guide for a discussion of the command_question condition.

Usage

```
answer ans -control_args- commandline
```

where:

1.	ans	is the desired answer to any question. If the answer is more than one word, it must be enclosed in quotation marks.
2.	control_args	can be chosen from the following list of control arguments:
	-brief, -bf	suppress printing (on the user's terminal) of both the question and the answer.
	-times n	give the prespecified answer n times only (where n is an integer); then act as if the answer command had not been called. The default action is to answer the question as often as it is asked.
3.	commandline	is any Multics command line. It can contain any number of separate arguments (i.e., have spaces within it) and need not be enclosed in quotation marks.

Note

If a question is asked that requires a yes or no answer, and the preset answer is neither "yes" nor "no", the on unit is not invoked.

Examples

To delete the test_dir directory without being interrogated by the delete_dir command, type:

```
answer yes -bf delete_dir test_dir
```

To automatically see the first three blocks of an info segment named fred.info and then be interrogated about seeing any more blocks, type:

```
answer yes -times 2 help fred
```

To see only the first three blocks of that info segment, type:

```
answer no answer yes -times 2 help fred
```

In the above example, the answer command is invoked twice. The first invocation is to answer "no" to the command line

```
answer yes -times 2 help fred
```

The second invocation is to answer "yes" twice to the command line

```
help fred
```

The help command prints the first block of fred.info and gets the answer "yes" from the second invocation of the answer command. It repeats this process and again obtains a "yes" answer. After help prints the third block of fred.info, however, the second invocation of the answer command has had its count run out and behaves as if it had not been called. Hence, the first invocation of the answer command supplies the answer "no" and execution ends.

Name: apl

The apl command invokes the Multics APL interpreter, which is completely described in the Multics APL User's Guide, Order No. AK95. The Multics APL language is nearly identical to the APL/360 language; differences are noted in the above mentioned document.

APL can be characterized as a line-at-a-time desk calculator with many sophisticated operators and a limited stored-program capability. Little or no prior acquaintance with digital computers is needed to make use of it. After invoking APL, one types an expression to be evaluated. The APL interpreter performs the calculations, prints the result, and awaits a new input line. The result of an expression evaluation can also be assigned to a variable and remembered from line to line. In addition, there is a capability for storing input lines and giving them a name, so that a later mention of the name causes the lines to be brought forth and interpreted as if they had been entered from the terminal at that time. Finally, there is also the ability to save the entire state of an APL session, complete with all variable values and stored programs, so that it can be taken up later.

Usage

```
     apl
```

Note

There are no arguments. The interpreter responds by typing six spaces and awaiting input. For further information, consult the Multics APL User's Guide.

Name: archive, ac

An archive segment is formed by combining an arbitrary number of separate segments into one single segment. The constituent segments that compose the archive are called components of the archive segment. The process of placing segments in an archive is particularly useful as a means of eliminating wasted space that occurs when individual segments do not occupy complete pages of storage. Archiving is also convenient as a means of packaging sets of related segments; it is used this way when interfacing with the Multics binder (see the bind command description in this document).

The archive command performs a variety of operations that the Multics user can employ to create new archive segments and to maintain existing ones. The operations are:

1. Table of contents operation; print a table of contents of an archive segment.
2. Append operation; append components to, or create a new archive segment.
3. Replace operation; replace components in, add to, or create a new archive segment.
4. Update operation; update an archive segment by replacing components with more recently modified ones.
5. Delete operation; delete specified components of an archive segment.
6. Extract operation; extract components from an archive segment and place them in segments in the storage system.

Each of these general operations can be specialized to perform several functions and, in many cases, can be combined with the copy and deletion features described below. Such combinations give the user extensive control over the maintenance of his archive segments.

The table of contents operation and the extract operation use the existing contents of an archive segment; the other operations change the contents of an archive segment. A new archive segment can be created with either the append or replace operation. In each of the operations that add to or replace components of the archive, the original segment is copied and the copy written into the archive, leaving the original segment untouched unless deletion is specified as part of the operation. Use of the various operations is illustrated in the "Examples" at the end of this description.

The table of contents operation is used to list the contents of an archive segment. The table of contents operation can be made to print information in long or brief form with or without column headings.

The append operation is used to add components to the archive segment and to create new archive segments. When adding to an existing archive, if a component of the same name as the segment requested for appending is already present in the archive segment, a diagnostic message is printed on the user's terminal and that segment is not appended. When several segments are requested for appending, only those segments whose names do not match existing components of the archive segment are added to the archive.

The replace operation is similar to the append operation in that it can be used to add components to the archive segment, and therefore, is also used to create new archive segments. However, unlike the append operation, if a component of the same name as the segment requested for replacing is already present in the archive segment, the requested replace operation is performed by overwriting the existing component. When several segments are requested for replacing, those segments whose names do not match existing components of the archive segment are added to the archive, as in the append operation.

The update operation replaces existing components only if the date-time modified of a segment requested for updating is later than that of the corresponding component currently in the archive segment. When a segment whose name does not match an existing component of the archive segment is requested for updating, it is not added to the archive segment.

The delete operation is used only to delete components from archive segments. It cannot delete segments from the storage system and is not analogous to the deletion feature described below.

The extract operation is used to create copies of components from the archive segment elsewhere in the storage system. The extract operation performs a function opposite to the append operation.

In addition to the operations described above, there are two features, copy and deletion, that can be combined with certain operations in order to specialize their performance. Since copy and deletion are features and not operations, they cannot stand alone, but must always be combined with those operations that permit their use. The deletion feature is distinct from the delete operation, as noted below.

The copy feature can be combined with the append, replace, update, and delete operations. Since an archive segment can be located anywhere in the storage system, it occasionally is convenient to move the segment during the maintenance process, or to modify the original segment while temporarily retaining an unmodified version. When the copy feature is used, the original archive segment is copied from its location in the storage system, updated, and placed in the user's working directory.

The deletion feature can be combined with the append, replace, and update operations to delete segments from the storage system after they have been added to or replaced in an archive segment. The deletion can be forced to bypass the system's safety function, i.e., the user is not asked whether he wants to delete a protected segment before the deletion is performed. (This is analogous to the operation of the delete_force command.) Nothing is deleted until after the archive segment has been successfully updated.

Deletion of segments (deletion feature) is not to be confused with deletion of components from archive segments. The delete operation is a stand-alone function of the archive command that operates only on components of archive segments, deleting them from the archive. The deletion feature, on the other hand, performs deletions only when combined with an operation of the archive command, and then deletes only segments from the storage system after copies of those segments have been added to, or used to update archive segments.

The archive command can operate in two ways: if no components are named on the command line, the requested operation is performed on all existing components of the archive segment; if components are named on the command line, the operation is performed only on the named components.

The star convention can be used in the archive segment pathname during extract and table of contents operations; it cannot be used during append, replace, update and delete operations. Component names cannot be specified using the star convention. See "Constructing and Interpreting Names" in Section I of this document for a discussion of the star convention.

No commands other than archive, archive_sort, and reorder_archive should be used to manipulate the contents of an archive segment; using a text editor or other command might result in unspecified behavior during subsequent manipulations of that archive segment. See the descriptions of the archive_sort command and the reorder_archive command in the MPM Subsystem Writers´ Guide.

Usage

```
     archive key archive_path paths
```

where:

1. archive_path	is the pathname of the archive segment to be created or used. The suffix archive is added if the user does not supply it. If the archive segment does not exist, it is created for replace and append operations as described above. The star convention can be used with extraction and table of contents operations.
2. paths	are the components to be operated on by table of contents and delete operations. For append, replace, update and extract operations, each path specifies the pathname of a segment corresponding to a component whose name is the entryname portion of the pathname. The star and equal conventions cannot be used.
3. key	is one of the key functions listed below.

Table of Contents Operation:

t	print the entire table of contents if no components are named by the path arguments; otherwise print information about the named components only. A title and column headings are printed at the top.
tl	print the table of contents in long form; operates like t, printing more information for each component.
tb	print the table of contents, briefly; operates like t, except that the title and column headings are suppressed.
tlb	print the table of contents in long form, briefly; operates like tl, except that the title and column headings are suppressed.

Append Operation:

a	append named components to the archive segment. (The segments corresponding to the appended components are not affected.) If a named component is already in the archive, a diagnostic is issued and the component is not replaced. At least one component must be named by the path arguments. If the archive segment does not exist, it is created.
ad	append and delete; operates like a and then deletes all segments that have been appended to the archive. If the safety switch is on for any of the corresponding segments, the user is asked whether he wishes to delete the segment.
adf	append and deleteforce; operates like a and then forces deletion of all segments that have been appended to the archive.
ca	copy and append; operates like a, appending components to a copy of the new archive segment created in the user's working directory.
cad	copy, append, and delete; operates like ad, appending components to a copy of the archive segment and deleting the appended segments.
cadf	copy, append, and deleteforce; operates like adf, appending components to a copy of the archive segment and forcibly deleting the segments requested for appending.

Replace Operation:

r	replace components in, or add components to the archive segment. When no components are named in the command line, all components of the archive for which segments by the same name are found in the user's working directory are replaced. When a component is named, it is either replaced or added. If the archive segment does not exist, it is created.
rd	replace and delete; operates like r, replacing or adding components, then deletes all segments that have been replaced or added.
rdf	replace and deleteforce; operates like r and forces deletion of all replaced or added segments.
cr	copy and replace; operates like r, placing an updated copy of the archive segment in the user's working directory instead of changing the original archive segment.
crd	copy, replace and delete; operates like rd, placing an updated copy of the archive segment in the user's working directory.
crdf	copy, replace, and deleteforce; operates like rdf, placing an updated copy of the archive segment in the user's working directory.

Update Operation:

u	update; operates like r except it replaces only those components for which the corresponding segment has a date-time modified later than that associated with the component in the archive. If the component is not found in the archive segment, it is not added.
ud	update and delete; operates like u and deletes all updated segments after the archive has been updated.
udf	update and deleteforce; operates like u and forces deletion of all updated segments.
cu	copy and update; operates like u, placing an updated copy of the archive segment in the user's working directory.
cud	copy, update, and delete; operates like ud, placing an updated copy of the archive segment in the user's working directory.

cudf	copy, update, and deleteforce; operates like udf, placing an updated copy of the archive segment in the user's working directory.

Delete Operation:

d	delete from the archive those components named by the path arguments.
cd	copy and delete; operates like d, placing an updated copy of the archive segment in the working directory.

Extract Operations:

x	extract from the archive those components named by the path arguments, placing them in segments in the storage system. The directory where a segment is placed is the directory portion of the path argument. The access mode stored with the archive component is placed on the segment for the user performing extraction. If a segment already exists, it observes the duplicated name convention in a manner similar to the copy command. If no component names are given, all components are extracted and placed in segments in the working directory. The archive segment is not modified.
xf	extract and deleteforce; operates like x, forcing deletion of any duplicate names or segments found where the new segment is to be created.

Notes

Each component of an archive segment retains certain attributes of the segment from which it was copied. These consist of a single name, the effective mode of the user who placed the component in the archive, the date-time the segment was last modified, and the bit count of the segment. In addition, the date-time that the component was placed in the archive segment is maintained. When a component is extracted from an archive segment and placed in the storage system, the new segment is given the mode associated with the archive component for the user performing the extraction, the name of the component, and the bit count of the component.

The date-time-modified value of a component has a precision of one tenth of a minute. This means that a copy of a component less than a tenth of a minute more recent than the archived copy is not updated. Users who update archives in exec_com segments should be aware of this limitation.

The archive command maintains the order of components within an archive segment. When new components are added, they are placed at the end. The archive_sort or reorder_archive commands (described in the MPM Subsystem Writers' Guide) can be used to change the order of components in an archive segment.

The archive command cannot be used recursively. Checks are made to prevent the misuse of the command in this fashion. The user is asked a question if the command detects an attempt to use the archive command prior to its completing the last operation.

Because the replacement and deletion operations are not indivisible, it is possible for them to be stopped before completion and after the original segment has been truncated. This can happen, for example, if one gets a record quota overflow. When this situation occurs, a message is printed informing the user of what has happened. In this case, the only good copy of the updated archive segment is contained in the process directory.

Archive segments can be placed as components inside other archive segments, preserving their identity as archives, and can later be extracted intact.

When the archive command detects an internal inconsistency, it prints a message and stops the requested operation. For table of contents and extraction operations, it will have already completed requests for those components appearing before the place where the format error is detected.

For segment deletions after replacement requests, if the specified component name was a link to a segment, the segment linked to is deleted. The link is not unlinked.

The archive command observes segment protection by interrogating the user when he requests (unforced) deletion of a segment to which he does not have write permission. If he could have given himself write permission (i.e., he has modify permission on the superior directory) and he replies that he wants the segment deleted, the segment is deleted.

Examples

Assume that the user has several short segments and wishes to consolidate them to save space. His directory might initially look like the following:

```
list

Segments= 5, Records= 5.

rw     1  epsilon
rw     1  delta
rw     1  gamma
rw     1  beta
rw     1  alpha
```

He creates an archive segment (using the append key) containing four of the five segments.

```
archive a greek alpha beta gamma delta
archive: Creating greek.archive
```

His directory then has one more segment (the archive segment), and a table of contents of the new archive segment shows the four components.

```
list

Segments= 6, Records= 6.

rw     1  greek.archive
rw     1  epsilon
rw     1  delta
rw     1  gamma
rw     1  beta
rw     1  alpha


archive tl greek


                    greek.archive

name                          updated     mode     modified  length

alpha                    09/12/74 1435.0 rw   09/12/74 1434.2  441
beta                     09/12/74 1435.0 rw   09/12/74 1434.2  257
gamma                    09/12/74 1435.0 rw   09/12/74 1434.2  694
delta                    09/12/74 1435.0 rw   09/12/74 1434.2  109
```

After changing the segment delta, he replaces it in the archive segment and adds (using the replace key) the segment epsilon to the archive segment. He also deletes the component gamma.

```
archive r greek delta epsilon
archive: epsilon appended to greek.archive


archive d greek gamma
```

A table of contents now shows a different set of components.

```
     archive t greek

                              greek.archive

      updated                   name

     09/12/74  1435.0         alpha
     09/12/74  1435.0         beta
     09/12/74  1437.5         delta
     09/12/74  1437.5         epsilon
```

He later replaces the component alpha with an updated copy and deletes the storage system segment alpha, causing the updated column of a table of contents to change and a list of his directory to show one less segment.

```
     archive rd greek alpha


     archive t greek

                              greek.archive

      updated                   name

     09/12/74  1641.5         alpha
     09/12/74  1435.0         beta
     09/12/74  1437.5         delta
     09/12/74  1437.5         epsilon


     list

     Segments= 5, Records= 5.

     rw      1  greek.archive
     rw      1  epsilon
     rw      1  delta
     rw      1  gamma
     rw      1  beta
```

In another directory (which contains a different version of the segment alpha), he copies and updates the archive segment, causing the component alpha to be replaced and the updated archive segment to be placed in the working directory.

```
     archive cu orig_dir>greek
     archive: Copying greek.archive
     archive: alpha updated in greek.archive
```

```
list

Segments= 2, Records= 2.

rw      1  greek.archive
rw      1  alpha


archive t greek


                      greek.archive

 updated                name

09/12/74 1641.5       alpha
09/12/74 1435.0       beta
09/12/74 1437.5       delta
09/12/74 1437.5       epsilon


ac t >orig_dir>greek


                      greek.archive

 updated     name

09/12/74 1439.1       alpha
09/12/74 1435.0       beta
09/12/74 1437.5       delta
09/12/74 1437.5       epsilon
```

Notice that the entry in the updated column for the component alpha differs in the two tables of contents. Lastly, the user extracts two components into his new working directory, presumably to work on them.

```
archive x greek beta delta


list

Segments= 4, Records= 4.

rw      1  delta
rw      1  beta
rw      1  greek.archive
rw      1  alpha
```

Name: assign_resource, ar

The assign_resource command calls the resource control package (RCP) to assign a resource to the user's process. Currently, only device resources can be assigned. An assigned device still must be attached by a call to some I/O module. If a device is successfully assigned, the name of the device is printed. (If the user requests a specific device that is successfully assigned, the name of the device is not printed unless the user asks for it. See the -device and -long control arguments below.)

Usage

```
     assign_resource resource_type -control_args-
```

where:

1. resource_type specifies the type of resource to be assigned. Currently, only device types may be specified. The -device control argument is used to name a specific device to assign. Other control arguments are used to specify characteristics of the device to be assigned. The following device type keywords are supported:
   - tape
   - disk
   - console
   - printer
   - punch
   - reader
   - special

2. control_args may be chosen from the following:

   -device XX, -dv XX
   specifies the name of the device to be assigned. If this control argument is specified, other control arguments that specify device characteristics are ignored. (See "Examples" below.) If the -long control argument (see below) is used in conjunction with this control argument, a message containing the name of the assigned device is printed on the user's terminal; otherwise, no message is printed.

   -model n
   specifies the device model number characteristic. Only a device that has this model number is assigned.

   -track n, -tk n
   specifies the track characteristic of a tape drive. The value may be either 9 or 7. A track value of 9 is used by default when assigning a tape type device, if this control argument is not specified and if the -volume control argument is not specified.

-density n, -den n	specifies the density capability characteristic of a tape drive. There may be more than one instance of this argument. A tape drive is assigned that is capable of being set to all of the specified densities. The acceptable values for this argument are: 200 556 800 1600
-train n, -tn n	specifies the print train characteristic of a printer.
-line_length n, -ll n	specifies the line length of a printer. Its value must be one that is found in the "line length" field of a printer PRPH configuration card. If this field is not specified on a printer PRPH configuration card, this device characteristic is ignored for this printer.
-volume XX, -vol XX	specifies the name of a volume. If possible the device assigned is one on which this volume has already been placed.
-number n, -nb n	specifies the number of resources to assign. All of the resources assigned have the device characteristics specified by any other arguments passed to this command. If this control argument is not specified, one resource is assigned.
-comment XX, -com XX	is a comment string that is displayed to the operator when the resource is assigned. If more than one string is required, the entire string must be in quotes. Only printable ASCII characters are allowed. Any unprintable characters (also tabs or new lines) found in this string are converted to blanks.
-long, -lg	specifies that all of the device characteristics of the assigned device should be printed. If this argument is not supplied, only the name of the assigned device is printed.
-system, -sys	specifies that the user wants to be treated as a system process during this assignment. If this argument is not specified or if the user does not have the appropriate access, then the RCP assumes that this assignment is for a nonsystem process.
-wait -n-, -wt -n-	specifies that the user wants to wait if the assignment cannot be made at this time because the resources are assigned to some other process. The value n specifies the maximum number of minutes to wait. If n minutes elapse and a resource is not yet assigned, an error message is printed. If n is not specified, it is assumed that the user wants to wait indefinitely.

Examples

In the example below, the user issues the assign_resource command with the "tape" keyword and the -model control argument. The system responds with the name of the assigned device.

```
assign_resource tape -model 500
```

```
Device tape_04 assigned
```

In the next example, the user issues the assign_resource command with the "tape" keyword and the -device and -long control arguments. The system responds with the name of the assigned device and the model number, track, and density characteristics.

```
assign_resource tape -device tape_05 -long
```

```
Device tape_05 assigned
Model     =  500
Tracks    =  9
Densities =  200 556 800 1600
```

Name: attach_lv, alv

The attach_lv command calls the resource control package (RCP) to attach a logical volume. Attaching a logical volume involves informing the storage system that a particular volume is attached for a particular process. A logical volume (unless it is a public logical volume) must be attached for each process that wishes to use it. To be attached, the logical volume must first be physically mounted. This mounting involves mounting all of the physical volumes that comprise the logical volume. Mounting must be completed by the system operators before the logical volume may be attached by any process.

A user must have rw access to the logical volume he wishes to attach. This access is defined by the access control segment (ACS) associated with the logical volume.

Usage

```
     attach_lv volume_name
```

where volume_name specifies the name of the volume to be attached.

Note

The status command issued with the -device control argument prints the name of the logical volume on which a segment resides.

Name: basic

The basic command invokes the BASIC compiler to translate a segment containing BASIC source code. If the -compile control argument is not specified, the compiled code is then executed and not saved for future execution. If the -compile control argument is specified, a standard object segment is created for subsequent execution.

For a description of the BASIC language on the Multics system, consult the Multics BASIC manual, Order No. AM82.

Usage

```
     basic path -control_arg-
```

where:

1. path — is the pathname of the segment to be translated. The suffix basic need not appear as part of the pathname. It must, however, be the last component of the name of the source segment.

2. control_arg — can be one of the following:

   -compile — requests BASIC to compile the program and generate a bindable Multics standard object segment. The resulting object segment is placed in the user´s working directory. For a description of the features common to all Multics programming languages, see "Programming Languages" in Section II of the MPM Reference Guide, and for a description of object segments see "Standard Segment Formats" in Section V of the MPM Reference Guide.

   -time N — where N is the character-string representation of a decimal number that requests a limit of N seconds on the execution of the BASIC program. If the limit is exceeded, the user is asked if he wishes to continue.

Name: basic_system, bs

The basic_system command is the standard BASIC source editor and run dispatcher. A BASIC source segment pathname must be specified. If the segment exists, it is picked up; otherwise a new segment is expected to be input.

This is an interactive BASIC, as opposed to the basic command, which only compiles a program.

Usage

```
basic_system -path-
```

where path is the pathname of a BASIC source segment. If path does not have a suffix of basic, one is assumed; however, the basic suffix must be the last component of the name of the source segment. If path does not exist, or if no path argument is specified, input is assumed.

Requests

The basic_system editing requests are:

line_number source_line

adds or replaces a BASIC source line (source_line) in proper sequence. The line number (line_number) must be either 0 or a positive integer less than 10,000.

line_number

deletes that source line if such a line number exists.

quit

exits from basic_system and returns to command level. The current internal segment is lost unless a save request is issued before this.

list

prints the entire current internal segment.

run

calls the BASIC compiler to run the current internal source segment.

exec command_line

passes the command_line argument to the Multics command processor. This argument can be any valid Multics command line.

time n

establishes a time limit of n CPU seconds on the execution of the program. Since this does not take effect until the run request is given, a time request can be overridden by a subsequent time request. The time limit feature can be turned off by giving a subsequent time request with n = 0.

rseq -first- -increment-

resequences the line numbers so that they differ by a fixed increment. If increment is not given, it is 10 by default. If first is not given, it is 100 by default. The first and increment arguments must be in the order shown above; therefore increment may not be given without first. Both must be positive integers.

delete all
or
delete first -last-

deletes the specified lines. If all is specified, every line is deleted; otherwise, the lines between first and last inclusive are deleted. If last is not given, only the line number identified by first is deleted.

get -path-

clears the internal buffers so that the user can work on a different program. If path is given, it is the pathname of the new source segment. If path is not given here, the internal buffers are cleared and input is assumed. (Issuing a get request without the path argument is equivalent to invoking the basic_system command without the path argument.)

save -path-

stores the current internal source segment in the segment whose pathname is specified by path. If no path is given, the current internal source segment is stored as specified by the path argument given in an earlier save request, a get request, or the invocation of the basic_system command, whichever is most recent. If none of these conditions can be met (e.g., the most recent request was a get request issued without a path argument), the basic_system command returns an error message and the user will have to specify a path argument in order to save the current internal source segment. To make sure an internal source segment is stored with the pathname intended, it is a good practice to specify the path argument in the save request.

Notes

If the user issues a quit signal out of a BASIC compilation or execution run, he can immediately issue the program_interrupt (pi) command in order to get back to basic_system. Otherwise, any unsaved BASIC program within basic_system is lost.

Refer to the Multics BASIC manual, Order No. AM82, for detailed information on the BASIC language syntax.

Name: bind, bd

This command produces a single bound object segment from one or more unbound object segments that are called the <u>components</u> of the bound segment. (Compilers and the assembler produce unbound object segments.) A reference in one component to an external symbol defined in another component may be resolved during the binding. This prelinking avoids the cost of dynamic linking, and it also ensures that the reference is linked to the component regardless of the state of a process at the moment that dynamic linking would take place. References to a symbol are prelinked unless the contrary is specified by an instruction in the <u>bindfile</u>. The bindfile is a segment containing instructions that control various aspects of the binding operation (see "The Bindfile" below).

Usage:

```
bind archive_paths -update- -update_paths- -control_arg-
```

where:

1. archive_paths are the pathnames of archive segments containing one or more component object segments to be bound. Up to 16 input archive segments can be specified. They are logically concatenated in a left-to-right order to produce a single sequence of input component object segments. The specified pathname of the archive segment need not contain an explicit archive suffix.

2. -update, -ud is an optional functional argument to the binder indicating that the following list of archive segments (update_paths) specifies update rather than input object segments. If this optional argument is used, it must be preceded by a hyphen.

3. update_paths are pathnames of optional archive segments containing update object segments. Up to a combined total of 16 input and update segments can be specified. The contained update object segments are matched against the input object segments by object segment name. Matching update object segments replace the corresponding input object segments; unmatched ones are appended to the sequence of input object segments. If several update object segments have the same name, only the last one encountered is bound into the bound segment. The specified pathname of the archive segment need not contain an explicit archive suffix.

4. control_arg can be one of the following two optional control arguments:

   -list, -ls produces a listing segment whose name is derived from the name of the bound object segment plus a suffix of list. The listing segment is generated for the purpose of dprinting; it contains the bound segment's bind control segment (see "The Bindfile" below), its bind map, and that information from the bound object segment that would be printed by the print_link_info command. (See the description of the dprint command in this document and the print_link_info command in the MPM Subsystem Writers' Guide.)

-map produces a listing segment (with the suffixes list and map) that contains only the bind map information.

In the absence of these control arguments, no listing segment is generated.

Output

The binder produces as its output two segments: an executable bound procedure object segment and an optional, printable ASCII listing segment. The name of the bound object segment is, by default, derived from the entryname of the first input archive segment encountered by stripping the archive suffix from it. The name of the listing segment is derived from the name of the bound segment by adding the list suffix to it. Use of the Objectname master statement in the bindfile (see "Master Key Words" below) allows the name of the bound object segment to be stated explicitly. In addition, use of the Addname master statement in the binding instructions causes additional segment names to be added to the bound segment. The primary name of the bound object segment must not be the same as the name of any component.

The Bindfile

The bindfile is a segment containing symbolic instructions that control the operation of the binder. Its entryname must contain the suffix bind and it must be archived into any one of the input archive segments (at any location within that archive segment) where it is automatically located and recognized by the binder.

In case two bindfiles are specified, one in an input archive segment and the other in an update archive segment, the latter takes precedence and an appropriate message is printed to that effect.

The binder's symbolic instructions have their own syntax that allows for statements consisting of a key word followed by zero or more parameters and then delimited by a statement delimiter. Master statements pertain to the entire bound object segment; normal statements pertain to a single component object within the bound object segment. Master statements are identified by master key words that are distinct from normal key words in that they begin with a capital letter; normal key words begin with a lowercase letter. A key word designates a certain action to be undertaken by the binder pertaining to parameters following the keyword.

Following is a list of the delimiters used:

:	key word delimiter. It is used to identify a key word followed by one or more parameters. A key word that is followed by no parameters is delimited by a statement delimiter.
;	statement delimiter.
,	parameter delimiter (the last parameter is delimited by a statement delimiter).
/*	begin comment.
*/	end comment.

Normal Key Words

objectname — the single parameter is the name of a component object as it appears in the archive segment. The objectname statement indicates that all following normal statements (up to but not including the next objectname statement) pertain to the component object whose name is the parameter of the objectname statement.

synonym — the parameters are symbolic segment names declared to be synonymous to the component object's objectname. When b is declared to be a synonym for a, references (in the bound components) of the form b or b$x (any x) are resolved during binding by searching for a definition of b or x in component a. A synonym instruction must be given if such references are to be prelinked. The synonym instruction also affects dynamic linking so that if b is a reference name for the bound segment, then references of the form b or b$x are resolved by searching component a. In this case, the synonym instruction may reduce the cost of dynamic linking, and it avoids possible ambiguities when two components contain definitions for the symbol b. Users should take care to state explicitly in a synonym statement all the normally used segment names of a component object. For example, the commands list, list_names, and list_totals are all implemented in one procedure, and all have abbreviations; thus a bindfile for the bound segment in which this procedure resides would contain:

```
objectname:    list;

synonym:       ls, list_names, ln, list_totals, lt;
```

Failure to state segment names results in inefficient linker performance.

retain	the parameters are the symbolic names of external symbols defined within the component object segment that the user wishes to retain as external symbols of the bound object segment.
delete	the parameters are the symbolic names of external symbols defined within the component object segment that the user does not wish to be retained as external symbols of the new bound segment.
	The retain and delete statements are considered exclusive. An error message is displayed if the binder recognizes that two or more such statements were made regarding any single external symbol.
no_link	the parameters are the symbolic names of external symbols that are <u>not</u> to be prelinked during binding. The no_link statement implies a retain statement for the specified symbols.
global	the global statement can have as its parameter either retain, delete, or no_link. The parameter selected becomes effective for all external symbols of the component object. An explicit retain, delete, or no_link statement concerning a given external symbol of the component object overrides the global statement for that specific external symbol. A global no_link causes all external references to the component object to be regenerated as links to external symbols; this allows execution time substitution of such a component by a free standing version of it, for example for debugging purposes.
table	does not require parameters. It causes the symbol table for the component to be retained and is needed to override the No_Table master key word, which is described below.

<u>Master Key Words</u>

Objectname	the parameter is the segment name of the new bound object.
Order	the parameters are a list of objectnames in the desired binding order. In the absence of an order statement, binding is done in the order of the input sequence. The order statement requires that there be a one-to-one correspondence between its list of parameters and the components of the input sequence.
Force_Order	same as Order, except that the list of parameters can be a subset of the input sequence, allowing the archive segments to contain additional segments that are not to be bound (e.g., source programs).

Global — is the same as the global statement except that it pertains to all component object segments within the bound segment. A global or explicit statement concerning a single component object or a single external symbol of a component object overrides the Global statement for that component object or symbol.

Addname — the parameters are the symbolic names to be added to the bound segment. If Addname has no parameters, it causes the segment names and synonyms of those component objects for which at least a single external symbol was retained to be added to the bound object segment.

No_Table — does not require parameters. It causes the symbol tables from all the component symbol sections containing symbol tables to be omitted from the bound segment. If this key word is not given, all symbol tables are kept.

If no bindfile is specified, the binder assumes default parameters corresponding to the following:

```
Objectname:  segment name of the first input archive file.

Global:      retain;  /*regenerate all definitions*/
```

<u>Error Messages</u>

The binder produces three types of error messages. Messages beginning with the word "Warning:" do not necessarily represent errors, but warns the user of possible inconsistencies in the input components or bindfile. Messages beginning with the word "binder_:" normally represent errors in the input components. Errors detected during the parsing of the bindfile have the format:

Bindfile Error Line #<u>n</u> ....

where <u>n</u> is the line number of the erroneous statement. If an error is detected during parsing, the binder aborts because it would not be able to bind according to the user´s specifications.

The message

"binder_: Fatal error has occurred; binding unsuccessful."

indicates that it was impossible for the binder to produce an executable object segment because of errors detected during binding. The bound object segment is left in an unpredictable state.

Examples

The bindfile for the apl command, which is named bound_apl.bind, is as follows:

```
Objectname:  bound_apl;
Global:      delete;    /*delete all old definitions*/


objectname:  apl;
retain:      apl;       /*retain definition for single entry*/
```

The bindfile for the debug command, which is named bound_debug.bind, is as follows:

```
Objectname:  bound_debug;
Global:      delete;    /*delete all old definitions*/


Addname;                /*add names debug, db, list_arg_
                          and gr_print to bound segment
                          bound_debug_*/


objectname:  debug;
synonym:     db;        /*indicate db is synonymous to debug*/
retain:      debug,
             db;        /*retain entrynames debug$debug and
                          debug$db*/


objectname:  list_arg_;
retain:      list_arg_; /*retain entryname list_arg_$list_arg_*/
objectname:  gr_print;
retain:      gr_print;  /*retain entryname gr_print$gr_print*/
```

Name: calc

The calc command provides the user with a calculator capable of evaluating arithmetic expressions with operator precedence, a set of often-used functions, and a memory that is symbolically addressable (i.e., by identifier).

Usage

```
    calc
```

initiates the command. The user can then type in expressions, assignment statements, list requests, or a quit request, separated from each other by one or more newline characters. All of these operations are described below.

Expressions

Arithmetic expressions involving real values and the operands +, -, *, /, and ** (addition, subtraction, multiplication, division, and exponentiation) can be typed in. A prefix of either plus or minus is allowed. Parentheses can be used, and blanks between operators and values are ignored. Calc evaluates the expression according to rules of precedence and prints out the results.

The order of evaluation is as follows:

1. expressions within parentheses
2. function references
3. prefix +, prefix -
4. **
5. *, /
6. +, -

For example, if the user types:

```
    2 + 3 * 4
```

calc responds:

```
    =   14
```

Operations of the same level are processed from left to right except for the prefix plus and minus, which are processed from right to left. This means 2**3**4 is evaluated as (2**3)**4.

Numbers can be integers (123), fixed point (1.23) and floating point (1.23e+2, 1.23e2, 1.23E2, or 1230E-1). All are stored as float bin(27). An accuracy of about seven figures is maintained. Variables (see below) can be used in place of constants, e.g., pi * r ** 2.

Seven functions are provided: sin, cos, tan, atan, abs, ln, and log (ln is base e, log is base 10). They can be nested to any level, e.g., sin(ln(var).5*pi/180).

Assignment Statement

The value of an expression can be assigned to a variable. The name of the variable must be from one to eight characters in length and must be made up of letters (uppercase and/or lowercase) and the underscore character (_). The form is:

```
<variable>=<expression>
```

For example, the following are legal assignment statements:

```
x = 2

Rho = sin(2*theta)
```

The calc command does not print any response to assignment statements. The variables "pi" and "e" have preassigned values of 3.14159265 and 2.7182818, respectively.

List Request

If "list" is typed, calc prints out the names and values of all the variables that have been declared so far. The value of any individual variable can be displayed by typing the name of the variable followed by a newline.

Quit Request

Typing "q" causes calc to return to the calling program, i.e., to command level.

Examples

The lines typed by the user are preceded by an exclamation mark (!).

```
!    calc

!    2+2

     =    4

!    r = 1.5

!    pi*r**2

     =    7.068583

!    sin(0.01)

     =    9.999832E-3

!    143e11+(12e13

     too few )

!    143e11+(12e13)

     =    1.343E+14

!    list

     r      =    1.5

     e      =    2.718282

     pi     =    3.141592

!    q
```

Name: cancel_abs_request, car

The cancel_abs_request command allows a user to delete a request for an absentee computation that is no longer required. Normally the deletion can be made only by the user who originated the request.

Usage

```
     cancel_abs_request path -control_args-
```

where:

1. path — is the pathname of the absentee control segment associated with this request.

2. control_args — are selected from the following list of control arguments and can appear anywhere on the command line:

-queue n, -q n	indicates which priority queue is to be searched. It must be followed by a decimal integer specifying the number of the queue. If this control argument is omitted, the third priority queue is searched unless the -all control argument is provided. (See below.)
-all, -a	indicates that all priority queues are to be searched starting with the highest priority queue and ending with the lowest priority queue.
-brief, -bf	indicates that the message "Absentee request path cancelled" is omitted.

Notes

The last request for an absentee computation is deleted if there is more than one request associated with the same absentee control segment in the same queue.

The segment name, path, must be specified in the same way it was at the time the absentee request was submitted. That is, >udd>Multics>Jones is not the same as >user_dir_dir>Multics>Jones. The list_abs_requests command may be used to ascertain the form in which path was originally given.

If the request refers to an absentee process that is already logged in, this command is not effective in stopping the absentee computation.

Example

```
     cancel_abs_request >udd>Multics>Jones>dump>translate
```

would delete the last absentee request that the user had made in queue 3 that was associated with the control segment >udd>Multics>Jones>dump>translate.absin.

This page intentionally left blank.

Name: cancel_cobol_program, ccp

The cancel_cobol_program command causes one or more programs in the current COBOL run unit to be cancelled. Cancelling ensures that the next time the program is invoked within the run unit, its data is in its initial state. Any files that have been opened by the program and are still open are closed and the COBOL data segment is truncated. Refer to the run_cobol command for information concerning the run unit and the COBOL runtime environment.

Usage

```
cancel_cobol_program names -control_arg-
```

where:

1.	names	are the reference names of COBOL programs that are active in the current run unit. If the name specified in the PROG-ID statement of the program is different from its associated name*i* argument, name*i* must be in the form refname$prog-id.
2.	control_arg	may be -retain_data or -retd to leave the data segment associated with the program intact for debugging purposes. (See "Notes" below.)

Notes

The results of the cancel_cobol_program command and the execution of the CANCEL statement from within a COBOL program are similar. The only difference is that if a name*i* argument is not actually a component of the current run unit, an error message is issued and no action is taken; for the CANCEL statement, no warning is given in such a case.

To preserve program data for debugging purposes, the -retain_data control argument should be used. The data associated with the cancelled program is in its last used state; it is not restored to its initial state until the next time the program is invoked in the run unit.

Refer to the following related commands:

display_cobol_run_unit, dcr
stop_cobol_run, scr
run_cobol, rc

<u>Name</u>: cancel_daemon_request, cdr

The cancel_daemon_request command deletes a dprint or dpunch request that is no longer required. Normally the deletion can be made only by the user who originated the request. See the descriptions of the dprint and dpunch commands.

<u>Usage</u>

```
     cancel_daemon_request path -control_args-
```

where:

1. path	is the pathname of the segment or multisegment file for which the dprint or dpunch request is to be cancelled.
2. control_args	are selected from the following list of control arguments and can appear anywhere in the command line:
-request_type XX, -rqt XX	indicates that the request to be cancelled is to be found in the queue for the request type identified by the string XX. If this control argument is not given, the default request type is "printer". Request types can be listed by the print_request_types command.
-queue <u>n</u>, -q <u>n</u>	indicates which priority queue is to be searched. It must be followed by an integer specifying the number of the queue. If this control argument is omitted, only the third priority queue is searched unless the -all control argument is provided (see below).
-all, -a	indicates that all priority queues for the specified device class are to be searched starting with the highest priority queue and ending with the lowest priority queue.
-brief, -bf	indicates that the message "dprint (dpunch) of path cancelled" is to be omitted.

<u>Notes</u>

The last request to print or punch a path is deleted if there is more than one request associated with the same user for that path in the same queue.

Only request types of generic type "printer" or "punch" can be specified by the -request_type control argument. These request types can be listed by invoking the print_request_types command.

If the request refers to a path that the I/O daemon is already processing, this command is not effective in stopping the print or punch operation.

Examples

```
     cancel_daemon_request >udd>Alpha>Jones>dump>translate.list
```

deletes the last request that the user made in queue 3 (for the default request type) to print the segment >udd>Alpha>Jones>dump>translate.list.

```
     cancel_daemon_request >udd>Alpha>Jones>dump>probe.pl1 -request_type punch
```

deletes the last request that the user made in queue 3 (for the request type "punch") to punch the segment >udd>Alpha>Jones>dump>probe.pl1.

Name: change_default_wdir, cdwd

The change_default_wdir command records a specified directory as the user's default working directory for the duration of the current process or until the next change_default_wdir command is issued.

Usage

```
     change_default_wdir -path-
```

where path is the pathname of the directory that is to become the default working directory. If path is not given, the current working directory becomes the default working directory.

Notes

The change_default_wdir command is used in conjunction with the change_wdir command. When the change_wdir command is issued with no argument, the default working directory becomes the current working directory.

The original default working directory is the user's home directory upon logging in.

See also the descriptions of the change_wdir and print_default_wdir commands.

Name: change_error_mode, cem

This command is used to control the amount of information printed by the default handler for system conditions. It determines the length of messages for the life of a process or until it is invoked again in the process.

Usage

```
     change_error_mode -control_args-
```

where control_arg can be chosen from the following list:

-brief, -bf	prints only the condition name.
-long, -lg	prints more complete messages. In particular, if the condition was detected in a support procedure, the name of that procedure is printed in addition to the name of the most recent user procedure. If a segment that signalled a condition (or caused it to be signalled) is bound, both the offset relative to the base of the procedure and the offset relative to the base of the segment are printed.

Notes

If this command is not issued the message is in normal mode. If this command is issued with no argument, normal mode is the default. It can be used to return the value -long or -brief to the default value. The normal mode prints a message intermediate in length between the brief and long messages.

For a complete discussion of conditions and their handling see Section VI of the MPM Reference Guide. Refer to the description of the reprint_error command for a similar, but more selective, capability.

Name: change_wdir

The change_wdir command changes the user's working directory to the directory specified as an argument. A working directory is a directory in which the user's activity is centered. Its pathname is remembered by the system so that the user need not type the absolute pathname of segments inferior to that directory.

Usage

```
    change_wdir -path-
```

where path is the pathname of the directory that is to become the working directory. If path is not given, the default working directory is assumed.

Notes

If path specifies a nonexistent directory, an error message is printed on the user's terminal and the current working directory is not changed.

A user must have status permission on the directory containing path, but no access to path is required for the command to be employed. However, once the working directory has been changed, the user can proceed only according to his access to path. That is, to effectively use path as a working directory, he must have sma access permission for path. Restricted uses are possible in accordance with the mode attributes on the directory. For example, the user must have at least status permission to list the directory.

See also the descriptions of the change_default_wdir and print_default_wdir commands.

Name: check_info_segs, cis

The check_info_segs command prints a list of new or modified segments. It saves the current time in the user profile, so that when it is invoked again, it lists segments created or modified since the last invocation.

Optional control arguments allow check_info_segs to be used to perform a specified command on each modified segment, or to search any directory for modified segments, or to use a time other than that of the last invocation for the comparison.

Usage

```
     check_info_segs -control_args-
```

where control_args can be selected from the following:

-date date_string, -dt date_string	If this argument is given, check_info_segs uses the date specified by date_string instead of the date in the user profile. The date_string argument must be acceptable to the convert_date_to_binary_ subroutine (described in the MPM Subroutines). The time of last invocation in the user profile is not updated to the current time.
-long, -lg	If this argument is specified, check_info_segs lists the date-time-entry-modified as well as the name of any segment selected as having been created or modified during the interval in question.
-brief, -bf	If this argument is specified, check_info_segs does not print the names of selected segments and suppresses the comment "no change" if no segments are selected as having been created or modified during the interval in question. This control argument is intended for use with the -call control argument described below.
-no_update, -nud	If this argument is specified, check_info_segs does not place the current time into the user profile.
-call cmdline	If this argument is specified, check_info_segs calls the command processor with a string of the form "cmdline path" for each selected segment, after the name of the segment is typed; path is the absolute pathname of the segment. The cmdline must be enclosed in quotes if it contains blanks.
-pathname spath, -pn spath	If this control argument is specified, check_info_segs assumes that spath is a pathname with one or more asterisks (stars) in the entryname portion. All new or modified segments that match spath are selected. Refer to "Constructing and Interpreting Names" in Section III of the MPM Reference Guide for a discussion of star names.

Up to 10 occurrences of this argument can appear in a call to check_info_segs. All specified directories are searched, in the order that the arguments were given. If the -pathname argument is not specified, the defaults are -pn >documentation>info>**.info and -pn >documentation>iml_info>**.info.

Notes

The first time check_info_segs is invoked by a particular user, it just initializes the time in the user profile to the current time, prints a comment, and does not list any segments. If a profile does not exist, check_info_segs creates one in the user's home directory. The profile segment has the name Person_id.profile, where Person_id is the Person_id given at login time.

The check_info_segs command checks the date-time-entry-modified for any segment pointed to by a link, not the time the link was modified.

The check_info_segs command cannot detect that a segment has been deleted since the last invocation of the command.

Examples

To check for info segments modified since the specified date, type:

```
check_info_segs -date "07/01/74 0900."
```

To print all modified info segments, type:

```
check_info_segs -call print -bf
```

The "-bf" argument is given to check_info_segs to suppress duplicate printing of segment names since the print command types the segment name in the heading.

To print just the first block of any modified info segment, type:

```
check_info_segs -call "answer no help -pn"
```

The -pn argument must be given to the help command, since check_info_segs supplies an absolute pathname as the last argument in the command line.

To check for all modified segments in a project-maintained directory as well as the system directories, type:

```
check_info_segs -nud -pn >udd>Project_id>documentation>*.info
check_info_segs
```

The use of the -nud argument prevents the time of last invocation from being updated in the first command line.

Name: close_file, cf

The close_file command closes specified FORTRAN and PL/I files. It closes all open FORTRAN and PL/I files if the -all control argument is specified.

Usage

```
close_file -control_arg- filenames
```

where:

1. control_arg can have the value -all to close all open files. In this case, no filename appears.

2. filenames are the names of open FORTRAN or PL/I files.

Notes

The format of a FORTRAN file name is file*nn* where *nn* is a two-digit number other than 00; e.g., file05. PL/I file names are selected by the user and can have any format.

If a specified file cannot be found, an error message is printed indicating the name of the file. The rest of the specified files are closed.

For each filename, all PL/I files of that name and, if applicable, the FORTRAN file of that name are closed.

The command "close_file -all" does not affect I/O switches that are not associated with FORTRAN or PL/I files.

Name: cobol

The cobol command invokes the COBOL compiler to translate a segment containing the text of a COBOL source program into a Multics object segment. A listing segment can also be produced. These segments are placed in the user's working directory. This command cannot be called recursively. For information on COBOL, refer to the Multics COBOL Users' Guide, Order No. AS43 and the Multics COBOL Reference Manual, Order No. AS44.

Usage

```
cobol path -control_args-
```

where:

1. path — is the pathname of a COBOL source segment that is to be translated by the COBOL compiler. If path does not have a suffix of cobol, then one is assumed. However, the suffix cobol must be the last component of the name of the source segment.

2. control_args — may be chosen from the following list:

   -source, -sc — produces a line-numbered, printable ASCII listing of the program.

   -symbols, -sb — produces a source program listing (like the -source control argument), followed by a cross-reference listing of all data names defined in the program.

   -map — produces a source program listing with symbols (like the -symbols control argument), followed by a map of the object code generated by this compilation. The -map control argument produces sufficient information to allow the user to debug most problems online.

   -list, -ls — produces a source program listing with symbols (like the -symbols control argument), followed by an assembly-like listing of the compiled object program. Use of the -list control argument significantly increases compilation time and should be avoided whenever possible by using the -map control argument.

   -brief, -bf — causes error messages written to the user_output I/O switch to contain only an error number and statement identification once the full message has been given on the first occurrence. In the normal, nonbrief mode, an explanatory message is always written.

   -severity*i*, -sv*i* — causes error messages whose severity is less than *i* (where *i* is 1, 2, 3, or 4) to not be written to the user_output I/O switch although all errors are written into the listing. If this control argument is not given, a severity level of 2 is assumed. See the description of severity levels under "Error Diagnostics" below.

-check, -ck	is used for syntactic and semantic checking of a COBOL program. No code is generated.
-table, -tb	generates a full symbol table for use by symbolic debuggers; the symbol table is part of the symbol section of the object program and consists of two parts: a statement table that gives the correspondence between source line numbers and object locations and an identifier table that contains information about every identifier actually referenced by the source program. The table appears in the symbol section of the object segment produced by the compilation. This control argument usually causes the object segment to become significantly longer.
-format, -fmt	accepts source segment in the format acceptable to the format_cobol_source command.

The following control arguments are available, but are probably not of interest to every user.

-debug, -db	leaves the work files generated by the compiler intact after a compilation. This control argument is used for debugging the compiler. The command cobol$clean_up may be used to discard these files. Also, this causes severity 4 errors to not unwind and abort the compilation, but rather to invoke a new level of the command processor at the point of the error.
-time, -tm	prints the time (in seconds) and the number of page faults taken by each phase of the compiler; prints the total time at the end of the compilation. This is directed to the user_output I/O switch.

Notes

The only result of invoking the cobol command without control arguments is to generate an object segment.

A normal compilation produces an object segment and leaves it in the user's working directory. If an entry with that name existed previously in the directory, its access control list (ACL) is saved and given to the new copy of the object segment. Otherwise, the user is given re access to the segment with ring brackets v,v,v where v is the validation level of the process that is active when the object segment is created.

If the user specifies the -source, -symbols, -map, or -list control arguments, the cobol command creates a listing segment named path.list. The ACL is set as described for the object segment except that the user is given rw access to it when newly created. Previous copies of the object segment and the listing segment are replaced by the new segments created by the compilation.

Error Diagnostics

The COBOL compiler can diagnose and issue messages for about 800 different errors. These messages are graded in severity as follows:

1	Warning only. Compilation continues without ill effect.
2	Correctable error. The compiler attempts to remedy the situation and continues, possibly without ill effect. The assumptions the compiler makes in remedying the situation, however, do not necessarily guarantee the right results.
3	Uncorrectable but recoverable error. That is, the program is definitely in error and no meaningful object code can be produced, but the compiler can continue executing and diagnosing further errors.
4	Unrecoverable error. The compiler cannot continue beyond this error. A message is printed and control is returned to the cobol command. The command writes an abort message on the error_output I/O switch and returns to its caller.

As indicated above, the user can set the severity level so that he is not bothered by minor error messages. He can also specify the -brief control argument so that the message is shorter. Since the default severity level is 2, the user must explicitly specify the -severity1 (or -sv1) control argument when he invokes the cobol command to have observation messages printed. Neither the -severity<u>i</u> nor -brief control argument has any effect on the contents of the listing segment if one is produced.

An example of an error message in its long form is:

```
22        use after error procedure on extend.
                                       1
** 1 5-250 A use procedure has already been associated with this processing
mode.
```

If the -brief control argument is specified and message 5-250 has previously been given in its long form, the user instead sees:

```
22        use after error procedure on extend.
                                       1
** 1 5-250
```

In the second case, the user could look up error number 5-250 in Appendix A of the <u>Multics COBOL Users' Guide</u> and get the full message (or of course he could refer to the previously printed message). If the user had set his severity level to 3, he would have seen no message at all. Notice that the number of asterisks immediately preceding the error indicator corresponds to the severity level of the error.

This page intentionally left blank.

This page intentionally left blank.

Name: compare

The compare command compares two segments and lists their differences. The comparison is a word-by-word check and can be made with a mask so that only specified parts of each word are compared.

Usage

```
compare path1|offset1 path2|offset2 -control_args-
```

where:

1. path1, path2	are the pathnames of the segments to be compared. The equal convention is allowed for path2.
2. offset1, offset2	are octal offsets within the segments to be compared. The comparison begins at the word specified or at the first word of the segment if no offset is specified. If an offset is omitted, the vertical bar should also be omitted.
3. control_args	can be chosen from the following control arguments:
-mask _n_	the octal mask _n_ is to be used in the comparison. If _n_ is less than 12 octal digits, it is padded on the left with zeros.
-length _n_, -ln _n_	the comparison should continue for no more than _n_ (octal) words.

Notes

The maximum number of words to be compared is the word count of the first segment minus its offset or the word count of the second segment minus its offset, whichever is greater. If the -length control argument is supplied, comparison stops after the specified number of words. If the segments are of unequal length, the remaining words of the longer segment are printed as discrepancies. The word count of a segment is computed by dividing the bit count plus 35 by 36. If the word count minus the offset is less than zero, an error message is printed and the command is aborted.

Any discrepancies found by the command are listed in the following format:

```
offset      contents        offset          contents
   4      404000000002        4           000777000023
   6      404000000023        6           677774300100
```

To compare segments containing only ASCII character-string data, use the compare_ascii command.

Name: compare_ascii, cpa

This command compares two ASCII segments and prints the changes made to the segment specified by pathA to yield the segment pathB. The output is organized with the assumption that the pathA segment was edited to produce pathB. This command prints lines that were added, replaced, or deleted; it identifies each line by line number within the respective segment and also by the letter A or B to indicate which segment the line is from (A for pathA and B for pathB).

Usage

```
     compare_ascii pathA pathB -minchars- -minlines-
```

where:

1.	pathA	is the pathname of the segment that is identified by the letter A in the command output.
2.	pathB	is the pathname of the segment that is identified by the letter B in the command output (presumably the result of editing the first segment).
3.	minchars	is an optional decimal number specifying the minimum number of characters that must be identical before compare_ascii assumes the segments are again synchronized ("in sync") after a difference in the two segments. See "Notes" below.
4.	minlines	is an optional decimal number analogous to minchars specifying the minimum number of lines that must be identical. It is required that minchars be specified in order to use this argument.

Notes

The equal convention can be used.

The defaults for minchars and minlines are 50 and 5 respectively. This means that if a difference between the segments is encountered, at least five lines must be identical before the segments are considered to be back in sync again, and if a group of five lines contains less than 50 characters, then at least 50 characters must be identical before the segments are considered to be in sync again. (Thus, both minima must be met or exceeded.)

When compare_ascii detects a difference (the next line in pathA is not equal to the next line in pathB), it attempts to get the segments back in sync--that is, to find the line in each segment where the difference ends and the next several lines in pathA are equal to the next several lines in pathB. The number of lines required to be equal is given by minlines, as described above.

The requirement that the number of identical lines specified by minlines appear in the two segments decreases the probability of the comparisons getting out of sync in segments where the same line or sequence of lines occurs at several different places. (For example, if minlines were set to 1, then any two blank lines would cause the program to assume that it had found the end of the difference--but blank lines occur at many different places in some ASCII segments.)

The requirement that minchars identical characters appear serves the same purpose for segments where the same sequence of short lines is repeated several times--for example, multiple blank lines, or a series of control lines that precedes each paragraph in a runoff segment, or a series of end statements terminating nested do's in a PL/I program. If a group of minlines lines is equal in both segments, but the group contains a total of fewer than minchars characters, the command adds enough additional characters to the group (possibly ending it with part of a line) so that the the group contains exactly minchars characters, and it requires that this group be identical in both segments before it assumes that the segments are back in sync.

From the above discussion, it can be seen that, if minchars or minlines is set too low, the command assumes that it has found the end of a difference when it is not really looking at the same spot in both segments. Once out of sync, compare_ascii probably never gets back into sync, and thus it reports that the remainder of pathA was completely replaced by the remainder of pathB.

On the other hand, if minchars and minlines are set too high (in comparison with the frequency of differences between the segments), then the command gets back into sync infrequently, if ever, and thus reports complete replacement of large sections, or of the entire segment. For example it every fifth line of pathA were replaced to produce pathB, then the pattern of four lines equal, one line unequal occurs throughout the segments. Using the default of minlines = 5, the command would never find five consecutive lines that match, and would thus report that pathA was completely replaced by pathB.

The user should understand how minlines and minchars are used in the comparison algorithm and choose values for them that are likely to give the most useful results for the particular segments being compared, considering both the nature of the differences between the segments and the likelihood of line sequences occurring repeatedly in the segments.

Example

```
compare_ascii oldprog.pl1 prog.pl1 20 2
```

This command line requests a comparison between the source segment prog.pl1 and a (presumably) old version of the same program, named oldprog.pl1, both in the user's working directory. The values for minchars and minlines (20 and 2, respectively) mean that the command assumes that it has found the end of a difference when it finds two consecutive lines that are the same in both segments--provided that the two lines contain a total of at least 20 characters. If they do not, then the command does not assume it has found the end of the difference unless there are enough additional matching characters in subsequent lines to make 20 characters the same.

Name: copy, cp

The copy command causes copies of specified segments and multisegment files to be created in the specified directories with the specified names. Access control lists (ACLs) and multiple names are optionally copied.

Usage

```
copy path11 path21 ... path1n -path2n- -control_args-
```

where:

1.	path1i	is the pathname of a segment or multisegment file to be copied.
2.	path2i	is the pathname of a copy to be created from path1i. If the last path2 argument is not given, the copy is placed in the working directory with the entryname of path1n.
3.	control_args	can be chosen from the following list of control arguments:

-name, -nm	copies multiple names.
-acl	copies the ACL.
-all, -a	copies multiple names and ACLs.
-brief, -bf	suppresses the warning messages "Bit count inconsistent with current length..." and "Current length is not the same as records used...".

The control arguments can appear once anywhere in the command line after the command name and apply to the entire command line.

Notes

Read access is required for path1i. Status permission is required for the directory containing path1i. Append permission is required for the directory containing path2i. Modify permission is required if the -name, -acl, or -all control argument is used.

The star and equal conventions can be used. See "Constructing and Interpreting Names" in Section I for a description of the star and equal conventions.

If the ACL of a segment or multisegment file is being copied, then the initial ACL of the target directory has no effect on the ACL of the segment or multisegment file after it has been copied into that directory. The ACL remains exactly as it was in the original directory.

Since two entries in a directory cannot have the same entryname, special action is taken by this command if the name of the segment or multisegment file being copied (specified by path1*i*) already exists in the directory specified by path2*i*. If the entry being copied has an alternate name, the entryname that would have resulted in a duplicate name is removed and the user is informed of this action; the copying operation then takes place. If the entry being copied has only one entryname, the entry that already exists in the directory must be deleted to remove the name. The user is asked if the deletion should be done; if the user answers "no", the copying operation does not take place.

The copy command prints a warning message if the bit count of path1*i* is less than its current length or if the current length is greater than the number of records used. These warnings are suppressed by the use of the -brief control argument.

Example

```
copy >old_dir>fred.list george.=
```

The segment or multisegment file named fred.list in the directory >old_dir is copied into the working directory as george.list.

Name: copy_cards

The copy_cards command copies specified card image segments from system pool storage into a user's directory. The segments to be copied must have been created using the Multics Card Input Facility. The user process executing this command must have the proper access to the card image segment in order to perform the copy. When there are multiple copies of the same deck in pool storage, all are copied.

Usage

```
copy_cards deck_name -new_deck_name-
```

1.	deck_name	is the name that was entered on the deck_id card when the card deck was submitted for reading.
2.	new_deck_name	is the pathname of the segment in which the matching card image segment is to be placed. If omitted, the working directory and deck_name are assumed.

Notes

The deck_name may use the star convention, and when there are matching card image segments in pool storage to which the user has access, all are copied. Similarly, new_deck_name may use the equal convention.

When an attempt is made to read a card deck having the same name as some previously read deck still in pool storage, a numeric suffix is added to the name of the new deck, e.g., "deck_name.1". Repeated name duplications cause successively larger numeric suffixes to be used. (Name duplicatons can only occur for decks of the same access class submitted by the same user.) The copy_cards command informs the user of such duplications (if any) and retrieves all copies of the specified deck.

Only those card decks having an access class equal to the user's current authorization can be copied. Other decks will not be found.

See the description of the card input facility in Section V of the MPM Reference Guide for the format of the control cards needed when submitting a card deck to be read by system operations.

Example

```
copy_cards my_deck
```

copies the user's card image segment named my_deck from the card pool storage into the user's current working directory.

Names: copy_file, cpf

The copy_file command copies records from an input file to an output file. The input and output file records must be structured. (See "Unstructured Files" below for an explanation of how unstructured files can be copied.) The input file can be copied either partially or in its entirety.

The copy command makes an exact duplicate of the input file, whereas copy_file produces an output file that has been restructured for maximum compactness. (See the description of the copy command in this manual.)

Usage

```
copy_file in_control_arg out_control_arg -cpf_control_args-
```

where:

1. in_control_arg	is one of two input control arguments that specifies the input file from which records are read. It may be either an I/O switch name or an attach description. (See "Notes" below.)
-input_switch XX, -isw XX	specifies the input file by means of an already attached I/O switch name, where XX is the switch name.
-input_description "XX", -ids "XX"	specifies the input file by means of an attach description, where XX is the attach description. The attach description string must be enclosed in quotes.
2. out_control_arg	is one of two output control arguments that specifies the output file to which these records are written. It may be either an I/O switch name or an attach description. (See "Notes" below.)
-output_switch XX, -osw XX	specifies the output file by means of an already attached I/O switch name, where XX is the switch name.
-output_description "XX", -ods "XX"	specifies the output file by means of an attach description, where XX is the attach description. The attach description string must be enclosed in quotes.
3. cpf_control_args	may be one or more of the following control arguments. (See "Notes" below.)
-keyed	copies both records and keys from a keyed sequential input file to a keyed sequential output file. The default is to copy records from an input file (either keyed or not) to a sequential output file. (See "Keyed Files" below.)

-from n, -fm n	copies records beginning with the nth record of the input file, where n is a positive integer. The default is to begin copying with the "next record." (See "Notes" below.)
-start XX, -sr XX	copies records beginning with the record whose key is XX, where XX is 256 or fewer ASCII characters. The default is to begin copying with the "next record."
-to n	copies until the nth record has been copied or the input file is exhausted, whichever occurs first, where n is a positive integer greater than or equal to the n given with the -from control argument. This control argument can only be specified if -from is also specified. The default is to perform copying until the input file is exhausted.
-stop XX, -sp XX	copies until the record whose key is XX has been copied or the input file is exhausted, whichever occurs first, where XX is 256 or fewer ASCII characters. This control argument can be specified without specifying the -start control argument. However, if -start is specified, the XX given with -stop must be greater than or equal to (according to the ASCII collating sequence) the XX given with -start. The default is to perform copying until the input file is exhausted.
-count n, -ct n	copies until n records have been copied or the input file is exhausted, whichever occurs first, where n is a positive integer. The default is to perform copying until the input file is exhausted.
-all, -a	copies until the input file is exhausted. This is the default.
-brief, -bf	suppresses an informative message indicating the number of records actually copied.
-long, -lg	prints an informative message indicating the number of records actually copied. This is the default.

Unstructured Files

The copy_file command operates by performing record I/O on structured files. If it is desired to copy from/to an unstructured file, the record_stream_ I/O module can be used, e.g.,

```
cpf -ids "record_stream_ -target vfile_ pathname" -osw OUT
```

The effect is to take lines from the file specified by pathname via the vfile_ I/O module, transform them into records via the record_stream_ I/O module, and then copy them to the I/O switch named OUT.

Keyed Files

The copy_file command can copy a keyed sequential file either as such, or as though it were purely sequential. By default, the command copies only records and does not place keys in the output file. To copy the keys, the -keyed control argument must be used. When -keyed is used, the input file must be a keyed sequential file. Whether keys are copied or not, control arguments can be used to delimit the range of records to be copied (i.e., -start, -stop, -from, -to, -count). Copying is always performed in key order.

Notes

If either the input or output specification is an attach description, it is used to attach a uniquely named I/O switch to the file. The switch is opened, the copy performed, and then the switch is closed and detached. Alternately, the input or output file may be specified by an I/O switch name. Either the io_call command or iox_ subroutine may be used to attach the file prior to the invocation of the copy_file command. (See the description of the io_call command in this manual and the iox_ subroutine in the MPM Subroutines.)

If the input file is specified by an I/O switch name and the switch is not open, the copy_file command opens it for (keyed_)sequential_input, performs the copy, and closes it. If the switch is already open when the copy_file command is invoked, the opening mode must be sequential_input, sequential_input_output, keyed_sequential_input, or keyed_sequential_update. The switch is not closed after the copy has been performed.

The "next record" must be defined if neither the -start nor -from control argument is used to specify an absolute starting position within the input file. If the I/O switch is opened by the copy_file command, the next record is the first record of the file; otherwise, the next record is that record at which the file is positioned when the copy_file command is invoked.

If the output file is specified by an I/O switch name and the switch is not open, the copy_file command opens it for (keyed_)sequential_output, performs the copy, and closes it. If the switch is already open when the copy_file command is invoked, the opening mode must be sequential_output, sequential_input_output, keyed_sequential_output, keyed_sequential_update, direct_output, or direct_update. (In update mode, output file records with keys that duplicate input file records are rewritten.) The switch is not closed after the copy has been performed.

The -from and -start control arguments are mutually exclusive. The -to, -stop, -count, and -all control arguments are mutually exclusive. The -brief and -long control arguments are mutually exclusive. The informative message, printed by default, appears as follows:

```
345 records copied.
```

Examples

Copy an entire file from an already attached file to the segment in_copy:

```
cpf -isw in -ods "vfile_ in_copy"
```

Copy the first 13 records from a tape file to an output file. The normal result of this command would be to print the first 13 records on the user's terminal. (The two lines below would actually be typed as only one line.)

```
cpf -ct 13 -ids "tape_ansi_ 887677 -name TEST21 -ret all"
   -ods "record_stream_ user_output"
```

Copy 13 records from an already attached file to another already attached file, starting with the 56th record of the input file:

```
cpf -isw in -osw out -from 56 -ct 13
```

Copy records 43 through 78 from an already attached file to an already attached file:

```
cpf -isw in -osw out -from 43 -to 78
```

Copy all but the first seven records from segment testdata.11 to an already attached file:

```
cpf -ids "vfile_ testdata.11" -osw out -fm 8
```

Copy an entire keyed sequential file with keys:

```
cpf -isw in -osw out -all -keyed
```

Copy 13 records of a keyed sequential file starting with the record whose key is ASD66 to a sequential output file. No keys are copied.

```
cpf -isw in -osw out -sr ASD66 -ct 13
```

Copy the records and keys from a keyed sequential file up to and including the record whose key is bb"bb:

```
cpf -keyed -isw in -osw out -sp "bb""bb"
```

This page intentionally left blank.

Name: create, cr

The create command causes a storage system segment to be created in a specified directory (or in the working directory). That is, it creates a storage system entry for an empty segment. See the description of the create_dir and link commands for an explanation of the creation of directories and links, respectively.

Usage

```
     create paths
```

where paths are the pathnames of the segments to be created.

Notes

The user must have append permission for the directories in question.

If the creation of a new segment would introduce a duplication of names within the directory, and if the old segment has only one name, the user is interrogated as to whether he wishes the segment bearing the old instance of the name to be deleted. If the old segment has multiple names, the conflicting name is removed and a message to that effect is issued to the user. In either case, since the directory is being changed, the user must also have modify permission for the directory.

The user creating the new segment is given rw access to the segment created.

All directories specified in paths must already exist. That is, only a single level of the storage system hierarchy can be created with one invocation of this command.

If any one of the paths is the name of an existing link, a segment is created in the place specified by that link. The user must have append access for the directory containing the link in order to create this segment.

Example

```
     create first_class_mail >new_dir>alpha>beta
```

would cause the segment first_class_mail to be created in the working directory and the segment beta to be created in the directory >new_dir>alpha. As explained above, the directories new_dir and alpha must already exist.

Name: create_dir, cd

The create_dir command causes a specified storage system directory branch to be created in a specified directory (or in the working directory). That is, it creates a storage system entry for an empty subdirectory. See the description of the create command for information on the creation of segments.

Usage

```
create_dir paths -control_args-
```

where:

1. paths — specify the names of the subdirectories to be created.

2. control_args — may be chosen from the following:

-access_class XX, -acc XX	applies to each path*i* and causes each directory created to be upgraded to the specified access class. The access class may be specified with either long or short names.
-logical_volume V, -lv V	specifies that each directory created is to be a master directory whose segments are to reside on logical volume named V.
-quota *n*	specifies the quota to be given to the directory when it is created. This argument must be specified if either the -access_class or -logical_volume control argument is given. If omitted, the directory is given zero quota. The value of *n* must be a positive integer, and applies to each path*i*.

Notes

The user must have append permission on the directories in question.

If a quota is specified and the directory being created is not a master directory, the containing directory must have sufficient quota to move quota to the directory being created. (See the move_quota command for additional information.)

If the creation of a new subdirectory would introduce a duplication of names within the directory, and if the old subdirectory has only one name, the operation is not performed. If the old subdirectory has multiple names, the conflicting name is removed and a message to that effect issued to the user.

The user is given sma access on the created subdirectory.

All superior directories specified in path*i* must already exist. That is, only a single level of storage system directory hierarchy can be created in a single invocation of the create_dir command.

In order to create a master directory, the user must have a quota account on the logical volume with sufficient volume quota to create the directory. A master directory must always have a nonzero quota; therefore, the -quota control argument must always be given when creating a master directory. A master directory can be created even though the logical volume is not mounted.

Each upgraded directory must have a quota greater than zero and must have an access class that is greater than its containing directory. The specified access class must also be less than or equal to the maximum access authorization of the process.

When the -access_class control argument is specified, the command does not create a new directory through a link. Creating through links is allowed only when the access class of the containing directory is taken as the default.

Examples

```
     create_dir sub >my_dir>alpha>new
```

creates the subdirectory sub immediately inferior to the current working directory and the subdirectory new immediately inferior to the directory >my_dir>alpha. As noted above, the directories my_dir and alpha must already exist. Both directories are assigned the access class of their containing directory.

```
     create_dir subA -access_class a,c1,c2 -quota 5
```

creates the subdirectory subA with an access class of a,c1,c2 and a quota of 5 pages. The directory subA would be created immediately inferior to the working directory. (The access class names a, c1, and c2 used in the example represent possible names defined for the site. See the print_auth_names command for more details on access class names.)

```
     create_dir subB -logical_volume volz -quota 100
```

creates a master directory subB immediately inferior to the working directory. Segments created in this new directory will reside on the logical volume named volz. The directory subB is given a quota of 100 records.

Name: cumulative_page_trace, cpt

The cumulative_page_trace command accumulates page trace data so that the total set of pages used during the invocation of a command or subsystem can be determined. The command accumulates data from one invocation of itself to the next. Output from the command is in tabular format showing all pages that have been referenced by the user's process. A trace in the format of that produced by the page_trace command can also be obtained.

The cumulative_page_trace command operates by sampling and reading the system trace array after invocation of a command and at repeated intervals. Control arguments are given to specify the detailed operation of the cumulative_page_trace command.

The command line used to invoke the cumulative_page_trace command includes the command or subsystem to be traced as well as optional control arguments.

Usage

```
    cumulative_page_trace command_line -control_args-
```

where:

1. command_line — is a character string to be interpreted by the command processor as a command line. If this character string contains blanks, it must be surrounded by quotes. All procedures invoked as a result of processing this command line are metered by the cumulative_page_trace command.

2. control_args — may be chosen from the following:

-reset, -rs	resets the table of accumulated data. If the table is not reset, data from the current use of cumulative_page_trace is added to that obtained earlier in the process.
-flush	clears primary memory before each invocation of the command line and after each interrupt. This helps the user determine the number of page faults but increases the cost.
-loop _n_	calls the command to be metered _n_ times.
-sleep _n_	waits for _n_ seconds after each call to the command being metered.
-interrupt _n_, -int _n_	interrupts execution every _n_ virtual CPU milliseconds for page fault sampling.
-timers	includes all faults between signal and restart.
-trace path	writes the trace on the segment path using I/O switch named cpt.out; cumulative_page_trace attaches and detaches this switch.

-print, -pr	prints the accumulated results, giving the number of each page referenced.
-total, -tt	prints the total number of page faults and the number of pages referenced for each segment.
-count, -ct	prints the accumulated results, giving the number of each page and the number of faults for each page.
-long, -lg	produces output in long format, giving full pathnames.
-short, -sh	formats output for a line length of 80.

Notes

At least one of three generic operations must be requested. They may all be combined and, if so, are performed in the following order: resetting the table of accumulated data, calling the command to be metered applying the specified options, and printing the results in the specified format.

The default mode of operation permits no interrupts for page fault sampling. If the command or subsystem to be metered will take more than several hundred page faults, linkage faults, or other system events that are indicated in the page trace array, it is recommended that interrupts be requested. If the user does not know a suitable value for the -interrupt control argument, the value recommended is 400 milliseconds. If this figure is too large, messages indicate that some page faults may have been missed; a smaller value can then be chosen. The cost of a smaller value is high and may cause additional side effects. If the command or subsystem to be metered includes the taking of CPUT interrupts, then the -timers control argument should be given. This control argument causes some of the page faults of the metering mechanism to be included as well.

Only one of the control arguments -print, -count, or -total may be given. Each of these control arguments produces printed output in a different format. If more than one format is desired, the command must be invoked once for each format.

Examples

```
cpt "pl1 test" -interrupt 400 -trace trace_out
```

calls the pl1 command to compile the program named test, requesting an interrupt every 400 milliseconds to obtain page trace information. Trace information is placed in a segment named trace_out.

```
     cpt "list -pn >udd>Multics" -loop 2 -sleep 10
```

calls the list command twice, and sleeps for ten seconds between calls.

```
     cpt -print
```

prints the accumulated results of previous metering.

Name: debug, db

The debug command is an interactive debugging aid to be used in the Multics environment. It allows the user to look at or modify data or code. The user may stop execution of his program and examine its state by inserting "breakpoints" in the program before and/or during execution. A concise syntax for user requests, coupled with a complete system of defaults for unspecified items, allows the user to make many inquiries with little effort. Symbolic references permit the user to retreat from the machine-oriented debugging techniques of conventional systems and to refer to variables of interest directly by name.

The command, debug, uses a segment in the home directory to keep track of information about breaks. The segment is created if not found. If the segment cannot be created, the break features of debug are disabled and unusable. The name of the break segment is Person_id.breaks where Person_id is the login name of the user.

Usage

```
     debug
```

Notes

Through the debug command, the user can:

1. Look at data or code;
2. Modify data or code;
3. Set a break;
4. Perform (possibly nonlocal) transfers;
5. Call procedures;
6. Trace the stack being used;
7. Look at procedure arguments;
8. Control and coordinate breaks;
9. Continue execution after a break fault;
10. Change the stack reference frame;
11. Print machine registers; and
12. Execute commands.

These functions are provided by two types of debug requests: data requests and control requests. The first five functions above are performed by data requests; the others, by control requests. Several debug requests (either data or control) can be placed on a line separated by semicolons (;).

Number Representation Conventions

Debug uses both octal and decimal representation of numbers. In general, machine-dependent numbers such as pointers, offsets, and registers are assumed to be octal, while counting arguments (e.g., specifying a source line number, printing the first 20 lines) and variables referenced by name are assumed to be decimal.

A decimal default can be changed to octal by preceding the number with the escape sequence "&o". An octal default can be changed to decimal by preceding the number with "&d".

Example	Description
x = 8	assign the value 8 to the program variable x. Program variables referenced by name are assumed to be decimal; if octal representation is preferred, type: x = &o10
$q = 77	assign the value of 77 to the q-register. Register values are machine dependent and assumed to be octal; if decimal representation is preferred, type: $q = &d63
/test/&a19	print line 19 of the source segment for test.
&a19,s8	print 8 source lines, beginning at line 19.

Data Requests

Data requests consist of three fields and have the following format:

```
<generalized address> <operator> <operands>
```

The generalized address defines the actual data or code of interest. It is ultimately reduced to segment number and offset by debug before being used. The operator field indicates to debug which function to perform, e.g., print or modify the data referenced by the generalized address. The operands field may or may not be necessary, depending on the operator. When these fields are specified, they are separated by blanks or commas.

As debug decodes a data request, it parses the generalized address and generates a pointer to the data being referenced. This pointer, called the working pointer, is changed whenever the generalized address is changed. It points into either the working segment, its stack frame, or its linkage section. The actual segment depends on the most recent specification in a generalized address. The form for a generalized address is as follows:

```
[/segment name/] [offset] [segment ID] [relative offset]
```

(The brackets are not part of the debug syntax.) The segment name is either a pathname, a reference name, or a segment number, and defines what is called the working segment. The segment ID specifies which of the data bases associated with the working segment is to be used in setting the working pointer. The segment ID can be one of the following:

&s	refers to the stack frame if the working segment is a procedure segment with an active stack frame
&l	refers to an active linkage section (i.e., one with an entry in the linkage offset table (LOT) for the user's ring)
&t	refers to the working segment itself
&a	refers to the source program for the working segment
&p	refers to the parameters of an active invocation of a procedure
&i	refers to an active internal static section (i.e., one with an entry in the internal static offset table (ISOT) for the user's ring)

The offset field is used as an offset within the segment referenced by the working pointer. For the working segment, this offset is relative to the base of the segment. If the working pointer points into an active stack frame, the offset is relative to the base of that frame. If the working pointer points into an active linkage section, the offset is relative to the beginning of that linkage section.

The offset can be either a number or a symbolic name. If a symbolic name is specified, a symbol table must exist for the working segment. See the pl1 command for a description of symbol table creation. If a symbolic name begins with a numeric character, the escape characters &n (for name) must precede the name, to avoid interpreting the name as a number. For example:

```
/test/&n10&t
```

might be used to specify the location associated with FORTRAN line number (i.e., label) 10 in a debug request.

The relative offset field allows the user to relocate the working pointer by a constant value or register. For example, if the user wished to reference the fourth word after the stack variable i he could use:

```
/test/i+4
```

as the generalized address. The relative offset can also assume the value of a register. For example, if the a-register contained the value 4 at the time of a break, then:

```
/test/100&s$a
```

would set the working pointer to offset 104 from the base of the stack frame. It is important to note that a + sign is not present when a register is used. (See "Registers" below.)

The three most common values for the segment ID field are &t, &s, and &l. These designate that the working pointer is to refer to, respectively, the working segment itself, its active stack frame, or its active linkage section. In addition, two other possible values of segment ID allow alternate methods of referring to locations in either the working segment or its stack frame.

A segment ID of &a refers to the ASCII source program for the working segment. Associated with this segment ID is a decimal line number, which must immediately follow the &a. This line number is used to generate a working pointer to the first word of code compiled for that line. A relative offset can follow the line number. Note that the line-number/code-location association can only be determined if a symbol table exists for the working segment. This example:

```
/test_seg/&a219+36
```

would generate a working pointer that points to the thirty-sixth (octal) word in the text after the first word of code generated for line 219 in the source for the segment test_seg. If an offset field is given before &a, the offset is ignored. The offset of the working pointer is generated solely from the line number and the relative offset.

A segment ID of &p refers to the parameters of an active invocation of a procedure. If the current defaults specify an active stack frame, a number following the &p specifies the parameter that is to be addressed. The offset field is ignored, but a relative offset can be specified. This example:

```
/test_seg/&s;&p4+36,a14
```

causes the stack frame for test_seg to be the working segment, and the first 14 characters of the data contained at a location 36 words after the beginning of the fourth parameter are printed in ASCII format.

It is not necessary to specify all four fields of a generalized address. In fact, every field is optional. If a field is not specified, a default value is assumed that is frequently the last value that the field had. For example:

```
/test_seg/line&s+3
```

followed by the generalized address:

```
+4
```

would be acceptable. The latter request would have been equivalent to:

```
/test_seg/line&s+7
```

One time that the defaults assumed are not the values of the previous data request is when a symbolic variable name or label is specified that would cause some field to change. If this is the case, debug might recognize that the segment ID, for example, of the previous data request is not valid and set it appropriately. For example:

```
/test_seg/760&s
```

followed by:

```
regp
```

would cause the defaults to be changed to:

```
/test_seg/140&l
```

if regp is found at a relative offset of 140 (octal) in the linkage section. Note that the segment ID was changed to &l where it remains until explicitly or implicitly changed again.

Defaults are also reset to values different from the previous values when the segment name field is specified in a generalized address. In this case, the following actions are taken:

1. If the segment name begins with &n, take the rest of the characters composing the segment name and go to step 3 below, treating the string as a name. This convention allows the use of debug on segments whose names are composed of numeric characters.

2. If the segment name is really a segment number, this number is used in a search of all active stack frames to see if one exists for this segment. The search is from the highest stack depth (deepest in recursion) to the base of the stack so that if an active stack frame is found, it is the one most recently used. If an active stack frame is found, the generalized address defaults are set as follows:

a.	working segment	the one specified by the given segment number
b.	offset	zero
c.	segment ID	&s, i.e., the working pointer points into the latest stack frame for the working segment
d.	relative offset	zero

   If no active stack frame is found, the defaults are set as above except that the segment ID is &t instead of &s, i.e., the working pointer points into the working segment itself.

3. If the segment name is a reference name known in this ring, the segment number for the segment being referenced is found, and then the defaults are calculated as if this segment number were given directly.

4. If the segment name is a pathname, the specified segment is initiated (it can already have been known) and the returned segment number is used as above.

5. If the segment name is of the form segname$entname, the stack is searched from the highest active frame (as in step 2) for the most recent frame associated with the entry point entname in the segment segname. The working segment becomes segname, and the remaining defaults are set as described in step 2.

The entire set of defaults that apply to a debug data request can be determined at any time by issuing the control request to print defaults. For the format and use of this request, see the description under "Control Requests" below.

Operator Field of Data Requests

After decoding the generalized address and determining the working pointer, debug checks the operator. The following five operators are recognized:

1.	,	print
2.	=	assign
3.	<	set a break
4.	>	alter program control (i.e., "go to")
5.	:=	call a procedure

If a debug request is terminated before an operator is encountered either by a semicolon or a newline character, the default operator used is ",", i.e., print. The one exception is that a blank line is ignored. The first, second, and fifth operators above have operands.

Print Request

For the print request, there are two operands (both optional). The first operand is a single character specifying the output mode desired. The second operand is a number indicating how much output is being requested. For example:

```
/test_seg/142&t,i12
```

requests that 12 (decimal) words starting at 142 (octal) in the text of test_seg be printed in instruction format.

The following output modes are available for print requests (see "Output Modes" below for a full description):

o	octal
h	half-carriage octal
d	decimal
a	ASCII
i	instruction
p	pointer
s	source statement
l	code for line number
n	no output (just change defaults)
e	floating point with exponent
f	floating point
b	bit string
g	graphic

The request:

```
+36,a16
```

requests that 16 (decimal) characters starting at 36 (octal) words after the current working pointer be printed in ASCII format. The output might be:

```
1416   1416  ">user_dir_dir>"
```

The two numbers printed in most output modes should be interpreted as follows:

1. If the data is from a stack frame, the first number is the relative offset from the base of the stack segment and the second number is the relative offset within the stack frame. If the second number is negative, the variable does not exist in the current stack frame and is a parameter or a global variable.
2. If the data is from a linkage section, the first number is the offset within the combined linkage segment and the second number is the offset within the linkage section.
3. For all other segments, both numbers are the same and represent the offset within the segment.

If a mode is not specified for output, the last specified mode is used unless debug realizes another mode is more appropriate (e.g., when a symbol specifies a variable of a different type). If the amount of output is not specified, it is assumed to be one unit, i.e., one word for octal output, one line for source output, one character for ASCII output, etc.

Assign Request

When modifying data or code, the operands (at least one is expected) specify the new values to use. For example:

```
i = 8;  p(1) = 206|10, 206|32
```

would assign the decimal value 8 to i and the values 206|10 and 206|32 to p(1) and p(2), respectively. (It is assumed that both are variables that are defined for the current working segment.) If more than one operand is specified in an assignment request, consecutive words starting at the working pointer are changed. This is illustrated by the assignment to the pointer array p.

There are nine acceptable forms for assignment operands:

1. octal number
2. decimal number
3. character string
4. register value (see "Registers" below)
5. instruction format input
6. floating point number
7. pointer
8. bit string
9. variable

Whether a number is assumed to be octal or decimal on input depends on the target. A variable referenced by name is assumed to be decimal unless overridden by "&o". Assignment to a location by using an offset is assumed octal unless overridden by "&d".

```
 x = 99 (decimal)
+2 = 77 (octal)
```

Character strings being input must be bracketed by quote characters ("). Bit strings being input must be bracketed by quote characters and followed by a b. Floating point numbers must not have exponents.

The word-offset portion of a pointer value being input can optionally be followed by either a decimal bit offset in parentheses, a ring number in square brackets, or both. If both a bit offset and a ring number are specified, the ring number must follow the bit offset, with no intervening blanks. For example:

```
p = 206|25(29); q = 252|104[5]; rp = 211|200(3)[4]
```

The format for instruction input is:

```
(opcode  address,tag)
```

The address can specify a base register or a number. For example:

```
/test/lab2 = (lda pr6|20) (sta pr0|2,*0) (nop 0)
```

Some value must be given for the address field. The zero opcode is specified by the opcode arg.

Input of bit strings and character strings changes only those bits or characters specified, i.e., a full word might not be completely changed.

Several types of input can be interspersed in the same assignment request. For example:

```
/145/13000 = "names" &d16 126
```

When different types of input are specified in one request, the user should be aware that the bit offset of the temporary working pointer might be ignored for certain types of input. In the example above, the ASCII for "name" was placed at 145¦13000 and the ASCII for "s" was placed in the first character position of 145¦13001. The next assignment argument (&d16) fills in 145¦13001 with the decimal 16 and hence overwrites the "s" of the previous argument.

In order to better specify more complicated assignments, a repetition factor is provided. If a single number (decimal) appears in parentheses in an assignment, the next data item is assigned repeatedly (i.e., the specified number of times), updating the working pointer each time. An example of this might be:

```
string = (32)" " "alpha"
```

which results in string being modified so that the first 32 (decimal) characters are blanks, and the 33rd through the 37th would get the string "alpha".

Set Break Request

A breakpoint is a special modification to the code of a program that, when executed, causes control to pass to debug. The user is then free to examine and change the states of variables, set other breaks, continue execution, etc. When setting a break, the working pointer is used directly unless it points into the stack. In that case, the working pointer is temporarily forced to the text. To set a break at the label loop_here in the program parse_words, one would say:

```
/parse_words/loop_here<
```

One could also say:

```
/parse_words/loop_here+23<
```

to set the breakpoint 23 (octal) locations after the first word of code for the statement labelled loop_here in the text segment.

One could also set a break by specifying a line number. For example:

```
/rand/&a26<
```

would set a break at the first word of code generated for line 26 (decimal) of the source program.

The break number printed by debug when setting a breakpoint is used as the name of the break when referring to breaks. After a break is reset, the break number is reused. (Resetting a break restores the code to its previous value.)

Once a break has been set at a given location, another break cannot be set there. To find which breaks are set, the user can use the list breaks control request (see "Control Requests" below).

A program with breakpoints in it must be run from inside debug. See "Control Requests" below for executing Multics commands.

Alter Program Control Request

To alter program control by issuing an explicit transfer, one might say

```
/216/2176>
```

to cause debug to search the stack for an active stack frame for the segment 216 (octal) and set the stack pointer to this frame. It then transfers to 2176 (octal) in the text associated with this stack frame.

If no active stack frame is found, debug prints a message and waits for further requests.

Call a Procedure Request

The user can cause debug to call a specified procedure and return values into specified locations. This is done by specifying := as the operator in a data request. This operator expects one operand that is a procedure name with its associated arguments. There are two slightly different ways to invoke this feature: first, to invoke a procedure as a function call (with the argument n+1 being the returned value); and second, to explicitly call a procedure. When a procedure is invoked as a function reference, the current working pointer is used as the last argument in the argument list and, hence, the procedure returns a value into wherever the working pointer is pointing. For example:

```
/test/fi := sqrt_(2.0)
```

This causes the sqrt_ function to be called with the first argument 2.0 and the return argument of fi; debug converts the 2.0 into a floating point number before the call.

If no fields are present before the := is encountered, debug does not specify a return argument in the call. (The := can be thought of as "call" in a PL/I program.) For example:

```
     := who
```

sets up a call to who$who with no arguments. The call:

```
     := rename ("foo","moo")
```

and:

```
     ..rename foo moo
```

are functionally equivalent. (See Multics command execution under "Control Requests" below.)

The method debug uses in setting up the call is to use ten temporary storage areas, one for each of ten possible arguments. debug converts the arguments appropriately and stores the values in these areas. Each area starts on an even location and consists of eight words. These temporary storage areas can be looked at or altered with standard data requests. They are named %1, ..., %10. For example:

```
     := cpu_time_and_paging_(0,0,0)
     %1,d
     %2,d
     %3,d
```

prints three decimal numbers, all being return values from hcs_$usage_values. The actual call that debug made had three arguments that were all 0. (The first words of the first three storage areas were zeroed out prior to the call.) The above call could also have been made as follows:

```
     %3 := cpu_time_and_paging_(0,0)
```

If this were done, the third argument would not have been zeroed before the call.

Variables can also be used as arguments. For example:

```
sum := sqrt_(n)
```

No conversion would be done by debug if n were fixed and sqrt_ expected a floating argument.

The above mentioned temporaries can be used to do simple mode conversion. For example, to get the floating point representation of 3.7 (in octal) one could say:

```
%1 = 3.7; ,o
```

To find the ASCII value for 137 (octal) one could type:

```
%1 = 137137137137 ; ,a4
```

A reference to one of these storage areas causes the working segment to be changed to the stack segment.

If one of the arguments in a procedure call is the character %, then the temporary storage for that argument is not changed (e.g., overwritten with the usual argument value). Results from some previous work can be passed in that argument position. For example:

```
%2 := sqrt_(2.0)
:= ioa_("^e",%)
```

Registers

The hardware registers at the time of a fault (in particular a break fault) are available to the user for inspection or change. These registers are referenced by preceding the register name immediately by a dollar sign ($). The register can be looked at by merely typing the register name. For example:

```
$a
```

prints the contents of the a-register at the time of the last fault. If the user would like the value in the a-register to be changed, he might type:

```
$a = 146
```

for example. Decimal input is allowed also:

```
$a = &d19
```

The predefined register names used by debug are:

1.	pr0	pointer register 0
2.	pr1	pointer register 1
3.	pr2	pointer register 2
4.	pr3	pointer register 3
5.	pr4	pointer register 4
6	pr5	pointer register 5
7.	pr6	pointer register 6
8.	pr7	pointer register 7
9.	prs	all pointer registers
10.	x0	index register 0
11.	x1	index register 1
12.	x2	index register 2
13.	x3	index register 3
14.	x4	index register 4
15.	x5	index register 5
16.	x6	index register 6
17.	x7	index register 7
18.	a	a-register
19.	q	q-register
20.	aq	the a- and q-registers considered as a single register
21.	exp	exponent register
22.	tr	timer register
23.	ralr	ring alarm register
24.	eaq	the exponent, a- and q-registers in floating point format
25.	regs	all of 10) through 23)
26.	ppr	procedure pointer register
27.	tpr	temporary pointer register
28.	even	even instruction of Store Control Unit (SCU) data
29.	odd	odd instruction of SCU data

30.	ind	the indicator register
31.	scu	all SCU data
32.	all	all machine conditions

The user can change the above registers at will (with the exception of "ind" and "eaq") with the understanding that if he continues execution after the break or transfers directly (via > in a data request), the values of the hardware registers are set to those of the above registers.

The values in the registers are automatically filled in by debug (when it is called or faulted into) with those associated with the last fault found in the stack. The user can override these values with the fill registers and crawlout registers control requests. See "Control Requests" below.

The user can also define his own registers and use them as a small symbolic memory. For example:

```
$sta1 = 600220757100;    $nop = 11003
```

would allow the user to later say:

```
/test/210&t = $sta1 $nop $nop
```

To print out the contents of all user-defined registers, the user can type:

```
$user
```

The setting and displaying of registers follows the syntax of data requests. However, only the register name and a possible new value can appear in a register request. Registers can be specified in a general data request only in the relative offset field and as operands in assignment requests. Register names must be less than or equal to four characters in length. Some examples of the use of registers follow:

```
/test/i =$q
/test/0 = $x0
/test/46$x0,a5
```

Control Requests

Control requests provide the user with useful functions not necessarily related to any specific data. The format for a control request is:

```
.<request name>
```

Control requests and data requests can be freely mixed on a command line if separated by semicolons. However, certain control requests use the entire input line and hence ignore any semicolons found therein. Spaces are not allowed in most control requests.

The following is a list of all control requests and the functions they perform. See "Summary of Data and Control Requests" below for a complete review of all requests.

TRACE STACK

The general form is:

.t<u>i</u>,<u>j</u>

The stack is traced from frame <u>i</u> (counting from 0 at the base of the stack) for <u>j</u> frames, where <u>i</u> and <u>j</u> are decimal integers. If <u>i</u> is less than 0, tracing begins at 0; if <u>i</u> is greater than the last valid frame, then only the last frame is traced. If <u>i</u> is not specified, it is assumed to be 0; if <u>j</u> is not specified, all valid stack frames from <u>i</u> on are traced. The name printed in the stack trace is the primary segment name unless the segment is a PL/I or FORTRAN program in which case it is the entryname invoked for the stack frame (i.e., the label on the entry or procedure statement).

Examples:

```
.t2,3
.t100
```

POP OR PUSH STACK

The general form is:

.+<u>i</u> or .-<u>i</u>

The working segment is changed by moving up or down the stack <u>i</u> frames, where <u>i</u> is a decimal integer. For example, if the working segment's active stack frame is at depth 4 in the stack, then:

```
.+3
```

changes the working segment to the segment whose stack frame is at depth 7 in the stack. The defaults for working pointer, segment ID, and offset are reinitialized to the base of the stack frame, &s, and 0, respectively.

SET STACK

The general form is:

```
.i
```

The working segment is set to that of stack frame *i* (starting at 0), where *i* is a decimal integer. The defaults are set as in pushing or popping the stack.

EXECUTE MULTICS COMMAND

The general form is:

```
..<Multics command line>
```

The rest of the input line after the .. is interpreted as a standard Multics command line and is passed to the standard command processor with any preceding characters blanked out. Any valid Multics command line can be given. When setting breaks, the program being debugged must be called in this manner because debug sets up a condition handler (for break faults) that is active only as long as debug´s stack frame is active.

PRINT DEFAULTS

The general form is:

```
.d   or   .D
```

The output might look like:

```
3 /test_seg/14(0)&t,i  212
```

or:

```
3 />udd>m>foo>test_seg/14(0)&t,i  212
```

The first number (3 above) is the stack frame depth in decimal, unless there is no stack frame for the working segment, in which case the number is -1. The name of the working segment appears between the slashes (test_seg above); if .D is used, the full pathname occurs here. The offset appears next (14 above); the bit offset (in decimal) of the working pointer appears next; the segment ID (&t above) appears next; the operator appears next (, for print); the output mode appears next (i for instruction); finally the segment number of the working segment appears (212 above). To find the name/segment number association for a given segment, the user might type:

```
     /206/,n;.d
```

yielding:

```
     60 /test_caller/0(0)&s,o  206
```

If he knew the name, he could obtain the same output by typing:

```
     /test_caller/,n;.d
```

CONTINUE EXECUTION AFTER A BREAK

The general form is:

.c,*i*

or:

.ct,*i*

or:

.cr,*i*

If *i* is not specified, it is assumed to be 0. If *i* is specified, the next *i* break faults for the current break are skipped. The first instruction executed upon continuation is the instruction on which the break occurred. If a t follows the c, debug continues in temporary break mode (see "Break Requests" below). If an r follows the c, debug resets the mode to normal (not temporary).

Examples:

.c	continue execution
.c,3	continue execution, but skip the next three break faults for the current break
.ct	continue execution in temporary break mode

QUIT

The general form is:

```
.q
```

This request returns from debug to its caller. Note that if debug was entered via a break, then typing .q returns to the last procedure that explicitly called debug.

CHANGE OUTPUT MODE

Requests pertaining to debug´s terminal output begin with ".m".

1. Enter brief output mode:

   ```
   .mb
   ```

   This request places debug in brief output mode, which is somewhat less verbose than its normal output mode. In particular, assignment requests and the resetting of breaks are not acknowledged on the user´s terminal; the column headings are not printed for a stack trace; the printing of register contents is somewhat more compact; some error messages are abbreviated.

2. Enter long output mode:

   ```
   .ml
   ```

   This returns debug to long output mode, which results in fuller and more explicit terminal output. Long mode is the initial default.

SET I/O SWITCH NAMES

These requests allow a user to debug a program that is run with file output because it generates extensive output or a program that is run from within an exec_com after "&attach" because it requires much input. The general form is:

```
.si switch_name
.so switch_name
```

where switch_name identifies the switch_name to use for input (.si) or output (.so). The named switch must be attached by the user before the request is made. If no switch name is given, debug creates one (either debug_input or debug_output).

1. User makes a switch request but does not give a switch name:

```
.si
.so
```

   debug creates a switch named debug_input or debug_output and attaches it to the user_i/o switch. This would be the usual request for debugging programs that require the user_input or user_output switches to be attached to a file unstead of to user_i/o. Debug detaches the debug_input and debug_output switches when the user quits debug.

2. User makes a switch request and gives the switch name:

```
.si input_switch
.so output_switch
```

   The user must attach the switch_name before making the request. This could be used when the user wants to read debug requests from a file. The switches can be restored by typing:

```
.si user_input
.so user_output
```

Examples:

The user has directed the output switch named user_output to a segment, but wants debug diagnostics to be printed on the terminal.

```
debug
.so
```

Since a switch name was not given with the request, debug sets up a new I/O switch named debug_output as a synonym for user_i/o, which is the terminal in this case. When the user quits debug, the switch named debug_output is detached.

The user wants to debug a procedure that uses the user_input switch and has a set of debug requests in another segment named debug_macro. An input switch named macro has been attached to the segment of debug requests.

```
debug
.si macro
```

debug will take requests from the switch named macro and will not detach the switch when the user exits debug. An attempt by debug to read beyond the end of the macro input stream results in an exit from debug.

BREAK REQUESTS

The following control requests are specific to breaks and are recognized by having a "b" immediately following the ".". Reference is made to the default object segment, which is merely that segment that debug is currently working with when performing break requests. The default object segment is generally specified implicitly when a break is set or hit. It can be changed and determined on request. The default object segment used for break requests is not necessarily the same as the segment addressed by the working pointer used in data requests.

Breaks are numbered (named) sequentially starting at 1 but the numbers are unique only for the object segment in which the break resides. A user can have several breaks with the same number defined in different object segments.

There are two types of global requests that can be performed on breaks. The first, or subglobal requests, refer to all breaks within the default object segment. The second, or global requests, refer to all breaks set by the user (as determined from the break segment in the home directory). The subglobal request is specified by omitting the break number in a break request. The global request is specified by a "g" immediately after the "b" of all break requests (see below).

The general form of all break requests is:

.bgx_i_ args

where the "g", the number _i_, and the arguments are optional. The "x" is replaced by the control character for the break request desired. The following break requests are currently defined:

1. Reset a break (or breaks). The forms of the requests are:

.br_i_	to reset break _i_ of the default object segment
.br	to reset all breaks of the default object segment
.bgr	to reset all breaks known to debug

2. List (print information about) a break. The forms of the request are:

.bl_i_	to list break _i_ of the default object segment
.bl	to list all breaks of the default object segment
.bgl	to list all breaks known to debug

3. Execute a debug request at break time. The forms for this request are:

```
.be_i_   <rest of line>
.be      <rest of line>
.bge     <rest of line>
```

Specifying the above request causes <rest of line> to be interpreted as a debug input line whenever the appropriate break(s) is encountered. If <rest of line> is null, the specified breaks have this execute feature reset to normal.

4. Disable a break (or breaks). The forms of this request are:

.bo_i_	disable (turn _o_ff) break _i_ of the default break segment
.bo	disable all breaks in the default break segment
.bgo	disable all breaks known to debug

Disabling a break has the effect of preventing the break from being taken without discarding the information associated with it. A user might disable a break if he wishes not to use it for the moment but thinks he might want to restore it later. A disabled break can be eliminated altogether by the .br request, or reenabled by the .bn request. If the break was already disabled, the request has no effect.

5. Enable a break or breaks. The forms of this request are:

.bn_i_	enable (turn o_n_) break _i_ of the default break segment
.bn	enable all breaks in the default break segment
.bgn	enable all breaks

This request restores a previously disabled break. If the break was not disabled, the request has no effect.

6. Establish a temporary command line to be executed whenever breaks are hit. This request is of the form:

```
.bgt     <rest of line>
```

This causes <rest of line> to be executed as a debug request whenever any break is hit during the current process. The difference between this request and .bge is that when .bge is typed, the associated line remains associated with all breaks until they are reset, or until they are changed by .be requests. It is possible to have a temporary global command without removing request lines associated with individual breaks. If <rest of line> is null, a previously-established temporary command line is disestablished.

7. Break conditionally. The following requests allow the user to change a break into a conditional break, i.e., a break that stops only if a certain condition is met.

```
.bci  arg1  -rel-  arg2
.bc   arg1  -rel-  arg2
```

arg1 and arg2 can be constants or variables; -rel- can be = or ^=. Whenever a specified break is encountered, a test is made to see if the equality exists and breaks according to whether the user specified = or ^= in setting up the conditional break. For example:

```
.bc3   i ^= 0
```

causes break 3 to fault whenever it is encountered and the value of i is nonzero. Also:

```
.bc3   i = j
```

causes break 3 to fault whenever it is encountered and the value of i is the same as the value of j. The comparison is a bit by bit comparison with the number of bits to compare being determined by the size and type of the second argument.

If no arguments are given to a set conditional request, the specified break is set back to a normal break. For example:

```
.bc
```

would cause all breaks of the default object segment to fault normally.

8. Specify the number of times a break should be ignored (skipped). The general form is:

   ```
   .bsi  n
   ```

   This causes the number of skips to be assigned to break i of the default object segment to be n.

9. Print or change the default object segment. The form for this request is:

   ```
   .bd  name
   ```

   where name is the (relative) pathname or segment number of the segment to become the default object segment. If name is not specified, the pathname of the default object segment is printed.

10. List the current segments that have breaks. The form for this request is:

    ```
    .bp
    ```

    This request merely interprets the break segment in the initial working directory.

PRINT ARGUMENTS

The general form is:

.a<u>i</u>,<u>m</u>

Argument <u>i</u> for the current stack frame is printed in the mode specified by <u>m</u>. If <u>i</u> is not specified, all arguments are printed. If <u>m</u> is not specified, debug decides the output mode. Valid values for <u>m</u> are:

1.	o	full word octal
2.	p	pointer
3.	d	decimal
4.	a	ASCII
5.	b	bit string
6.	l	location of argument
7.	e,f	floating point
8.	?	debug decides (the default value for <u>m</u>)

Examples:

```
.a3
ARG 3:  ">user_dir_dir"
.a3,o
ARG 3: 076165163145
```

GET FAULT REGISTERS

The general form is:

.f

For register requests debug uses the machine registers of the last fault found in the stack starting at the frame currently being looked at. (This is the default when debug is entered as a result of a break fault.)

CRAWLOUT REGISTERS

The general form is:

```
.C
```

For register requests debug uses the fault data associated with the last crawlout (abnormal exit from an inner ring).

Program Interrupt Feature

The user can interrupt debug by hitting the quit button at any time, in particular during unwanted output. To return to debug request level (i.e., to where debug waits for a new request), the user should type:

```
program_interrupt
```

which is the standard program interrupt manager. (See the description of the program_interrupt command).

Temporary Break Mode

When debug is in temporary break mode (placed there via a .ct control request), the following actions are taken automatically:

1. When the user continues any break, another (temporary) break is set at the first word of code for the next line of source code after the source statement containing the break being continued. If debug cannot determine the location of the next line of source code, the temporary break is set at the word of object code immediately following the break being continued.

2. A temporary break is restored automatically whenever it is continued. A temporary break must be explicitly reset by the user only when it is not continued.

Since temporary breaks are set sequentially in a program (i.e., at the next statement in the source program), any transfers within a program can either skip a temporary break or cause code to be executed that was stopped earlier with a temporary break. Temporary break mode is designed to be used in programs that are fairly uniform and sequential in their flow of control. A user should list his breaks after using temporary break mode to see if any breaks remain active.

Indirection

It is quite often desirable to reference the data pointed to by the pointer that is pointed to by the working pointer, i.e., to go indirect through the pointer. The user can instruct debug to do this by typing * instead of the segment name, offset, and segment ID in a generalized address. For example:

```
/test/regp
```

might print:

```
1260  110 214¦2360
```

To find what is at 214¦2360, the user need type only (assuming he wanted two octal words):

```
*,o2
```

This causes the working pointer to be set to 214¦2360 and not necessarily into the same segment as before the request.

Implementation of Breakpoints

Breakpoints are implemented by using a special instruction (mme2) that causes a hardware fault whenever it is executed. In effect, debug sets itself up as the handler for this fault and, whenever a break word is executed, debug gains control. When debug is entered via a break, it does the following:

1. fills the registers with those of the break fault;

2. prints the location of the break fault;

3. waits for requests.

When continuing after a break fault, debug changes the control unit information so that when it is restarted, it executes the instruction that used to exist where the break word was placed.

The debug command keeps track of a default object segment. All break requests made are relative to the default object segment. For example, any reference to break 3 really means break 3 of the default object segment. To change (or find out) the value of the default object segment, the .bd request should be used.

Variable Names for PL/I and FORTRAN Programs

If a symbol table was created for a PL/I or FORTRAN program using the table option, then names of labels, scalars, structures, and arrays can be used. The only restrictions are: 1) that the entire structure name must be specified; 2) the only expressions that are allowed for subscripts are of the form:

```
variable ± constant
```

where variable can be an arbitrary reference as above; and 3) all subscripts must appear last. If a variable is based on a particular pointer, that pointer need not be specified. Some examples of valid variable references are:

```
p-> a.b.c(j,3)
a.b
p(3,i+2) -> qp.a.b(x(x(4)+1))->j.a
```

Bit Addressing

When a working pointer is generated to a data item that is based on or is a part of a substructure, a bit offset might be required. This bit offset is indeed kept and used. When making references to data relative to a working pointer with a bit offset, the relocated addresses can still contain a bit offset. For example, if the working pointer has the value:

```
151¦3706(13)
```

then the request:

```
+16,b3
```

sets the working pointer to:

```
151¦3724(13)
```

and prints the three bits at this location.

Output Modes

The following output modes are acceptable to debug:

1. o octal
   The data pointed to by the working pointer is printed in full word octal format, eight words per line.

2. h   half carriage octal
   The data is printed as in o format except that only four words per line are printed.

3. d   decimal
   The data is printed in decimal format, eight words per line.

4. a   ASCII
   The data is interpreted as ASCII and printed as such. No more than 256 characters are printed in response to a single request.

5. i   instruction
   The data is printed in instruction format.

6. p   pointer
   The data is printed in pointer format, i.e., segment number and offset (and bit offset if it is nonzero).

7. s   source statement
   One or more source statement lines are printed starting with the line of source code that generated the code pointed to by the working pointer (assumed to be pointing into the text). For example:

   ```
   /test/loop_here+32,s2
   ```

   prints two lines of source code starting with the line that generated the code, 32 (octal) words after the label loop_here.

   Another example:

   ```
   /test/&a219,s
   ```

   prints line number 219 (decimal) of test.lang where lang is the appropriate language suffix. Note that if there was no code generated for the specified line, debug prints a message, increments the line number, and tries again for up to 10 lines.

8. l   code for line number
   The code associated with the specified line number is printed in instruction format. The line number is determined as in s type output. For example:

   ```
   /test/&a27,l
   ```

   prints the code generated for line 27 (decimal) of test.lang. Note that any number following the l is ignored.

9. n   no output
   No output. This is used to suppress output when changing defaults.

10. e   floating point with exponent

11. f   floating point

12. b bit string
    The data is printed as if it were a bit string. No more than 72 bit positions are printed in response to a single request.

13. g graphic
    The specified number of characters are interpreted as graphic characters (this is assumed to start in typewriter mode).

Summary of Data and Control Requests

DATA REQUESTS

/seg name/	offset	seg ID	rel offset	operator	operands
pathname	number	&t	number	,	operands
ref name	symbol	&s	register	=	input list
seg number		&l		<	function list
&n seg name		&an		>	
seg$entry		&pn		:=	

Segment ID	Operators	Registers	Output Modes
&t text	, print	$a	o octal
		$q	h half-carriage octal
&s stack	= assign	$aq	d decimal
		$eaq	a ASCII
		$x0	
&l linkage	< set break	.	i instruction
		.	
&i internal static		.	p pointer
&an source line	> transfer	$x7	s source statement
		$pr0	l code for line number
&pn parameter	:= call	.	n no output
		.	
		.	e floating point
		$pr7	f floating point
		$exp	b bit string
		$tr	g graphic
		$ralr	
		$ppr	
		$tpr	
		$even	
		$odd	
		$ind	
		$prs	
		$regs	
		$scu	
		$all	

CONTROL REQUESTS

.ti,j	trace stack from frame i for j frames
.+i or .-i	pop or push stack by i frames
.i	set stack to i'th frame
..	Multics command
.d or .D	print default values
.c,i	continue after break fault (ignore next i break fault
.ct,i	continue, in temporary break mode
.cr,i	continue, in normal mode
.q	return from debug to caller
.bri	reset break i
.br	reset the breaks of the default object segment
.bgr	reset all breaks
.bli	list break i
.bl	list the breaks of the default object segment
.bgl	list all breaks
.bei <line>	execution line for break i
.be <line>	execution line for all breaks of the default object segment
.bge <line>	execution line for all breaks
.boi	disable break i
.bo	disable the break of the default object segment
.bgo	disable all breaks
.bni	enable break i
.bn	enable the breaks of the default object segment
.bgn	enable all breaks
.bgt <line>	establish a temporary global command
.bci a1 -rel- a2	make conditional break i
.bc a1 -rel- a2	make conditional all breaks of default object segment
.bsi n	set skips of break i to n
.bd name/no.	set (or print) default object segment
.bp	print names of all segments with breaks
.ai,m	print argument i in mode m (modes: o, p, d, a, b, l, e, f, ?)
.f	use registers from last fault
.C	use crawlout registers
.mb	change to brief output mode
.ml	change to long output mode

Name: decode

The decode command is used to reconstruct an original segment from an enciphered segment according to a key that need not be stored in the system. The original segment has the same length as the enciphered segment. (See the encode command.)

Usage

```
     decode path1 -path2-
```

where:

1. path1 is the pathname of the enciphered segment. The code suffix should not be specified because the command attaches the code suffix to the path1 argument (e.g., if the user types alpha_x.code as the path1 argument, the command attaches the suffix and looks for a segment named alpha_x.code.code).

2. path2 is the pathname of the deciphered segment to be produced. If no path2 argument is given, the command constructs a pathname from the path1 argument (see "Notes" below).

Notes

The decode command requests the key from the terminal only once, and produces path2 from the enciphered segment named path1.code. (For more information on the key, see the encode command.)

If the path2 argument is not given, the command places the deciphered segment in a segment whose name is the path1 argument. The command strips the code suffix from the path1 entryname and uses that as the entryname for path2. For example, if the user types the command line:

```
     decode alpha_x
```

the command looks for an enciphered segment named alpha_x.code and places the deciphered segment produced in a segment named alpha_x.

Name: defer_messages, dm

The defer_messages command prevents messages sent by the send_message command from printing on the user's terminal. Instead, these messages are saved in the user's mailbox. For a description of the mailbox, refer to the accept_messages and mail commands.

Usage

```
defer_messages
```

Notes

The print_messages command prints messages that have been deferred.

The immediate_messages command restores the printing of messages as they are received.

Name: delete, dl

The delete command causes the specified segments and/or multisegment files to be deleted. See also the descriptions of the delete_dir and delete_force commands (for deleting directories and deleting protected segments or multisegment files without being interrogated, respectively).

Usage

```
delete paths
```

where paths are the pathnames of the segments or multisegment files to be deleted.

Notes

In order to delete a segment or multisegment file with the delete command, the entry must have its safety switch off and the user must have modify permission for the directory. If the safety switch is on, the user is interrogated as to whether he wishes to delete the entry. See also the description for the delete_force command to delete without interrogating the user.

If any one of the paths is a link, delete prints a message; it does not delete either the path in question or the link. (See the description of the unlink command.) If any one of the paths is a directory, delete prints a message; it does not delete the directory. (See the description of the delete_dir command.)

The star convention can be used.

Name: delete_acl, da

The delete_acl command removes entries from the access control lists (ACLs) of segments, multisegment files, and directories. For a description of ACLs, see "Access Control" in Section VI of the MPM Reference Guide.

Usage

```
delete_acl -path- -User_ids- -control_args-
```

where:

1.	path	is the pathname of a segment, multisegment file, or directory. If it is -wd, -working_directory, or omitted, the working directory is assumed. If path is omitted, no User_id can be specified. The star convention can be used.
2.	User_ids	are access control names that must be of the form Person_id.Project_id.tag. All ACL entries with matching names are deleted. (For a description of the matching strategy, refer to the set_acl command.) If no User_id is given, the user's Person_id and current Project_id are assumed.
3.	control_args	can be chosen from the following:
	-all, -a	causes the entire ACL to be deleted with the exception of an entry for *.SysDaemon.*.
	-directory, -dr	specifies that only directories are affected. The default is segments, multisegment files, and directories.
	-segment, -sm	specifies that only segments and multisegment files are affected.
	-brief, -bf	suppresses the message "User name not on ACL."

Notes

If the delete_acl command is invoked with no arguments, it deletes the entry for the user's Person_id and current Project_id on the ACL of the working directory.

An ACL entry for *.SysDaemon.* can be deleted only by specifying all three components. The user should be aware that in deleting access to the SysDaemon project he prevents Backup.SysDaemon.* from saving the segment or directory (including the hierarchy inferior to the directory) on tape, Dumper.SysDaemon.* from reloading it, and Retriever.SysDaemon.* from retrieving it.

The user needs modify permission on the containing directory.

Examples

```
     delete_acl news .Faculty. Jones
```

deletes from the ACL of news all entries with Project_id Faculty and the entry for Jones.*.*.

```
     da beta.** ..
```

deletes from the ACL of every segment, multisegment file, and directory (in the working directory) whose entryname has a first component of beta all entries except the one for *.SysDaemon.*.

```
     da beta.** .. -sm
```

deletes from the ACL of only all segments and multisegment files (in the working directory) whose entryname has a first component of beta all entries except the one for *.SysDaemon.*.

Name: delete_dir, dd

The delete_dir command causes the specified directories (and any segments, links, and multisegment files they contain) to be deleted. All inferior directories and their contents are also deleted. See the descriptions of the delete and delete_force commands for an explanation of deleting segments and deleting protected segments, respectively.

Usage

```
     delete_dir paths
```

where paths are the pathnames of the directories to be deleted.

Notes

The user must have modify permission for both the directory and its superior directory. The star convention can be used. Before deleting each specified directory, delete_dir asks the user if he wants to delete that directory. It is deleted only if the user types "yes".

When deleting a nonempty master directory, or a directory containing inferior nonempty master directories, the user must have previously mounted the logical volume(s). If a nonempty master directory for an unmounted volume is encountered, no subtrees of that master directory are deleted, even if they are mounted.

Warning: Protected segments in path*i* or any of its subdirectories are not deleted. Segments whose write bracket is less than the current ring (except for mailboxes and message segments) are also not deleted. Consequently, the subtree is not completely deleted if it contains any such segments. For a discussion of protected segments, see the safety switch attribute in Section II of the MPM Reference Guide. For a discussion of ring brackets, see "Intraprocess Access Control" in Section VI of the MPM Reference Guide.

Name: delete_force, df

The delete_force command causes the specified segments or multisegment files to be deleted, regardless of whether or not the safety switch is on.

Usage

```
     delete_force paths
```

where paths are the pathnames of the segments and/or multisegment files to be deleted.

Notes

In order to delete a segment or multisegment file using the delete command, the safety switch must be off and the user must have modify permission on its containing directory. However, the delete_force command requires only that the user have modify permission on the directory containing an entry in order to delete the entry. Since the user could turn off the safety switch of the entry by virtue of his modify permission to the directory, he has the power to delete the entry. Thus, this command provides a more convenient way of deleting the entry if the safety switch is on.

If path is a link, delete_force prints a message; it does not delete either the path in question or the link. (See the description of the unlink command.) If path is a directory, delete_force prints a message; it does not delete the directory. (See the description of the delete_dir command.)

The star convention can be used.

Name: delete_iacl_dir, did

This command deletes entries from a directory initial access control list (initial ACL) in a specified directory. A directory initial ACL contains the ACL entries to be placed on directories created in the specified directory. For a discussion of initial ACLs, see "Access Control" in Section III of the MPM Reference Guide.

Usage

```
     delete_iacl_dir -path- -User_ids- -control_args-
```

where:

1. path — specifies the pathname of the directory in which the directory initial ACL should be changed. If path is -wd, -working_directory, or omitted, the working directory is assumed. If path is omitted, no User_id can be specified. The star convention can be used.

2. User_ids — are access control names that must be of the form Person_id.Project_id.tag. All entries in the directory initial ACL that match User_id are deleted. (For a description of the matching strategy, refer to the set_acl command.) If no User_id is specified, the user's Person_id and current Project_id are assumed.

3. control_args — can be chosen from the following:

-all, -a	deletes the entire initial ACL with the exception of an entry for *.SysDaemon.*.
-ring _n_, -rg _n_	identifies the ring number whose directory initial ACL should be deleted. If it is present, it must be followed by _n_ (where user's ring $\leq n \leq 7$). It can appear anywhere on the line and affects the whole line. If this argument is not given, then the user's ring is assumed.
-brief, -bf	causes the message "User name not on ACL of path" to be suppressed.

Note

If the delete_iacl_dir command is given without any arguments, the ACL entry for the user's Person_id and current Project_id is deleted from the initial ACL of the working directory.

Examples

```
     delete_iacl_dir news .Faculty Jones..
```

deletes from the directory initial ACL of the news directory all entries ending in .Faculty.* and all entries with Person_id Jones.

```
     delete_iacl_dir -a
```

deletes all entries from the directory initial ACL of the working directory.

```
     delete_iacl_dir store Jones -rg 5
```

deletes the entry for Jones.*.* from the ring 5 directory initial ACL of the store directory.

Name: delete_iacl_seg, dis

This command deletes entries from a segment initial access control list (initial ACL) in a specified directory. A segment initial ACL contains the ACL entries to be placed on segments created in the specified directory. For a discussion of initial ACLs, see "Access Control" in Section III of the MPM Reference Guide.

Usage

```
     delete_iacl_seg -path- -User_ids- -control_args-
```

where:

1. path — specifies the pathname of the directory in which the segment initial ACL should be changed. If path is -wd, -working_directory, or omitted, the working directory is assumed. If path is omitted, no User_id can be specified. The star convention can be used.

2. User_ids — are access control names that must be of the form Person_id.Project_id.tag. All entries in the directory initial ACL that match the given User_ids are deleted. (For a description of the matching strategy, refer to the set_acl command.) If no User_id is specified, the user's Person_id and current Project_id are assumed.

3. control_args — can be chosen from the following:

-all, -a	deletes the entire initial ACL with the exception of an entry for *.SysDaemon.*.
-ring _n_, -rg _n_	identifies the ring number whose segment initial ACL should be deleted. If it is present, it must be followed by _n_ (where user's ring $\leq$ _n_ $\leq$ 7). It can appear anywhere on the line and affects the whole line. If this argument is not given, the user's ring is assumed.
-brief, -bf	causes the message "User name not on ACL of path" to be suppressed.

Note

If the delete_iacl_seg command is given without any arguments, the ACL entry for the user's Person_id and current Project_id is deleted from the segment initial ACL of the working directory.

Examples

```
     delete_iacl_seg news .Multics. Jones
```

deletes from the segment initial ACL of the news directory all entries with Project_id Multics and the entry for Jones.*.*.

```
     delete_iacl_seg -a
```

deletes all entries from the segment initial ACL of the working directory.

```
     delete_iacl_seg store Jones.. -rg 5
```

deletes all entries with Person_id Jones from the ring 5 segment initial ACL of the store directory.

Name: delete_name, dn

The delete_name command deletes specified names from segments, multisegment files, links, or directories that have multiple names. See the descriptions of the add_name and rename commands for adding and changing names, respectively, on storage system entries.

Usage

```
     delete_name paths
```

where paths are the pathnames that are to be deleted.

Notes

In keeping with standard practice, each path can be a relative pathname or an absolute pathname; its final portion (the storage system entryname in question) is deleted from the segment or directory it specifies, provided that doing so does not leave the segment or directory without a name. In this case, the user is interrogated as to whether or not he wishes the segment or directory in question to be deleted.

The user must have modify permission on the containing directory.

The star convention can be used. For a description of the star convention, see "Constructing and Interpreting Names" in Section I.

Example

```
     delete_name alpha >my_dir>beta
```

deletes the name alpha from the list of names for the appropriate entry in the current working directory and also deletes the name beta from the list of names for the appropriate entry in the directory >my_dir. Neither alpha nor beta can be the only name for their respective entries.

Name: delete_search_rules, dsr

The delete_search_rules command allows the user to delete current search rules.

Usage

```
     delete_search_rules paths
```

where paths are usually directory pathnames (relative or absolute) to be deleted from the current search rules. One of the paths may be the keyword working_dir (see "Notes" below).

Note

The keywords home_dir, process_dir, and system_libraries are not accepted by delete_search_rules although they are accepted by the add_search_rules command. Deletion of the keywords initiated_segments and referencing_dir is discouraged and may lead to unpredictable results.

Name: detach_lv, dlv

The detach_lv command detaches one or more logical volumes that have been attached for the user's process by the resource control package (RCP). The detaching of a logical volume involves telling the storage system that this logical volume is no longer attached for this process. The detaching of a logical volume does not affect the attached/detached state of the logical volume for any other process.

Usage

```
     detach_lv volume_names
```

where volume_names specify the volumes to be detached. A user may detach all logical volumes attached for the process by specifying the keyword "all".

Name: display_cobol_run_unit, dcr

The display_cobol_run_unit command displays the current state of a COBOL run unit. The minimal information displayed tells which programs compose the run unit. Optionally, more detailed information can be displayed concerning active files, data location, and other aspects of the run unit. Refer to the run_cobol command for information concerning the run unit and the COBOL runtime environment.

Usage

```
display_cobol_run_unit -control_args-
```

where control_args may be chosen from the following list:

-long, -lg	causes more detailed information about each COBOL program in the run unit to be displayed.
-files	displays information about the current state of the files that have been referenced during the execution of the current run unit.
-all, -a	prints information about all programs in the run unit, including those that have been cancelled.

Note

Refer to the following related commands:

run_cobol, rc
stop_cobol_run, scr
cancel_cobol_program, ccp

This page intentionally left blank.

Name: display_pl1io_error, dpe

The display_pl1io_error command is designed to be invoked after the occurrence of an I/O error signal during a PL/I I/O operation. It describes the most recent file on which a PL/I I/O error was raised and displays diagnostic information associated with that type of error.

Usage

```
display_pl1io_error
```

Example

The command:

```
display_pl1io_error
```

might respond with the following display:

```
Error on file afile
Title: vfile_ afile
Attributes: open input keyed record sequential
Last i/o operation attempted: write from
Attempted "write" operation  conflicts with file "input" attribute.
Attempted "from" operation conflicts with file "input" attribute.
```

Name: do

The do command expands a command line according to the arguments supplied following the command string. The expanded command line is then passed to the current Multics command processor for execution. If abbreviations are being expanded in the user's process, any abbreviations in the expanded command line are expanded. (Since the command line supplied to the do command is enclosed in quotation marks, abbreviations in it are not expanded before do operates on it. See the description of the abbrev command.) Control arguments can be used to set the mode of operations.

The do command can also be invoked as an active function. See "Active Function Usage" below.

Usage

```
     do "command_string" -args-
  or
     do control_args
```

where:

1. command_string is a command line enclosed in quotation marks. Each instance of the parameter designator &*i* (where *i* is a number from 1 to 9) found in command_string is replaced by arg*i*. If any arg*i* is not supplied, each instance of &*i* in command_string is replaced by the null string. Each instance of the parameter designator &f*i* is replaced by the arguments arg*i* through the last argument supplied. Each instance of the string &n is replaced by the number of arguments supplied. The parameters &q*i*, &r*i*, &qf*i*, and &rf*i* are replaced by quoted arguments. (See "Quote-Doubling and Requoting" below.) Each instance of the unique-name designator &! found in command_string is replaced by a 15-character identifier unique to the particular invocation of the do command. Finally, each instance of the ampersand pair && is replaced by a single ampersand. Any other ampersand discovered in command_string causes an error message to be printed and the expansion to be terminated.

2. arg*i* is a character string argument. Any argument supplied but not referenced in a parameter designator is ignored.

3. control_args set the mode of operation of the do command. Control arguments can only be specified if neither a command_string nor args are given. Control_args can be:

-long, -lg	prints the expanded command line on error_output before it is executed or passed back.
-brief, -bf	suppresses printing of the expanded command line. This is the default.
-nogo	does not pass the expanded command line to the command processor. This control argument is ignored if do is invoked as an active function.

-go	passes the expanded command line to the command processor. This is the default. This control argument is ignored if do is invoked as an active function.
-absentee	establishes an on unit for the any_other condition during the execution of the expanded command line. See "Modes" below for additional information about the -absentee control argument.
-interactive	does not catch any signals. This is the default. (See "Modes" below.)

Active Function Usage

The active function:

```
[do "command_string" args]
```

evaluates to the expanded command line, without executing it.

Modes

The do command has three modes, the long/brief mode, the nogo/go mode, and the absentee/interactive mode. These modes are kept in internal static storage and are thus remembered from call to call within a process. The modes are set by invoking the do command with control arguments and are described under "Usage" above.

*

The absentee mode is mainly of use in an absentee environment, in which any invocation of the default any_other on unit terminates the process. In the absentee mode, any signal caught by the do command merely terminates execution of the command line, not the process. A number of conditions, however, are not handled by the do command but are passed on for their standard Multics treatment; they are quit, program_interrupt, command_error, command_query_error, command_question, and record_quota_overflow. (For a description of these conditions see "List of System Conditions and Default Handlers" in Section VII of the MPM Reference Guide.)

Quote-Doubling and Requoting

In addition to the parameter designators &1 ... &9, the do command also recognizes two more sets of parameter designators. They are &q1 ... &q9, to request quote-doubling in the actual argument as it is substituted into the expanded command line, and &r1 ... &r9, to request that the actual argument be requoted as well as have its quotes doubled during substitution.

Quote-doubling can be described as follows. Each parameter designator in the command_string to be expanded is found nested a certain level deep in quotes. If a designator is found to not be within quotes, then its quote-level is zero; if it is found between a single pair of quotes, then its quote-level is one; and so on. If the parameter designator &q*i* is found nested to quote-level L, then, as arg*i* is substituted into the expanded command line each quote character found in arg*i* is replaced by 2**L quote characters during insertion. This permits the quote character to survive the quote-stripping action to which the command processor subsequently subjects the expanded command line. If &q*i* is not located between quotes, or if arg*i* contains no quotes, then the substitutions performed for &q*i* and for &*i* are identical. The string &qf*i* is replaced by a list of the *i*th through last arguments with their quotes doubled.

If the parameter designator &r*i* is specified, the substituted argument arg*i* is placed between an additional level of quotes before having its quotes doubled. More precisely, if the parameter designator &r*i* is found nested to quote-level L, 2**L quotes are inserted into the expanded line, arg*i* is substituted into the expanded line with each of its quotes replaced by 2**(L+1)quotes, and 2**L more quotes are placed following it. If argument arg*i* is not supplied, nothing is placed in the expanded line; this provides a way to distinguish between arguments that are not supplied and arguments that are supplied but are null. If argument arg*i* is present, the expansions of &r*i*, and of &q*i* written between an additional level of quotes, are identical. The string &rf*i* is replaced by a list of the *i*th through last arguments, requoted.

Accessing More than Nine Arguments

In addition to the normal parameter designators in which the argument to be substituted is specified by a single integer, the do command accepts the designators &(d...d), &f(d...d), &r(d...d), and &q(d...d) where d...d denotes a string of decimal digits. An error message is printed and the expansion is terminated if any character other than 0 ... 9 is found between the parentheses.

Examples

The do command is particularly useful when used in conjunction with the abbreviation processor, initialized by the abbrev command. Consider the following abbreviations:

```
ADDPLI  do "fo &1.list;ioa_  ^¦;pli &1;co"
AUTHOR  do "ioa_$nnl &1;status -author &1"
CREATE  do "cd &1;sis &1 re *.Demo rew Jay.*"
LIST    do "fo Jay.list;LISTAB;ws &1 LISTAC;co;dp -dl Jay.list"
LISTAB  do ".l"
LISTAC  "la;ls -dtem -a"
P       do "pl1 &1 -list &2 &3"
P2      do "pl1 &1 -list &f2"
```

The command line:

```
ADDPLI alpha
```

expands to:

```
fo alpha.list;ioa_  ^¦;pli alpha;co
```

The command line:

```
AUTHOR beta
```

prints the author of segment beta.

The command line:

```
CREATE games
```

expands to:

```
cd games;sis games re *.Demo rew Jay.*
```

This shows an easy method of automatically setting initial access on the segments that will be cataloged in a newly created directory.

The command line:

```
LIST >udd>Demo>Jay
```

expands to:

```
fo Jay.list;LISTAB;ws >udd>Demo>Jay LISTAC;co;dp -dl Jay.list
```

that is expanded by abbrev to:

```
fo Jay.list;do ".l";ws >udd>Demo>Jay "la;ls -dtem -a";co;dp -dl Jay.list
```

This shows how do can be used at several levels and how abbreviations can be used within abbreviations.

The command line:

```
P alpha
```

generates the expansion:

```
pl1 alpha -list
```

while the command line:

```
P alpha -table
```

expands to:

```
pl1 alpha -list -table
```

This shows how references to unsupplied arguments get deleted.

The abbreviation P2 is equivalent to P for three or fewer arguments. The command line:

```
P2 alpha -table -sv3 -optimize
```

executes the pl1 command with the -list, -table, -sv3, and -optimize control arguments, whereas:

```
P alpha -table -sv3 -optimize
```

omits the -optimize control argument.

Name: dprint, dp

The dprint (daemon print) command queues specified segments and/or multisegment files for printing on one of the Multics line printers. The output is identified by the requestor's User_id.

*

Usage

```
dprint -control_args- -paths-
```

where:

1. control_args	may be chosen from the following list of control arguments and can appear anywhere in the command line:
-brief, -bf	suppresses the message "j requests signalled, k already queued. (request_type queue)". This control argument cannot be overruled later in the command line. (See the -request_type and -queue control arguments below.)
-copy *n*, -cp *n*	prints *n* copies (*n* ≤ 4) of specified paths. This control argument can be overruled by a subsequent -copy control argument. If path*i* is to be deleted after printing, all *n* copies are printed first. If this control argument is not given, one copy is made.
-queue *n*, -q *n*	prints specified paths in priority queue *n* (*n* ≤ 3). This control argument can be overruled by a subsequent -queue control argument. If this control argument is not given, queue 3 is assumed. (See "Notes" below.)
-delete, -dl	deletes (after printing) specified paths.
-header XX, -he XX	identifies subsequent output by the string XX. If this control argument is not given, the default is the requestor's Person_id. This argument can be overruled by a subsequent -header control argument.
-destination XX, -ds XX	labels subsequent output with the string XX, which is used to determine where to deliver the output. If this control argument is not given, the default is the requestor's Project_id. This argument can be overruled by a subsequent -destination control argument.

-request_type XX, -rqt XX	places specified paths in the queue for requests of the type identified by the string XX (see "Notes" below). If this control argument is not given, the default request type is "printer".
-indent n, -in n	prints specified paths so that the left margin is indented n columns. If this control argument is not given, no indentation occurs.
-line_length n, -ll n	prints specified paths so that lines longer than n characters are continued on the following line, i.e., no line of output extends past column n. If this control argument is not given, a line length of 136 characters is used.
-page_length n, -pl n	prints specified paths so that no more than n lines are on a page. If this control argument is not given, a page length of 60 lines is used.
-no_endpage, -nep	prints specified paths so that the printer skips to the top of a page only when a form-feed character is encountered in the input path. This argument causes the -page_length control argument, if present, to be ignored.
-single, -sg	prints specified paths so that any form-feed or vertical-tab character in any of the paths is printed as a single newline character.
-truncate, -tc	prints specified paths so that any line exceeding the line length is truncated rather than "folded" onto subsequent lines.
-label XX, -lbl XX	uses the specified string as a label at the top and bottom of every page (see "Notes" below).
-top_label XX, -tlbl XX	uses the specified string as a label at the top of every page (see "Notes" below).
-bottom_label XX, -blbl XX	uses the specified string as a label at the bottom of every page (see "Notes" below).
-access_label, -albl	for each pathi specified, uses the access class of that segment as a label at the top and bottom of every page (see "Notes" below).
-no_label, -nlbl	does not place any labels on the printed output.
2. paths	are the pathnames of segments to be queued for printing.

Notes

If the dprint command is invoked without any arguments, the system prints a message giving the status of queue 3.

If control arguments are present, they affect only paths specified after their appearance in the command line. If control arguments are given without a following path_i_ argument, they are ignored for this invocation of the command and a warning message is returned.

The -queue 1 control argument places requests in the top priority queue, -queue 2 places them in the second priority queue, and -queue 3 (or not specifying a queue) places them in the lowest priority queue. All requests in the first queue are processed before any requests in the other queues, and so on. Higher priority queues usually have a higher cost associated with them.

The -brief, -delete, -single, -truncate, and -no_endpage control arguments cannot be reset in a given invocation of the command; e.g., once -delete appears in a line, all subsequently specified paths are deleted after printing.

The -request_type control argument is used to ensure that a request is performed by a member of a particular group of printers, e.g., to distinguish between onsite printers and remote printers at various locations, or between printers being charged to different projects. Only request types of generic type "printer" may be specified. Request types can be listed by the print_request_types command.

The -label, -top_label, -bottom_label, and -access_label control arguments allow the user to place labels on each page of printed output. The default labels are access labels, i.e. the -access_label control argument is assumed. (Note, however, that if the access class of path_i_ is system_low and the access class name defined for system_low is null, then the default access label is blank.) The default access label may be overridden by the -no_label control argument if labels are not wanted or by one of the other label-related control arguments. The top and bottom labels are treated independently. Thus, use of the -top_label control argument alone will leave an access label as the default bottom label. A page label that exceeds 136 characters will be truncated to that length. Only the first line of a page label will be printed, i.e., a new line terminates the page label. Form feeds and vertical tabs are not permitted. The various label control arguments are incompatible with the -no_endpage control argument. If both are specified, -no_endpage overrides.

Paths cannot be printed unless appropriate system processes have sufficient access. The process that runs devices of the specified class (normally IO.SysDaemon) must have read access to all paths to be printed and status permission on the containing directory. Path_i_ cannot be deleted after printing unless its safety switch is off and the system process has at least sm access on the containing directory. Also, path_i_ will not be deleted if it has a date-time-modified value later than the date-time-modified value at the time-of the dprint request.

The dprint command does not accept the star convention; it prints a warning message if a name containing asterisks is encountered and continues processing its other arguments.

Example

The command:

```
dp -he Jones -cp 2 -dl test1 test7 -he Doe -ds BIN-5 text.runout
```

causes two copies of each of the segments named test1 and test7 in the current working directory to be printed with the header "Jones" and then deleted; it also causes two copies of the segment named text.runout in the current working directory to be printed with the header "Doe" and destination "BIN-5", then deleted.

Name: dpunch, dpn

The dpunch (daemon punch) command queues specified segments and/or multisegment files for punching by the Multics card punch. It is similar to the dprint command.

See Section IV, "Input and Output Facilities," in the MPM Reference Guide for information on the input/output system.

Usage

```
dpunch -control_args- -paths-
```

where:

1. control_args	may be chosen from the following list of control arguments and can appear anywhere in the command line after the command:
-brief, -bf	suppresses the message "j requests signalled, k already queued. (request_type queue)". This control argument cannot be overruled later in the command line. (See the -request_type and -queue control arguments below.)
-copy n, -cp n	punches n copies ($n \leq 4$) of all specified paths. This control argument can be overruled by a subsequent -copy control argument. If pathi is to be deleted after punching, all n copies are punched first. If this control argument is not given, one copy is made.
-queue n, -q n	punches specified paths in priority queue n ($n \leq 3$). This control argument can be overruled by a subsequent -queue control argument. If this control argument is not given, queue 3 is assumed. (See "Notes" below.)
-delete, -dl	deletes (after punching) all specified paths.
-header XX, -he XX	identifies subsequent output by the string XX. If this control argument is not given, the default is the requestor's Person_id. This control argument can be overruled by a subsequent -header control argument.
-destination XX, -ds XX	uses the string XX to determine where to deliver the deck. If this control argument is not given, the default is the requestor's Project_id. This control argument can be overruled by a subsequent -destination control argument.

-request_type XX, -rqt XX	places specified paths in the queue for requests of the type identified by the string XX (see "Notes" below). If this control argument is not given, the default request type is "punch".
-mcc	punches the specified paths in the command line using character conversion. This control argument can be overruled by either the -raw or -7punch control arguments.
-raw	punches the specified paths in the command line using no conversion. This control argument can be overruled by either the -mcc or -7punch control arguments.
-7punch, -7p	punches the specified paths in the command line using 7-punch conversion. This is the default conversion mode and need only be specified when a number of segments are being requested by one invocation of dpunch and other modes (-mcc or -raw) have been specified earlier in the command line. For a description of conversion modes, see "Bulk Input/Output" in Section IV of the MPM Reference Guide.

2. paths — are the pathnames of segments and/or multisegment files to be queued for punching.

Notes

If the dpunch command is invoked without any arguments, the system prints a message giving the status of queue 3.

If control arguments are present, they affect only paths specified after their appearance on the command line. If control arguments are given without a following path*i* argument, they are ignored for this invocation of the command and a warning message is returned.

*

The -queue 1 control argument places requests in the top priority queue, -queue 2 places them in the second priority queue, and -queue 3 (or not specifying a queue) places them in the lowest priority queue. All requests in the first queue are processed before any requests in the other queues, and so on. Higher priority queues usually have a higher cost associated with them.

The -delete control argument is the only control argument affecting segments that cannot be reset in a given invocation of the command. Once -delete appears in a line, all subsequent segments are deleted after punching.

The -request_type control argument is used to ensure that a request is performed by a member of a particular group of punches, for example, to distinguish between onsite punches and remote punches at various locations, or between punches being charged to different projects. Only request types of generic type "punch" may be specified. Request types can be listed by the print_request_types command.

A path*i* cannot be punched unless appropriate system processes have sufficient access. The process (normally IO.SysDaemon) that runs devices of the specified class must have read access to all paths to be punched and status permission on the containing directory. A path*i* cannot be deleted after punching unless its safety switch is off and the system process has at least sm permission on the containing directory. Also, path*i* will not be deleted if it has a date-time-modified value later than the date-time-modified value at the time of the dpunch request.

The dpunch command does not accept the star convention; it prints a warning message if a name containing asterisks is encountered and continues processing its other arguments.

It is suggested that the user, before deleting the path*i* that was punched, read the deck back in and compare it with the original (using the compare command) to ensure the absence of errors.

Example

The command:

```
dpunch a b -mcc -he Doe c.pl1 -dl -7p -he "J. Roe" alpha
```

causes segments a and b in the current working directory to be punched using 7-punch conversion (the default conversion mode); segment c.pl1 to be punched using character conversion with "for Doe" added to the heading; and segment alpha to be punched using 7-punch conversion (and then deleted) with "for J. Roe" added to the heading.

This page intentionally left blank.

Name: dump_segment, ds

The dump_segment command prints, in octal format, selected portions of a segment. It prints out either four or eight words per line and can be instructed to print out an edited version of the ASCII or BCD representation.

Usage

```
     dump_segment path -first- -n_words- -control_args-
```

where:

1. path	is the pathname or (octal) segment number of the segment to be dumped. If path is a pathname, but looks like a number, the preceding argument should be -name or -nm.
2. first	is the (octal) offset of the first word to be dumped. If both first and n_words are omitted, the entire segment is dumped.
3. n_words	is the (octal) number of words to be dumped. If first is supplied and n_words is omitted, 1 is assumed.
4. control_args	can be chosen from the following list of control arguments:
-long, -lg	prints eight words on a line. Four is the default. This control argument cannot be used with -character, -bcd, or -short. (Its use with -bcd or -character would result in a line longer than 132 characters.)
-character, -ch	prints the ASCII representation of the words in addition to the octal dump. Characters that cannot be printed are represented as periods.
-bcd	prints the BCD representation of the words in addition to the octal dump. There are no nonprintable BCD characters, so periods can be taken literally. The -bcd control argument cannot be used with the -character control argument.
-short, -sh	compacts lines to fit on a terminal with a short line length. Single spaces are placed between fields, and only the two low-order digits of the address are printed, except when the high-order digits change. This shortens BCD output lines to less than 80 characters.
-name, -nm	indicates that the following argument is a pathname even though it may look like an octal segment number.

-header, -he	prints a header line containing the pathname (or segment number) of the segment being dumped as well as the date-time printed. The default is to print a header only if the entire segment is being dumped, i.e., neither the first nor the n_words arguments is specified.
-no_header, -nhe	suppresses printing of the header line even though the entire segment is being dumped.
-address, -addr	prints the address (relative to the base of the segment) with the data. This is the default.
-no_address, -nad	does not print the address.
-offset _n_, -ofs _n_	prints the offset (relative to _n_ words before the start of data being dumped) along with the data. If _n_ is not given, 0 is assumed.
-no_offset, -nofs	does not print the offset. This is the default.
-block _n_, -bk _n_	dumps words in blocks of _n_ words separated by a blank line. The offset, if being printed, is reset to initial value at the beginning of each block.

This page intentionally left blank.

Name: edm

The edm command invokes a simple Multics context editor. It is used for creating and editing ASCII segments. This command cannot be called recursively. See Section VI of the Multics User's Guide (Order No. AL40) for an introduction to the use of edm.

Usage

```
     edm -path-
```

where path specifies the pathname of the segment to be created or edited. The path argument can be either an absolute or a relative pathname. If path is not given, edm begins in input mode (see "Notes" below), ready to accept whatever is subsequently typed as input. If path is given, but the segment does not yet exist, edm also begins in input mode. If path specifies a segment that already exists, edm begins in edit mode.

Notes

This command operates in response to requests from the user. To issue a request, the user must cause edm to be in edit mode. This mode is entered in two ways: if the segment already exists, it is entered automatically when edm is invoked; if dealing with a new segment (and edm has been in input mode), the mode change character must be issued. The mode change character is the period (.), issued as the only character on a line. The command announces its mode by typing "Edit." or "Input." when the mode is entered. From edit mode, input mode is also entered via the mode change character.

The edm requests are predicated on the assumption that the segment consists of a series of lines to which there is a conceptual pointer that indicates the current line. (The "top" and "bottom" lines of the segment are also meaningful.) Various 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 (and not the full request name) is accepted by the command. Four requests have been considered sufficiently dangerous, or likely to confuse the unwary user, that their names must be specified in full.

If the user issues a quit signal while in edit mode and then invokes the program_interrupt command, the effect of the last request executed on the edited copy is nullified. (See the description of the program_interrupt command in this document.) In addition, any requests not yet executed are lost. If program_interrupt is typed after a quit in comment or input modes, then all input since last leaving edit mode is lost. If the user wishes to keep the input, he must invoke the start command following the quit.

In the examples that follow, the pointer that indicates the current line is represented by an arrow (->).

Requests

The requests are as follows (detailed descriptions follow the list, in the same order):

-	backup
=	print current line number
,	comment mode
.	mode change
b	bottom
c	change
d	delete
E	execute
f	find
i	insert
k	kill
l	locate
merge	insert segment
move	move lines within segment
n	next
p	print
q	quit
qf	quitforce
r	retype
s	substitute
t	top
updelete	delete to pointer
upwrite	write to pointer (upper portion of segment)
v	verbose
w	write

Backup (-)

Format: - n

Purpose: Move pointer backwards (toward the top of the segment) the number of lines specified by the integer n.

Spacing: A space is optional between the request and the integer argument.

Pointer: Set to the nth line specified before the current line.

Default: If n is null, the pointer is moved up only one line.

Example:

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

Request: -2

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

Print Current Line Number (=)

Format: =

Purpose: Print current line number.

Pointer: Unchanged.

Comment Mode (,)

Format: ,

Purpose: Establish a special inputting mode in which the lines, starting with the current one, are successively treated as follows. The line is printed without a carriage return and anything then typed by the user (e.g., comment, newline, etc.) is appended to the line. If the user types the mode change character (".") as his comment, the last line typed is unchanged and edit mode is reentered.

Pointer: Left pointing to the last line printed.

Mode Change (.)

Format: .

Purpose: Allow user to enter edit mode from input mode or vice versa. This request is also used to terminate the comment mode request and return edm to edit mode.

Pointer: Left pointing to the last line input, edited, or printed.

Bottom (b)

Format: b

Purpose: Move pointer to the end of the segment and switch to input mode.

Pointer: Set after the last line in the segment.

Change (c)

Format: c _n_ /string1/string2/

Purpose: Replace every instance of string1 by string2 in the _n_ consecutive lines beginning with the current line, where _n_ must be an integer. If the user is in verbose mode, edm prints each line that is changed (see the v request). If no line is changed, then edm prints "edm: Substitution failed."

Spacing: A space before _n_ and between _n_ and the string1 delimiter is optional.

Pointer: Set to the last line scanned.

Delimiters: Any character not appearing in string1 or string2 can delimit the strings (/ is used as the delimiter in the format line). A delimiter following string2 is optional.

Default: If the integer _n_ is absent, only the current line is treated. If string1 is absent, string2 is inserted at the beginning of the line.

Example:

```
Before:     a:  procedure;
            ->  x = y.
                q = r.
                s = t;
                end a;

Request:    c2/./;/

Response:   x = y;
            q = r;

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

Note: For compatibility with qedx, this request can also be given as s (for substitute).

Delete (d)

Format: d _n_

Purpose: Cause _n_ lines to be deleted where _n_ is an integer. Deletion begins at the current line.

Spacing: A space is optional between d and _n_.

Pointer: Set to "no line" following the line deleted. That is, an i (insert) request or a change to input mode would take effect before the next nondeleted line.

Default: If _n_ is null, only the current line is deleted.

Note: The requests c, d, n, and p count "no line" when issued immediately after a delete request.

Execute (E)

Format:	E commandline
Purpose:	Pass commandline to the command processor for execution as a command line.
Spacing:	A single space following E is not significant.
Pointer:	Unchanged.

Find (f)

Format:	f string
Purpose:	Search segment for a line beginning with string. Search starts at the line following the current line and continues around the entire segment until the string is found or until the pointer returns to the current line. The current line is not searched. If the string is not found, the error message "edm: Search failed." is printed. If the string is found and the user is in verbose mode, the line containing the string is printed.
Spacing:	A single space following f is not significant. All other leading and embedded spaces are used in searching.
Pointer:	Set to the line found, or remains at the current line if the line is not found.
Default:	If the string is null, the string used by the last f or l (locate) request is used.

Insert (i)

Format:	i newtextline
Purpose:	Insert newtextline after the current line.
Spacing:	The first space following i is not significant. All other leading and embedded spaces become part of the text of the newtextline.
Pointer:	Set to the inserted line.

Default:	If newtextline is null, a blank line is inserted.
Note:	Immediately after a t (top) request, an i request causes the newtextline to be inserted at the beginning of the segment.

<u>Kill (k)</u>

Format:	k
Purpose:	To inhibit (kill) responses following a c, f, l, n, or s request.
Pointer:	Unchanged.
Note:	See v (verbose) request for restoring responses.

<u>Locate (l)</u>

Format:	l string
Purpose:	Search segment for a line containing string. Search starts at the line following the current line and continues around the entire segment until the string is found or until the pointer returns to the current line. The current line is not searched. If the string is not found, the error message "edm: Search failed." is printed. If the string is found and the user is in verbose mode, the line containing the string is printed.
Spacing:	A single space following l is not significant. All other leading and embedded spaces are used in searching.
Pointer:	Set to the line found, or remains at the current line if the line is not found.
Default:	If the string is null, the string used by the last l or f (find) request is used.

Example:

Before:

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

Request:

```
1 x =
```

Response:

```
x = y;
```

After:

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

<u>Merge (merge)</u>

Format: merge path

Purpose: Insert the segment specified by the pathname path after the current line. The pathname specified by path can be either an absolute or a relative pathname.

Spacing: A single space following merge is not significant.

Pointer: Set to "no line" following the last line of the inserted segment.

Default: If path is not given, the pathname given in the invocation of edm is used. If a pathname is given neither in this request nor in the invocation of edm, an error message is printed and edm looks for another request.

<u>Move (move)</u>

Format: move <u>m</u> <u>n</u>

Purpose: Insert <u>n</u> lines beginning at line <u>m</u> after the current line and delete them from their original location.

Spacing: A space is optional before <u>m</u>.

Pointer: Set to "no line" following the lines moved. That is, an i request or change to input mode would take effect immediately following the moved text.

Default: If n is null, only the single line m is moved.

Note: The requests c, d, n, and p count "no line" when issued immediately after a move request.

Next (n)

Format: n n

Purpose: Move pointer down the segment n lines. The line so located is printed if the user is in verbose mode.

Spacing: A space is optional between n and the integer n.

Pointer: Set to the nth line specified after the current line.

Default: If the integer n is null, the pointer is moved down only one line.

Note: The printed response to this request can be shut off using the k (kill) request.

Print (p)

Format: p n

Purpose: Print n lines beginning with the current line.

Spacing: A space is optional between p and the integer n.

Pointer: Left pointing to the last line printed.

Default: If n is null, the current line is printed.

Note: A print request in edm can be aborted by issuing a quit signal and typing pi or program_interrupt. This puts edm in a state where it is ready to accept another request. (See the description of the program_interrupt command.)

Quit (q)

Format: q

Purpose: Exit edm and return to the caller, usually command level. If no write request has been made since the last change to the edited text, edm warns the user that the changes will be lost and asks if he still wishes to quit.

Pointer: If the user is queried and answers no, then the pointer is unchanged.

Quitforce (qf)

Format: qf

Purpose: Exit from edm directly without either warning or querying the user.

Retype (r)

Format: r newtextline

Purpose: Replace current line with newtextline.

Spacing: One space between r and newline is not significant. All other leading and embedded spaces become part of the text of the newtextline.

Pointer: Unchanged.

Default: If newtextline is null, a blank line replaces the current line.

Substitute (s)

Note: This request is identical to the c (change) request.

Top (t)

Format:	t
Purpose:	Move pointer to the top of the segment.
Pointer:	At "no line" immediately above the first line of text.
Note:	An i (insert) request immediately following a t request causes insertion of a text line at the beginning of the segment.

Delete to Pointer (updelete)

Format:	updelete
Purpose:	Delete all lines above (but not including) the current line.
Pointer:	Unchanged.

Write to Pointer (upwrite)

Format:	upwrite path
Purpose:	Save all the lines above the current line (but not including the current line) in the segment whose name is specified by path. The lines written out are deleted from the edit buffers and thus are no longer available for editing. They will replace the previous contents of path. The pathname specified by path can be either an absolute or a relative pathname.
Spacing:	A single space following upwrite is not significant.
Pointer:	Unchanged.
Default:	If path is not given, the pathname given in the invocation of edm is used. If a pathname is given neither in this request nor in the invocation of edm, an error message is printed and edm looks for another request.

Verbose (v)

Format:	v
Purpose:	Cause edm to print responses following a c, f, l, n, or s request. This is the default mode.
Pointer:	Unchanged.
Note:	See k (kill) for inhibiting verbose mode.

Write (w)

Format:	w path
Purpose:	Write out (save) the edited copy in the segment specified by path. The pathname specified by path can be either an absolute or a relative pathname.
Spacing:	A space between w and path is not significant.
Pointer:	If the w request is successful, set to "no line" at the end of the segment.
Default:	If path is not given, the pathname given in the invocation of edm is used. If a pathname is given neither in this request nor in the invocation of edm, an error message is printed and edm looks for another request.
Note:	To terminate editing without saving the edited copy, see the qf (quitforce) request.

This page intentionally left blank.

Name: encode

In order to provide additional security for data stored in a Multics segment, the encode command is provided to encipher a segment's contents according to a key that need not be stored in the system. The enciphered segment has the same length as the original segment.

Usage

```
encode path1 -path2-
```

where:

1.	path1	is the pathname of the segment to be enciphered.
2.	path2	is the pathname of the enciphered segment to be produced. If path2 is not provided, it is taken to be the same as path1. This command always appends the code suffix to path2 to produce the name of the enciphered segment.

Notes

The encode command requests an encipherment key (1-11 characters not including space, semicolon, or tab) from the terminal. Printing on the terminal is suppressed (the printer is turned off) while the key is typed. The command then requests that the key be typed again, to guard against the possibility of mistyping the key. If the two keys do not match, the key is requested twice again.

To reconstruct the original segment from the enciphered segment see the decode command.

Name: enter_abs_request, ear

The enter_abs_request command allows a user to request that an absentee process be created for him. An absentee process executes commands from a segment and places the output in another segment. The user can delay the creation of the absentee process until a specified time.

The principal difference between an absentee process and an interactive one is that the I/O switch user_input is attached to an absentee control segment containing commands and control lines, and the I/O switch user_output is attached to an absentee output segment. The absentee control segment has the same syntax as an exec_com segment. (See the description of the exec_com command.)

Usage

```
enter_abs_request path -control_args- -ag -optional_args-
```

where:

1. path	specifies the pathname of the absentee control segment associated with this request. The entryname must have the suffix absin although it can be omitted in the command. The first argument to the command must be path.
2. control_args	are selected from the following list of control arguments and can appear in any position:
-output_file path, -of path	indicates that the user wishes to specify the pathname of the output segment. It must be followed by the pathname of the absentee output segment. (See "Notes" below.)
-restart, -rt	indicates that the computation specified by this request should be started over again from the beginning if interrupted (e.g., by a system crash). The default is not to restart the computation.
-limit n, -li n	indicates that the user wants to place a CPU limit on the time the absentee process uses. It must be followed by a positive decimal integer specifying the limit, in seconds. The default is no user-supplied limit. There is an installation-defined limit to the amount of time that an absentee process can use that overrides the time limit given by the user.
-queue n, -q n	indicates in which priority queue the request is to be placed. It must be followed by a decimal integer specifying the number of the queue. If this option is omitted, the request is placed in the third queue. While the total number of queues is determined by the site, the default queue is 3.

-time dtime, -tm dtime	indicates that the user wishes to delay creation of the absentee process until a specified time. It must be followed by a character string representing this time. The format of the deferred time is any character string acceptable to the convert_date_to_binary_ subroutine (described in Section II of the MPM Subroutines). If the time string contains blanks, it must be enclosed in quotes.
-brief, -bf	indicates that the message "j already requested." is to be suppressed.

3. -arguments, -ag — is an optional control argument that indicates that the absentee control segment requires arguments. If present, it must be followed by at least one argument. All arguments following -ag on the command line are taken as arguments to the absentee control segment. Thus -ag, if present, must be the last control argument to the enter_abs_request command.

4. optional_args — are arguments to the absentee control segment.

Notes

If the pathname of the output segment is not specified, the output of the absentee process is directed to a segment whose pathname is the same as the absentee control segment, except that it has the suffix absout instead of absin. If the pathname of the output segment is specified, the named segment may or may not already exist and it need not have the suffix absout.

The command checks for the existence of the absentee input segment and rejects a request for an absentee process if it is not present.

The effect of specifying the -time option is as if the enter_abs_request command were issued at the deferred time.

See also the descriptions of the commands list_abs_requests and cancel_abs_request for information on displaying and cancelling outstanding absentee requests.

Examples

Suppose that a user wants to request an offline compilation. A control segment could be constructed called absentee_pl1.absin containing:

```
change_wdir current_dir

pl1 x -table -symbols

dprint -delete x.list

logout
```

The command line:

```
enter_abs_request absentee_pl1.absin
```

causes an absentee process to be created (some time in the future) that:

1. sets the working directory to the directory named current_dir, which is inferior to the user's normal home directory
2. compiles a PL/I program named x.pl1 with two control arguments
3. dprints one copy of the listing segment and then deletes it
4. logs out

The output of these tasks would appear in the directory containing absentee_pl1.absin in a segment called absentee_pl1.absout.

Suppose that an absentee control segment, trans.absin, contained the following:

```
change_wdir &1

&2 &3 -map &4

dprint -delete &3.list

&goto &2.b

&label pl1.b

&3

&label fortran.b

logout
```

The command:

```
ear trans -li 300 -rt -ag work pl1 x -table
```

causes a request for a restartable absentee process (having a CPU limit of 300 seconds) to be made in queue 3 that:

1. sets the working directory to the directory named work, which is inferior to the normal home directory
2. compiles a PL/I program x.pl1 in that directory
3. produces a listing segment containing a map
4. issues a dprint request for the listing segment
5. executes the program x just compiled in the absentee process
6. logs out

The command:

```
ear trans -rt -tm "Monday 2300.00" -q 2 -ag comp fortran yz
```

causes a request for a restartable absentee process to be created at the first occurrence of Monday, 11 P.M., and placed in queue 2 that:

1. sets the working directory to the directory named comp, which is inferior to the home directory
2. compiles a FORTRAN program named yz.fortran
3. produces a listing segment
4. issues a dprint request for the listing segment
5. logs out

All of the commands used in the above examples are described in this document under the name of the particular command.

Name: exec_com, ec

The exec_com command is used to execute a series of command lines contained in a segment. This command allows the user to construct command sequences that are invoked frequently without retyping the commands each time. In addition, the segment can contain control statements that permit more flexibility than the simple execution of commands. Facilities exist for:

- substitution of arguments to the command for special strings in the exec_com segment
- control of I/O switches
- generating command lines, control statements, and input lines conditionally
- combining several exec_com sequences into one segment
- altering the flow of control

Usage

```
     exec_com path -optional_args-
```

where:

1.	path	is the pathname of the segment containing the commands to be executed and control statements to be interpreted. The entryname of the segment must have the suffix ec, although the suffix can be omitted in the command invocation.
2.	optional_args	are character strings to be substituted for special strings in the exec_com segment.

Input Segment

The exec_com segment should contain only command lines, input lines, and control statements. Normally it is created using a text editor, such as edm or qedx. The exec_com command can be used in conjunction with the abbrev command to form abbreviations for command sequences that are used frequently.

When the character & appears in the exec_com segment, it is interpreted as a special character. It is used to denote a string used for argument substitution and to signify the start of a control statement.

Argument Substitution

Strings of the form &i in the exec_com segment are interpreted as dummy arguments and are replaced by the corresponding argument to the exec_com command. For instance, optional_arg1 is substituted for the string &1 and optional_arg10 is substituted for &10.

The strings &qi, &ri, &fi, &qfi, and &rfi also indicate argument substitution. The string &qi is replaced by the ith argument to the exec_com command with quotes doubled. The string &ri is replaced by the ith argument, requoted. Refer to the do command for a description of quote doubling and requoting and for examples of the use of &qi, &ri, &fi, &qfi, and &rfi. The string &fi is replaced by a string of the ith through last arguments to exec_com, separated by blanks. Likewise, &qfi is replaced by a string of the ith through last arguments with quotes doubled and &rfi is replaced by a string of the ith through last arguments, requoted.

The string &n is replaced by the number of arguments to the exec_com command. The string &ec_name is replaced by the entryname portion of the exec_com pathname without the ec suffix. The string &0 is replaced by the pathname argument to the exec_com command just as it was typed.

Argument substitution can take place in command lines, input lines or in control statements, since the replacement of arguments is done before the check for a control statement.

Control Statements

Control statements permit more variety and control in the execution of the command sequences. Currently the control statements are: &label, &goto, &attach, &detach, &input_line, &command_line, &ready, &print, &quit, &if, &then, and &else.

Control statements generally must start at the beginning of a line with no leading blanks. Exceptions to this rule are the &then and &else statements, that can appear elsewhere. Also when a control statement is part of a THEN_CLAUSE or an ELSE_CLAUSE, it does not have to start at the beginning of a line.

1. &label and &goto

These statements permit the transfer of control within an exec_com segment.

&label location	identifies the place to which a goto control statement transfers control. location is any string of 32 or fewer characters identifying the label.
&goto location	causes control to be transferred to the place in the exec_com segment specified by the label location. Execution then continues at the line immediately following the label.

2. &attach, &detach, and &input_line

These statements allow the control of the user_input I/O switch.

&attach	causes the user_input I/O switch to be attached to the exec_com segment. This means that if this control statement is executed, all input read by subsequent commands is taken from the segment rather than from the previous source of data to which the user_input I/O switch was attached.
&detach	causes the user_input I/O switch to be reverted to its original value. By default, the user_input I/O switch is left attached to its original source.
&input_line on	causes input lines returned when using the attach feature to be written on the user_output I/O switch. This is the default condition.
&input_line off	causes input lines not to be written out.

3. &command_line, &ready, and &print

These statements allow the control of the user_output I/O switch. They are useful as tools in observing the progress of the exec_com execution and in printing messages.

&command_line on	causes subsequent command lines to be written on the user_output I/O switch before they are executed. This is the default condition.
&command_line off	causes subsequent command lines not to be written out.
&ready on	causes the invocation of the user's ready procedure after the execution of each command line.
&ready off	causes the user's ready procedure not to be invoked. This is the default condition.
&print char_string	causes the character string following &print to be written out on the user_output I/O switch. The character ^ is treated as a special character in a print statement. The following is a list of strings that can appear and the characters that replace them:

string	replacement
^/ or ^n/	newline character
^¦ or ^n¦	form feed (new page)
^- or ^n-	horizontal tab
^^	^

where n expresses the number of special characters to be written out.

No other characters should appear following the ^ character in the print statement.

4. &quit — causes the current invocation of exec_com to return to its caller and not to execute subsequent command lines.

5. &if, &then, and &else

These statements provide the ability to have command lines, input lines, and control statements interpreted conditionally.

The form of these control statements is:

```
&if [ACTIVE_FUNCTION -arg1- ... -argn-]
&then THEN_CLAUSE
&else ELSE_CLAUSE
```

The active function reference in an &if control statement is evaluated. If the value of the active function is the string true, THEN_CLAUSE is executed. If the value is false, ELSE_CLAUSE is executed.

&if [ACTIVE_FUNCTION -arg1- ... -argn-]
This statement must start at the beginning of a line. The active function is any active function (user-provided or system-supplied) that returns as its value a varying character string with the value true or false. The arguments to the active function can themselves be active functions. (Nesting of active functions is permitted.) The active function and its optional arguments, enclosed in brackets, must be on the same line as the &if string.

&then THEN_CLAUSE
This statement must immediately follow the &if statement; it can appear on the same line or on the following line. THEN_CLAUSE is an exec_com statement, and can include a command line, an input line, the null statement and most control statements. The exceptions are &label, &if, &then, and &else. (Nesting of &if statements is not permitted.) THEN_CLAUSE must be on the same line as the &then string.

&else ELSE_CLAUSE
This statement is optional. When it appears, it must immediately follow the &then statement; it can appear on the same line or on the following line. ELSE_CLAUSE is an exec_com statement and can include a command line, an input line, the null statement and most control statements. The exceptions are &label, &if, &then and &else. ELSE_CLAUSE must be on the same line as the &else string.

The active functions described in "Logical Active Functions" in Section II are frequently used in the &if control statement.

Notes

If a line begins with the & character but is not one of the current control statements, the entire line is ignored. This is one way of including comments in the exec_com segment. The user is cautioned to leave a blank immediately following the & to ensure compatibility with control requests to be added to exec_com in the future.

The segment executed by exec_com can contain calls to exec_com. The user must exercise caution when invoking this feature in conjunction with the &attach feature. When exec_com is called from an exec_com using this feature, the input read by commands in the second exec_com is read from the first exec_com segment. Generally if the &attach feature is used, all calls to exec_com should be preceded by &detach control statements.

Several exec_coms can be combined into one segment, by using the dummy argument &ec_name together with the &label and &goto statements. If exec_coms are grouped together, the exec_com segment should have all the names (concatenated with an ec suffix) on its storage system entry that can replace &ec_name.

Examples

Assume that the segment a.ec in the user's working directory contains:

```
pl1 &1 -table -list
dprint -delete &1.list
&quit
```

The command:

```
exec_com a foo
```

would cause the following commands to be executed:

```
pl1 foo -table -list
dprint -delete foo.list
```

Assume that the segment b.ec in the user's working directory has an additional name a.ec and contains:

```
&goto &ec_name
&
&label b
print &1 1 99
&quit
&
&label a
pl1 &1 -table -list
dprint -delete &1.list
&quit
```

The command:

```
exec_com b my_file
```

would cause the following command to be executed:

```
print my_file 1 99
```

The command:

```
exec_com a foo
```

would cause the following commands to be executed:

```
pl1 foo -table -list
dprint -delete foo.list
```

Assume that the segment d.ec in the user's working directory contains the following:

```
&if [exists segment &1.pl1] &then
&else &goto not_found
pl1 &1 -table -list
dprint -delete &1.list
&quit
&label not_found
&print &1.pl1 not found
&quit
```

If the segment foo.pl1 exists, the command:

```
exec_com d foo
```

would cause the following commands to be executed:

```
pl1 foo -table -list
dprint -delete foo.list
```

If the segment foo.pl1 did not exist, the command:

```
exec_com d foo
```

would output the following:

```
foo.pl1 not found
```

Assume that the segment test.ec in the user's working directory contains:

```
&print begin &ec_name exec_com
&command_line off
create &1.pl1
&command_line on
&attach
edm &1.pl1
i &1: proc;
&input_line off
i end &1;
w
q
&detach
&goto &2
&label compile
pl1 &1
&label nocompile
&print end &ec_name &1 &2 exec_com
```

The command:

```
exec_com test x compile
```

produces the following output:

```
begin test exec_com
edm x.pl1
Edit.
i x: proc;

pl1 x
PL/I
end test x compile exec_com
```

Names: file_output, fo
console_output, co

The file_output command allows the user to direct the I/O output switch user_output to a specified file. The console_output command allows the user to direct it back to the terminal.

Usage

```
file_output -path-
console_output
```

where path is an optional file pathname. If is not present, the file_output command directs output to the file, output_file, in the user's working directory. If the specified file does not exist (pathname or output_file), it is created. If it does already exist, subsequent output is appended to the end of the file.

Note

To avoid getting ready messages in the output file, the file_output and console_output commands should appear on the same line of terminal input. (See the second example below.)

Examples

The sequence of commands:

```
file_output my_info
list -a
list -pn >sample_dir -dr
console_output
```

places in the file my_info, in the user's working directory, a listing of all entries in his working directory and a listing of all directories contained in the directory >sample_dir. If they have not been turned off, the ready messages from the file_output command and the two invocations of the list command also appear in my_info. (See the ready_off command.)

The command line:

```
fo my_info; list -a; list -pn >sample_dir -dr; co
```

has the same result as the first example except that no ready messages appear in my_info.

Name: format_cobol_source, fcs

The format_cobol_source command converts pseudo free-form COBOL source programs to the standard fixed-format COBOL source programs processed by the COBOL compiler.

Usage

```
format_cobol_source path1 path2
```

where:

1. path1 is the pathname of the input segment containing pseudo free-form COBOL source code. If path does not have a suffix of cobol, one is assumed. However, the suffix cobol must be the last component of the name of the input segment.

2. path2 is the pathname of the output segment that contains the converted fixed-format COBOL source. The cobol suffix is optional for the path2 argument; however, the output segment will have the cobol suffix. If the specified path2 argument is not found, a segment is created and given the path2 argument plus the cobol suffix as its name. If path2 is the same segment as path1, the converted output does not replace the input and an error message is printed.

Note

The free-form COBOL format is fully described in the Multics COBOL Users' Guide, Order No. AS43. The pseudo free-form COBOL source statements are translated as follows:

```
Pseudo free-form COBOL                     Output
¦column1                         ¦column7  ¦column12

*XXX                             *XXX
a*XXX                             XXX
d*XXX                            d         XXX
da*XXX                           dXXX
/XXX                             /XXX
-XXX                                       -XXX
YYYXXX                                     YYYXXX
```

where X is any character and YYY is any three characters except:

1. *XX
2. a*X
3. d*X
4. da*
5. /XX
6. -XX

This page intentionally left blank.

Name: fortran, ft

The fortran command invokes the FORTRAN compiler to translate a segment containing the text of a FORTRAN source program into a Multics object segment. A listing segment is optionally produced. These results are placed in the user's working directory. This command cannot be called recursively. For information on FORTRAN, refer to the Multics FORTRAN manual, Order No. AJ28.

Usage

```
     fortran path -control_args-
```

where:

1. path	is the pathname of a FORTRAN source segment that is to be translated by the FORTRAN compiler. If path does not have a suffix of fortran, then one is assumed. However, the suffix fortran must be the last component of the name of the source segment.
2. control_args	can be chosen from the following list of control arguments:
-source, -sc	produces a line-numbered, printable ASCII listing of the source program.
-symbols, -sb	produces a source program listing (like the -source control argument), followed by a list of all the names declared in the program with their attributes.
-map	produces a source program listing with symbols (like the -symbols control argument), followed by a map of the object code generated by the compilation. The -map control argument produces sufficient information to allow the user to debug most problems online.
-list, -ls	produces a source program listing with symbols (like the -symbols control argument), followed by an assembly-like listing of the compiled program. Use of the -list control argument significantly increases compilation time and should be avoided whenever possible by using the -map control argument.
-brief, -bf	causes error messages written into the I/O switch error_output to contain only an error number, statement identification, and, when appropriate, the identifier or constant in error. In the normal, nonbrief mode, an explanatory message of one or more sentences is also written, followed, in most cases, by the text of the erroneous statement.

-severityi, -svi	causes error messages whose severity is less than i (where i is 1, 2, 3, or 4) to not be written into the error_output switch although all errors are written into the listing. The default value for i is 1. See the description of severity levels under "Error Diagnostics" below.
-check, -ck	is used for syntactic and semantic checking of a FORTRAN program. Only the first phase of the compiler is executed. Code generation is skipped as is the manipulation of the working segments used by the code generator.
-optimize, -ot	invokes an extra compiler phase just before code generation to perform certain optimizations such as the removal of common subexpressions, which reduce the size and execution time of the object segment. Use of this control argument adds 10% to 20% to the compilation time.
-table, -tb	generates a full symbol table for use by symbolic debuggers; the symbol table is part of the symbol section of the object program and consists of two parts: a statement table that gives the correspondence between source line numbers and object locations and an identifier table that contains information about every identifier actually referenced by the source program. This control argument usually causes the object segment to become significantly longer.
-brief_table, -bftb	generates a partial symbol table consisting only of statement labels for use by symbolic debuggers. The table appears in the symbol section of the object segment produced for the compilation. This control argument does not significantly increase the size of the object program.
-subscriptrange, -subrg	causes extra code to be produced for all subscripted array references, all computed goto statements, and all alternate return statements to check for subscript values exceeding the declared dimension bounds. Such an error causes the subscriptrange condition to be signalled.
-profile, -pf	generates additional code to meter the execution of individual statements. Each statement in the object program contains an additional instruction to increment an internal counter associated with that statement. After a program has been executed, the profile command can be used to print the execution counts.

The following two control arguments are available for users who wish to maintain their FORTRAN source segments in ANSI card format. For a definition of card-image format, refer to the Multics FORTRAN manual, Order No. AJ28.

-card	specifies that the source segment is in card-image format. No converted segment is produced.
-convert	specifies that the source segment is in card-image format. The compiler generates a segment, path.converted, in the user's working directory, in Multics FORTRAN format. All alphabetic characters that are not part of character strings are mapped into their lowercase equivalent. The listing segment displays the segment as it appears after this mapping. Error messages refer to only the modified segment.

The segment produced by -convert differs from the source segment as follows:

1. Alphabetic characters not in character strings are mapped to lowercase.

2. Columns 73-80 of the card image are deleted. Trailing blanks that are not part of a character string are eliminated.

3. Columns 1-6 have three different forms and are converted accordingly:

   a. Column 1 contains a "C", "c", or "*". The card image is a comment line. A lowercase c is placed in column 1 and columns 2-6 are preserved as is.

   b. Column 6 contains a character other than zero or blank. The card image is a continuation line. Columns 1-6 are replaced by a tab and the preceding line is marked as being continued.

   c. For all other cards, column 6 is ignored and eliminated. The card image is an initial line. Columns 1-5 can contain blanks or numerals. The numerals present are concatenated to form a single string and are followed by a tab.

The following control arguments are available, but are probably not of interest to every user.

-time, -tm	prints a table after compilation giving the time (in seconds), the number of page faults, and the amount of free storage used by each of the phases of the compiler. This information is also available from the command fortran$times invoked immediately after a compilation.
-debug, -db	leaves the list-structured internal representation of the source programs intact after a compilation. This control argument is used for debugging the compiler. The command fortran$epilogue can be used to discard the list structure.

Further information on the above control arguments is contained below under "Error Diagnostics" and "Listing."

Notes

The only result of invoking the fortran command without control arguments is to generate an object segment.

A successful compilation produces an object segment and leaves it in the user's working directory. If an entry with that name existed previously in the directory, its access control list (ACL) is saved and given to the new copy of the object segment. Otherwise, the user is given re access to the segment with ring brackets v,v,v where v is the validation level of the process that is active when the object segment is created.

If the user specifies the control arguments -source, -symbols, -map, or -list, the fortran command creates a listing segment in the working directory and gives it a name consisting of the entryname portion of the source segment with the suffix list rather than fortran (e.g., a source segment named test.fortran would have a listing segment named test.list). The ACL is as described for the object segment except that the user is given rw access to it when newly created. Previous copies of the object segment and the listing segment are replaced by the new segments created by the compilation.

Because of the Multics standard that restricts the length of segment names, a FORTRAN source segment name cannot be longer than 24 characters.

Error Diagnostics

The FORTRAN compiler can diagnose and issue messages for about 200 different errors. These messages are graded in severity as follows:

1 Warning only. Compilation continues without ill effect.

2 Correctable error. The compiler remedies the situation and continues, probably without ill effect. For example, a missing end line can be and is corrected by simulating the appending of an end line to the source to complete the program. This does not guarantee the correct results, however.

3 An uncorrectable but recoverable error. That is, the program is definitely in error and cannot be corrected but the compiler can and does continue executing up to the point just before code is generated. Thus, any further errors are diagnosed. If the error is detected during code generation, code generation is completed although the code generated is not correct.

4 An unrecoverable error. The compiler cannot continue beyond this error. The message is printed and then control is returned to the fortran command unwinding the compiler. The command writes an abort message into the error_output I/O switch and returns to its caller.

Error messages are written into the error_output I/O switch as they occur. Thus, a user at his terminal can quit his compilation process immediately when he sees something is amiss. As indicated above, the user can set the severity level so that he is not bothered by minor error messages. He can also specify the -brief control argument so that the message is shorter. An example of an error message in its long form is:

```
WARNING 156 IN STATEMENT 1 ON LINE 5
Do loop control variable "j" has been modified within the
range of the do loop ending at this statement.
SOURCE:  5    continue
```

If the -brief control argument is specified, the user sees instead:

```
WARNING 156 IN STATEMENT 1 ON LINE 5
"j"
SOURCE:  5    continue
```

If the user had set his severity level to 2, he would receive no message at all.

Once a given error message is printed on the user's terminal in the long form, all further instances of that error result in a message printed in the brief mode.

If a listing is being produced, the error messages are also written into the listing segment. They appear, sorted by line number, after the listing of the source program. No more than 100 messages are printed in the listing.

Listing

The listing created by the fortran command is a line-numbered image of the source segment. This is followed by a table of all of the names declared within the program. The names are categorized by declaration type as follows:

1. declared by declarative statement (e.g., mode, dimension, etc.)
2. not declared (e.g., labels, format specifications, etc.)

Within these categories, the symbols are sorted alphabetically and then listed with their location, storage class, mode, attributes, and references. Next is a table of the program´s storage requirements. Then comes a listing of external operators used, external entries called, and common blocks referenced by the program. The symbol listing is followed by the error messages, if any.

The object code map follows the list of error messages. This table gives the starting location in the text segment of the instructions generated for statements starting on a given line. The table is sorted by ascending storage locations.

Finally, the listing contains the assembly-like listing of the object segment produced. The executable instructions are grouped under an identifying header that contains the source statement that produced the instruction. Operation code, base-register, and modifier mnemonics are printed beside the octal instruction. If the address field of the instruction uses the IC (self-relative) modifier, the absolute text location corresponding to the relative address is printed on the remarks field of the line. If the reference is to a constant, the octal value of the first word of the constant is also printed. If the reference is to a variable, the name of the variable is printed in the remarks field of the line.

Name: fortran_abs, fa

This command submits an absentee request to perform FORTRAN compilations. The absentee process for which fortran_abs submits a request compiles the segments named, appends the output of print_link_info for each segment to the segment path.list if it exists, and dprints and deletes path.list. If the -output_file control argument is not specified, an output segment, path.absout, is created in the user's working directory (if more than one path is specified, only the first is used). If none of the segments to be compiled can be found, no absentee request is submitted. The print_link_info command is described in the MPM Subsystem Writers' Guide.

Usage

```
fortran_abs paths -ft_args- -dp_args- -abs_control_args-
```

where:

1.	paths	are the pathnames of segments to be compiled.
2.	ft_args	may be one or more control arguments accepted by the fortran command.
3.	dp_args	may be one or more control arguments (except -delete) accepted by the dprint command.
4.	abs_control_args	may be one or more of the following control arguments:
	-queue n, -q n	specifies in which priority queue the request is to be placed (n $\leq$ 3). The default queue is 3; path.list is also dprinted in queue n.
	-hold	specifies that fortran_abs should not dprint or delete path.list.
	-output_file path, -of path	specifies that absentee output is to go to the segment whose pathname is path.

*

Notes

Control arguments and segment pathnames can be mixed freely and can appear anywhere on the command line after the command. All control arguments apply to all segment pathnames. If an unrecognizable control argument is given, the absentee request is not submitted.

Unpredictable results can occur if two absentee requests are submitted that could simultaneously attempt to compile the same segment or write into the same absout segment.

When doing several compilations, it is more efficient to give several segment names in one command rather than several commands. With one command, only one process is set up. Thus the dynamic intersegment links that need to be snapped when setting up a process and when invoking the compiler need be snapped only once.

Name: fs_chname

The fs_chname command is an interface to the storage system subroutine hcs_$chname_file (described in the MPM Subroutines). It causes an entryname of a specified segment to be replaced, deleted, or added. This command interprets none of the special command system symbols (e.g., *, >) and thus allows the user to bypass the star convention or to manipulate strangely named segments. For segments with ordinary names, the rename, add_name, and delete_name commands perform the same function.

Usage

```
fs_chname dir_name entryname oldname newname
```

where:

1.	dir_name	is the directory name portion of the pathname of the segment in question.
2.	entryname	is the entryname portion of the pathname of the segment in question.
3.	oldname	is an old entryname to be deleted. See "Notes" below.
4.	newname	is a new entryname to be added. See "Notes" below.

Notes

When both an old entryname and a new entryname appear in the command line, the new entryname replaces the old entryname. This is equivalent to using the rename command.

If the old entryname is a null character string (""), then the new entryname is added to the segment. This is equivalent to using the add_name command.

If the new entryname is a null character string (""), then the old entryname is deleted from the segment. This is equivalent to using the delete_name command.

The user must have modify access on the directory containing the entry in order to make any name changes.

Examples

```
     fs_chname >my_dir alpha>beta alpha>beta alpha_beta
```

replaces the incorrect entryname alpha>beta (that the rename command would interpret as designating the segment beta in the directory alpha) with a more appropriate name.

```
     fs_chname >my_dir story.equal "" story.=
```

adds the entryname story.= to the specified segment. The add_name command could not perform this operation because it would interpret the second component of story.= as use of the equals convention, and would attempt to add the entryname story.equal to the segment. See "Constructing and Interpreting Names" in Section I for a discussion of the equals convention.

Name: gcos, gc

The gcos command invokes the GCOS environment simulator to run a single GCOS job, immediately, in the user's process.

Related facilities include the GCOS daemon, which provides batch processing for GCOS jobs under Multics, and the gcos_sysprint, gcos_syspunch, and gcos_card_utility commands, which may be used to manipulate GCOS format files that reside in the Multics storage system. The commands are described in this document. More information on the Simulator and the GCOS daemon may be found in the Multics GCOS Environment Simulator manual, Order No. AN05.

Usage

```
gcos job_deck_path -control_args-
```

where:

1. job_deck_path	is the pathname of a file (segment or multisegment file) containing a GCOS job deck. The file may contain ASCII lines (as produced by one of the Multics editors) representing card images; or it may be a GCOS standard system format file, containing BCD and binary card images. It is assumed to contain ASCII lines, unless the GCOS format is specified. One way of specifying GCOS format is to have the name of the file end with the suffix gcos. The other way is the -gcos control argument, described below.
2. control_args	may be chosen from the following control arguments. They may appear in any order and may precede or follow the job_deck_path.

Control arguments specifying the input supplied to the simulator:

-gcos, -gc	the input file is in GCOS standard system format, containing BCD and binary card images. Such a file could have been produced as output of a previous GCOS job, or by the gcos_card_utility command.
-ascii, -aci	the input file contains ASCII lines, as produced by one of the Multics editors. This is the default, but this argument may be used if the name of an ASCII file ends with the suffix gcos (to avoid the necessity of renaming the file).
-no_canonicalize, -nocan, -no	may be used to save processing time, when an ASCII input file contains no tab characters, and the fields on all the card images are aligned in the columns required by GCOS. Normally, an ASCII input file may contain tabs separating the fields on each line. The process of replacing these tabs by the

	appropriate number of blanks to align the fields in the columns required by GCOS is known as canonicalization. Logical tab stops are known for GCOS $ control cards and for all the languages supported by the simulator. See the GCOS Environment Simulator manual for more information on canonicalization.
-truncate, -tc	if any ASCII input file (the job deck file, or any $ SELECTed file) contains lines longer than 80 characters (after canonicalization), the extra characters are assumed to be part of comments, and discarded without warnings. If this argument is not given, the first line longer than 80 characters causes the job to be rejected.

Control arguments specifying the disposition of output from the simulator:

-list, -ls	convert print files (both SYSOUT and simulated printer) from BCD to ASCII and delete the BCD copy. (This conversion is performed by a call to the gcos_sysprint command for each file.)
-lower_case, -lc	translate alphabetic BCD characters in print files to lowercase ASCII. The default is uppercase.
-dprint, -dp	queue the converted print files for printing followed by deletion, by the Multics I/O daemon. (The -list argument is implied and need not be given.)
-dprint_options "options", -dpo "options"	queue the converted print files for printing by the I/O daemon, but use the dprint control arguments supplied in the options string instead of the default of -dl. The options must be enclosed in quotation marks if they contain blanks or other delimiter characters recognized by the command processor. The dprint command is called via cu_$cp so that a user-defined abbreviation for dprint (that supplies default heading and destination arguments, for example) would be used in this call. (The -list and -dprint arguments are implied and need not be given.)
-raw	convert punch files (both sysout and simulated card punch) from BCD (or binary) to an internal format suitable for punching by the Multics I/O daemon in raw mode (960 bits per card image) and delete the BCD copy. (This conversion is performed by a call to the gcos_syspunch command for each file.)
-dpunch, -dpn	queue the converted punch files for punching by the I/O daemon in raw mode, followed by deletion. (The -raw argument is implied and need not be given.)

-dpunch_options "options", -dpno "options"	queue the converted punch files for punching by the I/O daemon, but use the dpunch control arguments supplied in the options string. The -raw argument is always used for dpunch, since the converted punch files are not suitable for punching in any other mode. The explanations under -dprint_options above, regarding quotation marks and abbreviations, apply to this argument as well. (The -raw and -dpunch arguments are implied and need not be given.)
-hold, -hd	do not perform the default conversion and daemon output of print and punch files. The default is: -dpo -dl -dpno "-dl -raw" Since the default for each file type (print or punch) is overridden when any of the above arguments is specified for the given file type, the -hold argument is only required when one of the file types is to be left in GCOS standard system format, with no conversion or daemon output being performed.

Control arguments governing the creation of files by the simulator:

-temp_dir path, -td path	use the pathname of a directory specified by path for temporary GCOS files. By default, the process directory is used.
-syot_dir path, -sd path	use the pathname of a directory specified by path for the GCOS format copies of print, punch, and sysout files. By default, the working directory is used. (The converted copies of these files are always placed in the working directory.)
-job_id id, -id id	use the job identification specified by id in the names of files created by the simulator for this job. See the <u>GCOS Environment Simulator</u> manual for more information on the simulator´s naming of files. The id may be any character string, of up to 18 characters, to be used in file names, or it may be one of the following control arguments:

-unique	use a Multics unique name as the job id. (A unique name is a 15-character string, generated by the unique_chars_ subroutine, beginning with an exclamation point and guaranteed to be unique within the system.)
-jd_seg, -jd	use the entryname of the job deck segment as the job id. If the entryname ends with gcos, that suffix is removed from the id. This is the default.

Other control arguments:

-brief, -bf	suppress the printing of all terminal output produced by the simulator except for fatal error messages. Output from the slave program is not suppressed.
-long, -lg	request that, in addition to the normal terminal output, certain lines from the execution report, including the begin and end activity lines (containing PSW, etc.) also be printed on the terminal.
-continue, -ctu	continue processing the job when a nonfatal error occurs. Unless the -brief control argument is given, a warning message is printed on the user's terminal. If this argument is not given, the first nonfatal error causes the job to be rejected. Nonfatal errors occur mainly in control card processing, and are described in detail in Section V of the GCOS Environment Simulator manual under "Control Cards."
-userlib	enable the use of GCOS slave software libraries supplied by the user, instead of, or in addition to, the copies of the libraries installed in the system. The use of this argument is described in Section II of the GCOS Environment Simulator manual.
-debug, -db	inform the simulator that: 1) it is being run interactively; 2) by a user who is familiar with the Multics debug command, and other Multics error recovery facilities; and 3) the user wishes to be given the opportunity to use these facilities to determine the cause of, and possibly correct, any error that would otherwise cause the simulation of the job to be aborted.
-no_bar, -nobar, -nb	request that the simulator run the slave programs in Multics mode instead of BAR mode. This is used for debugging the simulator. Use of this argument may produce abnormal slave program behavior.

Note

If no control arguments are given, the defaults are such that the command:

```
gcos path
```

is equivalent to the command:

```
gcos path -aci -dpo -dl -dpno "-dl -raw" -id -jd
```

Name: gcos_card_utility, gcu

The gcos_card_utility command copies GCOS card image files, altering their format, content, and medium, as specified by the user. The command can make a partial copy of an input file, separate the input file into several output files, or combine several input files into one output file.

The acceptable formats are: Multics ASCII (as produced by Multics editors); GCOS standard System format (SSF) (defined in Section III of the Series 60 (Level 66)/6000 File and Record Control manual, Order No. DD07); or raw (one of the formats used by the Multics I/O daemon for card input and output). GCOS SSF or raw format files can contain compressed source decks (COMDK) as described in Section IX under "RDREC" in the File and Record Control manual.

The content of the files may include: one or more complete GCOS jobs (IMCV); one or more source or object decks (in the format produced by the GCOS Source/Object Library Editor); binary card images; or any other card images (partial GCOS jobs) subject to certain restrictions.

The medium may be magnetic tape or a segment or multisegment file in the Multics storage system.

Usage

```
     gcos_card_utility  input_specification  output_specification
```

The two specifications may appear in either order. The first must be completed before the second is begun. They consist of pathnames (or tape numbers) and control arguments specifying format, content, medium, partial copying, and the amount of information that should be printed on the user's terminal.

The arguments that can be used in the specifications are described below. Some are meaningful only for input or for output; others have slightly different meanings for input and output. These restrictions are noted in the argument descriptions.

A group of arguments, preceded by certain control arguments, is defined to be a "list." A list must follow several of the control arguments described below. The format of a list is also described below, under "Input and Output Lists."

The argument descriptions should be read sequentially for best understanding of the operation of the command. The meanings of, and restrictions on, combinations of arguments are mentioned in the descriptions themselves, if they fit into the context. Otherwise, they are mentioned in the "Notes" portion of this command description. In addition, some examples are given at the end of this command description.

Except where otherwise noted, the arguments within the input and output specifications may appear in any order.

Input and Output Specifications

The following control arguments are used to signify the beginning of the input or output specifications:

-input, -in	This argument signifies the beginning of the input specification. It may be omitted if the input specification appears first, the input consists of a single file, and the pathname of that file is the first item of the input specification (i.e., if the pathname of the single input file is the first argument of the command).
-output, -out	This argument signifies the beginning of the output specification. It may be omitted if the output specification is second, the output consists of a single file, the pathname of that file is the first item of the output specification, and the input specification does not end with a list (i.e., if the pathname of the single output file can be recognized as such, thus ending the input specification and beginning the output specification).

Pathname

The following argument is used to specify an input or output file in the Multics storage system:

path

This is the relative or absolute pathname of an input or output file. If the entry portion of the pathname ends with any of the suffixes: ascii, gcos, raw, or comdk, these are recognized, and will have the same meanings as the control arguments: -ascii, -gcos, -raw, or -comdk, respectively.

A conflict between a suffix and a control argument is treated as an error, unless the control argument precedes the pathname, in which case the control argument overrides the suffix. This allows a file whose format does not match its suffix to be processed without having to be renamed first.

For multiple file input (or output), the format of the input (or output) files must be the same. If the format is not specified by a control argument (which must precede the entire list if the pathnames have suffixes), the suffixes determine the format, and they must be consistent.

File Formats

The following control arguments are used to specify the format of the input and output files:

-ascii, -aci
: Input and output:
The file is in Multics ASCII format, as produced by one of the Multics editors. The lines are of variable length and end with the ASCII newline character.

Input:
Canonicalization of the input lines is performed unless the -no_canonicalize control argument is given. The logical tabstops must be determinable from context or be given with the -tabs control argument. (These arguments are described below.) If the file is to be translated to BCD, it may only contain those ASCII characters that have BCD equivalents.

Output:
Any binary input card images are discarded. A warning message is printed for each contiguous group of card images discarded, giving the size of the group. If the input is BCD, an ASCII newline character is appended to each translated BCD card image, resulting in ASCII output lines of 81 characters each. (See the -truncate control argument below.)

-no_canonicalize, -no
: ASCII input only:
This control argument indicates that canonicalization should not be performed. By default, it is performed on all ASCII input files.

ASCII card images may be typed in free format, with tabs separating fields. Canonicalization is the process of replacing the tabs with the correct number of spaces to align the fields in the columns required by GCOS. Logical tabstops are known for dollar cards (columns 8, 16, and 32) and for all of the languages supported by the GCOS environment simulator.

If the card images contain no tabs, processing time can be saved by omitting the canonicalization.

-tabs tabstops
: ASCII input only:
This argument can be used to supply logical tabstops to be used for canonicalization. A list of from 1 to 10 column numbers, in increasing numerical order, separated by spaces, must follow this argument.

Normally, the set of logical tabstops to be used for each group of nondollar cards is determined from the system call card (e.g., $ FORTRAN, $ GMAP) that precedes them. If the input file contains nondollar cards not preceded by a system call card, the tabstops must be supplied with the -tabs control argument. When they are supplied this way, they are used for all nondollar cards in the file instead of the tabstops implied by any system call cards in the file.

-truncate, -tc	ASCII input: This argument indicates that ASCII input lines might be longer than 80 characters (after canonicalization, if it is being performed), and that any such lines are to be truncated to 80 characters without warning. In the absence of this argument, the first occurrence of an ASCII line longer than 80 characters causes the command to be aborted. ASCII Output: Trailing blanks are removed from ASCII output lines, resulting in lines of varying length, 81 characters or shorter.
-gcos, -gc	Input and output: The file is in GCOS standard system format. It contains 320-word blocks (320 or less, for tape). Blocks begin with a block control word (BCW) and contain variable length records, each beginning with a record control word (RCW). The records may contain BCD or binary card images, or ASCII lines (media codes 1, 2, or 6).
-gcos_ascii, -gca	Output only: The output file is written in GCOS standard system format, but the records contain ASCII characters (media code 6) and are of varying length. Trailing new lines are removed, and any unused characters in the last word are filled with ASCII PAD (octal 177) characters.
-raw	Input and output: The file contains card images, in the format produced (and accepted) by the Multics I/O daemon, for card input (and output) in raw mode. Each card image is a 960-bit string, 12 rows by 80 columns, with each 12-bit string representing the punches in one column, a "1"b signifying a punched hole. The actual reading and punching of cards is not done by this command. See the description of the dpunch command in this document, and also "Bulk Input and Output" in Section V of the MPM Reference Guide, for information on card input and output.

The format of the input or output files can be specified explicitly by one of the above control arguments or by a suffix on the pathname. It may also be implied by some other argument or combinations of arguments, such as specifying -tape or specifying -comdk and not -raw, both of which imply -gcos. In the absence of any indication to the contrary, ASCII format is assumed.

Only one of the following control arguments may be used in one I/O specification: -ascii, -gcos, -gcos_ascii, and -raw. The following control arguments may only be used for an ASCII file: -no_canonicalize, -tabs, and -truncate.

-comdk, -cdk	Input: This argument indicates that if any COMDK's are present

in the input file, they are to be decompressed. (This does not require that any COMDK's be present.) If this argument is not given, COMDK cards are treated as binary cards, which are copied unchanged if the output format is GCOS or raw, but discarded if the output format is ASCII.

Output:
Indicates that all cards except binary cards and dollar cards (GCOS control cards, that have a "$" in column 1) are to be compressed before being written in the output file.

This control argument is designed to compress individual source decks, and may produce unusable results if run on complete jobs, since there are some nondollar cards that GCOS slave programs do not expect to be compressed.

Input and output:
Only GCOS and raw format files may contain COMDK's. If raw format is not specified, GCOS format is assumed (and need not be specified explicitly) for a file containing COMDK's.

File Contents

The following control arguments are used to specify the content of the input and output files:

-imcv list

Input only:
This argument indicates that the input file contains several complete GCOS jobs that are to be separated, either to copy each into a separate output file to select a subset of all the jobs for copying.

This argument must be followed by one or more arguments specifying the snumbs of the jobs to be copied. The format of this list of snumbs is described below under "Input and Output Lists."

-library list,
-lib list

Input only:
This argument indicates that the input file contains several source or object decks, each preceded by a $ GMAP, $ 355MAP, or $ OBJECT card with the four-character edit name in columns 73-76 (the format produced by the GCOS Source/Object Library Editor), and that the decks are to be separated, either to copy each into a separate output file or to select a subset of all the decks for copying.

This argument must be followed by one or more arguments specifying the edit names of the source decks to be copied. The format of this list of edit names is described below under "Input and Output Lists."

Only one of these control arguments (-imcv or -library) may be used in an I/O specification.

<u>Tape Files</u>

The following control arguments are used when the medium of the input or output file is tape:

-tape tape_number
-tape7 tape_number
-tape9 tape_number

Input and output:
These arguments are used to specify a GCOS magnetic tape as input or output. For -tape the installation default number of tracks is used. For all three, the installation default density (for the given number of tracks) is used.

One of these -tape control arguments must precede the -retain and -label control arguments, if they are given.

The tape is assumed to be labeled with standard GCOS labels, as described in the <u>File and Record Control</u> manual, Section XI. (See -no_labels under the -label control argument below.) Multifile reels are supported but only as described below. Multireel files are not supported.

The tape number is the tape reel serial number (word four of the label). It may be up to five characters long. It is used in the mount message to the Multics operator and written in the header label, for output.

-attached, -att

Instead of a tape number, the argument -attached (-att) may be used, to indicate that the tape remains attached from a previous use of this command. Multifile reels may be read or written by using this command repeatedly and keeping the tape attached between uses with the -retain argument.

The -attached control argument may also follow (not necessarily immediately) the tape number, so that the number of an attached tape can be supplied for writing in the label.

*

-retain, -ret

This argument specifies that the tape is to remain attached, and positioned at the end of the file read or written, when the command finishes. It is then available to subsequent uses of this command without requiring the operator to mount and dismount it repeatedly. This argument must be given in each use of the command except the last, to keep the tape mounted.

-detach, -det

This argument causes a tape that was retained by a previous use of this command to be detached. It may be used in two ways:

1. If it is the only argument on the command line, any previously retained tapes, either input or output, are detached, and no other processing takes place.

2. If it is used within an input (or output) specification, any previously retained input (or output) tape is detached, allowing a different input (or output) tape to be attached.

-label label, -lbl label	This argument is used to select a file from a multifile input tape and to specify position and file name for an output tape. Use of this argument is optional. The label has three possible formats:

```
n
file_name
n,file_name
```

where n is the position on the tape of the file to be read or written and file_name is the 12-character name to be found (or written) in words 9-10 of the tape label.

On input, if -label is not given, or on output, if n is not given, current tape position is assumed. It is the beginning of the tape, except for tapes that are retained from previous uses of this command. (The -label control argument should not be given for a retained input tape that is positioned correctly, since it causes the tape to be rewound and searched from the beginning for the specified position or file name. An output tape remains positioned if only the file_name is given, but is rewound if the position is given.)

If both n and file_name are given on input, and file_name does not match words 9-10 of the nth label, the user is given the option to quit or continue.

If file_name is not given on output, words 9-10 are set to blanks. If file_name is the string -no_labels or -nl, then label processing is omitted and every record read or written is treated as a data record.

Only GCOS format files can be read from or written onto tape; the -gcos control argument is implied and need not be given.

If a tape file is specified for input (or output), it must be the only input (or output) file specified.

Partial Copying

The following control arguments may be used to specify that a subset of the card images in each input file should be copied:

-first n, -ft n	The number after this argument is the number of the first card image to be copied.
-last n, -lt n	The number after this argument is the number of the last card image to be copied.
-count n, -ct n	The number after this argument is the number of card images to be copied.

Counting is performed on card images sent to the output file, but before they are compressed, if COMDK output was specified. Input card images have already been decompressed, and binary card images discarded for ASCII output, before counting takes place.

If -first is not given, copying begins with the first card image. If neither -last nor -count is given, copying proceeds to the end of the file. The -last and -count control arguments may not both be used in the same specification.

For IMCV or Library content, only the specified subset is searched for the jobs or decks to be copied, and copying stops in the middle of a job or deck if the end of the subset is reached. Also, COMDK's that are not being copied are not decompressed, so COMDK card images, rather than decompressed card images, are counted during the search for the next job or deck to be copied.

Output File Name Duplication

The following control argument may be used to specify the action to be taken when the output file already exists:

-append, -app	Output (nontape) only: This argument specifies that, if an output file already exists, the output is to be appended to it, immediately following its current contents. The format of the existing file must be the same as that of the output to be written. If it is not, the results are unpredictable. If this argument is not given, and the output file already exists, the user is given the choice of aborting the command, or deleting the existing file and continuing.

Input and Output Lists

The following arguments are used to specify a list of Multics files, or of jobs or decks within a single file, to be copied:

-list list, -ls list	Input and output: This argument specifies a list of Multics files to be copied from, or into. The list that follows it has the same format as the lists after -imcv or -gmap, except that -all cannot be included in it.
-all	Input only: This argument may follow -imcv or -library to indicate that the entire contents of the input file are to be copied, each job or deck into a separate output file.

-name, -nm	Output only: This argument may follow -list to indicate that the names of output files should be taken from the snumbs or edit names of jobs or decks from an IMCV or Library input file. The output files are placed in the working directory and have entrynames of the form snumb.suffix or edit_name.suffix, where suffix is ascii, gcos, or raw, depending on the format of the output file.
name1 ... namen	Input and output: The names of input or output files, or input snumbs or input edit names may be given on the command line, after -list, -imcv, or -library.
-file_input path, -file path, -fi path	Input and output: A path is the pathname of a segment containing the file names, or snumbs, or edit names, one name per line. This form may be used (after -list, -imcv, or -library) when the list of names is long and inconvenient to type on the command line.

Terminal Output

The following control arguments are used to specify the amount of information to be printed on the user's terminal:

-brief, -bf	Input and output: This argument indicates that messages warning of nonfatal errors, or giving detailed information about errors, should not be printed on the terminal. The argument may be used in both the input and output specifications, to suppress messages connected with the reading of input or the writing of output.
-long, -lg	Input: The names of input items (snumbs, edit names, tape file names, Multics pathnames) are printed on the terminal as they are read from input. Output: The names of items selected for copying are printed on the terminal as they are written to output. Input and output: If this argument is given for both input and output, all input item names are printed, and those selected for copying are so indicated. This argument is not inconsistent with -brief. The two arguments control the printing of different types of information, and both may be given on the same command line.

-debug, -db	Input or output: This argument may be given once on the command line, in either the input or the output specification. Its meaning is independent of its location. It indicates that whenever an error occurs, the Multics debug command is to be called after the error message is printed, giving the user the opportunity to determine the cause of the error and possibly correct it and cause processing to continue.

Notes

Only one of the "listing" control arguments (-list, -imcv, or -library) may be given in the input specification since only one list of input items can be processed. (The items can be file names, snumbs, or edit names.)

If a list of output file names is given, and the number of input items does not match the number of output file names, copying proceeds until one of the lists is exhausted, and then a warning message is printed on the terminal.

When a list of snumbs or edit names is given, the input file is read sequentially and the list is searched for a match with each snumb or edit name in the input file to determine whether the job or deck should be copied. Thus the order of items in the output file is the same as their order in the input file, and the order of the list is not significant. However, a list of input (or output) file names is read (or written) in the order in which they appear in the list.

Alphabetic BCD input is translated to lowercase ASCII. Arguments containing edit names, tape labels, etc., must be typed in lowercase in order to compare with corresponding BCD names from the input file.

Examples

The following three commands specify the same translation (raw to GCOS):

```
gcu -in infile -raw -out outfile -gcos
gcu -in infile.raw -out outfile.gcos
gcu infile.raw outfile -gcos
```

In the second example, the suffixes of the file names specify the format. (Of course, the filename given in the command line must be identical to the name of the file, including any suffix: infile -raw and infile.raw are not equivalent ways of specifying the same file, since the file names are different.)

Name: gcos_sysprint, gsp

The gcos_sysprint command converts a print file (either SYSOUT or simulated printer) that was produced by the GCOS environment simulator, from BCD to ASCII. This command is called automatically by the simulator to convert any print files produced by the job, unless the -hold option is given on the gcos command line. It can also be called by the user, to convert a file that was not converted automatically.

Usage

```
gcos_sysprint input_path -output_path- -control_args-
```

where:

1. input_path	is the pathname of a print file produced by the simulator.
2. output_path	is the pathname of a file into which the ASCII output lines are to be written. If the file already exists, it is overwritten with no warning. If output_path is omitted, the lines are typed on the user's terminal as they are converted.
3. control_args	may be chosen from the following:
-lower_case, -lc	indicates that alphabetic BCD characters should be translated to lowercase ASCII. By default, they are translated to uppercase.
-temp_dir path, -td path	indicates that the temporary files used for sorting by report code should be placed in the directory indicated by path, rather than in the process directory. This is necessary for large (near 256K) files because of the record quota on the process directory.

Notes

The star and equal conventions are not implemented in this command.

If the first record in the file is the SYSOUT header record written by the simulator, then the records are sorted by report code (within each activity) before being printed, and all records from the execution report are printed first. Otherwise the records are printed in the order in which they appear in the input file, with no sorting or reordering of any kind.

The records are assumed to be GCOS print line images, ending with GCOS printer control characters. These latter are converted to ASCII newline and newpage characters. To convert a BCD card image file to ASCII, with newline characters being added to the end of each line, use the gcos_card_utility command.

Name: gcos_syspunch, gspn

The gcos_syspunch command converts a GCOS standard system format file, containing BCD and binary card images, to a format suitable for punching using the Multics dpunch command with the -raw argument. This command is called automatically by the GCOS environment simulator, to convert any punch files produced by the job, unless the -hold argument is given on the gcos command line. It can also be called by the user, to convert any GCOS card image file for dpunching.

Usage

```
     gcos_syspunch path
```

where path is the pathname of the card image file to be converted.

Notes

The output is written into a segment in the working directory whose entryname consists of the entryname portion of path plus the suffix raw. If this segment already exists, it will be overwritten with no warning.

The star convention is not implemented in this command.

No sorting by report code is performed by this command.

The conversion produces a 960-bit string for each input card image. Each 12-bit string represents one of the 80 card columns, with a 1-bit indicating that a hole should be punched in the corresponding row and column. This conversion is such that, when the output file is punched using the dpunch -raw command, the result is valid GCOS BCD or binary cards containing the same information that was in the records of the input file.

Name: get_com_line, gcl

The get_com_line command prints on the user's terminal the maximum length allowed for an expanded command line. An expanded command line is one obtained after all abbreviations and active functions have been processed. (See the abbrev command and Section II for a description of abbreviations and active functions, respectively.)

Usage

```
     get_com_line
```

Note

The default maximum length of an expanded command line is 132 characters. It can be changed using the set_com_line command. For a discussion of the command language (including the treatment of active functions), see "The Command Language" in Section I of the MPM Reference Guide.

Name: get_quota, gq

The get_quota command returns information about the secondary storage quota and pages used for a specified directory.

Usage

```
     get_quota paths -control_arg-
```

where:

1.	paths	are the names of the directories for which quota information is desired. If one of the paths is -wd or -wdir, the working directory is used. If no paths are given, the working directory is used. The star convention can be used to obtain quota information about several directories.
2.	control_arg	can be either -long or -lg to specify that the long form of output is to be used; this control_arg may appear anywhere on the command line.

Notes

The short form of output (the default case) prints the number of pages of quota assigned to the directory and the number of pages used by the segments in that directory and any inferior directories that are charging against that quota. The output is prepared in tabular format, with a total, when more than one pathname is specified. When only one pathname is specified, a single line of output is printed.

The long form of output gives the quota and pages-used information provided in the short output. In addition, it prints information about the number of immediately inferior directories with nonzero quotas. It also shows the time-page product in units of page-days, along with the date that this number was last updated. Thus, a user can see what secondary storage charges his accounts are accumulating. If the user has inferior directories with nonzero quotas, he has to print this product for all these directories in order to obtain the charge.

Name: help

The help command assists users in obtaining online information about the Multics system. This online information, in the form of documents called info segments, consists of command descriptions and subroutine descriptions, plus miscellaneous information about system status and system changes.

Info segments can be manipulated from command level, by using control arguments, and from within, by using info segment requests. Control arguments and requests are described below under "Usage" and "Requests," respectively.

Format of Info Segments

All info segments have a heading line consisting of a brief title and the date the segment was last modified. For command and subroutine descriptions, the program name, including abbreviations, is used for the title.

All info segments are divided into one or more sections, composed of one or more paragraphs. A paragraph is a block of text preceded by two blank lines. It may contain text that is separated by a single blank line. A paragraph that begins a new section is headed by a section title. Section titles are short -- usually one or two words followed by a colon (:). For commands and subroutines, section titles in their standard order are:

Function:	gives a brief description of what the program does.
Syntax:	gives a sample invocation of the program; for subroutines, a declaration as well as an invocation is given for each entry point (usually in the PL/I language).
Arguments:	gives a brief description of each argument; for subroutines, arguments are described for each entry point.
Control arguments:	gives a brief description of each control argument.
Notes:	gives comments, clarifications, or any special case information.
Examples:	gives sample invocations of the program.

An info segment may contain section titles other than those listed above. A user can issue the help command with the -title control argument or he can issue the title request (both described below) to determine section titles in a particular info segment.

Usage

```
help -name- -control_args-
```

where:

1. name — specifies the info segment that the user wishes to read. The command searches for the name first in the installation-dependent information directory, then in the system information directory. If name does not have the info suffix, one is assumed. The star convention may be used. (See "Notes" below for information about the use of the star convention with the help command.) The name argument and the -pathname control argument (described below) conflict -- only one can be used.

2. control_args — work "forward" in the info segment. They take effect at the header line and scan to the end of the segment; no wraparound feature is employed. The control arguments may be chosen from the following:

   -pathname path, -pn path — specifies the info segment path that the user wishes to read. The command does not search in the installation-dependent or system information directories. If path does not have the info suffix, one is assumed. The star convention may be used. The -pathname control argument and the name argument (described above) conflict -- only one can be used.

   -header, -he — prints only the heading line(s) (consisting of a title, segment modification date, and line count) of the info segment. The -header and -title control arguments conflict -- only one can be used.

   -title — prints only the section titles of the info segment. The -title and -header control arguments conflict -- only one can be used.

   -section SS, -sc SS — searches the info segment for the section title SS and starts printing from that section. If the section title SS is more than one word, it must be enclosed in quotes. Printing continues only to the end of the first paragraph of that section. Then the user is asked if he wants more help. (See "Requests" below for information about the query and possible user responses to it.) If the -section and -search control arguments are given, the -section search is done first.

-search S1 S2...Sn, -sh S1 S2...Sn	searches the info segment for a paragraph that contains the specified character strings. The character strings may be anywhere and in any order but only within one paragraph. Printing starts at the beginning of the paragraph and terminates at the end. Then the user is asked if he wants more help. Everything in the command line after the -search control argument is taken as part of the character string, so the -search control argument must be the last control argument given.

Requests

After printing a paragraph of an info segment, the help command asks the user if he wants more help before it prints the next paragraph. The query is of the form:

```
7 lines titled "XXX" follow:  More help?
```

where XXX is the title of the next section or is the previously encountered section title, if the paragraph to be printed does not begin a new section. The user may respond to the query with any of the requests listed below. These requests work "forward" in the info segment. They take effect at the user's current position within the info segment and scan to the end of the segment; no wraparound feature is employed.

yes	prints the next paragraph of information and then asks the user is he wants more help.
no	exits from the current info segment.
quit	causes the command to terminate.
rest	prints the remaining text of the current info segment.
skip	skips the next paragraph and asks the user if he wants to see the paragraph following it.
title	prints the section titles of all sections remaining in the current info segment.
section -SS-, sc -SS-	searches the info segment for the section title SS and starts printing from that section. Printing continues only to the end of the first paragraph of that section, then the user is asked if he wants more help. If no section title SS is found, the message Section "SS" not found is returned and the query is repeated. If SS is not specified, the first paragraph in the next section is printed.

search -S1 S2...S<u>n</u>-, sh -S1 S2...S<u>n</u>-	searches the info segment for a paragraph that contains the specified character strings. The character strings may be anywhere and in any order but only within one paragraph. Printing starts at the beginning of the paragraph and terminates at the end. Then the user is asked if he wants more help. If no paragraph containing the requested strings is found, the message search failed: search S1 S2 ... S<u>n</u> is returned and the query is repeated. If no strings are specified, the previous set (from a control argument or request), is used. If no strings are specified, and the -search control argument or request has not been issued previously, the command returns an error message and the query is repeated.

<u>Notes</u>

When the star convention is used with the help command:

1. The info segments whose entryname matches the star name are alphabetized within the directory and scanned in that order.

2. The -title and -header control arguments and the title request print the section titles and/or header line(s) for all info segments whose entryname matches the star name.

3. The -section and -search control arguments scan the info segments until the desired item is found. Then the user is questioned as usual. The section and search requests scan only the current info segment.

4. The yes, no, rest, and skip requests print the next selected paragraph. This paragraph may be the first paragraph of the next info segment whose entryname matches the star name or, if -search or -section were specified, the next paragraph meeting the specified criterion. (See the third example below.)

5. If the user issues a quit signal, the program_interrupt command can be used to reenter the help command at the next info segment whose entryname matches the star name. (See the third example below.)

Typing the command "help" or "help help" causes information about the help command to be printed.

For a list of info segments in the installation-dependent information directory, type:

```
list -pn >documentation>iis
```

For a list of info segments in the system information directory, type:

```
list -pn >documentation>info
```

Examples

Three examples follow. In all examples, command lines and requests typed by the user are preceded by an exclamation point (!).

```
! help list
(10 lines follow; 137 lines in segment)
01/21/76: list, ls


Function: list prints information about the entries in a single
directory.  Arguments...


1 line titled "Syntax" follows. More help?   ! yes
Syntax: list -entrynames- -control_args-


4 lines titled "Arguments" follow. More help?   ! title
Arguments
Control arguments
Directory argument
Entry type arguments
     .
     .
     .
Output format arguments
r 1020 0.121 2.340 46
```

In the example above, the user asks to see the info segment list.info. Notice the suffix info is assumed. The help command prints with header information, the first paragraph of the info segment, and the user query. The user responds with the yes request. The command prints the next paragraph and the query to which the user responds with the title request. The command then prints the titles of the remaining sections of the info segment and returns the user to command level.

```
! help list -section Arguments


Arguments: entrynames are the names of entries to be listed.  The star
convention may be used.  If no entrynames are given, all entries in the
directory (of the default types or the types specified by control
arguments) are listed.


6 lines titled "Arguments" follows. More help?   ! quit
r 1009 0.126 1.411 26
```

In the example above, the user asks to see the section titled "Arguments" of the list info segment. The command prints the first paragraph of that section and the query. In this instance, the query repeats the previous section title "Arguments." The user is being asked if he wants to see the second paragraph of that section. He responds with the quit request and is returned to command level.

```
! help **.info -search iox


>doc>iis>Installations.info
(3 lines follow; 402 lines in segment)
02/02/76
The component trouble_report of bound_counsultant_ (IML)
was modified to ask that input lines start with a '*'.  The
routine was also converted to use iox_.


01/30/76
20 lines follow. More help?  ! no


>doc>iis>Installations.old.info
(3 lines follow)
MCR's 270, 360, and 826:  A new absentee DIM and exec_com
were installed converting both to use iox_.  Please type "help
new_ec".


06/11/75
11 lines follow. More help?  ! rest
(186 lines follow.)
MCR 1566: The command status of bound_fscom2_ (SSS) was
modified to remove the error message printed if the
device id returned
QUIT
r 1347 0.856 13.440 151 level 2 14
```

```
! pi


>doc>info>fortran.info
(26 lines follow; 85 lines in segment)
28 January 1974


Type "help fortran.changes" for a list of recent changes to the
     implementation
Type "help fortran.io" for a description of how fortran relates
     to the Multics I/O System "iox_".
     .
     .
     .
Type "help fortran.status" for a list of bugs, restrictions, etc.


Rest of segment has 58 lines. More help?  ! search
search failed: search


Rest of segment has 58 lines. More help?  ! quit
r 1359 0.609 14.655 177
```

In the example above, the user asks to see the string "iox" in all the info segments whose entryname matches the star name. The help command prints the header information, the first paragraph that contains the string "iox", and the user query. The user responds with the no request. The command then exits that info segment and prints header information, the first paragraph that contains the string "iox", and the user query for the next info segment whose entryname matches the star name. The user responds with the rest request and then issues a quit signal after four lines are printed. The program_interrupt (pi) command is used to reenter the next info segment whose name matches the star name. Help prints the header information, the first paragraph that contains the string "iox", and the user query. The user responds with the search request. There are no more instances of the string "iox" in the current info segment, so the command responds with a message to that effect and the user query to which the user responds with the quit request.

Name: how_many_users, hmu

This command tells how many users are currently logged in on the system. In addition, it prints the name of the system, the load on the system, and the maximum load. If the absentee facility is running, the number of absentee users and the maximum number of absentee users is printed also.

Usage

```
how_many_users -control_args- -optional_args-
```

where:

1. control_args	can be chosen from the following control arguments:
-long, -lg	prints additional information including the name of the installation, the time the system was brought up, the time of the next shutdown, if it has been scheduled, and the time of the last shutdown or crash. Load information on absentee users is also printed.
-absentee, -as	prints load information on absentee users only, even if the absentee facility is not running.
-brief, -bf	suppresses the printing of the headers. Only used in conjuction with one of the optional_args.
2. optional_args	specifies that only selected users are to be listed and can be one of the following:
Person_id	lists a count of logged in users with the name Person_id.
.Project_id	lists a count of logged in users with the project name Project_id.
Person_id.Project_id	lists a count of logged in users with the name and project of Person_id.Project_id.

Notes

If this command is invoked without any arguments, basic summary information is printed (see the first example below).

Absentee counts in a selective use of how_many_users (i.e., when an optional_arg is specified) are denoted by an asterisk (*).

Up to 20 classes of selected users are permitted.

Examples

To print summary information, type:

```
how_many_users

Multics 2.0, load 5.0/50.0; 6 users
```

To print summary information on absentee users, type:

```
how_many_users -absentee

Absentee users 0/2
```

To print the additional information provided by the -long control argument, type:

```
how_many_users -long

Multics 2.0: PCO, Phoenix, Az.
Load = 13.0 out of 110.0 Units; users = 13
Absentee users = 0; Max absentee users = 45
System up since 06/25/74  0522.7
Last shutdown was at 06/25/74  0515.2
```

To print brief information about the SysDaemon project, type:

```
how_many_users -bf .SysDaemon

SysDaemon = 3 + 0*
```

To print brief information about the user whose Person_id is Smith, type:

```
how_many_users -bf Smith

Smith = 1 + 1*
```

Name: immediate_messages, im

This restores the immediate printing of messages sent by the send_message command (described in this document).

Usage

```
immediate_messages
```

Note

This command "cancels" the defer_messages command.

Name: indent, ind

The indent command improves the readability of a PL/I source segment by indenting it according to a set of standard conventions described below.

Usage

```
indent oldpath -newpath- -control_args-
```

where:

1. oldpath	is the pathname of the input PL/I source segment. If the input segment name does not have a suffix of pl1, the suffix is assumed.
2. newpath	is the pathname of the output PL/I source segment. If the output segment name does not have a suffix of pl1, the suffix is assumed. If this argument is omitted, newpath is assumed to be the same as oldpath, and the indented copy of the program replaces the original copy. However, if errors are detected during indentation and newpath is not given, the original copy is not replaced. Instead, the pathname of the temporary file containing the indented copy is printed in an error message.
3. control_args	can be any of the following:
-brief, -bf	suppress warning comments on illegal or non-PL/I characters found outside of a string or comment. (Such characters are never removed.) When this argument is given, those errors whose warning messages are suppressed do not prevent the original copy from being replaced.
-lmargin XX, -lm XX	set the left margin (indentation for normal program statements) to XX. If this argument is omitted, the default left margin is 11.
-comment YY, -cm YY	set the comment column to YY. Comments are lined up in this column unless they occur in the beginning of a line or are preceded by a blank line. If this argument is omitted, the default comment column is 61.
-indent ZZ, -in ZZ	set indentation for each level to ZZ. Each do, begin, proc, and procedure statement causes an additional ZZ spaces of indentation until the matching end statement is encountered. If this argument is omitted, the default indentation is 5.

Conventions

Declaration statements are indented five spaces for dcl declarations and ten for declare declarations. Identifiers appearing on different lines of the same declare statement are lined up under the first identifier on the first line of the statement. Structure declarations are indented according to level number; after level two, each additional level is indented two additional spaces.

An additional level of indentation is also provided for the then clause of an if-statement; else clauses are lined up with the corresponding if. Statements that continue over more than one line have an additional five spaces of indentation for the second and all succeeding lines.

Multiple spaces are replaced by a single space, except inside of strings or for nonleading spaces and tabs in comments. Trailing spaces and tabs are removed from all lines. The indent command inserts spaces before left parentheses, after commas, and around the constructs =, ->, <=, >=, and ^=. Spaces are deleted if they are found after a left parenthesis or before a right parenthesis. Tabs are used wherever possible to conserve storage in the output segment.

The indent command counts parentheses and expects them to balance at every semicolon. If parentheses do not balance at a semicolon, or if the input segment ends in a string or comment, indent prints a warning message. Language keywords (do, begin, end, etc.) are recognized only at parenthesis level zero, and most keywords are recognized only if they appear to begin a statement.

Restrictions

Lines longer than 350 characters are split, since they overflow indent's buffer size. This is the only case in which indent splits a line.

Labelled end statements do not close multiple open do statements.

The indent command assumes that the identifiers begin, end, procedure, proc, declare, and dcl are reserved words when they appear at the beginning of a statement. If the input contains a statement like:

```
do = do + 1;
```

the indent command interprets it to mean that the statement delimits a do group and does not indent correctly.

Structure level numbers greater than 99 do not indent correctly.

Name: initiate, in

The initiate command enables users to initiate segments directly, i.e., not using the normal search rules. For a discussion of search rules, see "The System Libraries and Search Rules" in Section III of the MPM Reference Guide. When a segment has been explicitly initiated (using this command) by some reference name, the first reference to that name does not cause a search of the directories specified by the search rules as would normally occur. Rather, the segment specified by this command is accessed.

Usage

```
initiate path -ref_names- -control_arg-
```

where:

1.	path	is the pathname of the segment to be initiated. A relative pathname may be used.
2.	ref_names	are optional reference names by which the segment can be known without further initiating. See "Notes" below.
3.	control_arg	can be -long or -lg and can appear anywhere in the command line. If present, the segment number assigned to the segment is printed on the user's terminal.

Notes

If no reference names are given in the command line, then the segment is initiated by the entryname part of the pathname. If any reference name is present in the command line, the segment is not initiated by its entryname, but by the reference names so given. If the pathname is a single element name, then the directory assumed is the working directory. The < and > symbols are recognized in the pathname; the star convention cannot be used to initiate a group of segments.

If a reference name cannot be initiated, an error message is given and the command continues initiating the segment by the other names.

To initiate a segment, the user must have nonnull access to that segment.

Examples

```
initiate >udd>m>mmm>gamma x y
```

initiates the segment >udd>m>mmm>gamma with the names x and y.

```
     initiate  pop
```

initiates the segment pop in the working directory and gives it the reference name pop.

```
     initiate xx u v -long
```

initiates the segment xx in the working directory with the reference names u and v and prints out the assigned segment number.

Name: io_call, io

This command performs an operation on a designated I/O switch.

Usage

```
io_call opname switchname -args-
```

where:

1.	opname	designates the operation to be performed.
2.	switchname	is the name of the I/O switch.
3.	args	may be one or more arguments, depending on the particular operation to be performed.

The opnames permitted, followed by their alternate forms where applicable, are:

```
attach                        look_iocb
close                         open
control                       position
delete_record, delete         print_iocb
detach_iocb, detach           put_chars
destroy_iocb                  read_key
find_iocb                     read_length
get_chars                     read_record, read
get_line                      rewrite_record, rewrite
modes                         seek_key
move_attach                   write_record, write
```

Usage is explained below under a separate heading for each designated operation. The explanations are arranged functionally rather than alphabetically.

Unless otherwise specified, if a control block for the I/O switch does not already exist, an error message is printed on error_output and the operation is not performed. If the requested operation is not supported for the switch's attachment and/or opening, an error message is printed on error_output.

The explanations of the operations cover only the main points of interest and, in general, treat only the cases where the I/O switch is attached to a file or device. For full details see the descriptions of the iox_ subroutine and the I/O modules in the MPM Subroutines and Section IV "Input and Output Facilities" of the MPM Reference Guide, respectively.

Operation: attach

```
io_call attach switchname modulename -args-
```

where:

1.	modulename	is the name of the I/O module to be used in the attachment.
2.	args	may be one or more arguments, depending on what is permitted by the particular I/O module.

This command attaches the I/O switch using the designated I/O module. The attach description is the concatenation of modulename and args separated by blanks. The attach description must conform to the requirements of the I/O module.

If a control block for the I/O switch does not already exist, one is created.

Operation: detach_iocb, detach

```
io_call detach switchname
```

This command detaches the I/O switch.

Operation: open

```
io_call open switchname mode
```

where mode is one of the following opening modes, which may be specified by its full name, or by an abbreviation:

```
stream_input, si                 keyed_sequential_input, ksqi
stream_output, so                keyed_sequential_output, ksqo
stream_input_output, sio         keyed_sequential_update, ksqu
sequential_input, sqi            direct_input, di
sequential_output, sqo           direct_output, do
sequential_input_output, sqio    direct_update, du
sequential_update, squ
```

This command opens the I/O switch with the specified opening mode.

Operation: close

```
io_call close switchname
```

This command closes the I/O switch.

Operation: get_line

```
io_call get_line switchname -n- -control_args-
```

where:

1.	n	is a decimal number greater than zero specifying the maximum number of characters to be read.
2.	control_args	can be selected from the following:
	-segment path -offset-, -sm path -offset-	specifies that the line read from the I/O switch is to be stored in the segment specified by path, at the location specified by offset.
	-nnl	specifies that the newline character, if present, is deleted from the end of the line.
	-nl	specifies that a newline character is added to the end of the line if one is not present.
	-lines	specifies that the offset, if given, is measured in lines rather than characters. This control argument only has meaning if the -segment control argument is also specified.

This command reads the next line from the file or device to which the I/O switch is attached. If n is given, and the line is longer than n, then only the first n characters are read.

If the -segment control argument is not specified, the line read is written onto the I/O switch user_output, with a newline character appended if one is not present and -nnl has not been specified.

If the -segment control argument is specified, the line is stored in the segment specified by path. If this segment does not exist, it is created. If offset is specifed, the line is stored at that position relative to the start of the segment. This is normally measured in characters, unless -lines has been used. If offset is omitted, the line is appended to the end of the segment. The bit count of the segment is always updated to a point beyond the newly added data.

Operation: get_chars

```
io_call get_chars switchname n -control_args-
```

where:

1. n — is a decimal number greater than zero specifying the number of characters to read.

2. control_args — can be selected from the same list as described under the get_line operation.

This command reads the next n characters from the file or device to which the I/O switch is attached. The disposition of the characters read is the same as described under the get_line operation; that is, they are written upon user_output if the -segment control argument is not specified, or stored in a segment if the -segment control argument is specified.

Operation: put_chars

```
io_call put_chars switchname -string- -control_args-
```

where:

1. string — may be any character string.

2. control_args — can be selected from the following:

-segment path -length-, -segment path -offset- -length-, -sm path -length-, -sm path -offset- -length-	specifies that the data for the output operation is to be found in the segment specified by pathname. The location and length of the data may be optionally described with offset and length parameters.
-nnl	specifies that a newline character is not to be appended to the end of the output string.
-nl	specifies that a newline character is to be added to the end of the output line if one is not present.
-lines	specifies that offsets and lengths are measured in lines instead of characters.

The string parameter and the -segment control argument are mutually exclusive. If a string is specifed, the contents of the string are the data output to the I/O switch. If the -segment control argument is specified, the data is taken from the segment specified by path, at the offset and length given. If offset is omitted, the beginning of the segment is assumed. If length is omitted, the entire segment is output.

If the I/O switch is attached to a device, this command transmits the characters from the string or the segment to the device. If the I/O switch is attached to an unstructured file, the data is added to the end of the file. The -nl control argument is the default on a put_chars operation: a newline character is added unless one is already present, or the -nnl control argument is specified.

Operation: read_record, read

```
     io_call read_record switchname n -control_args-
```

where:

1.	n	is a decimal integer greater than zero specifying the size of the buffer to use.
2.	control_args	can be selected from the same list as described under the get_line operation.

This command reads the next record from the file to which the I/O switch is attached into a buffer of length n. If the -segment control argument is not specified, the record (or the part of it that fits into the buffer) is printed on user_output. If the -segment control argument is specified, the record will be stored in a segment as explained under the get_chars operation.

Operation: write_record, write

```
     io_call write_record switchname -string- -control_args-
```

where:

1.	string	is any character string.
2.	control_args	may be selected from the same list as described under the put_chars operation.

This command adds a record to the file to which the I/O switch is attached. If the string parameter is specified, the record is equal to the string. If the -segment control argument is specified, the record will be extracted from the segment as described under the put_chars operation. In either case, the -nnl control argument is the default: a newline character is added only if the -nl control argument is specified. If the file is a sequential file, the record is added at the end of the file. If the file is an indexed file, the record's key must have been defined by a preceding seek_key operation.

<u>Operation</u>: rewrite_record, rewrite

```
io_call rewrite_record switchname -string- -control_args-
```

where:

1.	string	is any character string.
2.	control_args	may be selected from the same list as described under the put_chars operation.

This command replaces the current record in the file to which the I/O switch is attached. The new record is either the string parameter, or is taken from a segment, as described under the write_record operation. The current record must have been defined by a preceding read_record, seek_key, or position operation as follows:

read_record	current record is the last record read.
seek_key	current record is record with the designated key.
position	current record is the record preceding the record to which the file was positioned.

<u>Operation</u>: delete_record, delete

```
io_call delete_record switchname
```

This command deletes the current record in the file to which the I/O switch is attached. The current record is determined as in rewrite_record above.

<u>Operation</u>: position

```
io_call position switchname type
```

where type may be one of the following:

bof	set position to beginning of file
eof	set position to end of file
f <u>n</u>, fwd <u>n</u>, forward <u>n</u>	set position forward <u>n</u> records or lines (same as reverse -<u>n</u>)
r <u>n</u>, rev <u>n</u>, reverse <u>n</u>	set position back <u>n</u> records (same as forward -<u>n</u> records)

other	any other numeric argument or pair of numeric arguments may be specified, but its function depends on the I/O module being used and may not be implemented for all I/O modules.

This command positions the file to which the I/O switch is attached. If type is bof, the file is positioned to its beginning, so that the next record is the first record (structured files), or so that the next byte is the first byte (unstructured files). If type is eof, the file is positioned to its end; the next record (or next byte) is at the end-of-file position. If type is forward or reverse, the file is positioned forwards or backwards over records (structured files) or lines (unstructured files). The number of records or lines skipped is determined by the absolute value of n.

In the case of unstructured files, the next byte position after the operation is at a byte immediately following a newline character (or at the first byte in the file or at the end of the file); and the number of newline characters moved over is the absolute value of n.

If the I/O switch is attached to a device, only forward skips (where type is forward are allowed. The effect is to discard the next n lines input from the device.

Operation: seek_key

```
io_call seek_key switchname key
```

where key is a string of ASCII characters with $0 \leq$ length ≤ 256.

This command positions the indexed file to which the I/O switch is attached to the record with the given key. The record's length is printed on user_output. Trailing blanks in the key are ignored.

If the file does not contain a record with the specified key, it becomes the key for insertion. A following write_record operation adds a record with this key.

Operation: read_key

```
io_call read_key switchname
```

This command prints, on user_output, the key and record length of the next record in the indexed file to which the I/O switch is attached. The file's position is not changed.

Operation: read_length

```
io_call read_length switchname
```

This command prints, on user_output, the length of the next record in the structured file to which the I/O switch is attached. The file's position is not changed.

Operation: control

```
io_call control switchname order -args-
```

where:

1. order is one of the orders accepted by the I/O module used in the attachment of the I/O switch.

2. args are additional arguments dependent upon the order being issued and the I/O module being used.

This command applies only when the I/O switch is attached via an I/O module that supports the control I/O operation. The exact format of the command line depends on the order being issued and the I/O module being used. For more details, refer to "Control Operations from Command Level" in the appropriate I/O module in the MPM Subroutines. If the I/O module supports the control operation and the the paragraph just referenced does not appear, it can be assumed that only control orders that do not require an info_structure can be performed with the io_call command, as a null info_ptr is used. (See the description of the iox_$control entry point and the appropriate I/O module's control operation, both in the MPM Subroutines.)

Operation: modes

```
io_call modes switchname -string- -control_arg-
```

1. string is a sequence of modes separated by commas. The string must not contain blanks.

2. control_arg may be -brief or -bf

This command applies only when the I/O switch is attached via an I/O module that supports modes. The command sets only new modes specified in string, and then prints the old modes on user_output. Printing of the old modes is suppressed if the -brief control argument is used.

If the switch name is user_i/o, the command refers to the modes controlling the user's terminal. See the I/O module tty_ subroutine description in the MPM Subroutines for an explanation of applicable modes.

Operation: find_iocb

```
io_call find_iocb switchname
```

This command prints, on user_output, the location of the control block for the I/O switch. If it does not already exist, the control block is created.

Operation: look_iocb

```
io_call look_iocb switchname
```

This command prints, on user_output, the location of the control block for the I/O switch. If the I/O switch does not exist, an error is printed.

Operation: move_attach

```
io_call move_attach switchname switchname2
```

where switchname2 is the name of a second I/O switch.

This command moves the attachment of the first I/O switch (switchname) to the second I/O switch (switchname2). The original I/O switch is left in a detached state.

Operation: destroy_iocb

```
io_call destroy_iocb switchname
```

This command destroys the I/O switch by deleting its control block. The switch must be in a detached state before this command is used. Any pointers to the I/O switch become invalid.

Operation: print_iocb

```
io_call print_iocb switchname
```

This command prints, on user_output, all of the data in the control block for the I/O switch, including all pointers and entry variables.

Summary of Operations

```
Usage:  io attach switchname modulename -args-
Usage:  io detach switchname
Usage:  io open switchname mode
Usage:  io close switchname
Usage:  io get_line switchname -n- -control_args-
Usage:  io get_chars switchname n -control_args-
Usage:  io put_chars switchname -string- -control_args-
Usage:  io read_record switchname n -control_args-
Usage:  io write_record switchname -string- -control_args-
Usage:  io rewrite_record switchname -string- -control_args-
Usage:  io delete_record switchname
Usage:  io position switchname type
Usage:  io seek_key switchname key
Usage:  io read_key switchname
Usage:  io read_length switchname
Usage:  io control switchname order
Usage:  io modes switchname -string- -brief-
Usage:  io find_iocb switchname
Usage:  io look_iocb switchname
Usage:  io move_attach switchname switchname2
Usage:  io destroy_iocb switchname
Usage:  io print_iocb switchname
```

where:

1. switchname — is the name of the I/O switch.

2. modulename — is the name of I/O module used in the attachment.

3. args — are any arguments accepted by the I/O module used in the attachment.

4. mode — is one of the following modes:

stream_input, si	keyed_sequential_input, ksqi
stream_output, so	keyed_sequential_output, ksqo
stream_input_output, sio	keyed_sequential_update, ksqu
sequential_input, sqi	direct_input, di
sequential_output, sqo	direct_output, do
sequential_input_output, sqio	direct_update, du
sequential_update, squ	

5. n — is a decimal number.

6. string — is any character string.

7. type — sets the file position. It can be:

bof	forward n
eof	reverse n
othern	

8. key — is a string of ASCII characters with $0 \leq$ length ≤ 256.

9.	order	is one of the orders accepted by the I/O module used in the attachment of the I/O switch.
10.	control_args	can be chosen from the following: -segment path -offset-, -sm path -offset- -segment path -length-, -sm path -length- -segment path -offset- -length-, -sm path -offset- -length- -nnl -nl -lines -brief, -bf

This page intentionally left blank.

Name: line_length, ll

The line_length command allows the user to control the maximum length of a line output to the device that his process is connected to through the user_output I/O switch. This device is usually his terminal.

Usage

```
     line_length maxlength
```

where maxlength is a nonnegative decimal number greater than 2 that specifies the maximum number of characters that can henceforth be printed on a single line using the I/O switch named user_output. In most cases, this is the maximum length of a line of output printed at the user's terminal.

Name: link, lk

The link command causes a storage system link with a specified name to be created in a specified directory pointing to a specified segment or directory. For a discussion of links, see "Segment, Directory, and Link Attributes" in Section III of the MPM Reference Guide.

Usage

```
link path1_1 path2_1 ... path1_n -path2_n-
```

where:

1.	path1_i	specifies the pathname of the segment to which path2_i is to point. The pathnames must be specified in pairs.
2.	path2_i	specifies the pathname of the link to be created. If not given (in the final argument position of a command line only), a link to path1_i is created in the working directory with the entryname portion of path1_i as its entryname.

Notes

The user must have append permission for the directory in which the link is to be created.

Entrynames must be unique within the directory. If the creation of a specified link would introduce a duplication of names within the directory, and if the old entry has only one name, the user is interrogated as to whether he wishes the entry bearing the old instance of the name to be deleted. If he answers "no", the link is not created. (If the old entry has multiple names, the conflicting name is removed and a message to that effect is issued to the user.) In either case, since the directory in which the link is to be created is being changed, the user must also have modify permission for that directory.

The star and equal conventions can be used.

Example

```
link >my_dir>beta  alpha  >dictionary>grammar
```

creates two links in the working directory, named alpha and grammar; the first points to the segment beta in the directory >my_dir and the second points to the segment grammar in the directory >dictionary.

Name: list, ls

The list command prints information about entries contained in a single directory. A large selection of control arguments enable the user to specify the directory to be listed, which entries are to be listed, the amount and kind of information to be printed for each entry, and the order in which the entries are to be listed.

Usage

```
list -entrynames- -control_args-
```

where:

1. entrynames are the names of entries to be listed. If entrynames are given, only entries having at least one name matching an entryname argument are listed. The star convention may be used. If no entryname argument is given, all entries (of the types specified by control arguments) in the directory are listed.

2. control_args may be chosen from the arguments described in "Control Arguments for the list Command" below.

Except where otherwise noted in the descriptions of the control arguments ("Control Arguments for the list Command" below), the entrynames and control_args arguments may appear anywhere on the command line.

Basic Use of the list Command

If the list command is invoked without any arguments, it lists all segments and multisegment files in the working directory, printing the name(s), access mode, and length of each. Segments and multisegment files are listed separately (segments first), each preceded by a line giving the total entries of that type and the sum of their lengths. (This line is referred to as the totals information or the header.) Within each entry type, entries are listed in the order in which they are found in the directory.

The following example shows the results of invoking the list command without any arguments (the line typed by the user is preceded by an exclamation mark):

```
! list

  Segments = 8, Lengths = 41.

  r w   10  new_code_info.runout
  rew    9  new_code_info.runoff
  r w    3  work.pl1
  r w    7  work.list
  re     2  work
  r w    1  print.ec
  r w    1  output_file
  r      8  data_base

  Multisegment-files = 1, Lengths = 334.

  r w  334  info_segs
```

Notice that the information about the entries is arranged in columns without column headings. The set of columns printed by the list command depends on the control arguments given by the user and the type of entry being listed.

There are four entry types: segments, multisegment files, directories, and links. Segments and multisegment files are referred to collectively as files; segments, multisegment files, and directories are referred to collectively as branches. The set of possible columns is different for branches and links. For branches, the set of possible columns and their order (from left to right) is: modification date, date and time used, access mode, size, names, and number of names; for links: date and time entry modified, names, number of names, and link pathname. The modification-date column contains either the date and time the entry was modified or the date and time the contents were modified, and the size column contains either records used or length (in records) computed from the bit count, as specified by control arguments. Unless otherwise specified by control arguments, the items printed for branches are: access modes, length, and names; for links: names and link pathname.

The list command offers the user precise control over the command output. The various control arguments specify exactly what is to be printed. Most users will find that the following subset of list command control arguments allows them to adequately define the desired information.

-file, -f	lists information about files. (This is the default.)
-directory, -dr	lists information about directories.
-link, -lk	lists information about links.
-name, -nm	prints the names column, giving primary and any additional names of each entry.
-date_time_entry_modified, -dtem	prints the date and time the entry was last modified (e.g., by the changing of attributes such as names, ACL, or bit count).

-primary, -pri	prints only the primary name (in the names column) of each entry.
-sort XX, -sr XX	sorts the entries, within each entry type, according to the column name specified by XX. (The column names and their sorting order are described under "Entry Order" below.)
-total, -tt	prints only the heading line (totals information) for each entry type specified; this line gives the total number of entries and the sum of their sizes.

Detailed information on each of the above control arguments is given in "Control Arguments for the list Command" below.

Control Arguments for the list Command

The control arguments for the list command are described in detail on the following pages. For convenience, these arguments have been arranged in categories according to the function they perform. The categories and their respective control arguments are listed below (detailed descriptions follow the list, in the same order):

```
directory
     -pathname path, -pn path

entry type
     -segment, -sm
     -multisegment_file, -msf
     -file, -f
     -directory, -dr
     -branch, -br
     -link, -lk
     -all, -a

columns
     -mode, -md
     -length, -ln
     -record, -rec
     -name, -nm
     -date_time_used, -dtu
     -date_time_entry_modified, -dtem
     -date_time_contents_modified, -dtcm
     -count, -ct
     -link_path, -lp

totals/header line
     -total, -tt
     -no_header, -nhe

multiple-name entries
     -primary, -pri
     -match

entry order
     -sort XX, -sr XX
     -reverse, -rv
     entry exclusion
         -exclude entryname, -ex entryname
         -first N, -ft N
         -from D, -fm D
         -to D

     output format
         -brief, -bf
         -short, -sh
```

DIRECTORY

If no control argument from this category is given, the working directory is assumed.

-pathname path, -pn path
: causes entries in the directory specified by path to be listed.

ENTRY TYPE

If no control argument from this category is given, the -file control argument is assumed. More than one of the following control arguments may be given.

-segment, -sm
: lists information about segments.

-multisegment_file, -msf
: lists information about multisegment files.

-file, -f
: lists information about files (i.e., segments and multisegment files, in that order).

-directory, -dr
: lists information about directories.

-branch, -br
: lists information about branches (i.e., segments, multisegment files, and directories, in that order).

-link, -lk
: lists information about links.

-all, -a
: lists information about all entry types in the following order: segments, multisegment files, directories, and links.

COLUMNS

If no control argument from this category is given, the access-mode, length, and names columns (in that order) are printed for branches and the names and link-path columns (in that order) are printed for links. More than one of the control arguments listed below can be given in a single invocation of the list command. When the -brief, -mode, -record, -length, or -name control arguments are given, only the names column plus those columns explicitly selected by control arguments are printed.

The user is given a choice as to what can be printed in two of the columns for branches (size and modification date). For size, the user may choose between length computed from the bit count or a count of records used. For modification date, the user may choose between the date and time the entry was modified (e.g., by the changing of attributes such as names, ACL, or bit count) or the date and time the contents of the segment or directory were modified.

If sorting by a size or modification date is specified, the above choices also apply to sorting, and the specifications of what to sort on and what to print must be consistent. For example, it is not possible to print length computed from bit count while sorting on records used.

Because of the way the information is maintained by the storage system, the records-used, date-time-contents-modified, and date-time-used values are more expensive to obtain than the other items printed by the list command. It is recommended that these values not be used for printing or sorting except when absolutely necessary. Less expensive alternatives are provided that should be suitable in most cases (e.g., length computed from bit count, and date and time the entry was modified).

The names column is printed in every invocation of the list command except when the user explicitly requests only totals information (see "Totals/Header Line" below).

-mode, -md
    prints the access-mode column.

-length, -ln
    prints current length computed from the bit count. This control argument is inconsistent with the -record control argument; only one of the two may be given. The -length argument, which is the less expensive of the two, is the default.

-record, -rec
    prints the records used. This argument is inconsistent with the -length control argument; only one of the two may be given. The -record control argument is the more expensive of the two.

-name, -nm
    prints the names column, giving the primary name and any additional names of each entry.

-date_time_used, -dtu
    prints the date and time the entry was last used.

-date_time_entry_modified, -dtem
prints the date and time the entry was last modified. (e.g., by the changing of attributes such as names, ACL, or bit count). This argument is inconsistent with the -date_time_contents_modified control argument; only one of the two may be given. This argument is the less expensive of the two.

-date_time_contents_modified, -dtcm
prints the date and time the contents of the segment or directory were last modified. This argument is inconsistent with the -date_time_entry_modified control argument; only one of the two may be given. This argument is the more expensive of the two.

-count, -ct
prints the count column, which gives the total number of names for entries that have more than one name.

-link_path, -lp
prints the link-path column.

TOTALS/HEADER LINE

If no control argument from this category is given, both totals and detailed information are printed.

-total, -tt
prints only the heading line (totals information) for each entry type specified; this line gives the total number of entries and the sum of their sizes.

-no_header, -nhe
omits all heading lines.

MULTIPLE-NAME ENTRIES

The control arguments in this category are applicable only to entries that have more than one name. If no control argument from this category is given, all of the names of the specified entries are printed in the names column.

-primary, -pri
prints, in the names column, only the primary name of each entry. This control argument does not suppress the printing of any other columns; it merely suppresses the printing of secondary names.

-match
prints, in the names column, only those names that match one of the given entrynames.

ENTRY ORDER

If no control argument from this category is given, entries are printed in the order in which they are found in the directory.

-sort XX, -sr XX
: sorts entries, within each entry type, according to the sort column XX where XX can be one of the following:

name, nm	sort entries by primary name, according to the standard ASCII collating sequence.
length, ln	sort entries by length computed from the bit count, largest first. This argument is inconsistent with the -record control argument.
record, rec	sort entries by records used, largest first. This argument is inconsistent with the -length control argument. If this argument is given, and the size column is being printed, the value printed in that column will be records used, rather than length.
mode, md	sort entries by access mode in the following order: null, r (or s), rw (or sm), re, rew (or sma). (This order is the result of sorting by the internal representation of the mode.)
date_time_entry_modified, dtem	sort entries by the date and time the entry was last modified, most recent first. This argument is inconsistent with the -dtcm control argument. If the -dtem control argument is given and no sort key follows the -sort control argument, then this argument is implied as the default sort key.
date_time_contents_modified, dtcm	sort entries by the date and time the contents of the entry were last modified, most recent first. This argument is inconsistent with the -dtem control argument. If the -dtcm control argument is given and no sort key follows the -sort control argument, then this argument is implied as the default sort key.
date_time_used, dtu	sort entries by the date and time used, most recent first.
count, ct	sort entries by number of names, most names first.

It is not necessary for a column to be printed in order to sort on it, but note the restrictions described earlier, regarding sorting on and printing the modification-date and size columns.

If the sort column XX is omitted, the default sorting column is determined as follows: if no date column is being printed, sort by primary name; if only one of the date columns is being printed, sort by that date; if both the modification-date and date-time-used columns are being printed, sort by the modification-date column.

Links can only be sorted by the name, modification-date, or count columns. If sorting by any other column is specified, links are printed in the order in which they are found in the directory, while branches (if also being listed) are sorted by the specified column. (See "Notes" below.)

-reverse, -rv
: prints entries in the reverse of the order in which they are found in the directory. If the -sort control argument is also given, the specified sort is reversed.

ENTRY EXCLUSION

The following control arguments limit the amount of output produced by excluding entries according to either name or date or by setting an upper limit on the number of entries listed.

-exclude entryname, -ex entryname
: do not list any entries that have a name that matches the specified entryname. The star convention may be used.

If the user wishes to exclude more than one entryname, he must give an -exclude control argument for each one of them. The entrynames given in all -exclude control arguments and any names given in the entryname arguments (explained on the first page of the list command description) operate together to limit the entries that are listed. All entries that have at least one name that matches any one of the entrynames given in the -exclude control arguments are excluded from the listing. From the entries that remain, those matching any of the entryname arguments are listed; if no entryname arguments are given, all the remaining entries are listed. (See "Examples" below.)

-first N, -ft N
: list only the first N entries (after sorting, if specified) of each entry type being listed. The heading lines will contain the totals figures for all entries that would have been listed if the -first control argument had not been given. This argument is useful to avoid tying up a terminal by listing a large directory, when only the first few entries are of interest.

The following two arguments exclude entries on the basis of date. The date used in this comparison is the modification-date value in all cases except when the only date column being printed or sorted on is the date-time-used column. If no date column is being printed, the date-time-entry-modified value is used.

-from D, -fm D
: do not list any entries that have a date value (selected as described above) before the one specified by D.

-to D
: do not list any entries that have a date value (selected as described above) after the one specified by D.

The D value after the -from or -to control arguments must be a string acceptable to the convert_date_to_binary_ subroutine, described in the MPM Subroutines. If the date-time string contains spaces, the string must be enclosed in quotation marks. The D value should specify both a date and a time; if only a date is given, then the convert_date_to_binary_ subroutine uses, as the default time, the current time of day.

If both the -from and -to control arguments are given, the -from D value must be earlier than the -to D value.

OUTPUT FORMAT

If no control argument from this category is given, the output format of the list command is not changed.

-brief, -bf
: if just totals information is being printed, this argument causes the totals information for all selected entry types to be abbreviated and printed on a single line. Otherwise, it suppresses the printing of the default columns when they are not explicitly named in control arguments. For example, typing:

```
     list -dtu -brief
```

: causes the names and date-time-used columns, but not the access-mode and length columns, to be printed.

-short, -sh
: prints link pathnames starting two spaces after their entrynames, instead of aligning them in column position 35.

Notes

The obsolete name for a modification date (date_time_modified, dtm) is accepted, in both the control argument and sort key form, as a synonym for the date-time-entry-modified value.

Links do not have a date-time-contents-modified value. If links are being listed and either modification-date value is specified for printing, sorting, or entry exclusion (using the -from and -to control arguments), the date-time-entry-modified value of links is used.

Examples

```
     list -primary -count
```

lists all files in the working directory (the default directory); the names column contains only the primary names of all entries; the total number of names (for those entries having more than one name) is printed after the primary name. In addition to the names column, the access-mode and length columns are printed.

```
     list -exclude *.*
```

lists all the files in the working directory having other than two-component names, printing the three default columns (access mode, length, and names).

```
     list -segment *.* -exclude *.pl1
```

lists all the segments in the working directory having two-component names whose second component is not pl1, printing the three default columns.

```
     list -date_time_entry_modified -sort
```

lists all files in the working directory, sorted by the date-time-entry-modified column (the default sort key since the user specifically requested that date column). The date-time-entry-modified column is printed in addition to the three default columns.

```
     list -name -sort date_time_modified
```

lists all files in the working directory, sorted by the date-time-entry-modified value. Only the names column is printed. Note the use of date_time_modified as a synonym for date_time_entry_modified.

```
     list -segment -name -primary -no_header
```

lists only the primary name of each segment in the working directory without printing the heading line or any blank lines. This combination of arguments, together with the file_output command, is useful for generating a file that contains the primary names of a selected set of entries.

```
     list -mode -primary
```

lists the access mode and primary name of each file in the working directory.

```
     list -total -to "7/1/75 0000.0" -dtu -rec
```

prints the totals (number of entries and total records used) for all files that have not been used since the end of June 1975. Notice that the -dtu control argument is used to specify that the -to date refers to the date and time used.

Name: list_abs_requests, lar

The list_abs_requests command allows the user to obtain information about absentee requests. Normally the user is allowed information concerning only requests that he has made. The user makes absentee requests using the enter_abs_request command.

Usage

```
     list_abs_requests -control_args-
```

where control_args are selected from the following list of control arguments and can appear anywhere on the command line:

-total, -tt	indicates that the user wants only the total number of the requests in the queue.
-long, -lg	indicates that all of the information pertaining to an absentee request is to be printed. If this control argument is omitted, only the full pathname of the absentee control segment is printed.
-queue *n*, -q *n*	indicates which queue is to be searched. It must be followed by a decimal integer specifying the number of the queue. If this control argument is omitted, the third priority queue is searched unless the -all control argument is provided. (See below.)
-all, -a	indicates that all priority queues are to be searched, starting with the highest priority queue and ending with the lowest priority queue.

Notes

The -total and -long control arguments are incompatible and cannot be used in the same list_abs_requests command line.

The default condition is to list only pathnames for queue 3.

Examples

To find out what absentee requests he has in queue 3, the user types:

```
list_abs_requests


Queue 3: 3 requests. 6 total requests.

     >udd>Multics>Jones>dump>translate.absin
     >udd>Multics>Jones>abs>tasks.absin
     >udd>Multics>Jones>abs>bindings.absin
```

To get all information about his absentee requests in the highest-priority queue, he types:

```
list_abs_requests -long -queue 1


Queue 1: 2 requests. 27 total requests.

Absentee input segment:     >udd>Multics>Jones>dump>translate.absin
Restartable:                yes
Deferred time:              09/16/74  2300.0 edt Thu
Argument string:            "pl1"
                            "abcd"
                            "-table"
                            "-map"


Absentee input segment:     >udd>Multics>Jones>bind>auto_bind.absin
Restartable:                no
Cpu limit:                  600 seconds
Absentee output file:       >udd>Multics>Day>bind>bd.out
```

To find out the total number of absentee requests he has in all queues, he types:

```
list_abs_requests -total -all


Queue 1: 2 requests. 15 total requests.

Queue 2: 0 requests. 0 total requests.

Queue 3: 0 requests. 39 total requests.
```

Name: list_acl, la

The list_acl command lists the access control lists (ACLs) of segments, multisegment files, and directories. For a description of ACLs, see "Access Control" in Section VI of the MPM Reference Guide.

Usage

```
list_acl -path- -User_ids- -control_args-
```

where:

1. path	is the pathname of a segment, multisegment file, or directory. If it is -wd, -working_directory, or omitted, the working directory is assumed. If it is omitted, no User_ids can be specified. The star convention can be used.
2. User_ids	are access control names that must be of the form Person_id.Project_id.tag. All ACL entries with matching names are listed. (For a description of the matching strategy, refer to the set_acl command.) If User_id is omitted, the entire ACL is listed.
3. control_args	can be chosen from the following control arguments:
-ring_brackets, -rb	lists the ring brackets. This control argument can appear anywhere on the line and affects the whole line. Ring brackets are discussed under "Intraprocess Access Control" in Section VI of the MPM Reference Guide.
-brief, -bf	suppresses the message "User name not on ACL of path."
-directory, -dr	lists the ACLs of directories only. The default is segments, multisegment files, and directories. (See "Notes" below.)
-segment, -sm	lists the ACLs of segments and multisegment files only.

Notes

The -directory and -segment control arguments are used to resolve an ambiguous choice that may occur when path is a star name.

If the list_acl command is invoked with no arguments, it lists the entire ACL of the working directory.

Example

```
     list_acl notice.runoff .Faculty. Doe
```

lists, from the ACL of notice.runoff, all entries with Project_id Faculty and the entry for Doe.*.*.

```
     list_acl *.pl1  -rb
```

lists the whole ACL and the ring brackets of every segment in the working directory that has a two-component name with a second component of pl1.

```
     la -wd -rb .Faculty. *.*.*
```

lists access modes and ring brackets for all entries on the working directory's ACL whose middle component is Faculty and for the *.*.* entry.

Name: list_daemon_requests, ldr

The list_daemon_requests command prints information about dprint and dpunch requests. Normally, the user is allowed to obtain information only on requests that he has made. See the descriptions of the dprint and dpunch commands.

Usage

```
list_daemon_requests -control_args-
```

where control_args are selected from the following list of control arguments and can appear anywhere in the command line:

-total, -tt	indicates that the user wants only the total number of the requests in the queue.
-long, -lg	indicates that all of the information pertaining to a request should be printed. If this control argument is not given, only the full pathname of the segment or file to be dprinted or dpunched is printed.
-request_type XX, -rqt XX	indicates that information on requests for the request type specified by XX should be printed. If this argument is not given, the default request type is "printer". Request types can be listed by the print_request_types command.
-queue n, -q n	indicates which priority queue is to be searched. It must be followed by an integer specifying the number of the queue. If this control argument is not given, only the third priority queue is searched unless the -all control argument is provided (see below).
-all, -a	indicates that all priority queues for the specified request type are to be searched, starting with the highest priority queue and ending with the lowest priority queue.
-admin -XX-, am -XX-	indicates that information on requests of all, or specified, users should be printed. The default is to print information about the user's own requests. If XX is specified, it may be in one of three forms: Person_id .Project_id Person_id.Project to specify listing by Person_id, Project_id, or a combination of both. If XX is not specified, information on all requests is printed. Special access is required to use this control argument. If the user does not have the appropriate access, a message to that effect is printed.

Notes

The -total and -long control arguments are incompatible and cannot be used in the same list_daemon_requests command line.

Only request types of generic type "printer" or "punch" can be specified by the -request_type control argument. These request types can be listed by invoking the print_request_types command.

If no control arguments are given, the default action of this command is to list only pathnames for queue 3 of the default request type ("printer").

Examples

To find out what dprint requests he has in queue 3 for the default request type, the user types:

```
list_daemon_requests


Queue 3: 3 requests. 6 total requests.

     >udd>Alpha>Jones>dump>translate.list
     >udd>Alpha>Jones>doc>ldr.runoff
     >udd>Alpha>Jones>Jones.profile
```

To get all information about his dprint requests in the highest priority queue for the default request type, the user types:

```
list_daemon_requests -long -queue 1


Queue 1: 2 requests. 27 total requests.

Pathname:      >udd>Beta>Smith>foo.list
Type:          print
Copies:        1
Time:          12/02/74  1124.7 mst Mon
Delete:        yes
For:           Jones


Pathname:      >doc>info>motd.info
Type:          print
Copies:        3
Time:          12/02/74  1145.0 mst Mon
Delete:        no
To:            575 Tech Sq.
```

To find out the total number of requests he has in all queues for the request type "punch", the user types:

```
list_daemon_requests -total -all -request_type punch

Queue 1: 0 requests. 0 total requests.

Queue 2: 0 requests. 15 total requests.

Queue 3: 2 requests. 39 total requests.
```

This page intentionally left blank.

Name: list_iacl_dir, lid

This command lists some or all of the entries on a directory initial access control list (initial ACL) in a specified directory. A directory initial ACL contains the ACL entries to be placed on directories created in the specified directory. For a discussion of initial ACLs, see "Access Control" in Section VI of the MPM Reference Guide.

Usage

```
list_iacl_dir -path- User_ids -control_args-
```

where:

1. path	specifies the directory in which the directory initial ACL should be listed. If path is -wd, -working_directory, or omitted, then the working directory is assumed. If path is omitted, no User_ids can be specified. The star convention can be used.
2. User_ids	are access control names that must be of the form Person_id.Project_id.tag. All access names that match the given User_ids are listed. (For a description of the matching strategy, refer to the set_acl command.) If no User_id is specified, the entire initial ACL is listed.
3. control_args	are selected from the following control arguments:
-ring *n*, -rg *n*	identifies the ring number whose directory initial ACL should be listed. If present, it must be followed by *n* (where $0 \leq n \leq 7$). This argument can appear anywhere on the line and affects the whole line. If this argument is not specified, the ACL of the user's current ring is listed.
-brief, -bf	suppresses the message "User name not on ACL of path."

Note

If the list_iacl_dir command is given without any arguments, the entire initial ACL for the working directory is listed.

Examples

```
     list_iacl_dir all_runoff .Faculty Fred..
```

lists, from the directory initial ACL in the all_runoff directory, all entries ending in Faculty.* and all entries with the Person_id Fred.

```
     list_iacl_dir -wd -a -rg 5
```

lists all entries in the ring 5 directory initial ACL of the working directory.

Name: list_iacl_seg, lis

This command lists some or all of the entries on a segment initial access control list (initial ACL) in a specified directory. A segment initial ACL contains the ACL entries to be placed on segments created in the specified directory. For a discussion of initial ACLs, see "Access Control" in Section VI of the MPM Reference Guide.

Usage

```
list_iacl_seg -path- -User_ids- -control_args-
```

where:

1. path — specifies the directory in which the segment initial ACL should be listed. If path is -wd, -working_directory, or omitted, then the working directory is assumed. If path is omitted, then no User_ids can be specified. The star convention can be used.

2. User_ids — are access control names that must be of the form Person_id.Project_id.tag. All access names that match the given components are listed. (For a description of the matching strategy, refer to the set_acl command.) If no User_id is specified, the entire initial ACL is listed.

3. control_args — are selected from the following control arguments:

   -ring n, -rg n — identifies the ring number whose segment initial ACL should be listed. If present, it must be followed by n (where 0 $\leq$ n $\leq$ 7). This argument can appear anywhere on the line and affects the whole line. If this argument is not specified, the ACL of the user's current ring is listed.

   -brief, -bf — suppresses the message "User name not on ACL of path."

Note

If the list_iacl_seg command is given without any arguments, the entire segment initial ACL for the working directory is listed.

Examples

```
     list_iacl_seg all_runoff .Faculty.  Fred
```

lists, from the segment initial ACL in all_runoff directory, all entries with the Project_id Faculty and the entry for Fred.*.*.

```
     list_iacl_seg -wd -a -rg 5
```

lists all entries in the ring 5 segment initial ACL of the working directory.

Name: list_ref_names, lrn

This command lists the reference names associated with a specified segment, it accepts both segment numbers and pathnames as the segment specification. When a pathname is specified, the segment number by which it is known is printed. When a segment number is specified, it also prints the pathname of the segment.

Usage

```
list_ref_names paths -control_args-
```

where:

1.	paths	can be segment numbers or pathnames of segments known to the user's process. If path is a segment number, the pathname and reference names of the segment are printed. If path is a pathname, the segment number (in octal) and the reference names of the segment are printed. If a pathname looks like a control argument (i.e., if it is preceded by a minus sign) or a number, then path should be preceded by -name or -nm.
2.	control_args	can be any of the following control arguments:

-from i -to k
allows the user to specify a range of segment numbers (segments i through k). The pathnames and reference names of the segments in this range are printed. If the -from control argument is omitted, and the -to argument is given then the segment number of the first segment not in ring 0 is assumed, unless -all is used (see below). If the -to control argument is omitted, and the -from argument is given then the highest used segment number is assumed.

-brief, -bf
suppresses printing of the reference names for the entire execution of the command. This control argument can appear anywhere in the line.

-all, -a
prints the pathnames and reference names of all known segments, as well as the reference names of ring 0 segments. This control argument can appear anywhere in the line. The -all control argument is equivalent to -from 0.

Notes

All of the above arguments (segment specifiers and control arguments) can be mixed. For example, in the command line:

```
list_ref_names  156 -from 230 path_one
```

the pathname and reference names of segment 156 and of all segments from 230 on are also printed. The segment number (in octal) and the reference names of path_one are printed.

In the default condition, when called with no arguments, list_ref_names prints information on all segments that are not in ring 0.

Name: list_resources, lr

The list_resources command lists some or all of the resources that are assigned or attached to the calling process by the resource control package (RCP).

Usage

```
     list_resources  -control_args-
```

where control_args may be chosen from the following:

-long, -lg	prints all the information known about each resource listed. If this control argument is not supplied, only the name is printed for each resource listed.
-type XX, -tp XX	lists only resources of the type XX. Currently, this argument must specify a device type.
-device XX, -dv XX	lists device resources with the name XX. No other resources are listed.
-logical_volume, -lv	lists only logical volumes that are currently attached.
-assignments, -asm	lists only resource assignments.
-attachments, -atm	lists only device attachments.

Note

If this command is invoked without any arguments, all resources assigned and devices attached to the calling process are listed.

Examples

In the example below, the user issues the list_resources command with no control arguments. The system responds with the name of the assigned devices.

```
     list_resources

     Device Assignments
          Device tape_05
          Device tape_02
```

In the next example, the user issues the list_resources command with the -long control argument. The system responds with all the information known about each resource listed.

```
     list_resources -long

     Device Assignments
     2 devices assigned
          Device tape_05
               State     =  assigned
               Time      =  04/30/76  1316.2 edt Fri
               Disp      =  retain
               Level     =  4
               Model     =  500
               Tracks    =  9
               Densities =  200 556 800 1600
          Device tape_02
               State     =  assigned
               Time      =  04/30/76  1314.7 edt Fri
               Disp      =  retain
               Level     =  4
               Model     =  500
               Tracks    =  9
               Densities =  200 556 800 1600
```

This page intentionally left blank.

Name: logout

The logout command terminates a user session and ends communication with the Multics system. It signals the finish condition for the process; and, after the default on unit for the finish condition closes all open files and returns, it destroys the process.

Usage

```
     logout -control_args-
```

where control_args can be chosen from the following:

-hold, -hd	the user's session is terminated. However, communication with the Multics system is not terminated, and a user can immediately log in without redialing.
-brief, -bf	no logout message is printed, and if the -hold control argument has been specified, no login message is printed either.

Note

See Section II, "How to Access the Multics System" in the Multics Users' Guide, Order No. AL40.

Name: mail, ml

The mail command sends a message to another user or prints messages in any mailbox to which the user has sufficient access. The extended access used on mailboxes, which are ring 1 segments, permits the creator of a mailbox to control other users' access to his mailbox. Adding, reading, and deleting messages are independent privileges under extended access. For example, one user can be given access to only add messages, and another user to add messages and also read and delete only the messages he has added. For more information on extended access, see "Extended Access" below. Mail and interactive messages sent to a user are placed in the mailbox named Person_id.mbx in his home directory.

Usage

To send mail:

```
     mail path Person_id1 Project_id1 ... -Person_idn- -Project_idn-
```

where:

1.	path	is the pathname of a segment to be sent or is an asterisk (*) to indicate that the user wishes to type a message to be sent (see "Composing Mail" below).
2.	Person_idi	is the name of a person to whom mail is to be sent.
3.	Project_idi	is the name of a project on which Person_idi is registered.

To print messages sent by the mail and send_message commands:

```
     mail -path- -control_arg-
```

where:

1.	path	is the pathname of a mailbox. If the mbx suffix is not given, it is assumed. If no path argument is given, the contents of the default mailbox is printed (see "Creating a Mailbox" below).
2.	control_arg	can be -brief or -bf so that only the total number of messages in the mailbox is printed. If the mailbox is empty, nothing is printed.

Printing Mail

When the contents of the mailbox named by path is printed, it is preceded by a line of the form:

<u>n</u> messages.

where <u>n</u> is a decimal integer greater than zero.

Each message is preceded by a line of the form:

<u>i</u>) From: Person_id.Project_id (sent_from) date time (<u>n</u> lines)

where:

1.	<u>i</u>	is the incremental number of the message. The first message is numbered 1, the second is 2, etc. The messages are printed beginning with message 1; that is, the oldest message is printed first and the newest is printed last.
2.	Person_id	is the registered person identifier of the user who sent the message.
3.	Project_id	is the name of the project on which the sender was logged in when he sent the message.
4.	sent_from	is an optional field that further identifies the sender, e.g., the log-in name of an anonymous user, if he has one.
5.	date	is the date the message was sent, of the form mm/dd/yy to indicate the month, day, and year.
6.	time	is the time the message was sent, of the form hhmm.m to indicate the hours and minutes (including tenths of minutes) in 24-hour time.
7.	<u>n</u> lines	is the number of lines in the message.

*

After printing all messages, the mail command asks whether the user wants the messages deleted. If the answer is yes, all messages in the mailbox are deleted. If the answer is no, no messages are deleted. In either case, the user returns to command level.

If the user issues a quit signal while the messages in the mailbox are being printed and then issues the program_interrupt command, the mail command stops printing and asks whether to delete all messages in the mailbox, including those that were not printed.

Sending a Segment

The contents of the segment specified by path is sent to the mailbox:

>user_dir_dir>Project_idi>Person_idi>Person_idi.mbx

for each Person_id-Project_id pair specified in the command line.

The segment to be mailed must be less than one record long (4096 ASCII characters).

Composing Mail

If path is *, mail responds with "Input:" and accepts lines from the terminal until a line consisting only of a period (.) is typed. The typed lines are then sent to the specified user(s).

Creating a Mailbox

A default mailbox is created automatically the first time a user prints his own mail or issues the accept_messages or print_messages commands. The default mailbox is:

>user_dir_dir>Project_id>Person_id>Person_id.mbx

Extended Access

Access on a newly created mailbox is automatically set to adrosw for the user who created it, as for *.SysDaemon.*, and aow for *.*.*. The types of extended access for mailboxes are:

add	a	add a message
delete	d	delete any message
read	r	read any message
own	o	read or delete only your own messages, i.e., those sent by you
status	s	find out how many messages are in the mailbox
wakeup	w	send a normal priority wakeup

The modes "n", "null", or "" specify null access.

Related Commands

Special commands exist to create additional mailboxes and to change the attributes of mailboxes. These commands, described in the MPM Subsystem Writers' Guide, are:

mbx_create	create a mailbox
mbx_delete	delete a mailbox
mbx_add_name	add a name to a mailbox
mbx_delete_name	delete a name from a mailbox
mbx_rename	rename a mailbox
mbx_list_acl	list the access control list (ACL) of a mailbox
mbx_set_acl	change or add entries to the ACL of a mailbox
mbx_delete_acl	delete entries from the ACL of a mailbox
mbx_set_max_length	set the maximum length of a mailbox

Name: memo

The memo command makes it possible to use Multics as an interactive notebook and reminder list containing memos. This command allows the user to specify a maturity time for each memo (a time before which the memo will not appear). By use of the alarm feature, the user can specify the exact time the memo is to be printed on the terminal. Memos can also be set that are passed directly to the command processor and executed as a normal Multics command line. Using these features jointly, the user can set a memo that actually performs a specified action at a specified time by itself rather than merely reminding him to perform the action. Finally, the user can specify that the memo is to be repeated at regular intervals.

In the default case, memo maintains its information in segment Person_id.memo in the user's home directory. If memo is invoked and such a segment does not exist, memo attempts to create and initialize it. Optionally a different memo segment can be specified and used. Each memo in the memo segment consists of a text portion containing up to 132 characters, the maturity date, a sequence number (memo_number) assigned by the memo command and additional information telling whether the memo is to be repeated or not and whether it is to be printed or executed.

For the user's convenience, control arguments allow the printing, listing, and deletion of memos selected by subsequent optional arguments. Memos can be selected by number, type, maturity time, and content. Other control arguments enable or disable memo alarms.

It should be noted that if a date, time, repeat interval, or match string contains embedded blanks, that string must be enclosed in quotes so that the command processor passes it to memo as a single argument.

Usage

```
     memo -control_arg- -optional_args- -memo_text-
```

where:

1. control_arg	is one of the following control arguments. Only one can appear on the command line, and it must be the first argument. If no control argument appears, the rest of the line is used to set a memo. If no arguments are specified, mature memos are printed or executed and alarms are enabled.
-pathname path, -pn path	uses the memo segment specified by the pathname path. If the segment specified by path does not exist, memo attempts to create it. If the user does not specify the suffix memo, it is added. This control argument must not be followed by any optional arguments.

-list, -ls	lists memos selected by the optional arguments in full detail, including their maturity times, text, memo_numbers and information about the optional arguments (optional_args) used when the memos were set. No memos are executed.
-print, -pr	prints the text of all memos selected by the optional arguments. No memos are executed.
-delete, -dl	deletes all memos selected by the optional arguments.
-off	suppresses all memo alarms, until the next memo -on, memo, or memo -brief command. This control argument must not be followed by any optional arguments.
-on	enables memo alarms without printing or executing any nonalarm memos. This control argument must not be followed by any optional arguments.
-brief, -bf	suppresses the message "No memos" if none are found. Mature memos are printed or executed and alarms are enabled. This control argument must not be followed by any optional arguments.
2. optional_args	can be selected from the following optional arguments. Some of the arguments can be used for setting memos; some for selecting memos to be printed, listed, or deleted; and others for both setting and selecting memos.
memo_number	specifies which memos are to be selected. memo_number is printed when the user types memo, memo -brief, or memo -list.
-date mtime, -dt mtime	identifies a time (mtime) in a form suitable for input to the subroutine convert_date_to_binary_ (see the description for this subroutine in the MPM Subroutines). The mtime is truncated to midnight preceding the date in which mtime falls. If used while setting a memo, then the truncated mtime becomes the maturity time of the new memo. If memos are being selected, then only those memos with maturity times prior to or equal to the truncated mtime are selected.
-time mtime, -tm mtime	identifies a time (mtime) in a form suitable for input to the subroutine convert_date_to_binary_. This optional argument is used in the same manner as the -date optional argument above except that mtime is not truncated.
-alarm, -al	if a memo is being set, this specifies that the memo is to be an alarm. When mature, it is to be printed or executed immediately (or as soon as alarms are enabled) and then deleted. If memos are being selected, this argument selects any memos that are alarms.

-repeat intvl, -rp intvl	identifies the interval (where intvl must be greater than or equal to 1 minute) at which this memo is to appear. This optional argument is used when setting a memo. When the memo is mature, an identical memo is set with a maturity time that is "interval" in the future. The interval specification should be in the format of the offset field suitable for input to convert_date_to_binary_ and must be no more than 32 characters in length.
-invisible, -iv	specifies that the memo is never to be mature and will never be printed during a normal memo print. (Used only when setting a memo.)
-call	if a memo is being set, this argument specifies that the memo is to be passed to the command processor as a command. If memos are being selected, this argument selects any memos that are such calls.
-match string	selects memos containing substrings matching all of the strings (where each string is a character string). The remainder of the command line is interpreted as the set of the strings to be matched. The maximum number of strings that can be specified is 32, and the maximum length of any one string is 32 characters.
3. memo_text	is the text of the memo being set.

Notes

If the -pathname control argument is used, the argument that follows must be the pathname of the memo segment that is to be used. If a memo segment is specified by this means, it continues to be used for the duration of the user's process, unless changed again by the -pathname control argument. If a segment with the specified pathname does not exist, memo attempts to create it.

To set a memo, no control arguments are given. Any of the optional arguments except -match and memo_number can be used to specify the type of memo being set and the time it is to mature. If no maturity time or date is specified, the maturity time is assumed to be the current time.

If memo is invoked with no arguments or only the -brief control argument, then all mature memos are printed or passed to the command processor. Alarms are enabled, and any alarms pending are printed or executed.

If either the -print or -list control argument is given, then all memos selected by the optional arguments are printed. The contents of the memo segment do not change in any way, and memos that would ordinarily be passed to the command processor are printed instead. If no optional arguments are used to select which memos are to be printed or listed, then all memos are printed or listed. If the -date or -time optional arguments are given, then only those memos that mature before the specified date or time are printed. If the -call argument is given, any such memo is printed. If the -alarm argument is given, any alarm memos are printed.

If the -delete control argument is used, memos selected by the optional argument are deleted. If no optional arguments have been used to specify which memos are to be deleted, then none are deleted.

The -off control argument is useful for times when the user does not wish any extraneous output, such as when using the Multics runoff command. The command line memo -on can be given to reenable alarms after they have been turned off, or it can be used at login or new_proc time to enable alarms without printing or executing other mature memos. Memo alarms are enabled by memo, memo -brief and memo -on commands, only.

Examples

In the following sequence of memo examples, input typed by the user is marked by an exclamation mark (!). Ready messages from the system are omitted. First, the user's memo segment is initialized and is demonstrated to have no mature memos. Four memos are set and then listed, first in their entirety, then alarm memos, then only mature memos, then all memos maturing before a specified date. Finally, the only mature memo is deleted, and its successful deletion is demonstrated. The time of the example is 5/15/73 1729.

```
! memo
  memo:  Creating >udd>Demo>Jones>Jones.memo.


! memo
  No memos.


! memo get bookshelves
! memo -alarm -date 5/23/73 -repeat 2weeks Staff meeting at two.
! memo -call -alarm -date 6/1/73 -repeat 1month list -dtem -rv
! memo -time "Thursday 9am" -repeat 1week Weekly report due Friday.


! memo -list
   1) Tue 05/15/73 1729  get bookshelves
   2) Wed 05/23/73 0000  Staff meeting at two.  (alarm, "2weeks")
   3) Fri 06/01/73 0000  list -dtem -rv  (call, alarm,  "1month")
   4) Thu 05/17/73 0900  Weekly report due Friday.  ("1week")


! memo -list -alarm
   2) Wed 05/23/73 0000  Staff meeting at two.  (alarm, "2weeks")
   3) Fri 06/01/73 0000  list -dtem -rv  (call, alarm, "1month")
```

```
!  memo
    1) get bookshelves


!  memo -print -date 5/30/73
   get bookshelves
   Staff meeting at two.
   Weekly report due Friday.


!  memo -delete -match book


!  memo
   No memos.
```

This page intentionally left blank.

Name: merge

The merge command provides a generalized file merging capability, which is specialized for execution by user-supplied parameters. The basic function of the merge is to read one or more input files of records, which are ordered according to the values of one or more key fields (i.e., the files have been sorted using the sort command), merge (collate) those records according to the values of those key fields, and write a single file of ordered (or "ranked") records.

For a detailed description of both the sort and merge commands, refer to the Multics Sort/Merge Reference Manual, Order No. AW32.

Name: move

The move command causes a designated segment or multisegment file (and its access control list (ACL) and all names on the designated file) to be moved to a new position in the storage system hierarchy.

Usage

```
move path11 path21 ... path1n -path2n- -control_arg-
```

where:

1.	path1*i*	is the pathname of the segment or multisegment file to be moved.
2.	path2*i*	is the pathname to which path1*i* is to be moved. If the last path2 segment is not given, path1*n* is moved to the working directory, and is given the entryname path1*n*.
3.	control_arg	can be -brief or -bf. This argument causes the messages "Bit count inconsistent with current length..." and "Current length is not the same as records used..." to be suppressed.

Notes

The star and equal conventions can be used.

When an entry is moved, it is given all of the names that the path1*i* argument already has plus the entryname specified in the path2*i* argument.

Since two entries in a directory cannot have the same entryname, special action is taken by this command if the creation of a segment or multisegment file would introduce a duplication of names within the directory. If an entry with the entryname path2*i* already exists in the target directory and this entry has an alternate name, the conflicting name is removed and the user is informed of this action; the move then takes place. If the entry having the entryname path2*i* has only one name, the entry must be deleted in order to remove the name. The user is asked if the deletion should be done; if the user answers "no", the move does not take place.

Read access is required for path1*i*. Status and modify permissions are required for the directory containing path1*i*. Status, modify, and append permissions are required for the directory containing path2*i*.

If path*i* is protected by the safety switch, the user is asked whether path*i* is to be deleted after it has been copied.

The initial ACL of the target directory has no effect on the ACL of the segment or multisegment file after it has been moved. The ACL remains exactly as it was in the original directory.

Example

```
     move alpha >Doe>= >Doe>beta b
```

moves alpha from the current working directory to the directory >Doe, keeping the name alpha, and moves beta from the directory >Doe to the current working directory with the names b and beta.

Name: move_quota, mq

The move_quota command allows a user to move records of quota between two directories, one immediately inferior to (contained in) the other.

Usage

```
     move_quota path1 quota_change1 ... pathn quota_changen
```

where:

1.	pathi	is the pathname of a directory. The quota change takes place between this branch and its containing directory. A pathi of -wd or -wdir specifies the working directory. The star convention cannot be used.
2.	quota_changei	is the number of records to be moved between the immediately superior (containing) directory quota and the pathi quota. The quota_change argument can be either a positive or negative number. If it is positive, the quota is moved from the containing directory to pathi; if it is negative, the move is from pathi to the containing directory.

Notes

The user must have modify permission on both the directory specified by pathi and its containing directory.

After the change, the quota on pathi must be greater than or equal to the number of records used in pathi unless the change makes the quota zero.

If the change makes the quota on pathi zero, there must be no immediately inferior directory with nonzero quota, and the records used and the record-time product for pathi are reflected up to the superior directory.

If pathi is an upgraded directory (its access class is greater than the access class of its containing directory), quota_changei must be positive. Quota can only be moved back to the containing directory of an upgraded directory by deleting the upgraded directory.

Quota may not be moved between a master directory and its containing directory. It may, however, be moved between a master directory and an inferior directory as described above. See the set_mdir_quota command in the MPM Subsystem Writers' Guide for information on changing the quota on a master directory.

Example

```
     move_quota >udd>m>Smith>sub1_dir 1000 >udd>m>Smith>sub1_dir>sub2_dir -50
```

adds 1000 records to the quota on >udd>m>Smith>sub1_dir and subtracts 1000 records from the quota on >udd>m>Smith. It then subtracts 50 records from the quota on >udd>m>Smith>sub1_dir>sub2_dir and adds 50 records to the quota on >udd>m>Smith>sub1_dir.

This page intentionally left blank.

<u>Name</u>: new_proc

This command destroys the user's current process and creates a new one, using the control arguments given initially with the login command, and the optional argument to the new_proc command itself. Just before the old process is destroyed, the "finish" condition is signalled. The default on unit closes all open files and returns. The search rules, I/O attachments, and working directory for the new process are as if the user had just logged in.

Usage

```
     new_proc  -control_arg-
```

where control_arg may be -authorization XX or -auth XX to create the new process at authorization XX, where XX is any authorization acceptable to the convert_authorization_ subroutine. (See the convert_authorization_ subroutine in the MPM Subroutines.) The authorization must be less than or equal to both the maximum authorization of the process and the access class of the terminal. The default is to create the new process at the same authorization.

Note

If the user's initial working directory contains a segment named start_up.ec, and the user did not log in with the -no_start_up control argument, new_proc causes the command:

```
     exec_com start_up new_proc interactive
```

to be automatically issued in the new process. This feature can be used to initialize per-process static variables.

Name: page_trace, pgt

The page_trace command prints a recent history of page faults and other system events within the calling process.

Usage

```
page_trace -count- -control_arg-
```

where:

1. count prints the last count of system events (mostly page faults) recorded for the calling process. If count is not specified, then all the entries in the system trace list for the calling process are printed. Currently, there is room for approximately 350 entries in the system trace list.

2. control_arg is either -long or -lg to print full pathnames, where appropriate. The default is to print only entrynames.

Output

The first column of output describes the type of trace entry. An empty column indicates that the entry is for a page fault. The second column of output is the real time, in milliseconds, since the previous entry's event occurred. The third column of output contains the page number for entries, where appropriate. The fourth column gives the segment number for entries, where appropriate. The last column is the entryname (or pathname) of the segment for entries, where appropriate.

Whenever the real time between successive entries is greater than one second, a blank line is printed between the entries. This blank line usually appears between interactions, where the user interposes a think time longer than one second, and on long running programs, between scheduling quanta.

Notes

Since it is possible for segment numbers to be reused within a process, and since only segment numbers (not entrynames or pathnames) are kept in the trace array, the entrynames and pathnames associated with a trace entry may be for previous uses of the segment numbers, not the latest ones. In fact, the entry and pathnames printed are the current ones appropriate for the given segment number.

For completeness, events occurring while inside the supervisor are also listed in the trace. The interpretation of these events sometimes requires detailed knowledge of the system structure; in particular, they may depend on activities of other users. For many purposes, the user will find it appropriate to identify the points at which he enters and leaves the supervisor and ignore the events in between.

Typically, any single invocation of a program does not induce a page fault on every page touched by the program, since some pages may still be in primary memory from previous uses or use by another process. It may be necessary to obtain several traces to fully identify the extent of pages used.

Name: pl1

The pl1 command invokes the PL/I compiler to translate a segment containing the text of a PL/I source program into a Multics object segment. A listing segment is optionally produced. These results are placed in the user's working directory. This command cannot be called recursively. For information on PL/I, refer to the Multics PL/I Language manual, Order No. AG94.

Usage

```
pl1 path -control_args-
```

where:

1. path	is the pathname of a PL/I source segment that is to be translated by the PL/I compiler. If path does not have a suffix of pl1, then one is assumed. However, the suffix pl1 must be the last component of the name of the source segment.
2. control_args	can be chosen from the following list of control arguments:
-source -sc	produces a line-numbered, printable ASCII listing of the program.
-symbols -sb	produces a source program listing (like the -source control argument), followed by a list of all the names declared in the program with their attributes.
-map	produces a source program listing with symbols (like the -symbols control argument), followed by a map of the object code generated by the compilation. The -map control argument produces sufficient information to allow the user to debug most problems online.
-list, -ls	produces a source program listing with symbols (like the -symbols control argument), followed by an assembly-like listing of the compiled object program. Use of the -list control argument significantly increases compilation time and should be avoided whenever possible by using the -map control argument.
-brief, -bf	causes error messages written into the error_output I/O switch to contain only an error number, statement identification, and, when appropriate, the identifier or constant in error. In the normal, nonbrief mode, an explanatory message of one or more sentences is also written, followed, in most cases, by the text of the erroneous statement.

-severityi, -svi	causes error messages whose severity is less than i (where i is 1, 2, 3, or 4) to not be written into the error_output switch although all errors are written into the listing. The default value for i is 1. For a description of severity levels, see "Error Diagnostics" below.
-check, -ck	is used for syntactic and semantic checking of a PL/I program. Only the first three phases of the compiler are executed. Code generation is skipped as is the manipulation of the working segments used by the code generator.
-optimize, -ot	invokes an extra compiler phase just before code generation to perform certain optimizations, such as the removal of common subexpressions, which reduce the size and execution time of the object segment. Use of this control argument adds 10% to 20% to the compilation time.
-table, -tb	generates a full symbol table for use by symbolic debuggers; the symbol table is part of the symbol section of the object program and consists of two parts: a statement table that gives the correspondence between source line numbers and object locations and an identifier table containing information about every identifier actually referenced by the source program. This control argument usually causes the object segment to become significantly longer.
-brief_table, -bftb	generates a partial symbol table consisting of only statement labels for use by symbolic debuggers. The table appears in the symbol section of the object segment produced for the compilation. This control argument does not significantly increase the size of the object program.
-profile, -pf	generates additional code to meter the execution of individual statements. Each statement in the object program contains an additional instruction to increment an internal counter associated with that statement. After a program has been executed, the profile command can be used to print the execution counts. See the description of the profile command in this document.

The following control arguments are available, but are probably not of interest to every user.

-debug, -db	leaves the list-structured internal representation of the source programs intact after a compilation. This control argument is used for debugging the compiler. The command pl1$clean_up can be used to discard the list structure.

-time, -tm	prints a table after compilation, a table giving the time (in seconds), the number of page faults, and the amount of free storage used by each of the phases of the compiler. This information is also available from the command pl1$times invoked immediately after a compilation.

Further information on the above control arguments is contained under "Error Diagnostics" and "Listing" below.

Notes

The only result of invoking the pl1 command without control arguments is to generate an object segment.

A successful compilation produces an object segment and leaves it in the user's working directory. If an entry with that name existed previously in the directory, its access control list (ACL) is saved and given to the new copy. Otherwise, the user is given re access to the segment with ring brackets v,v,v where v is the validation level of the process that is active when the object segment is created.

If the user specifies the control arguments -source, -symbols, -map, or -list, the pl1 command creates a listing segment in the working directory and gives it a name consisting of the entryname portion of the source segment with the suffix list rather than pl1 (e.g., a source segment named valid.pl1 would have a listing segment named valid.list). The ACL is as described for the object segment except that the user is given rw access to the newly created segment. Previous copies of the object segment and the listing segment are replaced by the new segments created by the compilation.

Because of the Multics standard that restricts the length of segment names, a PL/I source segment name cannot be longer than 27 characters.

Error Diagnostics

The PL/I compiler can diagnose and issue messages for about 350 different errors. These messages are graded in severity as follows:

1	Warning only. Compilation continues without ill effect.
2	Correctable error. The compiler remedies the situation and continues, probably without ill effect. For example, a missing end statement can be and is corrected by simulating the appending of the string ";end;" to the source to complete the program. This does not guarantee the correct results, however.

3	An uncorrectable but recoverable error. That is, the program is definitely in error and cannot be corrected but the compiler can and does continue executing up to the point just before code is generated. Thus, any further errors are diagnosed. If the error is detected during code generation, code generation is completed although the code generated is not correct.
4	An unrecoverable error. The compiler cannot continue beyond this error. The message is printed and then control is returned to the pl1 command unwinding the compiler. The command writes an abort message into the error_output I/O switch and returns to its caller.

Error messages are written into the error_output I/O switch as they occur. Thus, a user at his terminal can quit his compilation process immediately when he sees something is amiss. As indicated above, the user can set the severity level so that he is not bothered by minor error messages. He can also specify the -brief control argument so that the message is shorter. An example of an error message in its long form is:

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

If the -brief control argument is specified, the user sees instead:

```
ERROR 158, SEVERITY 2 ON LINE 30
"zilch"
```

If the user had set his severity level to 3, he would have seen no message at all.

Once a given error message has been printed on the user's terminal in the long form, all further instances of that error message use the brief mode.

If a listing is being produced, the error messages are also written into the listing segment. They appear, sorted by line number, after the listing of the source program. No more than 100 messages are printed in the listing.

Listing

The listing created by the pl1 command begins with a line-numbered image of the source segment. This is followed by a table of all of the names declared within the program. The names are categorized by declaration type as follows:

1. declare statement
2. explicit context (labels, entries, and parameters)
3. implicit context

Within these categories, the symbols are sorted alphabetically and then listed with their location; storage class; data type; size or precision; level; attributes such as initial, array, internal, external, aligned, and unaligned; and a cross-reference list. Next is a table of the program's storage requirements, followed by a listing of external operators used, external entries called, and external variables referenced by the program. The symbol listing is followed by the error messages, if any.

The object code map follows the list of error messages. This table gives the starting location in the text segment of the instructions generated for statements starting on a given line. The table is sorted by ascending storage locations.

Finally, the listing contains the assembly-like listing of the object segment produced. The executable instructions are grouped under an identifying header that contains the source statement that produced the instruction. Operation code, base-register, and modifier mnemonics are printed beside the octal instruction. If the address field of the instruction uses the IC (self-relative) modifier, the absolute text location corresponding to the relative address is printed on the remarks field of the line. If the reference is to a constant, the octal value of the first word of the constant is also printed. If the address field of the instruction references a symbol declared by the user, its name appears in the remarks field of the line.

Name: pl1_abs, pa

This command submits an absentee request to perform PL/I compilations. The absentee process for which pl1_abs submits a request compiles the segments named, appends the output of print_link_info for each segment to the segment path.list if it exists, and dprints and deletes path.list. If the -output_file control argument is not specified, an output segment, path.absout, is created in the user's working directory (if more than one path is specified, only the first is used). If none of the segments to be compiled can be found, no absentee request is submitted. The print_link_info command is described in the MPM Subsystem Writers' Guide.

Usage

```
     pl1_abs paths -pl1_args- -dp_args- -abs_control_args-
```

where:

1. paths	are the pathnames of segments to be compiled.
2. pl1_args	may be one or more control arguments accepted by the pl1 command.
3. dp_args	may be one or more control arguments (except -delete) accepted by the dprint command.
4. abs_control_args	may be one or more of the following control arguments:
-queue _n_, -q _n_	specifies in which priority queue the request is to be placed ($n \leq 3$). The default queue is 3; path.list is also dprinted in queue _n_.
-hold	specifies that pl1_abs should not dprint or delete path.list.
-output_file path, -of path	specifies that absentee output is to go to the segment whose pathname is path.

*

Notes

Control arguments and segment pathnames can be mixed freely and can appear anywhere on the command line after the command. All control arguments apply to all segment pathnames. If an unrecognizable control argument is given, the absentee request is not submitted.

Unpredictable results can occur if two absentee requests are submitted that could simultaneously attempt to compile the same segment or write into the same absout segment.

When doing several compilations, it is more efficient to give several segment pathnames in one command rather than several commands. With one command, only one process is set up. Thus the dynamic intersegment links that need to be snapped when setting up a process and when invoking the compiler need be snapped only once.

Name: print, pr

The print command prints a specified ASCII segment on the user's terminal. Unless the user specifies a range of line numbers, the command prints the entire segment. Multisegment files cannot be printed by invoking the print command.

Usage

```
     print path -optional_args-
```

where:

1.	path	is the pathname of the segment to be printed.
2.	optional_args	can be selected from the following:
	begin	is the line number that identifies where printing begins and is optional. If it is not specified, printing starts on the first line of this segment (after an identifying header, see "Notes" below). If begin is not specified, then end cannot be specified either.
	end	is the line number that identifies where printing ends; it is optional and if not specified, printing ends with the last line of the segment followed by two blank lines and a ready message. If the line number specified is greater than the number of lines in the segment, the entire segment is printed.

Notes

If neither a beginning line nor an ending line is supplied, a short identifying header followed by two blank lines is printed preceding the printing of the segment. This header is suppressed whenever begin is nonnull. See "Examples" below.

The command assumes that newline characters are appropriately embedded in the text. Output is written through the I/O switch, user_output, which is usually directed to the user's terminal.

The user must have read access to the segment to be printed.

Examples

```
     print alpha
```

prints the segment alpha in the user's working directory in its entirety.

```
     print alpha 1
```

has the same effect, but omits the identifying header.

```
     print alpha 10 20
```

prints lines 10 through 20 of the segment.

```
     print alpha 10
```

prints lines 10 through the end of the segment.

```
     print alpha 1 10
```

prints the first ten lines of the segment.

Name: print_attach_table, pat

This command prints information on the user's terminal about the I/O switch name associations created by attach calls in the user's current ring.

Usage

```
print_attach_table -switch_names-
```

where switch_names are the names of I/O switches about which the user wishes to obtain information. Information about only the specified switches is printed. If a specified switch is not currently attached, a message to that effect is printed on the user's terminal.

Notes

The attach and open operations associated with the indicated switch names are printed. If no arguments are specified, the information for all switches currently attached is printed.

For further information, refer to the description of the io_call command.

This page intentionally left blank.

Name: print_auth_names, pan

The print_auth_names command prints the names of the sensitivity levels and access categories defined for the installation. Only the names that can be used to describe an access class or access authorization between system low and system high are printed, unless the -all control argument is given.

Usage

```
print_auth_names -control_args-
```

where control_args can be chosen from the following:

-level	lists the sensitivity levels. (This is the default)
-category, -cat	lists the access categories. (This is the default)
-brief, -bf	suppresses the title and headings.
-all, -a	lists all possible names (above system high).

Note

This command lists the names that are acceptable to the convert_authorization_ subroutine (described in the MPM Subroutines) to define an access class or access authorization. (All commands and system interfaces that use a character string to describe an access class use this subroutine.) Both the long and short names are printed.

Name: print_default_wdir, pdwd

The print_default_wdir command prints out the pathname of the current default working directory on the user's terminal.

Usage

```
     print_default_wdir
```

Note

See also the descriptions for the change_wdir and change_default_wdir commands.

Name: print_messages, pm

The print_messages command prints any interprocess messages that were received (and saved in the user's mailbox) while the user was not accepting messages. (For a description of the mailbox, refer to the accept_messages and mail commands.)

Usage

```
     print_messages -control_arg-
```

where control_arg can be -last or -lt to reprint only the latest message received. This feature is useful for repeating a garbled message.

Notes

Messages are deleted after they are printed; however, the "last" message remains available for the life of the process.

If messages are deferred, it is a good practice to print out pending messages periodically.

Name: print_motd, pmotd

The print_motd command is intended to be used within a start_up.ec segment. It prints out changes to the message of the day since the last time the command was called. For a description of the function of a start_up.ec segment see "Protocol for Logging In" in Section I of the MPM Reference Guide.

Usage

```
print_motd
```

Notes

When a user logs in, the current message of the day is first compared with the segment Person_id.motd in his home directory. Next, all lines that have been appended or modified since he last saw the message are printed on his terminal. Then, the Person_id.motd segment is updated for use the next time print_motd is invoked.

If the segment Person_id.motd does not exist, print_motd attempts to create it, prints the current message of the day, and initializes Person_id.motd.

Name: print_proc_auth, ppa

The print_proc_auth command causes the access authorization of the current process and current system privileges (if any) to be printed on the terminal.

Usage

```
     print_proc_auth -control_args-
```

where the following optional control_args may be specified:

-long, -lg	prints the installation-defined long names (up to 32 characters) for the sensitivity levels and categories.
-all, -a	prints the maximum access authorization of this process.

Notes

If the -long control argument is not specified, the access authorization printed is composed of the installation-defined short names (eight characters or less) for sensitivity levels and categories.

The maximum authorization printed for the -all control argument is the maximum authorization that this process could have been given at login, and corresponds to the maximum access class of upgraded directories that may be created by this process.

Name: print_request_types, prt

The print_request_types command prints a list of all request types handled by the I/O daemon. For each request type, two items of information are provided: the access name of the I/O daemon driver process that performs requests of that type, and the generic type to which the request type belongs. An asterisk (*) immediately preceding a request type indicates that the request type is the default for its generic type.

Usage

```
     print_request_types -control_args-
```

where control_args can be one or more of the following:

-brief, -bf	suppresses printing of a heading line.
-access_name XX, -an XX	specifies that only those request types having an access name of XX are to be listed; XX must be of the form Person_id.Project_id.
-gen_type XX, -gt XX	specifies that only these request types of generic type XX are to be listed.

Note

The access name IO.SysDaemon indicates a standard request type available to all users. Any other access name indicates a nonstandard request type that is generally not available to all users.

The generic type of a request type determines which commands can be used to submit requests of that request type. For example, the dprint command uses only request types of generic type "printer". The dpunch command uses only request types of generic type "punch".

Name: print_search_rules, psr

The print_search_rules command prints the search rules currently in use.

Usage

```
print_search_rules
```

Note

See also the description of the set_search_rules, add_search_rules, and delete_search_rules commands. The standard search rules are described under "System Libraries and Search Rules" in Section III of the MPM Reference Guide.

Name: print_wdir, pwd

The print_wdir command prints the pathname of the current working directory on the user's terminal. A working directory is a directory in which the user's activity is centered. Its pathname is remembered by the system so that the user need not type the absolute pathname of segments inferior to that directory.

Usage

```
     print_wdir
```

Note

See the descriptions of the change_wdir and change_default_wdir commands. See also "The System Libraries and Search Rules" in Section III of the MPM Reference Guide.

Name: probe, pb

The probe command provides symbolic, interactive debugging facilities for programs compiled with PL/I, FORTRAN, or COBOL. Its features permit a user to interrupt a running program at a particular statement, examine and modify program variables in their initial state or during execution, examine the stack of block invocations, and list portions of the source program. External subroutines and functions may be invoked, with arguments as required, for execution under probe control. The probe command may be called recursively.

Usage

```
probe -procedure_name-
```

where procedure_name is an optional argument that gives the symbolic name of an entry to the procedure or subroutine that is to be examined with probe. It can take the form reference_name$offset_name. If no procedure_name argument is specified, the procedure owning the frame in which the last condition was raised is assumed, if one exists; otherwise, an error is reported.

Overview of Processing

The probe command is generally used to examine an active program at points where execution has been suspended by one of the following:

1. Breakpoint. Execution is temporarily halted at a point selected by the user and probe entered directly. Debugging requests associated with the breakpoint are automatically carried out and/or requests issued from the user's terminal. Program execution can be resumed at the point of interruption.

2. Error. An error such as zerodivide or subscriptrange can interrupt program execution. After an error message is printed, a new command level is established. The user can then call probe to examine the state of the program.

3. Quit signal. A run-away or looping program can be stopped by issuing a quit signal. A new command level is established and the user can call probe to determine the source of the problem.

In all of these cases, variables of all storage classes (including automatic) are accessible.

The probe command can also be used to examine a nonactive program--one that has never been run or that has completed execution--by specifying a procedure_name argument in the command line. In this case, the user can examine static variables and the program source. However, the most common use is to set breaks before actually running the program.

A program to be debugged with probe must have a standard symbol table that contains information about variables defined in the program and a statement map that gives the correspondence between source statements and object code. A symbol table and statement map are produced for the languages supported if the -table control argument is given at compilation. (A program may also be compiled with the -brief_table control argument, which produces only a statement map. The variables of a program compiled in this way cannot be examined with probe; however, the user may retrieve information about source statements and where the program was interrupted and also may set breakpoints at particular statements.)

Information about programs being debugged is stored by probe in a segment in the user's home directory called Person_id.probe where Person_id is the user's log-in name. This segment is created automatically when needed.

Probe Pointers

Three internal "pointers" are used by probe to keep track of the program's state. They are:

source pointer	indicates the current source-program statement
block pointer	indicates the current block
control pointer	indicates the current control point

These values are affected by certain probe requests. A user can, for example, position the source pointer to a particular statement, then list a portion of the source program beginning at that point.

The block pointer serves two purposes. It identifies the procedure, subprogram, or begin block whose variables are to be examined. Further, it specifies the stack frame associated with the block and is used to distinguish among different occurrences of an automatic variable in a recursively invoked procedure. The control pointer marks the point at which a program is suspended.

The initial values of these pointers are determined as described below. If a procedure_name argument is given in the command line and if the designated program is active, the control and source pointers are set to the last statement executed, and the block pointer is set to the most recent invocation of the procedure. If the designated program is not active, then the control and source pointers are set to the entry statement, and the block pointer to the outermost block (but with no active frame).

If no procedure_name argument is given and the default rule applies (i.e., a condition has been raised), then the procedure in which the condition was raised is used. The source and control pointers are set to the statement where the condition was raised, and the block pointer to the block containing that statement.

Similarly, when probe is entered because of a breakpoint encountered during the execution of a program, the source and control pointers are set to the statement at which the break has been set; and the block pointer to the block containing that statement.

Breakpoints

A breakpoint causes a temporary interruption of program execution, during which debugging operations can be performed. Using probe requests, a user can set a breakpoint before or after any statement and can associate a list of probe requests with the break. When the break is encountered during execution, probe is entered and the list of requests interpreted automatically. These requests might, for example, display the value of a variable or alter its value (effectively allowing source level patching of the program), tell what line was just executed, or cause probe to read a list of requests from the terminal to permit the user to interactively examine the state of his program. When the request list associated with the break is exhausted, the execution of the program is resumed from the point at which it was interrupted.

The implementation of a breakpoint by probe consists of patching a call to the probe command into the appropriate location in the object segment of the program. As a result, there need not be an active invocation of probe for a break to occur; also, breakpoints may be set in a program before it is run, while the program is suspended by another break, or before a program interrupted by a quit signal or error condition has been restarted.

Probe Requests

A probe request consists of a keyword (or its abbreviation) that specifies the desired function and any arguments required by the particular request.

A series of requests may be given in the form of a request list. Here, individual requests are separated by semicolons or newline characters.

A single request or a parenthesized request list may be preceded by a conditional predicate whose value determines if and when the requests it modifies will be executed.

The following pages present the format and function of each probe request. Requests are grouped according to function. Required arguments are indicated for each. The syntax and semantics of generic arguments such as expressions, procedures, labels, and variables are defined under separate headings following the request descriptions.

The following descriptions first give the name of the request and its abbreviated form (if any). This line is followed by the general format line(s) of the request.

BASIC REQUESTS

1. value, v

```
value  expression
value  cross-section
```

The request "value expression" causes the value of the given expression to be displayed. Allowable expressions are variables, builtin functions such as addr and octal, and the value returned by an external function. The evaluation of expressions is described later (following the descriptions of all the requests) under "Evaluation of Expressions."

Examples:

```
value var
value p -> a.b(j).c
value addr (i)
value octal (ptr)
value function (2)
```

The request "value cross-section" is used to display values contained in a cross-section of an array. A cross-section is specified by giving the upper and lower bounds of one or more subscripts, as in:

```
value array (1:5, 1)
```

The notation 1:5 indicates the range one through five for the first subscript. The example above prints array(1,1), array(2,1), ..., array(5,1). More than one dimension can be iterated; for instance, array(1:2,1:2) prints, in order, array(1,1), array(1,2), array(2,1), array(2,2).

2. let, l

```
let variable = expression
let cross-section = expression
```

This request sets the specified variable or array elements to the value of the expression. If the variable and the expression are of different data types, conversion is performed according to the rules of PL/I. Array cross-sections are expressed as shown in the value request above. One array cross-section may not be assigned to another.

Examples:

```
let var = 2
let array (2,3) = i + 1
let p -> a.b(1:2).c = 10b
let ptr = null
```

Because of compiler optimization, the change may not take immediate effect in the program, though the value request shows the value to be altered.

3. call, cl

    call procedure (arg1, ..., argn)

This request calls the procedure named with the arguments given. If the procedure expects arguments of a certain type, those given are converted to the expected type; otherwise, they are passed without conversion. The value request (see above) can be used to invoke a function, with the same sort of argument conversion taking place. If the procedure has no arguments, a null argument list, "()", must be given.

Examples:

```
call sub ("abc", p -> p2 -> bv, 25o, addr(j))
call sub_noargs ()
value function ("010"b)
```

4. goto, g

    goto label

This request transfers control from probe to the statement specified and initiates program execution at that point.

Examples:

goto label_var	transfer to value of label variable
goto action (3)	transfer to label constant
goto 29	transfer to statement on line 29 of current program
goto $110	transfer to line with label 110 in the FORTRAN program
goto $c,1	transfer to the statement following the current statement

Because of compiler optimization, unpredictable results may occur when using this request.

5. quit, q

    quit

This request causes a return to command level.

6. continue, c

    continue

This request restarts a program that has been suspended by a break. If this request is issued in any other context, probe returns to its caller (generally command level).

SOURCE REQUESTS

1. source, sc

   source n

This request displays one or more statements beginning with the current statement (i.e., the source pointer). If n is not specified, one line is printed; otherwise, n lines are printed. Only executable statements for which code has been generated can be listed; however, if a range of statements is requested, intervening text, such as comments and nonexecutable statements (for example, declarations), is included in the output.

2. position, ps

   position label
   position ±n

This request sets the source pointer to the statement indicated by label or to an executable statement relative to the current statement as indicated by the value of n. If +n is given, the pointer is set forward n statements; if -n is given, the pointer is set back n statements. If no label or offset is given, the statement designated by the control pointer is assumed.

Examples:

position here	set the source ptr to the statement labeled here
position action (3)	to the statement labeled action (3)
position 2-14	to the statement on line 14 of include file 2 of the program
position +2	move forward two statements in the source
position -5	move back five statements

The position request can also be used to search for an executable statement that contains a specified string, using the form:

```
position "string"
```

The search begins with the statement following the current statement and continues through the program, if necessary, until the current statement is again reached. If a match is found, the source pointer is set to that statement. If the specified string contains a quotation mark, it must be doubled when given in the request line. Because statements are reordered by the compiler, the search may not necessarily find statements in the same order as the source listing of the program would indicate.

Examples:

position "write (6,10)"	locate the statement in the program
position "str = ""a"	locate str = "a
position "q+2"; source	locate and print the statement

SYMBOL REQUESTS

1. stack, sk

    stack i, n all

This request traces the stack backward beginning at the ith frame and continuing for n frames. If i is not given, then the trace begins with the most recent frame and continues for n frames. If no limits are given, the entire stack is traced. The trace lists all active procedures and block invocations (including quick blocks) beginning with the most recent. For each block, a frame or level number is given, as is the name of any conditions raised in the frame.

Examples:

stack	trace the whole stack
stack 2	trace the two most recent frames
stack 3, 2	trace the third and second frames

Normally, system or subsystem support procedures are not included in the stack trace. These may be included by specifying "all".

Examples:

stack all	trace the whole stack including all support stack frames
stack 5,3 all	trace the fifth, fourth, and third frames including all support stack frames

2. use, u

    use block

This request selects the block to be used for subsequent probe requests. It may be specified by the name of an entry, a label, or a stack frame number (level i). If no block is specified, then the block originally used (when probe was entered) is assumed. The block pointer is set to the specified block so that variables in that block can be referenced. In addition, the source pointer is set to the last statement executed in the block. In this way, the point at which the block exited can be found through use of the source request. Acceptable block specifications include:

    procedure_name
    label
    level i
    -n

In this context, procedure_name is the name of a procedure or subprogram entry point whose frame is desired; its usage is essentially the same as if used on the command line. A label denotes the block that contains the statement identified by the label or line number; for instance, the label on a begin statement denotes that begin block. If the label's block is not active, the source pointer is set to the statement specified. The block specification level i uses the block with level number i from a stack trace; -n uses the nth previous instance of the current block, allowing one to move back to a previous recursion level. If more frames are requested than actually exist, the last one found is used.

Examples:

use sub	use the block that procedure sub occupies
use label	use the block that contains the statement labeled label
use level 2	use the second frame in the stack trace
use -1	use the previous instance of the current block
use -999	use the last (oldest) instance

When a level is specified, the last trace mode (support procedures included or excluded) specified is used to find the level requested.

3. symbol, sb

   symbol identifier

This request displays the attributes of the variable specified and the name of the block in which its declaration is found. If the size or dimensions of the variable are not constant, an attempt is made to evaluate the size or extent expression; if the value is not available, an asterisk (*) is used instead.

4. where, wh

```
where  source
where  block
where  control
```

This request displays the current value of one or all of the pointers. Source and control give the statement number of the corresponding statement. Block gives the name of the block currently being used; if the block is active, its level number is also given. If neither source, block, or control appears, the information for all three is given.

Examples:

where	give the value of all three pointers
where source	give the value of the source pointer

BREAK REQUESTS

1. before, b

```
before label:  request
before label:  (request list)
```

This request sets a breakpoint before the statement specified by label and causes the given request(s) to be associated with the break. If no label is given, the current statement is assumed. If no requests are given, a halt is assumed (see the halt request described below).

When the running program arrives at the statement specified, probe is entered before the statement is executed, and associated requests are processed automatically. When all requests are done, execution of the program resumes at the statement before which the break was set. A breakpoint set before a statement takes effect whether the statement is arrived at in sequence or as the result of a branch or call from some other location.

Examples:

before: (value var; value var2)	set a break before the current statement to display the value of the variables var and var2
before quick: value x	set a break before the statement labeled quick
before	set a break containing the halt request before the current statement

The request list may extend across line boundaries if necessary.

2. after, a

```
after label:  request
after label:  (request list)
```

This request is the same as the before request except that the break is set after the designated statement. This means that the request list is interpreted after the statement has been executed. If the statement branches to another location in the program, the breakpoint does not take effect.

Notice the distinction between two breakpoints in sequence. The one that is <u>after</u> statement x is not effective when control is passed to statement x+1 from elsewhere. The break <u>before</u> statement x+1 <u>does</u> take place.

3. halt, h

    halt

This request causes probe to stop processing its current input and to read requests from the terminal. A new invocation of probe is created with new pointers set to the values at the time the halt request was executed. As part of a break request list, it enables the user to enter requests while a program is suspended by the break. A running program can be halted in this way. A subsequent continue request causes probe to resume what it was doing before it stopped; for example, finish a break request list and resume execution of the program.

Examples:

before 29: halt	causes the program to halt at statement 29 and allows the user to enter probe requests (the continue request can be used to restart the program)
after: (value a; halt; value b)	causes the value of a to be printed before the program halts; later, after the user enters a continue request, the value of b is printed, and the execution of the program is resumed

4. reset, r

```
reset
reset at/after/before label
reset procedure
reset *
```

This request deletes breaks set by the before and after requests. When no argument is supplied, reset deletes the current break. With a label argument, breaks set before and/or after a statement are deleted; with a procedure or asterisk (*) argument, all the breaks in a specified segment or all breaks in all segments, respectively, can be deleted.

Examples:

reset	delete the current break
reset at 34	delete breaks set before and after the first statement on line 34
reset after 34	delete the break set after line 34
reset sub	delete all breaks in sub
reset *	delete all known breaks

5. status, st

```
status
status at/after/before label
status procedure
status *
```

This request gives information about breaks that have been set by the user. The scope of the requests is similar to reset except that status without arguments specifies all breaks in the current program (the program containing the statement designated by the source pointer).

Examples:

status	list the breaks set in the current program
status before label	give the break set before the statement at label
status sub	list the breaks set in sub
status *	list the procedures that have breaks set in them

6. pause, pa

```
pause
```

This request is equivalent to "halt; reset" in a break request list. It causes the procedure to execute a break once and then reset it. If the statement after which the break is set transfers elsewhere, the break does not occur and remains set until encountered sometime in the future or explicitly reset at some other point.

7. step, s

```
step
```

This request enables the user to step through his program one statement at a time. It sets a break consisting of a pause request after the next statement to be executed (as indicated by the control pointer) and resumes the execution of the program as with a continue request.

MISCELLANEOUS REQUESTS

1. mode

```
mode brief
mode long
```

This request turns the brief message mode on or off. In brief mode, most messages generated by probe are shortened and others are suppressed altogether. The default is long.

2. execute, e

```
execute "string"
```

This request passes one or more Multics command lines, represented above by "string", to the command processor for execution.

3. acknowledge

```
.
```

This request causes probe to identify itself by printing "probe" on the terminal. It may be used, for example, to determine if a called procedure has returned.

CONDITIONAL PREDICATES

1. if

```
if conditional expression:  request
if conditional expression:  (request list)
```

The request or request list is executed if the conditional expression is true. The expression must be of the form:

```
expression operator expression
```

where operator can be <=, <, =, ^=, >, or >=.

Example:

```
if a < b: let p = addr (a)
```

This predicate is most useful in a break request list where it can be used to cause a conditional halt. For example,

```
before: if z ^= "10"b: halt
```

causes the program to stop only when z ^= "10"b.

2. while, wl

```
while conditional expression:  request
while conditional expression:  (request list)
```

The request or request list is executed repeatedly as long as the conditional expression is true.

Example:

```
while p ^= null: (value p -> r.val; let p = p -> r.next)
```

<u>Evaluation of Expressions</u>

Allowable expressions include simple scalar variables, constants, and probe builtin functions. The sum and difference of computational (arithmetic and string) values can also be used.

Variables can be simple identifiers, subscripted references, structure qualified references, and locator qualified references. Subscripts are also expressions. Locators must be offsets, pointer variables, or constants.

Examples:

```
running_total
salaries (p -> i - 2)
a.b(2).c(3)
a.b.c(2,3)
x.y -> var
```

Constants can be arithmetic, string, bit, and pointer. Arithmetic constants can be either decimal or binary, fixed or floating point, real or complex. Also, octal numbers are permitted as abbreviations for binary integers (e.g., 12o = 10).

Examples:

```
-123
10b
45.37
4.73e10
2.1-0.3i
12345670o
```

Character and bit strings without repetition factors are allowed. Character strings can include newline characters. Octal strings can be used in place of bit strings (e.g., "123"o = "001010011"b).

Examples:

```
"abc"
"quote""instring"
"1010"b
"01234567"o
```

A pointer constant is of the form:

```
segment_number¦word_offset(bit_offset)
```

where the segment_number and word_offset must be in octal. The bit_offset is optional but if given must be in decimal. The pointer constant can be used as a locator.

Examples:

```
214¦5764
232¦7413(9)
```

Four builtin functions are provided by probe: addr, null, octal, and substr. The function of addr and null is the same as in PL/I: addr takes one argument and returns a pointer to its argument; null, taking no arguments, returns a null pointer. The function octal acts very much like the PL/I unspec builtin function in that it treats its argument as a bit string of the same length as the raw data value and can be used in a similar manner as a pseudo-variable. However, when used in the value request, the value is displayed in octal. Data items not occupying a multiple of three bits are padded on the right. The substr builtin function may be used as a function or pseudo-variable. It takes two or three arguments. The first argument must be a character or bit string or a reference to the octal builtin function; the second and optional third arguments give the offset and length of the desired substring as with the PL/I substr builtin function.

Examples:

For the following examples, assume that p is declared as an aligned pointer, i as fixed binary initial(-2), and cs as character(8) initial ("abcdefgh").

value addr (i)	displays the address of i
let p = null	sets the pointer, p, to null
value octal (i)	displays the storage containing i in octal, giving: 777777777776
value substr (cs, 2, 3)	displays "bcd"
let substr (cs, 4, 1)=""	sets cs to "abc fgh"

Label References

A label identifies a source program statement and can be a label variable or constant, a line number in source-listing format, or one of the following special statement designators:

$c	designates the "current statement"
$b	designates the statement on which the most recent break occurred
$number	designates a FORTRAN label

An optional offset of the form ",<u>s</u>" is also allowed.

Examples:

label	statement at label
label_var	statement to which label_var is set
17	statement on line 17 of program
3-14,2	statement 2 on line 14 of include file 3
$b	statement at which last break occurred
$c,1	statement after current statement
$100	FORTRAN statement labeled 100

Generally, a label can also be the name of a procedure, entry, or subroutine statement.

Procedure References

A procedure_name is an identifier representing an entry variable or constant. External reference names, representing entry points not declared in the current block, can be used.

Evaluation of Variable References

When a variable is referenced in a request, probe first attempts to evaluate it by checking for an applicable declaration in the current block and, if necessary, in its parents. If no declaration is found, the list of builtin functions is searched. Finally, when the context allows a procedure_name, a search is made following the user's search rules.

The block in which a variable reference is resolved can be altered by the use request that sets the current block. For example, if "value var" displays the value of var in the current block, then "use -1; value var" displays the value of var at the previous level of recursion. An optional block specification is available for referencing variables in other blocks:

```
variable [block]
```

where block is the same as in the use request. The use of blocks in this manner does not alter the block pointer.

Examples:

var[-1]	looks for the previous value of var
abc[other_block]	looks in "other_block" for abc
xyz[39]	looks in the block that contains line 39 for xyz
n.m[level 4]	looks in the block at level 4 for n.m
q(2)[sub]	looks in the procedure sub for q(2)

A block specification can be used to qualify a variable reference in any context the variable could be used. However, a block specification on a label or entry constant is ignored unless the relative (-n) format is used and the label or entry is itself used in a block specification. In such a case, it is taken to mean the nth previous instance of the block designated by the label or entry; that is, "var[sub[-2]]" references var in the second previous invocation (third on the stack) of sub.

Sample Debugging Sessions

Two extensive examples are given on the following pages to illustrate both how probe requests are used and how to get useful debugging information out of them. the first example was devised principally to demonstrate the application of probe requests. A listing of the source of the program, test, is given on the next page. The program has been compiled with the -table control argument (line 1). The sample output follows with an exclamation point (!) denoting lines typed by the user. Unless otherwise indicated, line numbers referenced in the following paragraphs are from the sample output.

The user first calls his program (line 5); noticing that it seems to be looping, he stops it by issuing the quit signal (line 6). After the user invokes probe (line 10), it responds by telling him that the internal function fun was executing line 38 when interrupted. Since the source pointer was automatically set to that line, the source request (line 12) causes the current source statement to be displayed. A statement causing an error could be displayed in a similar manner.

```
 1 test: procedure;
 2
 3      declare
 4
 5           (i, j) fixed binary,
 6           1 s structure based (p),
 7             2 num fixed binary,
 8             2 b (n refer (s.num)) float binary,
 9           p pointer, n fixed binary,
10           sysprint file;
11
12
13      n = 5;
14      allocate s set (p);
15
16      do i = 1 to s.num;
17           s.b(i) = fun (i, 1);
18      end;
19      put skip list (s.b);
20
21      do j = s.num to 1 by -1;
22           s.b(j) = fun (-j, -1);
23      end;
24      put skip list(s.b);
25
26      return;
27
28
29      fun: procedure (b, i) returns (float binary);
30
31           declare
32                (b, i) fixed binary;
33
34           if b = 0
35                then return (1);
36                else do;
37                     b = b - i;
38                     return (2**b + fun (b, i));
39                end;
40
41      end fun;
42
43
44 end test;
```

```
    ! pl1 test -table
     PL/I
     r 1248 3.211 28.336 280

    ! test
!(quit)
     QUIT
     r 1250 5.371 6.702 52  level 2, 10

    ! probe
     Condition quit raised at line 38 of fun.
    ! source
               return (2**b + fun (b, i));
    ! stack
      11    command_processor_
      10    release_stack
       9    unclaimed_signal
       8    real_sdh_
       7    return_to_ring_0_
       6    fun                    quit
       5    test
       4    command_processor_
       3    listen_
       2    process_overseer_
       1    user_init_admin_
    ! use level 5
    ! source
             s.b(i) = fun (i, 1);
    ! value s.num
        5
    ! position "i = 1"; source
         do i = 1 to s.num;
    ! after: value i
     Break set after line 16 of test.
    ! quit
     r 1252 1.375 16.394 354  level 2, 10

    ! release
     r 1252 .126 .922 19

    ! test
        1
        1
        1
        1
!(quit)
     QUIT
     r 1252 3.069 .650 25  level 2, 12

    ! release
     r 1253 .092 .937 20

    ! probe test
    ! status
     Break after line 16.
    ! status after 16
     Break after line 16:  value i
    ! reset at 16
     Break reset after line 16 of test.
```

```
 60        ! position 34
 61        ! source
                      if b = 0
 62                            then return (1);
 63
 64        ! before: halt
 65          Break set before line 34 of test.
 66        ! quit
 67          r 1255 .781 12.356 333
 68
 69        ! test
 70          Stopped before line 34 of fun.
 71        ! value b
 72              1
 73        ! where
 74          Current line is line 34 of test.
 75          Using level 6: fun.
 76          Control at line 34 of fun.
 77        ! value i
 78              1
 79        ! c
 80          Stopped before line 34 of fun.
 81        ! stack 5
 82             8    break
 83             7    fun
 84             6    fun
 85             5    test
 86             4    command_processor_
 87        ! value b
 88              0
 89        ! value b[-1]
 90              0
 91        ! value i
 92              1
 93        ! symbol i
 94          fixed binary(17,0) aligned parameter
 95          Declared in fun.
 96        ! use test
 97        ! value i
 98              0
 99        ! reset
100          Break reset before line 34 of test.
101        ! quit
102          r 1307 4.870 64.788 1544
```

The stack command is then used (line 14) to see in what order the procedures were called. The output shows that procedure test was called from command level, and then called fun. While fun was executing, a quit signal was issued and established a new command level.

The use request (line 26) sets the block pointer to the outermost block of procedure test, and the source pointer to the last statement executed in that block--the statement which invoked the function fun.

The source request (line 27) is issued to display the current statement (as set above) to determine from which line of the program (17 or 27) fun was actually invoked.

Since the block pointer has also been set, the user can check the value of "s.num" with the value request (line 28) and ascertain that it is as desired. Since there is no new declaration of "s.num" within the procedure fun, the declaration made in the parent block, test, is known and the value of "s.num" could be displayed without changing the block pointer as would be necessary if there were a conflicting declaration.

The user decides that it is worthwhile to trace the value of i. Rather than recompiling his program with a "put statement" added in a strategic location, probe allows him to set a break containing a value request to accomplish the same thing. The user wants to set the break after the do statement on line 16 of the program and searches for it with the position request (line 31). The source request is used to verify that the correct line was found. The after request is used to actually set the break (line 33). The quit request (line 35) then causes probe to return command level.

To abort the suspended program test, the user invokes the Multics release command (line 38). If he had done this just after issuing the quit signal, he could not have used probe to examine automatic variables inside the program or to determine where the program had been interrupted.

The program is restarted (line 41) but now, after each execution of line 16, the break occurs and probe displays the value of i. Clearly, it is not being incremented as it should. Since this approach is not producing any useful information, the user aborts the program and tries to delete the break. The status request is used to tell what breaks have been set in the procedure test (line 54), and then (line 56) to see the probe request associated with that break. The break is then deleted with the reset request (line 58). If there had also been a "Break before 16", then the request "reset at 16" would have deleted both.

The user next decides to examine fun, so he sets a break that will halt every time fun is invoked (lines 60 through 64). Looking at the listing, he sees that the first statement in fun is on line 34, so he sets the source pointer to that statement with the position request and sets a break to halt the program. To accomplish the same thing, "before 34: halt" could have been used.

The program is called (line 69) and then halts when the break before line 34 is reached. The user displays b and i (lines 71 and 77), getting the values he expected. The where request is also used (line 73) to check on the current state of things. The continue request (line 79) restarts fun, which calls itself recursively and stops again. The stack request (line 81, showing the last five frames) verifies that fact. The user displays the b in the current instance of fun (line 87, at level 7) and in the previous one (line 89, at level 6). Mistakenly expecting the b's at different levels to be different, he gets suspicious. The variable i has the value expected (line 91), but the symbol command (line 93) shows that it is the wrong instance of i--the parameter to fun, not the loop index. To get the correct instance, he must look in the frame belonging to the procedure test (line 96) and display that i (line 97). This i has been set to 0. The user then realizes his error. The function is modifying its argument (the loop index i) on line 37 (line 94). When the user has finished debugging the program, the reset request (line 99) is used to delete the currently active break (the one that just occurred), and the program is aborted with the quit request (line 101).

The preceding example was constructed to give a user a feeling for applying probe requests. The following example is taken from an actual debugging session using probe and illustrates several additional techniques available to the user.

The program of interest is a subroutine, sort_strings, that is supposed to sort a character array of arbitrary dimension; the array is passed as an argument to the subroutine. Since very large strings are being compared, it would be time consuming to exchange the strings themselves. Therefore, an array of pointers to the strings (actually, the indices of the strings in the original array) is first sorted by a simple bubble sort, and the strings moved afterwards into the correct order. There are (at least) two bugs in the program as it appears in the listing. The next two paragraphs further describe the algorithm intended.

A bubble sort involves making repeated passes over an input array, comparing adjacent pairs of values, and interchanging them as necessary. This moves the larger (smaller) values toward the end of the array. The sort only covers that portion of the array that is out of order (i.e., up to the element where the final exchange took place on the previous pass--all elements following this point are clearly correctly arranged). The example below illustrates how a bubble sort works in one case. (The hyphen delimits the end of the search.)

```
Original  First Pass   Second Pass   Third Pass

   d           a            a            -
   a           c            b            a
   c    ->     b     ->     -     ->     b
   b           -            c            c
   e           d            d            d
   -           e            e            e
```

In the sort_strings subroutine (see source listing below), "k" determines the last element of the array needing to be sorted. Sorting continues until no exchanges occurred during the last pass (i.e., until the test, k <= 1, fails). The "order" array contains the indices that are actually sorted.

The reordering method used is to scan for unordered items and then move the entire chain (a replaces b; b replaces c; and c replaces a) containing the element. For example:

```
Initial Ordering   Desired Ordering            Movements

   1      e           3      a        temp <- 1 (e)  temp <- 2 (d)
   2      d           4      b        1 <- 3 (a)     2 <- 4 (b)
   3      a           5      c        3 <- 5 (c)     4 <- temp
   4      b           2      d        5 <- temp
   5      c           1      e
```

All elements that have been moved into the correct location are flagged as having been moved by setting their order values to -1.

Source listings of the program and subroutine, named testss and sort_strings respectively, are given below.

```
testss: procedure;

     /* test caller for sort_strings */

     declare

          i fixed binary, sysprint file,
          sort_strings entry (character(256) varying dimension(*)),
          array (6) character(256) varying initial
             (  "probe", "hello", "xray", "nice", "def", "abc"  );


     call sort_strings (array);
     do i = 1 to 6;
          put list (array (i));
          put skip;
     end;

end testss;
```

```
sort_strings: procedure (strings);

     declare

          strings character(256) varying dimension(*),
          order fixed binary dimension (hbound (strings, 1)),
          temp character(256) varying,
          (i, k, l, t) fixed binary;


     /* initialize order array */

     do i = 1 to hbound (order, 1);
          order (i) = i;
     end;

     /* perform bubble sort */

     k, l = hbound (strings, 1);
     do while (k <= l);
          do i = 2 to k;
               l = i - 1;
               if strings (order (l)) > strings (order (i)) then do;
                  t = order (l); order (l) = order (i); order (i) = t;
                  k = l;
               end;
          end;
     end;

```

```
30       /* move strings into above ordering */
31
32       do i = 1 to hbound (strings, 1);
33           if order (i) ^= -1 then do;
34               temp = strings (i);
35
36               /* follow chain 'til reach start again */
37
38               do k = i repeat l while (k ^= -1);
39                   l = order (k);
40                   strings (k) = strings (l);
41                   order (k) = -1;
42               end;
43               strings (l) = temp;
44           end;
45       end;
46
47
48   end sort_strings;
```

The debugging session begins below. Again, an exclamation point (!) indicates lines typed by the user.

```
 1        !  testss
 2  !(quit)
 3           QUIT
 4           r 736 6.068 0.132 9 level 2, 10
 5
 6        !  probe
 7           Condition quit raised at line 21 of sort_strings.
 8        !  source
 9                   do i = 2 to k;
10        !  value k
11               1
12        !  value l
13               1
```

First the program testss, used to test the sort_strings subroutine, is called from command level (line 1). When no output is produced, the program is aborted by issuing a quit signal, and probe is invoked to determine where the program was looping (line 6).

When probe is entered, it responds by giving the procedure and line where execution was interrupted. The source pointer is set by default to that line, so that the source request (line 8) may be used to display the text of the statement. The output does not indicate whether the infinite loop is occurring in the inner (do i = 2 to k) or outer (do while (k <= l)) loop. The value of k (line 11) is 1, which implies that the inner loop is not being entered; the value of l (line 13) is also 1 explaining why the outer loop never terminates.

An examination of the program shows that k and l could take on these values if elements 1 and 2 are exhanged on a pass with k = 2; on subsequent passes, no exchanges are made (as the inner loop is not entered), and the termination condition is never met. What is needed is to force l to be less than k on all passes unless an exchange actually occurs. This can be done by setting l = -1 before attempting the inner loop.

```
14        !  before: let l = -1
15           Break set before line 21 of sort_strings.
16        !  quit
17           r 737 1.217 3.562 97 level 2, 10
18
19        !  start
20
21           def
22           hello
23           probe
24           abc
25           xray
26           r 737 0.359 0.182 0
```

The probe command can be used to modify the value of variables either interactively or as part of a break request list. In the latter case, the change is made every time the program is executed. A breakpoint is set before the current statement (line 21 of the program--the inner loop) to set the value of l to -1 with the before request (line 14). The quit request (line 16) causes a return to command level, and the Multics start command (line 19) restarts the program from where it was interrupted. This time output is generated. However, the strings are not being sorted correctly.

```
27        !  probe sort_strings
28        !  position "i = 1";source
29                do i = 1 to hbound (order, 1);
30        !  position "i = 1";source
31                do i = 1 to hbound (strings , 1);
32        !  before
33           Break set before line 32 of sort_strings.
34        !  quit
35           r 738 0.218 0.002 14
36
37        !  testss
38           Stopped before line 32 of sort_strings.
39        !  symbol order
40           fixed binary(17,0) aligned automatic dimension(6)
41           Declared in sort_strings.
42        !  value order(1:6)
43               6
44               5
45               2
46               4
47               1
48               3
```

One way to determine whether it is the sorting or ordering section of the program that is functioning incorrectly, is to stop the program before the ordering section and look at its input, the array "order." The position request (line 28) is an attempt to locate the desired statement, but the source request (line 28), used to check that the correct line has been found, shows that the wrong one was found. The process is repeated (line 30), and the source pointer set to the correct line. A break is set (line 32) to cause the program to "halt" at that statement and enter probe. The driving program is begun once again (line 37), and sort_strings halts at the desired location. The symbol request (line 39) is used to check that the correct dimensions are being received for the array order. The value request (line 42) is used to display order(1), ..., order(6). It can be seen that these are the correct values ("abc", in position 6, is to be moved to position 1, etc.).

```
49          !  position 39; source
50                                 l = order (k)
51          !  after: (value k; value l)
52             Break set after line 39 of sort_strings.
53          !  continue
54                  1
55                  6
56                  6
57                  3
58                  3
59                  2
60                  2
61                  5
62                  5
63                  1
64                  1
65                 -1
66                  4
67                  4
68                  4
69                 -1
70              nice
71              def
72              hello
73              probe
74              abc
75              xray
76              r 740 0.602 0.000 0
```

It appears that the sorting code is working properly (with the patch in it). Therefore, the reordering of the array is failing for some other reason. The user then begins to trace the exchanges that are made. A break is set (lines 49 and 51) to display the values of k (the element to which the string is to be moved) and l (the element from which the string is to be moved) as the program is running. As stated previously, the effect of recompiling the program with a put statement added can be duplicated in this manner. The break is set after the line where both values have been determined for the exchange. The continue request (line 53) restarts the program from where it was suspended by the break.

The output shows that extra exchanges are taking place. When k = 5, the next element on the chain is the first element (l = 1), and the fifth element should therefore be replaced by the copy of the first value stored in "temp." It should not be replaced by the current first element (the old element 6, "abc"). Nor should the program continue to move the undefined element -1 into element 1.

```
77          !  probe sort_strings
78          !  reset at 39
79             Break reset after line 39 of sort_strings.
80          !  before 39: if order(k) = i: (
81          !       let strings(k) = temp
82          !       let order(k) = -1
83          !       goto 42
84          !  )
85             Break set before line 39 of sort_strings.
86          !  quit
87             r 742 0.280 0.966 56
```

For the program to work properly, the movement through the chain must stop when the next element is the first (i.e., when order (k) = i). The saved value of the first (temp) should then be copied into the current element (strings(k)), and the search for additional unreordered elements continued. If the user were to recompile the program, the following code should achieve the desired effect.

```
if order (i) ^= -1 then do;
     temp = strings (i);
     do k = i repeat l while (order (k) ^= i);
           l = order (k);
           strings (k) = strings (l);
           order (k) = -1;
     end;
     strings (k) = temp;
     order (k) = -1;
end;
```

This approach may be checked before recompilation by making a slightly more elaborate patch than the one made previously. The probe command may be used to place a check for the correct terminating condition as the first thing in the loop on k and, if the condition is met, cause strings(k) to be set and the loop exited. First the break (containing the two value requests) previously set after the statement (line 78) is reset. Then a break, containing several requests and extending across line boundaries, is set (lines 80 through 84) before the statement on line 39 of the program.

```
 88     !  testss
 89        Stopped before line 32 of sort_strings.
 90     !  reset
 91        Break reset before line 32 of sort_strings.
 92     !  continue
 93        abc
 94        def
 95        hello
 96        nice
 97        probe
 98        xray
 99        r 743 0.357 1.582 42
100
101     !  probe sort_strings
102     !  status
103        Break before line 39.
104        Break before line 21.
105     !  status at 21
106        Break before line 21:  let l = -1
107     !  quit
108        r 744 0.184 1.146 72
```

The program is run once again (line 88), and the break set between the two sections is encountered again. As it is no longer of any use, the reset request (line 90), assuming the default of the last break encountered, is used to delete the break. The continue request (line 92) resumes the execution of the program. This time it works!

The probe command is invoked once again. This time the status request is used to recall the breaks set, and, hence, the changes to be made to the program. Two forms of the status request are used. Just "status" (line 102) gives a list of all breaks set in the program; "status at line_number" (line 105) gives the text of the associated break request list. The user can now edit and recompile the program and expect it to work correctly. The remaining breaks need not be reset, because a recompilation has the same effect.

Terminology

active - a procedure is said to be active if its execution is ongoing or suspended by an error, quit signal, breakpoint, or call. An active procedure should be distinguished from one that has never been run, has completed execution, or has been interrupted and aborted by a Multics release command.

automatic storage - a storage class for which space is allocated dynamically in a stack frame upon block invocation. As a result, variables of this class only have storage assigned to them, and hence a legitimate address and value, when the block in which they are declared has an active invocation. PL/I variables, by default, belong to this class. FORTRAN variables must appear in an "automatic" statement in order to belong to this class.

block - corresponds to a PL/I procedure or begin block or FORTRAN program or subroutine, and identifies a particular group of variable declarations.

breakpoint - a point at which program execution is temporarily interrupted and probe requests executed.

invocation - when a procedure is called recursively, it will appear on the stack two or more times, and will have storage allocated for it the same number of times. Each instance of the procedure on the stack is considered a separate and distinguishable invocation of the block. The values of automatic variables can be different in different invocations of the same block. The most recent invocation is the topmost in stack trace.

level number - an integer used by probe to uniquely designate each block invocation (i.e., each entry in a stack trace). Level one is the first (least recent) procedure invoked. Level number is _not_ necessarily the same as either of the numbers given after the word "level" in a ready message. The first of this pair gives the count of command levels in effect and gives the value _n_+1, where _n_ is the number of programs (or groups of programs) whose execution has been suspended, the second gives the number of stack frames in existence and since the probe stack includes quick blocks, this number is less than or equal to the level number of the last command level in the stack trace.

quick block - internal procedures and begin blocks that satisfy certain requirements (e.g., are not called recursively, do not contain on, signal, or revert statements, etc.) have their automatic storage allocated by the blocks that call them. Hence, they do not actually have their own stack frames, but share the one of the caller. Certain system commands, such as trace_stack, ignore these blocks. The probe command, however, includes them in a stack trace, and treats them as if they were the same as any other blocks. The quickness of a block may be determined from a program listing containing information about the storage requirement of the program (produced with the -symbols, -map, or -list control arguments). For examnle, procedure "quick" shares stack frame of external procedure "main".

stack - if a procedure A calls another procedure B, then the execution of A is suspended until B returns. If B in turn calls C, then this is an ordered list of procedure or subroutine calls indicating which program called which other program, and which will return to which. This ordered list is called the "stack". In probe, a trace of the stack may be displayed by use of the stack request. The list is given in top-down fashion with the most recently called procedure listed first:

```
3        C
2        B
1        A
```

The numbers are level numbers (see below).

stack frame - when a block is invoked (that is, a procedure is called or a begin block is entered), storage is allocated for its automatic variables. The area allocated is called a stack frame and logically corresponds to each entry in the stack.

static storage - a storage class for which space is allocated once per process, effectively at the time the procedure is first referenced. As a result, variables of this class always have a legitimate address and value. Regular FORTRAN variables, and those in a common block, have static storage. PL/I variables must be explicitly declared.

support procedure - a system utility routine that provides runtime support for other procedures (e.g., the procedure that allocates storage as requested by a PL/I allocate statement).

Summary of Requests

after	a	Set a break after a statement.
before	b	Set a break before a statement.
call	cl	Call an external procedure.
continue	c	Return from probe.
execute	e	Execute a Multics command.
goto	g	Transfer to a statement.
halt	h	Stop the program.
if	(none)	Execute commands if condition is true.
let	l	Assign a value to a variable.
mode	(none)	Turn brief message mode on or off.
pause	pa	Stop a program once.
position	ps	Examine a specified statement or locate a string in the program.
quit	q	Return to command level.
reset	r	Delete one or more breaks.
source	sc	Display source statements.
stack	sk	Trace the stack.
status	st	Display information about breaks.
step	s	Advance one statement and halt.
symbol	sb	Display the attributes of a variable.
use	u	Examine the block specified.
value	v	Display the value of a variable.
where	wh	Display the value of probe pointers.
while	wl	Execute commands while condition is true.

Name: profile

The profile command is a debugging tool used in conjunction with the -profile control argument of the pl1 and fortran commands. The profile command prints information about the execution of each statement in the PL/I or FORTRAN program.

The -profile control argument causes the compiler to generate an internal static table containing an entry for each statement in the source program; the table entry contains information about the statement as well as a counter that starts out as zero. The counter associated with a statement is increased by one each time the statement is executed. The profile command prints and resets these counters.

Usage

```
profile paths -control_args-
```

where:

1. paths — are the pathnames or reference names of programs whose counters are to be printed or reset.

2. control_args — are selected from the following list. Control arguments take effect immediately and apply to all programs whose names appear in the command line.

   -print, -pr — print the following information for each statement in the specified programs:

   1. line number
   2. statement number
   3. number of times the statement has been executed
   4. cost of executing the statement measured in number of instructions executed online plus the number of PL/I operators invoked. Each instruction and each operator invocation count as only one unit.
   5. the names of all the PL/I operators used by this statement
   6. total cost for all statements is printed at the end

   -brief, -bf — omits from the statement list statements that have never been executed.

   -long, -lg — includes in the statement list statements that have never been executed.

   -reset, -rs — causes profile to reset to zero all counters in the specified program.

Note

The default control arguments are -print and -brief.

Example

The PL/I program shown below counts the number of occurrences of one string in another string. It was compiled with the -profile control argument and executed once. The output from the profile command, which is printed below, shows an anomaly of the current implementation--there is only one counter for the statement:

```
if ... then ...
```

so that one cannot determine the number of times the condition was satisfied.

The source code for the program is:

```
example:  proc(s1,s2);

declare   (s1,s2) char(*),
          (i,k) fixed bin,
          ioa_ options (variable);

          k = 0;
          do i = 1 to length(s1) - length(s2);
               if substr(s1,i,length(s2)) = s2 then k = k + 1;
               end;

          call ioa_("^d",k);
          end example;
```

After executing the program once and invoking the profile command without any control arguments, the output is:

```
LINE   STM   COUNT   COST                 PROGRAM

                                          example
   7     1       1      1
   8     1       1      8
   9     1      26    234
  10     1      26     52
  12     1       1     14 + 1 (call_ext_out_desc)
  13     1       1      0 + 1 (return)

TOTAL                 309 + 2
```

Name: program_interrupt, pi

The program_interrupt command allows the users of certain editors, subsystems, and other interactive programs to reenter those programs at known places after having interrupted the process by issuing a quit signal. The documentation of each individual program specifies whether or not it accommodates the program_interrupt feature and exactly what its behavior is following such an interrupt.

Generally speaking, when the user wants to reenter a program known to accommodate the program_interrupt feature, he issues the program_interrupt command (or pi) after returning to command level by issuing the quit signal. The interrupted program usually aborts what it was doing and resumes interaction with the user.

If a program_interrupt is mistakenly directed at a program not having this feature, then the system default error handler prints a message and returns the user to command level. At this point, the system is still holding the work that was originally interrupted by the quit signal. If the user desires to restart the program he quit out of, he should issue the start command. If the user does not desire to continue execution of the program, he should issue the release command to return to the command level prior to the command he just "quit" out of. Refer to the descriptions of the release and start commands.

Usage

```
program_interrupt
```

Note

To make use of the program interrupt facility, a program or subsystem must establish a condition handler for the program_interrupt condition. When the user invokes the program_interrupt command, it signals the program_interrupt condition, and the handler established by the program or subsystem is invoked. For a discussion of conditions see "The Multics Condition Mechanism" and "List of System Conditions and Default Handlers" in Section VI of the MPM Reference Guide.

Example

The edm command has a handler for the program_interrupt condition that stops whatever the editor is doing and looks for a request from the user's terminal. Thus, a user of edm who inadvertently typed "p100" (to print 100 lines) could kill this printout by issuing a quit signal and then typing program_interrupt. The edm command responds by printing "Edit." and then waiting for editing requests.

Name: progress, pg

The progress command executes a specified command line and prints information about how its execution is progressing in terms of CPU time, real time, and page faults.

Usage

```
     progress -control_arg- -command_line-
```

where:

1. control_arg	if present, progress performs only the function specified by that control argument. No command_line argument can follow except in the case of -brief. The control argument can be one of the following:
-off	suppresses the incremental messages (see "Output Messages" below) printed during execution of a command line previously initiated, but does not suppress the message printed when that command line is finished. This control argument can be used to suppress messages while debugging.
-on	restores the printing of incremental messages during execution of the command line.
-brief, -bf	permits only the message at completion of the command line to be printed. The command_line argument is used following this control argument.
-output_switch name, -os name	directs output from the progress command to be printed on the I/O switch named name. The default switch is user_i/o.
-cput n	causes progress to print its incremental message every n seconds of virtual CPU time. The default is -cput 10.
-realt n	causes progress to print its incremental message every n seconds of real time instead of virtual CPU time.
2. command_line	is a character string made up by concatenating all the arguments to progress (excluding the first if it is a control argument) with blanks between them. The string is executed as a command line. It can appear as the only argument or following the -brief control argument.

Output Messages

After every 10 seconds of virtual CPU time (assuming the default triggering value is used), progress prints out a message of the form:

```
ct/rt = pt%, ci/ri = pi% (pfi)
```

where:

1.	ct	is the number of virtual CPU seconds used by the command line so far.
2.	rt	is the total real seconds used so far.
3.	pt	is the ratio of virtual to real time used by the command so far.
4.	ci	is the incremental virtual CPU time (since the last message).
5.	ri	is the incremental real time.
6.	pi	is ci expressed as a percentage of ri.
7.	pfi	is the number of page faults per second of virtual CPU time (since the last message).

When the command line finishes, progress prints the following message:

```
finished: ct/rt = pt% (pft)
```

where pft is the number of page faults per second of virtual CPU time for the execution of the entire command.

Example

In the following example, the user wants to see how execution is progressing as he compiles a PL/I source program (named newseg.pl1) using the -list control argument of the pl1 command.

```
progress pl1 newseg -list
PL/I
10/30 = 33%, 10/30 = 33% (26)
20/50 = 40%, 10/20 = 50% (17)
30/123 = 24%, 10/73 = 13% (20)
finished: 33/150 = 22% (22)
```

Name: qedx, qx

The qedx context editor can be used to create and edit ASCII segments in Multics. The qedx editor is similar to the simpler edm editor also described in this document. The qedx editor cannot be called recursively.

Usage

```
     qedx -inst_path- -optional_args-
```

where:

1. inst_path is an optional argument and, if present, specifies the pathname of an ASCII segment from which the editor is to take its initial instructions. Such a set of instructions is commonly referred to as a macro. The editor automatically concatenates the suffix qedx to inst_path to obtain the complete pathname of the segment containing the qedx instructions.

2. optional_args are optional arguments that are appended, each as a separate line, to the buffer named args; (the first optional argument becomes the first line in the buffer and the last optional argument becomes the last line). Arguments are used in conjunction with a macro specified by instr_path. See "Initialization of Macros" below.

If inst_path is provided, the editor executes the qedx requests contained in the specified segment and then waits for the user to type further requests. If inst_path is omitted, the editor waits for the user to type a qedx request. The use of the inst_path argument requires a fairly detailed understanding of the editor and further discussion of this feature is delayed until later in this description.

Once the qedx editor is invoked, the user can immediately begin to issue qedx requests from his terminal. Requests fall into one of two general categories, input requests and edit requests. Input requests place the editor into input mode, and allow the user to enter new ASCII text from his terminal until an appropriate escape character sequence is typed to switch the editor back to edit mode. Edit requests allow the user to read and write ASCII segments and perform various editing functions on ASCII data. Input and editing operations are not performed directly on the target segments but in a temporary workspace known as a buffer.

To create a new ASCII segment, a user might perform the following steps:

1. Invoke qedx and enter input mode by typing one of the input requests (e.g., append) as the first qedx request.

   a. Enter ASCII text lines into the buffer from the terminal.

   b. Leave input mode by typing the appropriate escape character sequence as the first characters of a new line.

2. Inspect the contents of the buffer and make any necessary corrections using edit or input requests.

3. Write the contents of the buffer into a new segment using the write request.

4. Exit from the editor using the quit request.

To edit an existing ASCII segment, a user might perform the following steps:

1. Invoke qedx and read the segment into the buffer by giving a read request as the first qedx request.

2. Edit the contents of the buffer using edit and input requests as necessary. (The editor makes all changes on a copy of the segment, not on the original. Only when the user issues a write request does the editor overwrite the original segment with the edited version.)

3. Using the write request, write the contents of the modified buffer either back into the original segment or, perhaps, into a segment of a different name.

4. Exit from the editor using the quit request.

The user can create and edit any number of segments with a single invocation of the editor as long as the contents of the buffer are deleted before work is started on each new segment.

Note

While most users interact with the qedx editor through a terminal, the editor is designed to accept input through the user_input I/O switch and transmit output through the user_output I/O switch. These switches can be controlled (using the iox_ subroutine described in the MPM Subroutines) to interface with other devices/files in addition to the user's terminal. For convenience, the qedx editor description assumes that the user's input/output device is a terminal.

List of Editor Requests

In the list given below, editor requests are divided into four categories: input requests, basic edit requests, extended edit requests, and buffer requests. The input requests and basic edit requests are sufficient to allow a user to create and edit ASCII segments and provide a functional capability quite similar to edm. The extended requests give the user the ability to execute Multics commands without leaving the editor and also to effect global changes. Because the requests are, in general, more difficult to use properly, they should be learned only after mastering the input and basic edit requests. The buffer requests require a knowledge of auxiliary buffers. (Since the nothing and comment requests are generally used in macros, they are included with the buffer requests.) The buffer requests, used with any of the other requests, and special escape sequences allow the user to make qedx function as an interpretive programming language through the use of macros.

The character given in parentheses is the actual character used to invoke the request in qedx and does not always bear a relation to the name of the request.

INPUT REQUESTS

append (a)	Enter input mode, append lines typed from the terminal after a specified line.
change (c)	Enter input mode, replace the specified line or lines with lines typed from the terminal.
insert (i)	Enter input mode, insert lines typed from the terminal before a specified line.

BASIC EDIT REQUESTS

delete (d)	Delete specified line or lines from the buffer.
print (p)	Print specified line or lines on the terminal.
print line number (=)	Print line number of specified line.
quit (q)	Exit from the editor.
read (r)	Read specified segment into the buffer.
substitute (s)	Replace specific character strings in specified line or lines.
write (w)	Write current buffer into specified segment.
locate	Address specified line and print it (this request is not invoked by typing a special character).

EXTENDED EDIT REQUESTS

execute (e)	Pass remainder of request line to the Multics command processor (i.e., escape to execute other Multics commands).
global (g)	Print, delete, or print line number of all addressed lines that contain a specified character string.
exclude (v)	Print, delete, or print line number of all addressed lines that do not contain a specified character string.

BUFFER REQUESTS

buffer (b)	Switch to specified buffer (i.e., switch all subsequent editor operations to the specified buffer).
move (m)	Move specified line or lines into the specified buffer.
status (x)	Print a summary of the status of all buffers currently in use.
nothing (n)	Do nothing (used to address a line with no other action).
comment (")	Ignore the remainder of this request line.

Addressing

The qedx editor is basically a line-oriented editor in that editing requests usually operate on an integral number of ASCII lines. As a result, most editing requests are preceded by an address specifying the line or lines in the buffer on which the request is to operate. There are three basic means by which lines in the buffer can be addressed:

1. Addressing by absolute line number
2. Addressing by relative line number, i.e., relative to the "current" line
3. Addressing by context

In addition, a line address can be formed using a combination of the above techniques.

ADDRESSING BY ABSOLUTE LINE NUMBER

Each line in the buffer can be addressed by a decimal integer indicating the current position of the line within the buffer. The first line in the buffer is line 1, the second line 2, etc. The last line in the buffer can be addressed either by line number or by using the $ character, which is interpreted to mean "the last line currently in the buffer." In certain cases it is possible to address the (fictitious) line preceding line 1 in the buffer by addressing line 0.

As lines are added to or deleted from the buffer, the line numbers of all lines that follow the added or deleted lines are changed accordingly. For example, if line 15 is deleted from the buffer, line 16 becomes line 15, 17 becomes 16, and so on.

If an attempt is made to address a line not contained in the buffer, an error message is printed by the editor. If the buffer is currently empty, as it is when the editor is first entered, only the line numbers 0 and $ are considered valid.

ADDRESSING BY RELATIVE LINE NUMBER

The qedx editor maintains the notion of a "current" line that is specified by using the character "." (period). Normally, the current line is the last line addressed by an edit request or the last line entered from the terminal by an input request. The value of "." after each editor request is documented in the description of the request.

Lines can be addressed relative to the current line number by using an address consisting of "." followed by a signed decimal integer specifying the position of the desired line relative to the current line. For example, the address .+1 specifies the line immediately following the current line and the address .-1 specifies the line immediately preceding the current line.

When specifying an increment to the current line number, the + sign can be omitted (e.g., .5 is interpreted as .+5). In addition, when specifying a decrement to the current line number, the "." itself can be omitted (e.g., -3)is interpreted as .-3). It is also possible to follow the "." with a series of signed decimal integers (e.g., .5+5-3 is interpreted as .+7).

ADDRESSING BY CONTEXT

Lines can be addressed by context by using a "regular expression" to match a string of characters on a line. When used as an address, a regular expression specifies the first line encountered that contains a string of characters that matches the regular expression. In its simplest form, a regular expression is a character or a string of characters delimited by the right slant character (/). For example, in the following text, the regular expression /abc/ matches line 2.

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

To use a regular expression as an address, the user types /regexp/, where regexp is any valid regular expression as described below. The search for a regular expression begins on the line following the current line (i.e., .+1) and continues through the entire buffer, if necessary, until it again reaches the current line. In other words, the search proceeds from .+1 to $ and then from line 1 to the current line. If the search is successful, /regexp/ specifies the first line encountered during the search in which a match was found.

A regular expression can consist of any character in the ASCII set except the newline character. However, the following characters have specialized meanings in regular expressions.

/	Delimits a regular expression used as an address.
*	Signifies "any number (or none) of the preceding character".
^	When used as the first character of a regular expression, the ^ character signifies the character preceding the first character on a line.
$	When used as the last character of a regular expression, the $ character signifies the character following the last character on a line.
.	Matches any character on a line.

Some examples follow:

/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".
/.*/	Matches any line.
/^$/	Matches an empty line (a line containing only a newline character).

The special meanings of "/", "*", "$", "^", and "." within a regular expression can be removed by preceding the special character with the escape sequence \c.

/\c/\c*/	Matches the string "/*" anywhere on a line.

The editor remembers the last regular expression used in any context. The user can reinvoke the last used regular expression by using a null regular expression (i.e., //). In addition, a regular expression can be followed by a signed decimal integer in the same manner as when addressing relative to the current line number. For example, the addresses /abc/+5-3, /abc/+2 or /abc/2 all address the second line following a line containing "abc".

The two uses of "." and "$" (as line numbers and as special characters in regular expressions) are distinguished by context.

COMPOUND ADDRESSES

An address can be formed using a combination of the techniques described above. The following rules are intended as a general guide in the formation of these compound addresses.

1. If an absolute line number is to appear in an address, it must be the first component of the address.
2. A relative line number can appear anywhere in a compound address.
3. A regular expression can appear anywhere in a compound address.
    a. An absolute line number can be followed by a regular expression. This construct is used to begin the regular expression search after a specific line number. For example, the address 10/abc/ starts the search for /abc/ immediately after line 10.
    b. A regular expression can follow or be followed by an address specified by a relative line number. For example, the address .-8/abc/ starts the search eight lines before the current line, while /abc/.8 addresses the line eight lines after the first occurrence of /abc/.

c. A regular expression can be followed by another regular expression. For example, the address /abc//def/ matches the first line containing "def" appearing after the first line containing "abc". As mentioned earlier, a regular expression can be followed by a decimal integer. For example, the address /abc/-10/def/.5 starts the search for /def/ 10 lines before the first line to match /abc/ and if /def/ is matched, the value of the compound address is the fifth line following the line containing the match for /def/.

ADDRESSING A SERIES OF LINES

Several of the editor requests can be used to operate on a series of lines in the buffer. To specify a series of lines, two addresses must be given in the following general form:

```
ADR1,ADR2
```

The pair of addresses specifies the series of lines starting with the line addressed by the address ADR1 through the line addressed by ADR2, inclusive.

Examples:

1,5	specifies line 1 through line 5.
1,$	specifies the entire contents of the buffer.
.1,/abc/	specifies the line following the current line through the first line (after the current line) containing "abc".

When a comma is used to separate addresses, the address computation of the second address is unaffected by the computation of the first address (i.e., the value of "." is not changed by the evaluation of the first address). For example, the address pair:

```
.1,.2
```

specifies a series of two lines, the line immediately after the current line through the second line after the current line.

However, if a semicolon is used to separate addresses instead of a comma, the value of "." is set to the line addressed by ADR1 before the evaluation of ADR2 begins. In contrast to the example given immediately above, the address pair:

```
.1;.2
```

specifies a series of three lines, the line immediately following the original current line through the second line following the line specified by ADR1. As a further example, the address pair:

```
/abc/;.+10
```

is equivalent to the address pair:

```
/abc/,/abc/+10
```

ADDRESSING ERRORS

The following list describes the various errors that can occur when the editor is attempting to evaluate an address.

1. "Buffer empty" -- An attempt has been made to reference a specific line when the buffer is empty. (Only "$", "." and "0" are legal addresses within an empty buffer and only if used with a read, append, or insert request.)

2. "Address out of buffer" -- An attempt has been made to refer to a nonexistent line (e.g., an address of 20 when there are fewer than 20 lines in the buffer or an address of .+5 when the current line is fewer than 5 lines from the last line in the buffer).

3. "Address wrap around" -- An attempt has been made to address a series of lines in which the line number of the second line addressed is less than the line number of the first (e.g., $,1).

4. "Search failed" -- A regular expression search initiated from the user´s terminal has failed to find a match.

5. "Syntax error in regular expression" -- A regular expression used as an address has not been properly delimited, or successive asterisks have been encountered without an escape sequence character (e.g., /ab**c/).

6. "// undefined" -- A null regular expression has been used and no previously typed regular expression is available.

Editor Request Format

A request to the editor can take any one of the following three forms depending on the number of addresses to be specified with the request.

1. <request>

2. ADR<request>

3. ADR1,ADR2<request> or ADR1;ADR2<request>

ADR, ADR1, and ADR2 are any legal addresses as specified above, and <request> is any valid editor request.

Some editor requests require no address, some require a single address, and others require a pair of addresses. In all cases, however, the user can issue a request omitting one or both of the required addresses and let the editor provide the missing address information by default. The following general rules apply to the use of addresses specified by default.

1. If a request requiring an address pair is issued with the second address missing, the (missing) second address is assumed to be the same as the first. For example:

   ```
   ADR<request>
   ```

   is interpreted as:

   ```
   ADR,ADR<request>
   ```

   and addresses a single line in the buffer (i.e., the line addressed by ADR).

2. If a request requiring an address pair is issued with both addresses missing, one of the following address pairs is assumed depending on the request issued.

.,.<request>	for all editor requests except write, global, and exclude
1,$<request>	for write, global and exclude

3. If a request requiring a single address is issued with no address specified, one of the following addresses is assumed depending on the request issued.

.<request>	for all editor requests except read
$<request>	for read requests

Value of "."

All editor requests that alter the contents of the buffer or cause information to be output on the user's terminal change the value of "." (i.e., the current line). Usually, the value of "." is set to the last address specified (either explicitly or by default) in the editor request. The one major exception to this rule is the delete request, which sets "." to the line <u>after</u> the last line deleted. (If the line deleted was the last one in the buffer, then "." is set to "$+1".)

Multiple Requests on a Line

In general, any number of editor requests can be issued in a single input line. However, each of the requests listed below must be terminated with a newline character, and, thus, each must appear on a line by itself or at the end of a line containing multiple editor requests.

read	(r)
write	(w)
quit	(q)
execute	(e)

Spacing

The following rules govern the use of spaces in editor requests.

1. Spaces are taken as literal text when appearing inside of regular expressions. Thus, /the n/ is not the same as /then/.

2. Spaces cannot appear in numbers, e.g., if 13 is written as 1 3, it is interpreted as 1+3 or 4.

3. Spaces within addresses except as indicated above are ignored.

4. The treatment of spaces in the body of an editor request depends on the nature of the request.

Responses From the Editor

In general, the editor does not respond with output on the terminal unless explicitly requested to do so (e.g., with a print or print line number request). The editor does not comment when the user enters or exits from the editor or changes to and from input and edit modes. The use of frequent print requests is recommended for new users of the qedx editor.

Stopping qedx Execution

If the user inadvertently requests a large amount of terminal output from the editor and wishes to abort the output without abandoning all previous editing, he can issue the quit signal (by pressing the proper key on his terminal, e.g., BRK, ATTN, INTERRUPT), and, after the quit response, he can reenter the editor by invoking the program_interrupt (pi) command (described in this document) This action causes the editor to abandon its printout, but leaves the value of "." as if the printout had gone to completion.

If an error is encountered by the editor, an error message is printed on the user's terminal and any editor requests already input (i.e., read ahead from the terminal) are discarded.

If a user exits from qedx by issuing the quit signal, and subsequently invokes qedx in the same process, the message "qedx: Pending work in previous invocation will be lost if you proceed; do you wish to proceed?" is printed on the terminal. The user must type a "yes" or "no" answer.

Input Mode

The editor can be placed in input mode with the use of one of the three input requests (append, change, and insert). The input request must immediately be followed by a blank or a newline character, which in turn is followed by the literal text to be input to the buffer. The literal text can contain any number of ASCII lines. To exit from input mode and terminate the input request, the escape sequence \f is typed, usually as the first characters of a new line. The \f escape sequence can be followed immediately with more editor requests on the same line. (To see how the contents of a buffer can be used as literal text with an input request, see "Use of Buffers for Moving Text" later in this description.) The usual form of an input request is as follows:

```
ADR1,ADR2 <input request>
TEXT
.
.
.
\f
```

It is important to remember to terminate the input request with the \f escape sequence before typing another request. Otherwise, the editor remains in input mode, and the (would be) editor request is regarded as input and included in the text rather than executed as a request.

Upon leaving input mode, the value of "." is set to point to the last line input from the terminal.

The special meaning of any of the escape sequences used by qedx (i.e., \f, \c, \b, \r) can be suppressed by preceding the escape sequence with \c thus allowing these escape sequences to be input as literal text. (The \b and \r escape sequences are described under "Special Escape Sequences" later in this description.)

Input Requests

There are three input requests accepted by qedx: append, change, and insert.

APPEND (a)

The append request is used to enter input lines from the terminal, appending these lines after the line addressed by the append request. The append request is one of the few requests that can operate correctly when the buffer is empty.

```
Format:         ADRa
                TEXT
                \f

Default:        a is taken to mean .a

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.
```

CHANGE (c)

The change request is used to delete an addressed line or set of lines and replace the deleted line(s) with new text entered from the terminal.

```
Format:         ADR1,ADR2c
                TEXT
                \f

Default:        c is taken to mean .,.c

Value of ".":   Set to last line entered from terminal.

Example:        Buffer contents:      a:  procedure;
                                          x=y;
                                          q=r;
                                          end a;

                Request sequence:     2,3c              /x=/,/q=/c
                                      s=t;      or      s=t;
                                      u=v;              u=v;
                                      w=z;              w=z;
                                      \f                \f

                Result:               a:  procedure;
                                          s=t;
                                          u=v;
                                ".">      w=z;
                                          end a;
```

INSERT (i)

The insert request is used to enter input lines from the terminal and insert the new text immediately before the addressed line. The insert request is one of the few requests that can operate on an empty buffer.

```
Format:         ADRi
                TEXT
                \f

Default:        i is taken to mean .i

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;
```

Note: Since the append request puts input lines after the current line and insert puts them before the current line, the request ADRi has the same effect as the request ADR-1a.

Basic Edit Requests

The edit requests described below represent a subset of qedx suitable for most editing situations. Additional requests are described later in this description under "Extended Edit Requests" and "Buffer Requests."

DELETE (d)

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

Format: ADR1,ADR2d

Default: d is taken to mean .,.d

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;
```

Notes: If the line deleted was the last one in the buffer, then "." is set to "$+1".

The user can create and edit any number of segments with just one invocation of qedx as long as he deletes the contents of the buffer (1,$d) before work is started on each new segment.

PRINT (p)

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

Format: ADR1,ADR2p

Default: p is taken to mean .,.p

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

Example:

```
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;
```

PRINT LINE NUMBER (=)

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

Format: ADR=

Default: = is taken to mean .=

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

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

	Request:	/q;/=
	Result:	3

QUIT (q)

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 (see below).

Format:	q
Default:	The quit request cannot have an address.
Note:	The quit request must be followed immediately by a newline character.

READ (r)

The read request is used to append the contents of a specified ASCII segment after the addressed line. The read request is one of the few requests that operate correctly when the buffer is empty.

Format:	ADRr PATH 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.

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.

SUBSTITUTE (s)

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/ is taken to mean .,.s/REGEXP/STRING/

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 escape sequence \c.

Examples:

Buffer contents:	The quick brown sox
Request:	s/sox/fox/
Result:	The quick brown fox

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;

WRITE (w)

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

Format: ADR1,ADR2w -PATH-

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 the default pathname is null, then the message "No pathname given" is printed and qedx awaits another request. The default pathname is the first pathname used with either a read or write request in this invocation of qedx. The default pathname is set to null if no pathname has been given in this invocation of qedx or if a pathname (either the same one or a different one) is used with a second read request.

Default: w PATH is taken to mean 1,$w PATH

Value of ".": Unchanged.

Example: Buffer contents:

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

Request: 1,4w sam.pl1

Result:	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).

LOCATE

This request sets the value of "." to a specific line and prints the line. This request is not invoked by typing a character; the user merely types a valid address (absolute, relative, or context) followed by a newline character.

Format: ADR

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

Value of ".": Set to a line addressed by request.

Example:

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

Request:                4  or +2  or /^k/

Result:                 kiwi
```

Extended Edit Requests

The editor requests discussed up to this point comprise a basic subset sufficient for most applications. The following requests offer the user more advanced and, in general, more powerful capabilities.

EXECUTE (e)

The execute request is used to invoke the Multics command system without exiting from the editor. Whenever an execute request is recognized, the remaining characters in the request line are passed to the Multics command processor. The execute request can be followed by any legal Multics command line. However, the user should not invoke qedx while in qedx since qedx is not recursive.

Format: e <command line>

Value of ".": Unchanged.

Example: The request line:

```
     e print alpha.pl1
```

can be used to print the segment in the user's working directory named alpha.pl1. After the segment is printed on the user's terminal, he can continue his work in qedx as though he had not issued the execute request. (See also "Stopping qedx Execution" earlier in this description.)

The request line:

```
     e list; mail
```

lists the contents of the user's working directory and prints the contents of his mailbox (if any).

Note: If the user wishes to abort a command line invoked with the execute request, he can issue the quit signal and then invoke the program_interrupt (pi) command (described in this document) to abort the command line and restore control to qedx.

GLOBAL (g)

The global request is used in conjunction with one of the following requests: print, delete, or print line number. The other request operates only on those lines addressed by the global request that contain a match for a specified regular expression.

Format: ADR1,ADR2gX/REGEXP/

where X must be one of the following requests:

d	delete lines containing REGEXP
p	print lines containing REGEXP
=	print line numbers of lines containing REGEXP

The character immediately following the request X is taken to be the regular expression delimiter and can be any character not appearing in REGEXP.

Default: gX/REGEXP/ is taken to mean 1,$gX/REGEXP/

Value of ".": Set to ADR2 of request.

Example: Buffer contents:

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

Request: gd/q/

Result:

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

EXCLUDE (v)

The exclude request is also used in conjunction with one of the following requests: print, delete, or print line number. The other request operates only on those lines addressed by the exclude request that do <u>not</u> contain a match for a specified regular expression.

Format: ADR1,ADR2vX/REGEXP/

where X must be one of the following requests:

d	delete lines not containing REGEXP
p	print lines not containing REGEXP
=	print line numbers of lines not containing REGEXP

The character immediately following the request X is taken to be the regular expression delimiter and can be any character not appearing in REGEXP.

Default: vX/REGEXP/ is taken to mean 1,$vX/REGEXP/

Value of ".": Set to ADR2 of request.

Example: Buffer contents:

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

Request: v=/q/

Result:

```
1
3
5
```

Auxiliary Buffers

The discussion up to this point has assumed the existence of only a single buffer. Actually, qedx supports a virtually unlimited number of buffers. One buffer at a time can be designated as the "current buffer"; any other buffers at this time are referred to as auxiliary buffers. All of the editor requests described so far operate within the current buffer.

Each buffer is given a symbolic name of 1 to 16 ASCII characters. When the editor is invoked, a single buffer (buffer 0) is created by the editor and designated as the current buffer. Additional buffers can be created merely by referencing a previously undefined buffer name. Each buffer is implemented as a separate segment in the user's process directory and, thus, is capable of holding any ASCII segment.

Buffer names of more than one character are always enclosed in parentheses; for example, the buffer name Fred is typed as (Fred). A buffer name consisting of a single character can be typed with or without the enclosing parentheses (e.g., "y" is taken to be "(y)").

NOTE: Auxiliary buffers exist for only the current invocation of qedx. That is, if the user creates several auxiliary buffers, issues the quit request and then invokes the qedx editor again, the auxiliary buffers he created earlier will be gone.

Buffer Requests

The buffer requests allow the user to create auxiliary buffers, move text from one buffer to another, and check on the status of all buffers currently in use. In addition, these requests provide an interpretive programming capability when used in conjunction with certain escape sequences (see "Special Escape Sequences" below).

CHANGE BUFFER (b)

The change buffer request is used to designate an auxiliary buffer as the current buffer. The previously designated current buffer becomes an auxiliary buffer.

Format:	b(X) where X is the name of the buffer that is to become the current buffer.
Value of ".":	Restored to the value of "." when buffer X was last used as the current buffer (i.e., the value of "." is maintained separately for each buffer and saved as part of the buffer status). If X is a new buffer, then "." is set to line 0.

MOVE (m)

The move request is used to move one or more lines from the current buffer to a specified auxiliary buffer. (The "moved" lines are deleted from the current buffer.) The addressed lines replace the previous contents (if any) of the auxiliary buffer.

Format: ADR1,ADR2m(X)

where X is the name of the auxiliary buffer to which the lines are to be moved.

Default: m(X) is taken to mean .,.m(X)

Value of ".": Set to the line after the last line moved in the current buffer. Set to line 0 in the specified auxiliary buffer.

Example:

```
Contents of:  Current Buffer           Buffer B
              a:  procedure;           abc=def;
                  x=y;                 end bin;
                  y=k;
                  k=r;
                  end a;

Request:      3,4m(B)    or    /k;/,/r;/m(B)

Result:       Current Buffer           Buffer B
                                   "."->
              a:  procedure;           y=k;
                  x=y;                 k=r;
         "."->    end a;
```

BUFFER STATUS (x)

The buffer status request is used to print a summary of the status of all buffers currently in use. The name and length (in lines) of each buffer is listed; the current buffer is specially marked with a right arrow "->" immediately to the left of the buffer name. Finally, each buffer's default pathname, if any, is listed.

Format: x

Value of ".": Unchanged.

Example: If the user has created the additional buffers alpha and beta and has designated alpha as his current buffer, the output from the buffer status request might be as follows.

```
157     (0)      demo.runoff
 32   ->(alpha)
 53     (beta)
```

This output indicates 157 lines in buffer 0 (the initial buffer), 32 lines in alpha (the current buffer) and 53 lines in beta. It also indicates that the default pathname for buffer 0 is demo.runoff (in the user's working directory) and that buffers alpha and beta have no default pathnames.

NOTHING (n)

The nothing request is used to address a particular line in the segment (i.e., set the value of "." to a particular line). No other action is taken.

Format: ADRn

Default: n is taken to mean .n

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

Example:

```
Buffer contents:          a:  procedure:
                   "."->      x=y;
                              y=k;
                              k=r;
                              end a;

Request:                 /k=/n

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

COMMENT (")

The comment request is generally used to annotate qedx macros and also can be used to annotate online work. The editor ignores the remainder of the request line.

Format: ADR"

Default: " is taken to mean ."

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

Example:

Buffer contents:

```
b (heads)
a
.
.
.
.
```

Request:

```
0a
"This macro enters special headers
"for subroutine descriptions
\f
w
```

Result:

```
"This macro enters special headers
"for subroutine descriptions
b (heads)
a
.
.
.
.
```

Special Escape Sequences

The input to qedx can be viewed as a stream of ASCII characters. Depending on the context, some of these characters are interpreted as editor requests and others are interpreted as literal text. The following escape sequences are recognized by the editor, in either context, as directives to alter the input character stream in some fashion.

\b(X) This sequence is used to redirect the editor input stream to read subsequent input from buffer X. When the editor encounters this sequence, the entire escape sequence is removed from the input stream and replaced with the literal contents of the specified buffer. If another \b escape sequence is encountered while accepting input from buffer X, the stream is again redirected and the newly encountered escape sequence is also replaced by the contents of the named buffer. The editor allows the recursive replacement of \b escape sequences by the contents of named buffers to a recursion depth of 500 nested \b escape sequences.

The buffer to which the input stream is redirected can contain editor requests, literal text or both. If the editor is executing a request obtained from a buffer (rather than from the terminal) and the request specifies a regular expression search for which no match is found, the usual error comment is suppressed and the remaining contents of the buffer are skipped. If one thinks of the escape sequence \b(X) as a subroutine call statement, the failure to match a regular expression specified by some request in buffer X can be thought of as a return statement.

Reading the contents of buffer X does not delete or alter in any way the contents of the buffer. The contents of buffer X remain constant. Thus, qedx can read input from buffer X many times (see the discussion on "Repeated Editor Sequences" below).

\r This escape sequence is used to temporarily redirect the input stream to read a single line from the user's terminal and is normally used when executing editor requests contained in a buffer. The \r is removed from the input stream and replaced with the next complete line entered from the user's terminal. In the line that replaces the \r sequence, additional \r or \b escape sequences have no effect.

NOTE: The special meanings of \b and \r can be suppressed by preceding the escape sequence with a \c escape sequence.

Use of Buffers for Moving Text

Perhaps the most common use of buffers in qedx is for moving text from one part of a segment to another. A typical pattern is to place the text to be moved into an auxiliary buffer with a move request. For example, the request:

```
1,5m(temp)
```

moves lines 1 through 5 of the current buffer into the auxiliary buffer temp and deletes them from the current buffer. Once the lines have been moved to an auxiliary buffer, they can be used as literal text in conjunction with an input request. For example, to insert the lines in buffer temp immediately before the last line in the current buffer, the following sequence might be used.

```
$i
\b(temp)\f
```

In this case, the literal text in buffer temp replaces the \b escape sequence and thus is treated as input to the editor already placed in input mode by the insert (i) request. Notice that the \f immediately follows the \b escape sequence. If the \f is put on the line following the \b escape sequence, a blank line will follow the literal text of buffer temp that was just inserted in the current buffer. (The blank line is caused by the two successive newline characters: one is the last character in buffer temp and the other is inserted between the \b and the \f escape sequences.)

Repeated Editor Sequences

Another common use for buffers is for the definition of frequently used editing sequences. For example, if a programmer were faced with the task of adding the same source code sequence in several places in a program, he might elect to type the editing sequence into a buffer only once and then invoke the contents of the buffer as many times as necessary. In the example given below, buffer NEW contains the necessary editor requests and literal text to append four lines of text at any point in the current buffer.

Example: Buffer NEW contents:

```
a
if  code ^=0 then do;
    call error (code);
    return;
    end;
\f
.-4,.1p
```

Request:

```
ADR\b(new)
```

Result: ADR becomes the address of the append request in buffer NEW and specifies the point at which the literal text is to be appended. The four lines of text in buffer NEW (lines 2-5) are appended to the current buffer; the \f terminates the append request; and the print request prints the line preceding the appended lines, the four appended lines, and the line following the appended lines.

Use of Editor Macros

The use of buffers in qedx allows a user to place more elaborate editor request sequences (commonly called macros) into auxiliary buffers and use the editor as an interpretive programming language. In this context, it is useful to regard a buffer containing executable editor requests as a subroutine and to view the \b escape sequence as a call statement.

In the example discussed below, a macro is implemented to read ASCII text from the terminal until an input terminating sequence, a line consisting only of ".", is typed. When the terminating sequence is typed, the macro asks the user for a name under which the input is filed and exits from the editor. The macro is implemented with two executable buffers (subroutines) named read and test and is invoked by diverting the input stream to the read buffer (i.e., by calling the read subroutine).

Example: Buffer read contents:

```
e ioa_ Type
$a
\r\f
\b(test)
\b(read)
```

Buffer test contents:

```
s/^\c.$//
d
e ioa_ "Give me a segment name."
w \r
q
```

Explanation of the read buffer:

1. The first request is an escape to the command processor to call ioa_ to print the message "Type" on the user's terminal.

2. The second request ($a) places the editor in input mode to append text to the end of the current buffer (presumably buffer 0).

3. One line is read from the user's terminal (\r) and the append request is terminated (\f).

4. The contents of buffer test is executed (i.e., read "calls" the test "subroutine").

5. When and if test "returns", the contents of buffer read are executed again (i.e., read "calls" itself).

Explanation of the test buffer:

1. The first line uses a substitute request to test the current line (i.e., the line just read in by the above append request) for the input terminating sequence (a line consisting only of "."). If the regular expression in the substitute request fails to find the terminating sequence, the remaining requests in buffer test are ignored (i.e., the test subroutine "returns" to its caller).

2. If the terminating sequence is found in step 1, the blank line that previously contained the terminating sequence is deleted.

3. Again ioa_ is used to type a message to the user. This time, the macro asks the user for a segment name in which the input lines appended by the read subroutine are to be stored.

4. The contents of the current buffer containing the input lines are written into a segment, the name of which is read from the terminal by the \r escape sequence.

5. The macro exits from the editor with a quit request. If the quit request were not included, qedx would expect further instructions from the user's terminal at this point.

Initialization of Macros

The editor provides a means through which a qedx macro can be initiated directly from command level. As indicated earlier, qedx can be invoked in the following fashion:

```
qedx path
```

The above command is equivalent to entering the editor with the simple command:

```
qedx
```

and immediately executing the following series of requests:

```
b(exec)
r path.qedx
b0
\b(exec)
```

This request sequence reads the initial macro segment into buffer exec, changes the current buffer back to buffer 0 and executes the contents of buffer exec. This series of requests is sufficient to allow a multibuffer macro to be initialized. For example, the macro given in the previous example can be initialized and run from a segment with the following contents:

```
b(read)$a
e ioa_ Type
$a
\c\r\c\f
\c\b(text)
\c\b(read)
\f
b(test)$a
s/^\c\c.$//
d
e ioa_ "Give me a segment name."
w \c\r
q
\f
b0
\b(read)
```

The contents of the read and test buffers are initialized with append requests. Notice that all escape sequences placed into a buffer as literal text must be preceded by a \c escape sequence. Thus, the second line input to the read buffer is input as:

```
     \c\r\c\f
```

and produces the following line:

```
     \r\f
```

In addition to the above, the qedx editor can be invoked with more than one argument. Thus, the command line:

```
     qedx read path
```

is the equivalent of:

```
     qedx
     b(exec)
     r read.qedx
     b(args)
     a
     path
     \f
     b0
     \b(exec)
```

If the contents of read.qedx is:

```
     r \b(args)
```

then the contents of the exec and args buffers become:

exec	args
r \b(args)	path

and the request \b(exec) reads the segment path into buffer 0. The editor then waits for further commands from the user.

With the same contents of read.qedx, the invocation:

```
qedx read path 1,$s/x/y/ w q
```

enters into the exec and args buffers the following:

```
exec           args

r \b(args)     path
               1,$s/x/y/
               w
               q
```

This causes the editor to read the segment path into buffer 0, substitute for every occurrence of x the character y, write out the segment path, and then quit and return to command level.

Notes on Macro Use

Since the name of the segment to be read in appears on the command line, this feature allows users to use abbreviations (see the description of the abbrev command) for the names of segments to be edited.

There is no safeguard to keep the editor from changing a buffer from which it is also accepting editor requests. If this is attempted, havoc can well be the result.

Name: ready, rdy

The ready command types out an up-to-date ready message giving the time of day as well as the amount of CPU time and page faults used since the last ready message was typed. If the user is not at the first command level, i.e., if some computation has been suspended and the stack involved not released, the ready message also contains the number of the current command level and the stack frame number of the ready command.

Usage

```
     ready
```

Note

See the descriptions of the ready_on and ready_off commands.

Name: ready_off, rdf

The ready_off command allows the user to turn off the ready message typed on the terminal after the processing of each command line. Automatic typing of the message is suspended until a ready_on command is given.

Usage

```
     ready_off
```

Note

See the descriptions of the ready and ready_on commands.

Name: ready_on, rdn

The ready_on command causes the ready message to be automatically typed on the terminal after each command line has been processed. Since automatic printing of the ready message is in effect until a ready_off command is called, the ready_on command is generally used only to "cancel" the ready_off command.

Usage

```
ready_on
```

Note

See the descriptions of the ready and ready_off commands.

Name: release, rl

The release command releases the stack history that was automatically preserved after a quit signal or unclaimed signal. That is, the Multics stack is returned to a point immediately prior to the stack frame of the command that was being executed when the most recent quit signal or unclaimed signal occurred.

Usage

```
     release -control_arg-
```

where control_arg is an optional control argument (either -all or -a) that releases the stack history preserved (and not already released) after all previous quit and/or unclaimed signals rather than after only the most recent quit or unclaimed signal.

Name: rename, rn

The rename command replaces a specified segment, multisegment file, directory, or link name by a specified new name, without affecting any other names the entry might have.

Usage

```
rename path1 name1 ... pathn namen
```

where:

1. path*i* specifies the old name that is to be replaced; it can be a pathname or an entryname.

2. name*i* specifies the new name that replaces the storage system entryname portion of path*i*.

Notes

The star and equals conventions can be used.

The user's access mode with respect to the directory specified by path*i* must contain the modify attribute.

Since two entries in a directory cannot have the same entryname, special action is taken by this command if name*i* already exists in the directory specified by path*i*. If the entry having the entryname name*i* has an alternate name, entryname name*i* is removed and the user is informed of this action; the renaming operation then takes place. If the entry having the entryname name*i* has only one name, the entry must be deleted in order to remove the name. The user is asked if the deletion should be done; if the user answers "no", the renaming operation does not take place.

Example

```
rename alpha beta >sample_dir>gamma delta
```

causes alpha, in the user's working directory, to be renamed beta and also causes gamma, in the directory >sample_dir, to be renamed delta.

Name: repeat_query

The repeat_query command repeats the last query (by the command_query_ subroutine) if it has not yet been answered.

This command is useful for reinterpreting questions (asked by other commands) that are garbled.

Usage

```
repeat_query
```

Notes

If no question has been asked, or if the latest question was answered, the error message "no pending query" is printed.

The repeat_query command does not completely restore the environment in effect at the time of the original query. For example, nonstandard attachments of I/O switches are not restored.

Example

Suppose that the system starts to print a question while the user is typing. The query looks like:

```
E@foo.pl1 ?
```

The user signals QUIT and invokes the repeat_query command. The system prints:

```
Do you want to delete the old segment foo.pl1?
```

The user answers and continues.

This page intentionally left blank.

Name: reprint_error, re

This command causes the system condition handler to print its message for a condition that has already been handled and for which stack history is preserved.

Usage

```
reprint_error -control_args-
```

where control_args can be chosen (in any order) from the following list of control arguments:

-depth i, -dh i	indicates which instance of saved fault information is to be used for the message (the most recent instance is depth 1). This control argument can appear only once per command line.
-all, -a	prints messages corresponding to all existing sets of condition information.
-brief, -bf	prints the short form of the message.
-long, -lg	prints the long form of the message.

Notes

If no control argument is specified, the default action results in the selection of slightly less extensive condition information than that printed by the -long control argument, and the default depth is 1.

The message mode options for this command have no effect on the operation of the default error handler as such.

Example

The following example illustrates some ways the reprint_error command might be used.

```
change_error_mode -brief

simf

Error: seg_fault_error

reprint_error -long
depth 1:
Error: Segment-fault error by command_processor_¦404
(>system_library_1>bound_command_loop_¦3046
referencing >udd>m>Smith>simf¦3
(offset is relative to base of segment)
Incorrect access on entry.

setacl simf re

simf

Error: simfault_000001

call_den_test_gate$ob

Error: out_of_bounds while in ring 1

reprint_error  -all -brief
depth 1:
Error: out_of_bounds while in ring 1
depth 2:
Error: simfault_000001
depth 3:
Error: seg_fault_error

reprint_error -long -depth 2
depth 2:
Error: Attempt by >udd>m>Smith>simf$simf¦13
to reference through null pointer
(simfault_000001 condition)

reprint_error -long -depth 1
depth 1:
Error while processing in ring 1:
out_of_bounds at deh_test1¦103
(>system_library_tools>bound_deh_test_¦347
referencing stack_1¦720703 (in process dir)
Attempt to access beyond end of segment.
Entry into lower ring was by
call_den_test_gate$ob¦115
(system_library_tools>bound_deh_test_¦115)
referencing den_test_gate_$gob
```

Name: resource_usage, ru

The resource_usage command enables the user to print a month-to-date report of his resource consumption. It is not possible for a user to invoke this command to obtain information about another user´s resource consumption.

Usage

```
     resource_usage -control_arg-
```

where control_arg provides the optional feature of selecting variable portions of available resource usage information. The valid control arguments can be one of the following:

-total, -tt	prints only total dollar figures including the user´s dollar limit stop and his month-to-date spending (see "Example" below).
-brief, -bf	prints the information selected by the -total control argument, preceding this information with a header and following it by total dollar figures depicting the user´s interactive, absentee, and I/O daemon usage.
-long, -lg	prints the most comprehensive picture of the user´s resource usage. This display includes the information selected by the -brief control argument and includes an expanded report of interactive, absentee, and I/O daemon usage that breaks down the total dollar charges according to shift and queue; breaks down the charged virtual CPU time and terminal connect time into hours; minutes; and seconds; and displays the charged memory units and terminal I/O operations, both expressed in thousands (see "Example" below).

Notes

If no control argument is specified, the default action results in the selection of slightly less extensive resource usage information than that printed by the -long control argument; namely, all dollar charges are printed but resource usage expressed as time is not printed.

The system calculates a user´s month-to-date dollar charges when it creates his process. If a user wishes the most up-to-date figures, he should issue the new_proc command prior to typing resource_usage.

Notice that in a given usage report, shift and queue numbers may not appear in consecutive order because only shifts or queues with accrued charges are listed.

If no dollar limit stop has been set by a user´s project administrator, the resource usage report indicates this by the printing of "open" as the dollar limit entry.

Example

```
     resource_usage -long

Doe.Example  Report from 04/01/74 1014.7 to 04/10/74  1345.8

  Month-To-Date Charge: $  292.61;
  Resource Limit:       $ 1000.00;

     Interactive Usage: $ 254.31; 19 logins, 2 crashes.

          shift  $charge    $limit       cpu   connect  terminal i/o  memory*K
            1     173.91     open    0:14:32  14:59:17           0.0       6.0
            2      54.30     open    0:05:26   4:52:47           0.0       1.9
            4      26.10     open    0:04:13   5:25:18           0.0        .4

     Absentee Usage:    $   18.97;

          queue  $charge      jobs       cpu  memory*K
            1       8.74         1   0:00:49        .4
            3      10.23         1   0:00:49        .6

     IO Daemon Usage:   $   19.32;

          queue  $charge    pieces       cpu   lines/K
            3      19.32        10   0:01:06        12
```

Name: run_cobol, rc

The run_cobol command explicitly initiates execution of a COBOL run unit in a specified "main program". This command is not needed to execute COBOL object programs on Multics; it is used to simulate an environment in which traditional COBOL concepts may be easily defined. This command cannot be called recursively.

Usage

```
run_cobol name -control_args-
```

1. name	is the reference name or pathname of the "main program" in which execution is to be initiated. If a pathname is given, then the specified segment is initiated with a reference name identical to the entryname portion of the pathname. Otherwise, the search rules are used to locate the segment. If the name specified in the PROG-ID statement of the COBOL program (i.e., the entry point name) is different from the current reference name of the object segment, then the name specified here must be in the form A$B where A is the pathname or reference name of the segment and B is the PROG-ID as defined in the IDENTIFICATION DIVISION of the source program.
2. control_args	can be chosen from the following:
-cobol_switch n, -cs n	sets one or more of the eight COBOL-defined "external switches" on, where n is a number from 1 to 8 (or a series of numbers separated by spaces) that corresponds to the numbered external switch. At the outset of the run unit, the default setting of these external switches is off. (The eight external switches are defined in the Multics COBOL Reference Manual, Order No. AS44.)
-no_stop_run, -nsr	avoids establishment of a handler for the stop_run condition. (See "Notes" below.)

Notes

This command enables the user to explicitly define and start execution of a COBOL run unit. A run unit is either explicitly started by the execution of the run_cobol command or implicitly started by the execution of a COBOL object program either by invocation from command level or from a call by another program written in COBOL or another language. A run unit is stopped either by the execution of the STOP RUN statement in a COBOL object program or by invocation of the stop_cobol_run command. For the duration of time after a run unit is started and before it is stopped, it is said to be active. All COBOL programs executed while a run unit is active are considered part of that run unit.

A run unit is a subset of a Multics process; it is stopped when the process is ended. Also, when all programs contained in a run unit are cancelled, the run unit is stopped (refer to the cancel_cobol_program command). Only one run unit may be active at any given time in a process; thus, the run_cobol command cannot be invoked recursively. Additionally, if a run unit has been started implicitly (as described above), the run_cobol command may not be used until that run unit has been stopped; i.e., the run_cobol command does not terminate a currently active run unit.

The explicit creation of a run unit with the run_cobol command performs the following functions:

1. Establishment of a "main program", i.e., a program from which control does not return to the caller. The EXIT PROGRAM statements, when encountered in such a program, have no effect, as required in the COBOL definition. An implicitly started run unit has no "main program". The EXIT PROGRAM statement in all programs contained in such a run unit always causes control to be returned to the caller, even if the caller is a system program, e.g., the command processor.

2. Setting of the COBOL external switches. These switches are set to off unless otherwise specified by the -cobol_switch control argument.

3. User control of the action taken when a STOP RUN statement is executed in a COBOL object program. The action normally taken for STOP RUN is cancellation of all programs in the run unit, closing any files left open. After this has been done, the data associated with any of the programs is no longer available. Thus in a debugging environment, it may be useful to redefine the action taken for STOP RUN. When the run unit is explicitly initiated with the run_cobol command, the STOP RUN statement causes the signalling of the stop_run condition for which a handler is established that performs the normal action described above. If the -no_stop_run control argument is specified, the handler is not established, thus allowing the user to handle the signal himself using other Multics commands. If the user has not provided a handler himself for stop_run and specifies the -no_stop_run control argument, an unclaimed signal results.

The name given in the run_cobol command need not be a COBOL object program. It may be a program produced by any language compiler that provides a meaningful interface with COBOL programs (e.g., PL/I, FORTRAN).

Refer to the following related commands:

display_cobol_run_unit, dcr
stop_cobol_run, scr
cancel_cobol_program, ccp

Name: runoff, rf

The runoff command is used to type out text segments in manuscript form. Output lines are built from the left margin by adding text words until no more words fit on the line; the line is then justified by inserting extra blanks to make an even right margin. Up to 20 lines each of headers and footers can be printed on each page. The pages can be numbered, lines can be centered, and equations can be formatted. Space can be allowed for diagrams. Detailed control over margins, spacing, headers, justification, numbering, and other aspects of format is provided by control lines that begin with a period. Although the control lines are interspersed within the text, they do not appear in the output segment. The output can be printed page by page to allow positioning of paper, or it can be directed into a segment. Characters not available on the device to which output is directed are replaced by blanks. If special symbols must be hand drawn, a separate segment can be created that indicates where each symbol should be placed. The user can define variables and cause expressions to be evaluated; he also has the ability to refer to (and sometimes modify) variables connected with the workings of the runoff command.

Usage

```
runoff paths -control_args-
```

where:

1. paths — are the pathnames of input segments or multisegment files named entryname.runoff. The runoff suffix must be the last component of each entryname; however, the suffix need not be supplied in the command line. If two or more pathnames are specified, they are treated as if runoff had been invoked separately for each one. The segments are printed in the order in which they occur in the invocation of the command.

2. control_args — can be chosen from the following list. Any control argument specified anywhere in the command invocation applies to all segments; control arguments can be intermixed arbitrarily with segment names. Control arguments must be preceded by a minus sign.

   -ball n, -bl n — converts output to a form suitable for an n typeball on a unit equipped with a selectric-type typing element. Acceptable ball numbers are 041, 012, 015, and 963. The default is the form of the terminal device being used. Use of this control argument overrides any specification set by the -device control argument (below).

   -character, -ch — flags certain key characters in the output by putting the line containing the key character in a segment named entryname.chars. The normal output is not affected. Page and line numbers referring to the normal output appear with each flagged line, and reminder characters, enclosed by color-shift characters, are substituted for the key characters. The default set of key and reminder characters corresponds to those unavailable with a 963 typeball, as follows:

Key	Reminder
left square bracket	<
right square bracket	>
left brace	(
right brace	)
tilde	t
grave accent	´

The key and reminder characters can be changed by use of the .ch control line; specifying a blank reminder character removes the associated key character from the set of key characters. If a key character would print normally in the output, it should also appear in a .tr control line to turn it into a blank in the output.

-device n, -dv n
: prepares output compatible with the device specified. This is usually used when the output is stored in a segment to be printed elsewhere. Suitable devices are terminals 2741, 1050, 37, and the bulk output printers, 202 or 300. Use of this control argument overrides any specification set by use of the -ball control argument; if both are used in one invocation of runoff, the last one encountered prevails.

If neither -device nor -ball was specified, the default device type is that from which the user is logged in; any unrecognized device type is assumed to support the entire ASCII character set.

-from n, -fm n
: starts printing at the page numbered n. If the -page control argument is used, printing starts at the renumbered page n.

-hyphenate, -hph
: When this control argument is used, a procedure named hyphenate_word_, that the user supplies, is invoked to perform hyphenation when the next word to be output does not fit in the space remaining in a line (see "Hyphenation Procedure Calling Sequence" at the end of this description). Otherwise, no attempt is made to hyphenate words.

-indent n, -in n
: indents output n spaces from the left margin (default indentation is 0 except for "-device 202", which is the default for -segment and has a default indentation of 20; see also -number below). This space is in addition to whatever indentation is established by use of the .in control word.

-no_pagination, -npgn
: suppresses page breaks in the output.

-number, -nb
: prints source line numbers in the left margin of the output; minimum indentation of 10 is forced.

-page n, -pg n
: changes the initial page number to n. All subsequent pages are similarly renumbered. If the control line .pa

	is used within the segment, the -page control argument is overridden and the page is numbered according to the .pa control line.
-parameter _arg_, -pm _arg_	assigns the argument _arg_ as a string to the internal variable "Parameter".
-pass _n_	processes the source segments _n_ times to permit proper evaluation of expressions containing symbols that are defined at a subsequent point in the input. No output is produced until the last pass.
-segment, -sm	directs output to the segment or multisegment file named entryname.runout. This control argument assumes by default that the material is to be dprinted, so the segment is prepared compatible with device 202 unless another device is specified; thus, unless overridden by the -indent control argument, each printed line in the output segment is preceded by 20 leading spaces so that the text is approximately centered on the page when dprinted.
-stop, -sp	waits for a carriage return from the user before beginning typing and after each page of output (including after the last page of output).
-to _n_	ends printing after the page numbered _n_.
-wait, -wt	waits for a carriage return from the user before starting output, but not between pages.

Notes

A runoff input segment contains two types of lines: control lines and text lines. A control line begins with a period; all other lines are considered text lines. A two-character control word appears in the second and third character positions of each control line. The control word can take a parameter that is separated from the control word by one or more spaces. Lines that are entirely blank are treated as if they contained a .sp 1 control line.

Text lines contain the material to be printed. If an input line is too short or too long to fill an output line, material is taken from or deferred to the next text line. A line beginning with a space is interpreted as a break in the text (e.g., the beginning of a new paragraph) and the previous line is printed as is.

Tab characters (ASCII HT) encountered in the input stream are converted to the number of spaces required to get to the next tab position (11, 21, ...).

When an input text line ends with any of the characters ".", "?", "!", ";", or ":", or with ".", "?", or "!" followed by a double quote or ")", two blanks precede the following word (if it is placed on the same output line), instead of the normal single blank.

The maximum number of characters per input or output line is 361; this permits 120 underlined characters plus the newline character.

Terminology

The following paragraphs describe various terms that are used throughout the runoff description.

FILL AND ADJUST MODES

Two separate concepts are relevant to understanding how runoff formats output: fill mode and adjust mode. In fill mode, text is moved from line to line when the input either exceeds or cannot fill an output line. Adjust mode right justifies the text by inserting extra spaces in the output line, with successive lines being padded alternately from the right and from the left. Initial spaces on a line are not subject to adjustment. Fill mode can be used without adjust, but in order for adjust to work, fill mode must be in effect.

LINE LENGTH

The line length is the maximum number of print positions in an output line, including all spaces and indentations, but not including margins set or implied by the -device, -indent, or -number control arguments.

BREAK

A break ensures that the text that follows is not run together with the text before the break. The previous line is printed out as is, without padding.

SPACING BETWEEN LINES

Vertical spacing within the body of the text is controlled by the three control words: .ss, .ds, and .ms (for single, double, and multiple spacing respectively). Single spacing is the default. Multiple spacing is set by the control line .ms n where n-1 is the number of blank lines between text lines.

PAGE EJECT

A page eject ensures that no text after the control line causing the page eject (e.g., .bp for "begin page") is printed on the current page. The current page is finished with only footers and footnotes at the bottom, and the next text line begins the following page.

MARGINS

There are four margins on the page vertically. The first margin on the page is the number of blank lines between the top of the page and the first header; this margin is set by the .m1 control word. The second, set by .m2, specifies the number of lines between the last header and the first line of text. The third (.m3) is between the last line of text and the first footer. The fourth (.m4) is between the last footer and the bottom of the page. The default for the first and fourth margins is four lines; for the second and third, two lines.

PAGE NUMBERS

As the output is being prepared, a page number counter is kept. This counter can be incremented or set by the user. The current value of the counter can be used in a header or footer through the use of the symbol "%". A page is called odd (even) if the current value of the counter is an odd (even) number.

The page numbers can be output as either arabic (the default) or roman (using the .ro control word).

HEADERS AND FOOTERS

A header is a line printed at the top of each page. A footer is a line printed at the bottom of each page. A page can have up to 20 headers and 20 footers. Headers are numbered from the top down, footers from the bottom up. The two groups are completely independent of each other. Provision is made for different headers and footers for odd and even numbered pages. Both odd and even headers (footers) can be set together by using the .he (.fo) control words. They are set separately by using the .eh, .oh, .ef, and .of control words.

A header/footer control line has two arguments, the <u>line</u> <u>number</u> (denoted in the control line descriptions as "#"), and the <u>title</u>.

The line number parameter of the control line determines which header or footer line is being set. If the number is omitted, it is assumed to be 1, and all previously defined headers or footers of the type specified (odd or even) are cancelled. Once set, a line is printed on each page until reset or cancelled.

The title part of the control line begins at the first nonblank character after the line number. This character is taken to be the delimiting character, and can be any character not used in the rest of the title. If the delimiting character appears less than four times, the missing last parts of the title are taken to be blank. The three parts of the title are printed left justified, centered, and right justified, respectively, on the line. Any or all parts of the title can be null. Justification and centering of a header or footer line are derived from the line length and indentation in effect at the time of the definition of the header or footer, and are used whenever that line is output, regardless of the values at the time of use. Any occurrence of the special character "%" within a title is replaced by the current value of the page counter whenever the title is printed. To cause a percent character to be printed, "%%" must be written in the title. The special character can be changed; see the .cc control word.

Omitting the title in the control line cancels the header or footer with that number, including its space on the page (e.g., ".he 4" cancels the fourth header). A blank line in the header or footer can be achieved by a title consisting entirely of one delimiting character (e.g., ".fo 3 $" makes the third footer a blank line). Omitting both number and title of a header (footer) cancels all headers (footers) of the type specified (e.g., ".oh" cancels all headers that were specified by any .oh control line).

Expressions and Expression Evaluation

An expression can be either arithmetic or string, and consists of numbers and operators in appropriate combinations. All operations are performed in integer format, except that string comparisons are performed on the full lengths of the strings.

The order of precedence for the operators is:
- ^ (bit-wise negation), - (unary)
- *, /, \ (remainder)
- +, - (binary)
- =, <, >, ≠, ≤, ≥ (all are comparison operators that yield -1 for true or 0 for false)
- & (bit-wise AND)
- ¦ (bit-wise OR), ≡ (bit-wise equivalence)

Other guidelines in the use of expressions are as follows:

1. Parentheses can be used for grouping.
2. Blanks are ignored outside of constants.
3. Octal numbers consist of "#" followed by a sequence of octal digits.
4. String constants are surrounded by the double quote character; certain special characters are defined by multiple-character sequences that begin with the * character, as follows:

**	asterisk character
*"	double-quote character
*b	backspace character
*n	newline character
*t	horizontal-tab character
*s	space character
*cnnn	character whose decimal value is nnn (1 to 3 digits)

5. Concatenation of strings is performed by the juxtaposition of the strings involved, in order, left to right.

6. For positive i, k,

   ```
   string_expression(i)
   ```

   and

   ```
   string_expression(i, k)
   ```

   are equivalent to the PL/I substr builtin function references

   ```
   substr(string_expression, i)
   ```

   and

   ```
   substr(string_expression, i, k)
   ```

   respectively.

7. For negative i, the substring is defined as starting -i characters from the rightmost end of the string; for negative k, the substring ends -k characters from the end of the string.

8. Evaluation of substrings takes place after any indicated concatenations; string operations have higher precedence than all the binary operations.

9. In any context other than a .sr control line or in a string comparison, a string expression is converted to an integer in such a way that a one-character string results in the ASCII numeric value of the character.

Expression evaluation takes place under the following conditions:

1. In .sr and .ts control lines

2. In all control lines that accept an "n" or "±n" argument

Definition and Substitution of Variables

Variables can be defined by the use of the .sr control line; their values can be retrieved thereafter by a symbolic reference. Names of the variables are composed of the uppercase and lowercase alphabetic characters, decimal digits, and "_", with a maximum length of 361 characters. When a variable is defined, it is given a type based on the type of the expression that is to be its value, either arithmetic or string. Variables that are undefined at the time of reference yield the null string, which is equivalent to an arithmetic 0.

In substitution of variables, the name of the variable is enclosed by "%"; other occurrences of the character "%" encountered during substitution of variables are replaced by the value of the page counter; if a "%" character is to occur in the resulting output, it must be coded as "%%" (but see also the .cc control word).

Substitution of variables can occur:

1. In expressions if a "%" is found as either the first or second character following the spacing after the control word (substitution of variables takes place before expression evaluation)
2. In .ur control lines
3. In all titles ('part1'part2'part3'), whether in header/footer control lines or as equation lines

Many of the variables internal to runoff are available to the user (a complete list is given at the end of this description). These variables include control argument values (or their defaults), values of switches and counters, and certain special functions. However, the user need not worry about naming conflicts, since an attempt to redefine an internal variable that is not explicitly modifiable is ignored; i.e., the operation of the command is unaffected.

Two special builtin symbols in runoff are provided for use in footnote and equation numbering: "Foot" contains the value of the next footnote number available (or the current footnote if referred to from within the text of the footnote) and "Eqcnt" is provided for equation numbering. The value of "Foot" is incremented by one when the closing .ft of a footnote is encountered. Any reference to "Eqcnt" provides the current value and causes its value to be incremented by one automatically; thus its value should be assigned to a variable, and the variable should then be used in all further references to that equation number.

Default Conditions

When no control words are given, runoff prints the text single spaced, right adjusted, with no headers, no footers, and no page numbers.

If page numbers are substituted in headers or equations, they are arabic.

A page consists of 66 lines, numbered 1 through 66. The first line is printed on line 7, and the last on line 60, if no headers or footers are used. If headers are used, there are four lines of top margin (.m1 4), the headers, two blank lines (.m2 2), and then the text. If footers are used, there are two lines skipped after the text (.m3 2), footers printed, and four lines of bottom margin (.m4 4).

A line is 65 characters long; the left margin is that of the typewriter. The output is compatible with whatever is normal for the device from which the runoff command is executed. The entire segment is printed, with no wait before beginning or between pages.

Control Line Formats

The following discussion gives a description of each of the control words that can be interspersed with the text for format control. Control lines do not cause an automatic break unless otherwise specified. Arguments of the control words are in the following form:

#	integer constant
n	integer expression
±n	integer expression preceded by optional + or - sign
<expression>	arbitrary expression (string or integer)
c	character
cd	character pair
f	segment name
´part1´part2´part3´	a title whose parts are to be left justified, centered, and right justified respectively.

.ad Adjust: text is printed right justified. Fill mode must be in effect for right justification to occur. Fill mode and adjust mode are the default conditions. This control line causes a break.

.ar Arabic numerals: when page numbers (% variable) are substituted into text or control lines as a result of a .ur control line or into a title or equation as it is printed, they are in arabic notation. This is the default condition.

.bp Begin page: the next line of text begins on a new page of output. This control line causes a break.

.br Break: the current output line is finished as is, and the next text line begins on a new output line.

.cc c Control character: this control line changes the character used to surround the names of symbolic variables when they are referenced to _c_. The default special character is "%". The character specified by _c_ must thereafter be used to refer to symbolic variables, while percent signs are treated literally. Either ".cc %" or .cc restores the percent sign as the special character.

.ce n Center: the next _n_ text lines are centered. Control lines and blank lines are not counted as part of the _n_ lines being centered. If _n_ is missing, 1 is assumed. This control line implies ".ne _n_" (or ".ne 2_n_" if double spacing) so that all lines centered are on the same page. A break occurs.

.ch cd.. Characters: each occurrence of the character _c_ is replaced in the chars segment (the segment named entryname.chars) by the character _d_, set off by color-shift characters. If the _d_ character is blank, or an unpaired _c_ character appears at the end of the line, the _c_ character is not flagged; it either occurs as itself in the chars segment or not at all if no other character on the line was flagged.

.ds Double space: begin double spacing the text. This control line causes a break.

.ef # ´part1´part2´part3´
Even footer: this defines even page footer line number #. If # is omitted, 1 is assumed. If both # and the title (parts 1 to 3) are omitted, all even footers defined by any .ef control line are cancelled. For more information, see the previous discussion entitled "Headers and Footers."

.eh # ´part1´part2´part3´
Even header: this defines even page header line number #. If # is omitted, 1 is assumed. If both # and the title (parts 1 to 3) are omitted, all even headers defined by any .eh control line are cancelled. For more information, see the previous discussion entitled "Headers and Footers."

.eq n Equation: the next n text lines are taken to be equations. If n is missing, 1 is assumed. This control line implies ".ne n" (or ".ne 2n" if double spacing) so that all equations are on the same page. The format of the equations should be ´part1´part2´part3´ just as in headers and footers.

.ex text Execute: the remainder of the control line (text) is passed to the Multics command processor. Substitution of variables can occur if the first or second character of text is "%".

.fh ´part1´part2´part3´
Footnote header: before footnotes are printed, a demarcation line is printed to separate them from the text. The format of this line can be specified through the title in the .fh control line. This title is printed in the same manner as headers/footers and equations. The default footnote header is a line of underscores from column one to the right margin.

.fi Fill: this control line sets the fill mode. In fill mode, text is moved from line to line to even the right margin, but blanks are not padded to justify exactly. Fill mode is the default condition. This control line causes a break.

.fo # ´part1´part2´part3´
Footer: even and odd footers are set at the same time; this is equivalent to:

```
     .ef # ´part1´part2´part3´
     .of # ´part1´part2´part3´
```

If # is omitted, 1 is assumed. If both # and the title (parts 1 to 3) are omitted, all footers are cancelled. For more information, see the discussion entitled "Headers and Footers."

.fr c Footnote reset: this control line controls footnote numbering according to the argument c. Permitted values of c are:

- t Footnote counter is reset at the top of each page. This is the default condition.
- f Footnote counter runs continuously through the text.
- u Suppresses numbering on the next footnote.

.ft Footnote: when .ft is encountered, all subsequent text until the next .ft line is treated as a footnote. Any further text on the .ft line is ignored. If a footnote occurring near the bottom of a page does not fit on the page, as much as necessary is continued at the bottom of the next page. If a footnote reference occurs in the bottom or next to bottom line of a page, the current page is terminated and the line with the footnote reference is printed at the top of the next text page.

.gb xxx Go back: the current input segment is searched from the beginning until a line of the form ".la xxx" is found; "xxx" in this case means "the rest of the line." Processing is continued from that point.

.gf xxx Go forward: same as .gb, except search forward from the current position in the input segment.

.he # ´part1´part2´part3´
Header: even and odd headers are set at the same time. This is equivalent to:

```
    .eh # ´part1´part2´part3´
    .oh # ´part1´part2´part3´
```

If # is omitted, 1 is assumed. If both # and the title (parts 1 to 3) are omitted, all headers are cancelled. For more information, see the discussion entitled "Headers and Footers."

.if f <expression>
Insert file: the segment with pathname f.runoff is inserted into the text at the point of the ".if f" control line. The inserted segment can contain both text and control lines. No break occurs. The effect is as if the control line were replaced by the segment. Inserts can be nested to a maximum depth of 30. If a second argument is provided, it is evaluated in the same fashion as the expression in .sr, and its value and type are associated with the identifier "Parameter"; if no second argument is provided the value of "Parameter" remains unchanged (or undefined). (In either case, the prior value of "Parameter" is not pushed down).

.in ±n Indent: the left margin is indented n spaces by padding n leading spaces on each line. The right margin remains unchanged. By default n is 0. The margin can be reset with another ".in n" request. Either .in or ".in 0" resets the original margin. If n is preceded by a plus or a minus sign, the indentation is changed by n rather than reset. This control line causes a break.

.la xxx Label: defines the label xxx for use as the target of the .gb or .gf control word.

.li n	Literal: this request causes the next _n_ lines to be treated as text, even if they begin with ".". If _n_ is not given, 1 is assumed.
.ll ±n	Line length: the line length is set to _n_. The left margin stays the same, and no break occurs. If _n_ is not given, 65 is assumed. If _n_ is preceded by a plus or a minus sign, the line length is changed by _n_ rather than reset.
.ma ±n	Margins: top and bottom margins are set to _n_ lines. If _n_ is preceded by a plus or a minus sign, the margin is changed by _n_ rather than reset. The margin is the number of lines printed above the first header and below the last footer. If _n_ is not given, 4 is assumed. This control line is equivalent to: .m1 ±n .m4 ±n
.mp ±n	Multiple pages: format the output text so that it prints on every _n_th page. This control line is valid only for output intended for the bulk printer. If _n_ is not given, 1 is assumed.
.ms ±n	Multiple space: begin multiple spacing text, leaving (_n_-1) blank lines between text lines. If _n_ is preceded by a plus or a minus sign, the spacing is changed by _n_ rather than reset. If _n_ is not given, 1 is assumed. This control line causes a break.
.m1 ±n	Margin 1: the margin between the top of the page and the first header is set to _n_ lines, or changed by _n_ if _n_ is signed. If _n_ is not given, 4 is assumed.
.m2 ±n	Margin 2: the number of blank lines between the last header and the first line of text is set to _n_, or changed by _n_ if _n_ is signed. If _n_ is not given, 2 is assumed.
.m3 ±n	Margin 3: the number of blank lines printed between the last line of text and the first footer is set to _n_, or changed by _n_ if _n_ is signed. If _n_ is not given, 2 is assumed.
.m4 ±n	Margin 4: the margin between the last footer and the bottom of the page is set to _n_ lines, or changed by _n_ if _n_ is signed. If _n_ is not given, 4 is assumed.
.na	No adjust: the right margin is not adjusted. This does not affect fill mode; text is still moved from one line to another. This control line causes a break.
.ne n	Need: a block of _n_ lines is needed. If _n_ or more lines remain on the current page, text continues as before; otherwise, the current page is ejected and text continued on the next page. No break is implied. If _n_ is not given, 1 is assumed. If several .ne control lines occur consecutively, the _n_'s are _not_ added together; only the largest _n_ has effect.

.nf No fill: fill mode is suppressed, so that a break is caused after each text line. Text is printed exactly as it is in the input segment. This control line causes a break.

.of # ´part1´part2´part3´
Odd footer: this defines odd page footer line number #. If # is omitted, 1 is assumed. If both # and the title (parts 1 to 3) are omitted, all footers defined by any .of control line are cancelled. For more information, see the discussion entitled "Headers and Footers."

.oh # ´part1´part2´part3´
Odd header: this defines odd page header line number #. If # is omitted, 1 is assumed. If both # and the title (parts 1 to 3) are omitted, all headers defined by any .oh control line are cancelled. For more information, see the discussion entitled "Headers and Footers."

.op Odd page: the next page number is forced to be odd by adding 1 to the page number counter if necessary. A break is caused and the current page is ejected. No blank even page is made; the even page number is merely skipped.

.pa ±n Page: the current line is finished as is (i.e., a break occurs) and the current page is ejected. The page number counter is set to n, or is changed by n if n was signed. If n is omitted, the page number counter is not changed.

.pi n Picture: if n lines remain on the present page, then n lines are spaced over; otherwise, the text continues as before until the bottom of the page is reached, then n lines are skipped on the next page before any text is printed. Headers are printed normally; the space resolved is below the headers. This option can be used to allow for pictures and diagrams. If several .pi control lines occur consecutively, each n is added to the number of lines pending and the total is checked against the space remaining on the page. All pending space is allotted together. If the total is greater than the usable space on a page, the next page contains only headers and footers and the rest of the space is left on the following page. If n is not given, 1 is assumed.

.pl ±n Page length: the page length is set to n lines. If n is not given, 66 is assumed. If n is preceded by a plus or a minus sign, the page length is changed by n rather than reset.

.rd Read: one line of input is read from the user_input I/O switch; this input line is then processed as if it had been encountered instead of the .rd control line. Thus it can be either a text line or a control line; a break occurs only if the replacement line is one that would cause a break.

.ro Roman numerals: when page numbers (% variable) are substituted into text or control lines as a result of a .ur control line or into a title or equation as it is printed, they are in lowercase roman notation. This can be reset to arabic numerals (the default) by use of the .ar control line.

.rt Return: cease processing characters from the current input segment. If the current input segment was entered by a .if control line in another segment, return to the line following the .if control line.

.sk n Skip: n page numbers are skipped before the next new page by adding n to the current page number counter. No break in text occurs. This control line can be used to leave out a page number for a figure. If n is not given, 1 is assumed.

.sp n Space n lines: If n is not given, 1 is assumed. If not enough lines remain on the current page, footers are printed and the page ejected, but the remaining space is not carried over to the next page. The n blank lines are added to those caused by either a .ds or .ms control line. For example, using a .ms 4 control line causes a .sp 1 to space 4 lines, .sp 2 to space 5 lines, and .sp 3 to space 6 lines. This control line causes a break. A blank line occurring in text is treated as if it were a .sp 1 control line.

.sr name <expression>
Set reference: associates value of <expression> with the identifier name. The type of name is set to the type of <expression> (either numeric or string); if the expression is not provided or cannot be properly evaluated, a diagnostic message is printed. The name identifier can be either a user-defined identifier or one of the built-in symbols that the user can set (see "Built-In Symbols" below).

.ss Single space: begin single spacing text. This is the default condition. This control line causes a break.

.tr cd.. Translate: the nonblank character c is translated to d in the output. An arbitrary number of cd pairs can follow the initial pair on the same line without intervening spaces. An unpaired c character at the end of a line translates to a blank character. (Translation of a graphic character to a blank only in the output is useful for preserving the identity of a particular string of characters, so that the string is neither split across a line, nor has padding inserted within it.) If several .tr control lines are used in a segment, the cd pairs are "added together." Also a particular c character can be translated to a different d character by using a new .tr control line to override the previous translation. To cancel a cd pair (i.e., have the c character print out as itself), use another .tr control line of the form ".tr cc". A .tr control line with no cd pair is ignored.

.ts n Test: process the next input line if the value of n does not equal zero (false). If n is not given, 1 is assumed.

.ty xxx	Type: write xxx (i.e., the rest of the control line) onto the error_output I/O switch. Substitution of variables can occur if the first or second character of xxx is "%". If xxx is omitted, a blank line is written onto the I/O switch.
.un n	Undent: the next output line is indented n spaces less than the current indentation. Adjustment, if in effect, occurs only on that part of the line between the normal left indentation and the right margin. If n is not specified, its value is the current indentation value (i.e., the next output line begins at the current left margin). This control line causes a break.
.ur text	Use reference: the remainder of the .ur control line (text) is scanned, with variables of the form "%name%" replaced by their corresponding values (converted back to character string form if they were numeric). The line thus constructed is then processed as if it had been encountered in the original input stream (e.g., it can be another control line, including possibly another .ur).
.wt	Wait: read one line from the user_input I/O switch and discard it (see the .rd control word description).
.*	This line is treated as a comment and ignored. No break occurs.
.~	This line is treated as a comment and ignored with respect to the output segment. However, the line is printed in the appropriate place in the chars output segment.

Summary of Control Arguments

-ball n, -bl n	Convert output to a form suitable for an n typeball.
-character, -ch	Create entryname.chars, listing page and line numbers with red reminder characters where certain characters, normally not printable, must be drawn in by hand.
-device n, -dv n	Prepare output compatible with device n.
-from n, -fm n	Start printing at the page numbered n.
-hyphenate, -hph	Call user-supplied procedure to perform hyphenation.
-indent n, -in n	Set initial indentation to n.
-no_pagination, -npgn	Suppress page breaks.
-number, -nb	Print source segment line numbers in output.
-page n, -pg n	Change the initial page number to n.
-parameter arg, -pm arg	Assign arg as a string to the internal variable "Parameter".
-pass n	Make n passes over the input.
-segment, -sm	Direct output to the segment or multisegment file named entryname.runout, where entryname is the name of the input segment.
-stop, -sp	Wait for a carriage return before each page.
-to n	Finish printing after the page numbered n.
-wait, -wt	Wait for a carriage return before the first page.

Summary of Control Words

The following conventions are used to specify arguments of control words:

#	integer constant
c	character
cd	character pair
exp	expression (either numeric or string)
n	integer expression
±n	± indicates update by n; if sign not present, set to n
f	segment name
t	title of the form ´part1´part2´part3´

Request	Break	Default	Meaning
.ad	yes	on	Right justify text
.ar	no	arabic	Arabic page numbers
.bp	yes		Begin new page
.br	yes		Break, begin new line
.cc c	no	%	Change special character from % to c
.ce n	yes	n=1	Center next n lines
.ch cd....	no		Note "c" in chars segment as "d"
.ds	yes	off	Double space
.ef # t	no		Defines even footer line #
.eh # t	no		Defines even header line #
.eq n	yes	n=1	Next n lines are equations
.ex text	no		Call command processor with "text"
.fh t	no	line of underscores	Format of footnote demarcation line
.fi	yes	on	Fill output lines
.fo # t	no		Equivalent to: .ef # t .of # t
.fr c	no	t	Controls footnote numbering: "t" reset each page "f" continuous "u" numbering suppressed for each footnote
.ft	no		Delimits footnotes
.gb xxx	no		"go back" to label xxx
.gf xxx	no		"go forward" to label xxx
.he # t	no		Equivalent to: .eh # t .oh # t
.if f exp	no		Segment f.runoff inserted at point of request; value of "exp" assigned to "Parameter"
.in ±n	yes	n=0	Indent left margin n spaces
.la xxx	no		Define label xxx
.li n	no	n=1	Next n lines treated as text
.ll ±n	no	n=65	Line length is n
.ma ±n	no	n=4	Equivalent to: .m1 ±n .m4 ±n
.mp ±n	no	n=1	Print only every n-th page
.ms ±n	yes	n=1	Multiple space of n lines
.m1 ±n	no	n=4	Margin above headers set to n
.m2 ±n	no	n=2	Margin between headers and text set to n
.m3 ±n	no	n=2	Margin between text and footers set to n
.m4 ±n	no	n=4	Margin below footers set to n
.na	yes	off	Do not right justify
.ne n	no	n=1	Need n lines; begin new page if not enough remain

Request	Break	Default	Meaning
.nf	yes	off	Do not fill output lines; print them exactly as entered
.of # t	no		Defines odd footer line #
.oh # t	no		Defines odd header line #
.op	yes		Next page number is odd
.pa ±n	yes		Begin page n
.pi n	no	n=1	Skip n lines if n remain; otherwise skip n on next page before any text
.pl ±n	no	n=66	Page length is n
.rd	no		Read one line of text from the user_input I/O switch and process it in place of .rd line
.ro	no	arabic	Roman numeral page numbers
.rt	no		"Return" from this input segment
.sk n	no	n=1	Skip n page numbers before next new page
.sp n	yes	n=1	Space n lines
.sr sym exp	no		Assign value of "exp" to variable named "sym"
.ss	yes	on	Single space
.tr cd....	no		Translate nonblank character c into d on output
.ts n	no	n=1	Process the next input line only if n is not zero
.ty xxx	no		Write "xxx" onto the error_output I/O switch
.un n	yes	left margin	Indent next text line n spaces less
.ur text	no		Substitute values of variables in "text", and scan the line again
.wt	no		Read one line of text from the user_input I/O switch and discard it (for synchronization with terminal)
.*	no		Comment line; ignored
.~	no		Comment line; ignored, but included in chars output segment

Built-In Symbols

Only those symbols marked yes in the Set column can have values assigned by the user.

All symbols are of type Number unless they are specified to be of type String.

Control words and control arguments that affect the values of the variables are indicated in parentheses: (x/y) indicates that x sets the switch to true (-1), and y sets it false (0); (a) or (a, b, c) indicates that it is affected by a or by a, b and c.

Symbol	Set	Value
Ad		Adjust (.ad/.na)
Ce		Number of lines remaining to be centered (.ce)
CharsTable	yes	Translation table for chars segment output (String) (.ch)
Charsw	yes	A chars segment is being created (-character)

Symbol	Set	Value
ConvTable	yes	Translation table for output. Product of DeviceTable and TrTable (String) (.tr, -device)
Date		Date of this invocation of runoff; format is mm/dd/yy (String)
Device	yes	Type of device output is to be formatted for (-device, -ball, -segment)
DeviceTable	yes	Translation table for physical device (String) (-device)
Eq		Equation line counter (.eq)
Eqcnt	yes	Equation reference counter (incremented each reference)
ExtraMargin	yes	Indent entire text this many spaces (-segment, -device, -indent)
Fi		Fill switch (.fi/.nf)
FileName		Name of current primary input segment (String)
Filesw		True if output is going to a segment (-segment)
Foot	yes	Footnote counter (.ft, .fr)
FootRef	yes	Footnote reference string in footnote body (String)
Fp	yes	First page to print (set at the beginning of each pass to the value of From)
Fr		Footnote counter reset switch
From	yes	First page to print (-from)
Ft		Footnote processing switch (.ft)
Hyphenating	yes	True if an attempt to break a word should be made (-hyphenate)
In		Indent to here (.in)
InputFileName		Name of current input segment (String) (.if)
InputLines		Current line number in current source file
LinesLeft		Number of usable text lines left on this page
Ll		Line length (.ll)
Lp	yes	Last page to print (initialized each pass from To)
Ma1		Space above header (.ma, .m1)
Ma2		Space below header (.m2)
Ma3		Space above foot (.m3)
Ma4		Space below foot (.ma, .m4)
Ms		Spacing between lines (ss = 1, ds = 2, etc.) (.ms, .ss, .ds)
MultiplePagecount		Form feeds between pages to printer (.mp)
NestingDepth		Index into stack of input files (.if)
Nl		Last used line number
NNp	yes	Next page number (-page, .pa)
NoFtNo		True to suppress number on next footnote reference (.fr)
NoPaging	yes	True if no pagination is desired (-no_pagination)
Np	yes	Current page number (.pa, -page, initialized each pass from Start)
PadLeft		Alternate left/right padding switch (.un, .ad)
Parameter	yes	Argument passed during insert processing (-parameter, .if)
Passes	yes	Number of passes left to make (= 1 when printing is being performed) (-pass)
Pi		Space needed for pictures (.pi)
Pl		Page length (.pl)
Print	yes	Whether or not to print (($Fp \leq Np \leq Lp$) & ($Passes \leq 1$))
Printersw		Output is intended for bulk printer (-device, -segment)
PrintLineNumbers	yes	True if source line numbers are to be printed in output (-number)

Symbol	Set	Value
Roman		Roman numeral pagination (.ro/.ar)
Selsw		True if typeball other than 963 is being used (-ball)
Start	yes	Initial page number (-page)
Stopsw	yes	Stop between pages of output (-stop)
TextRef	yes	Footnote reference string in main text (String)
Time		Local time, in seconds, since January 1, 1901.
To	yes	Last page to be printed (-to)
TrTable	yes	Translation table for user-supplied substitutions (String) (.tr)
Un		Undent to here (.un)
Waitsw	yes	Wait for input before printing first page (-wait)

Hyphenation Procedure Calling Sequence

The runoff command provides a means whereby a user-supplied program can be called whenever the space available on a line is less than the length of the next word (including attached punctuation, if any). The mechanism is activated by use of the -hyphenate control argument, and the PL/I calling sequence is provided below.

```
     declare hyphenate_word_ entry(char(*) unaligned, fixed bin, fixed bin);

     call hyphenate_word_(string, space, break);
```

where:

1.	string	is the text word that is to be split. (Input)
2.	space	is the number of print positions remaining in the line. (Input)
3.	break	is the number of characters from the word that should be placed on the current line; it should be at least one less than the value of <u>space</u> (to allow for the hyphen), and can be 0 to specify that the word is not to be broken. Thus if the word "calling" is to be split, and 6 spaces remain in the line, the procedure should return the value 4 (adjustment is performed after hyphenation). (Output)

Example

The following pages show the creation of a runoff segment and the result of invoking the runoff command on that segment. For an explanation of any of the control lines, refer to the respective control word definition earlier in this command description. Particularly notice the following:

1. The line length control is given before any headers and footers. If the user wants a line length other than the default one, he must specify it before he specifies his headers and footers; if he does not, the headers and footers on the first page are formatted for the default line length.

2. The .sr control line associates the page number count, at the time the title "RUNOFF SAMPLE PAGE" is printed, with the identifier rfsample. Refer to the last line of the segment (the .ur control line) to see how this reference is used.

3. The translate character (!) is used both to "count" spaces (see the a, b, and c items of 2, below) and to prevent an unattractive line split (see the last line of the segment.

```
qedx
a
.pl 84
.ll 80
.tr !
.fo 1 $$3-%$AG92$
.fo 2 $
.fo 3 $runoff sample page$runoff sample page$runoff sample page$
.he "______""______"
.he 2 $
.he 3 XrunoffXXrunoffX
.he 4 8______88______8
.m1 6
.m2 3
.m3 2
.m4 6
.sp 7
.ce
RUNOFF SAMPLE PAGE
.sr rfsample %
.sp 2
     The runoff command lets the user format his text segments through
a variety of control words.  The control words specify such things as:
.sp 2
.in 10
.un 5
1.   Page length and line length (.pl and .ll respectively).
If not specified by the user, these control words are given
default values of:
.sp
.in +5
.li
.pl 66
.br
.li
.ll 65
.in -5
```

```
.sp
.un 5
2.   Headers and footers, for all pages or for just
odd numbered or just even numbered pages.  The control words for
headers and footers are as follows:
.sp
.in +5
.un 5
a.!!!Headers and footers on both odd and even numbered pages (.he and .fo)
.sp 1
.un 5
b.!!!Headers and footers on just odd numbered pages (.oh and .of)
.sp 1
.un 5
c.!!!Headers and footers on just even numbered pages (.eh and .ef)
.sp 1
.in -5
.un 5
3.   Margins that control vertical spacing in relation to the top
of the page, headers, text, footers, and the bottom of the page.
These margins are defined as follows:
.sp
.in +5
.un 5
a.   Between top of page and first header (.m1)
.sp
.un 5
b.   Between last header and first line of text (.m2)
.sp
.un 5
c.   Between last line of text and first footer (.m3)
.sp
.un 5
d.   Between the last footer and bottom of page (.m4)
.in -5
.sp
If not specified by the user, these margins are given default
values of:
.sp
.in +5
.nf
.li 4
.m1 4
.m2 2
.m3 2
.m4 4
.fi
.in 0
.ce 3
.li 3
.
.
.
.sp 2
     To see the runoff segment that created this page,
.ur refer to page 3-%rfsample% in this document (Section!III).
\f
w example.runoff
q
<ready message>

rf example -wt
```

RUNOFF SAMPLE PAGE

The runoff command lets the user format his text segments through a variety of control words. The control words specify such things as:

1. Page length and line length (.pl and .ll respectively). If not specified by the user, these control words are given default values of:

   .pl 66
   .ll 65

2. Headers and footers, for all pages or for just odd numbered or just even numbered pages. The control words for headers and footers are as follows:

   a. Headers and footers on both odd and even numbered pages (.he and .fo)

   b. Headers and footers on just odd numbered pages (.oh and .of)

   c. Headers and footers on just even numbered pages (.eh and .ef)

3. Margins that control vertical spacing in relation to the top of the page, headers, text, footers, and the bottom of the page. These margins are defined as follows:

   a. Between top of page and first header (.m1)

   b. Between last header and first line of text (.m2)

   c. Between last line of text and first footer (.m3)

   d. Between the last footer and bottom of page (.m4)

   If not specified by the user, these margins are given default values of:

   .m1 4
   .m2 2
   .m3 2
   .m4 4

.
.
.

To see the runoff segment that created this page, refer to page 3-299 in this document (Section III).

Name: runoff_abs, rfa

This command submits an absentee request to process text segments using the runoff command. The absentee process prepares, in manuscript form, an output segment for each text segment and stores each output segment in the user's working directory. The name of the output segment is the name of the text segment with the suffix runoff replaced by runout. The absentee process then uses the dprint command to queue each output segment for printing and deletion. Printing and deletion can be withheld if desired. If the -output_file control argument (one of those recognized by the enter_abs_request command) is not specified, the absentee process's output segment is placed in the user's working directory with the name path1.absout, where path1 is the first argument of the command. (See "Usage" below.)

Usage

```
runoff_abs paths -rf_args- -ear_args- -dp_args- -abs_control_args-
```

where:

1.	paths	are absolute or relative pathnames specifying the segments to be processed by the runoff command. They need not specify the runoff suffix; however, this suffix appears in the actual segment name. If more than one pathname is given, each segment is considered a separate runoff task.
2.	rf_args	can be one or more control arguments accepted by the runoff command.
3.	ear_args	can be one or more control arguments accepted by the enter_abs_request command. The -brief (-bf) control argument is not permitted here.
4.	dp_args	can be one or more control arguments (except -brief and -truncate) accepted by the dprint command.
5.	abs_control_args	can be one of the following:
	-queue n, -q n	specifies in which priority queue the request is to be placed ($n \leq 3$). The default queue is 3.
	-copy n, -cp n	specifies the number of copies of the segment to be dprinted ($n \leq 4$). The default is 1.
	-hold	specifies that the output segments created by runoff should neither be queued for printing nor deleted. Each output segment is formatted for printing on a selectric-type terminal, with a 963 type ball, unless some other output form is specified by one of the runoff control arguments.

Notes

When doing several runoffs, it is more efficient to give several pathnames in one command, since only one process is set up with one command. Thus the cost of process initialization need be incurred only once.

Control arguments and pathnames can be mixed freely in the command line. All control arguments apply to all pathnames. An unrecognizable control argument causes the absentee request not to be submitted.

Name: safety_sw_off, ssf

This command turns off the safety switch of a segment, directory, or multisegment file, thus permitting the segment, directory, or multisegment file to be deleted. See "Segment, Directory, and Link Attributes" in Section III of the MPM Reference Guide for a description of the safety switch.

Usage

```
     safety_sw_off -paths-
```

where paths are the pathnames of the segments, directories, or multisegment files that should have their safety switches turned off. If one of the paths is -wd or -working_directory, the safety switch of the working directory is turned off. If no paths are given, the safety switch of only the working directory is turned off. The star convention can be used to specify a set of segments, directories, and/or multisegment files.

Note

the safety switch of the working directory.

Examples

```
     safety_sw_off test.pl1 check.fortran
```

turns off the safety switch of the segments test.pl1 and check.fortran in the working directory.

```
     ssf *.temp_dir -wd
```

turns off the safety switch of all segments, directories, and multisegment files (in the working directory) with a two-component name ending in temp_dir, and turns off the safety switch of the working directory.

```
     ssf
```

turns off the safety switch of the working directory.

Control arguments and pathnames can be mixed freely in the command line. All control arguments apply to all pathnames. An unrecognizable control argument causes the absentee request not to be submitted.

Name: safety_sw_off, ssf

This command turns off the safety switch of a segment, directory, or multisegment file, thus permitting the segment, directory, or multisegment file to be deleted. See "Segment, Directory, and Link Attributes" in Section III of the MPM Reference Guide for a description of the safety switch.

Usage

```
safety_sw_off -paths-
```

where paths are the pathnames of the segments, directories, or multisegment files that should have their safety switches turned off. If one of the paths is - d or -working_directory, the safety switch of the working directory is turned off. If no paths are given, the safety switch of only the working directory is turned off. The star convention can be used to specify a set of segments, directories, and/or multisegment files.

Note

If the safety_sw_off command is given with no arguments, the default is to turn off the safety switch of the working directory.

Examples

```
safety_sw_off test.pl1 check.fortran
```

turns off the safety switch of the segments test.pl1 and check.fortran in the working directory.

```
ssf *.temp_dir -wd
```

turns off the safety switch of all segments, directories, and multisegment files (in the working directory) with a two-component name ending in temp_dir, and turns off the safety switch of the working directory.

```
ssf
```

turns off the safety switch of the working directory.

Name: safety_sw_on, ssn

This command turns on the safety switch of a segment, directory, or multisegment file, thus preventing deletion of that segment, directory, or multisegment file. When the user invokes a command to delete a segment, directory, or multisegment file that has the safety switch turned on, he is asked if he wishes to make the deletion. The specified item is deleted only if the user answers "yes". See "Segment, Directory, and Link Attributes" in Section III of the MPM Reference Guide for a description of the safety switch.

Usage

```
safety_sw_on -paths-
```

where paths are the pathnames of the segments, directories, or multisegment files that should have their safety switches on. If one of the paths is -wd or -working_directory, the safety switch of the working directory is turned on. If no paths are given, the safety switch of only the working directory is turned on. The star convention can be used to specify a set of directories, segments, or multisegment files.

Note

Since the delete_dir command already asks the user if he wants to delete the specified directories, no further query is made. That is, the delete_dir command functions in the same manner whether the safety switch is on or off. If the safety_sw_on command is given with no arguments, the default is to turn on the safety switch of the working directory.

Examples

```
safety_sw_on *.alpha
```

turns on the safety switch of all segments, or multisegment files, and directories found in the working directory with two-component names ending in alpha.

```
ssn
```

turns on the safety switch of the working directory.

Name: send_message, sm

The send_message command sends messages (one or more, always sent one line at a time) to a given user on a given project. The messages are placed in the mailbox:

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

where Person_id and Project_id are arguments to the send_message command.

If the recipient is accepting messages (see the accept_messages and defer_messages commands), send_message causes each message to be printed immediately on the recipient's terminal.

Usage

```
send_message Person_id Project_id -message-
```

where:

1.	Person_id	is the registered name of the recipient.
2.	Project_id	is the name of the recipient's project.
3.	message	is an optional string that can be up to 132 characters long. If message is missing from the command line, send_message types "Input." and accepts lines that it sends, one line at a time, with each newline character. In this case, input is terminated by a line consisting solely of a period.

Notes

Parentheses, quotes, brackets, and semicolons in the command line have their usual command language interpretation. This means, for example, that:

```
sm Person_id Project_id testing complete; installation this week
```

sends:

```
testing complete
```

and prints an appropriate error message (probably "Segment installation not found.") because the characters typed after the semicolon are another command line.

```
     sm Person_id Project_id so long (for now)
```

sends two lines:

```
     so long for
     so long now
```

In both of these examples, the sender's intended message would have been sent if he had enclosed the message in quotes (e.g., "so long (for now)").

*

The user can receive messages while he is in send_message input mode. He can therefore carry on a conversation with a single invocation of the command.

Example

If WJones on the Alpha project sends the following to RTSmith on the Beta project:

```
     sm RTSmith Beta need access to your lsg command
```

the message would print on RTSmith's terminal (if RTSmith is accepting messages) as follows:

```
     from WJones.Alpha 04/20/76 1200.6 mst Tue: need access to your lsg  command
```

Name: set_acl, sa

The set_acl command manipulates the access control lists (ACLs) of segments, multisegment files, and directories. See "Access Control" in Section III of the MPM Reference Guide for a discussion of ACLs.

Usage

```
     set_acl path mode1 User_id1 ... moden -User_idn- -control_args-
```

where:

1. path	is the pathname of a segment, multisegment file, or directory. If it is -wd or -working_directory, the working directory is assumed. The star convention can be used and applies to either segments and multisegment files or directories, depending on the type of mode specified in mode1.
2. modei	is a valid access mode. For segments or multisegment files, any or all of the letters rew; for directories, any or all of the letters sma with the requirement that if "m" is present, "s" must also be present. Use null, "n" or "" to specify null access.
3. User_idi	is an access control name that must be of the form Person_id.Project_id.tag. All ACL entries with matching names receive the mode modei. (For a description of the matching strategy, see "Notes" below.) If no match is found and all three components are present, an entry is added to the ACL. If the last modei has no User_id following it, the user's Person_id and current Project_id are assumed.
4. control_args	may be either of the following control arguments:
-directory, -dr	specifies that only directories are affected.
-segment, -sm	specifies that only segments and multisegment files are affected. This is the default.
	Either control argument is used to resolve an ambiguous choice between segments and directories that occurs only when modei is null and the star convention is used in path.

Notes

The arguments are processed from left to right. Therefore, the effect of a particular pair of arguments can be changed by a later pair of arguments.

The user needs modify permission on the containing directory.

The strategy for matching an access control name argument is defined by three rules:

1. A literal component, including "*", matches only a component of the same name.
2. A missing component not delimited by a period is treated the same as a literal "*" (e.g., "*.Multics" is treated as "*.Multics.*"). Missing components on the left must be delimited by periods.
3. A missing component delimited by a period matches any component.

Some examples of User_ids and which ACL entries they match are:

..*	matches only the literal ACL entry "*.*.*".
Multics	matches only the ACL entry "Multics.*.*". (The absence of a leading period makes Multics the first component.)
JRSmith..	matches any ACL entry with a first component of JRSmith.
..	matches any ACL entry.
.	matches any ACL entry with a last component of *.
""	(null string) matches any ACL entry ending in ".*.*".

Examples

```
     set_acl *.pl1 rew *
```

adds to the ACL of every segment in the working directory that has a two-component name with a second component of pl1 an entry with mode rew to *.*.* (everyone) if that entry does not exist; otherwise it changes the mode of the *.*.* entry to rew.

```
     sa -wd sm Jones.Faculty
```

adds to the ACL of the working directory an entry with mode sm for Jones.Faculty.* if that entry does not exist; otherwise it changes the mode of the Jones.Faculty.* entry to sm.

```
     sa alpha.basic rew .Faculty. r Jones.Faculty.
```

changes the mode of every entry on the ACL of alpha.basic with a middle component of Faculty to rew, then changes the mode of every entry that starts with Jones.Faculty to r.

Name: set_bit_count, sbc

This command sets a specified bit count on a specified entry, and changes the bit count author for that entry to be the user who invoked the command. See "Segment, Directory, and Link Attributes" in Section III of the MPM Reference Guide for a description of the bit count of an entry.

Usage

```
set_bit_count path1 count1 ... pathn countn
```

where:

1. path*i* is the pathname of the entry whose bit count is to be set. If path*i* is a link, the bit count of the entry linked to is set.

2. count*i* is the bit count, in decimal, desired for path*i*.

Notes

Setting the bit count on a directory is permitted, but several system modules then regard the directory as a multisegment file.

The user must have write access on the entry for which the bit count is to be set.

Name: set_cc

This command sets the carriage control transformation for a specified FORTRAN formatted file either on or off.

When the transformation is on, the first character of each line written to the file is changed to a control character in accordance with the following table.

Character	Resulting Control Character
0	Newline 012 (double space)
1	Newpage 014 (page eject)
blank	None (single space)
+	The previous line and the current line are written as a single line split by a carriage return character. This causes the second line to overprint the first. If the file is attached to a terminal, the + is ignored. The result is a single space between lines.

When the transformation is off, the first character is not changed. The default is off for all files except for file06 and file42, for which the default is on. For more information, see Section IV of the Multics FORTRAN manual, Order No. AJ28.

Usage

```
set_cc filenn -control_arg-
```

where:

1. filenn is the name of the FORTRAN file in the range of file01 to file99. If filenn is out of range, an error message is printed.

2. control_arg can be one of the following:

   -off turns the carriage control transformation off for the specified FORTRAN file.

   -on turns the carriage control transformation on for the specified FORTRAN file.

Example

To turn off the carriage control transformation for the FORTRAN file named file06, type:

```
set_cc file06 -off
```

Name: set_com_line, scl

The set_com_line command allows the user to change the maximum size of expanded command lines. The default size is 132 characters. An expanded command line is one obtained after all active functions have been processed.

Usage

```
     set_com_line -size-
```

where size is the new maximum expanded command line size. If size is not specified, the line size is restored to its default of 132 characters.

Notes

The get_com_line command prints the current value of the maximum size of expanded command lines.

For a discussion of the command language (including the treatment of active functions), see "The Command Language" in Section I of the MPM Reference Guide.

Name: set_iacl_dir, sid

This command adds entries to a directory initial access control list (initial ACL) in a specified directory or modifies the access mode in an existing directory initial ACL entry. A directory initial ACL contains the ACL entries to be placed on directories created in the specified directory. For a discussion of initial ACLs, see "Access Control" in Section III of the MPM Reference Guide.

Usage

```
set_iacl_dir path mode1 User_id1 ... moden -User_idn- -control_arg-
```

where:

1.	path	specifies the directory in which the directory initial ACL should be changed. If it is -wd or -working_directory, the directory initial ACL for the working directory is changed. The star convention can be used.
2.	mode*i*	is the mode associated with User_id*i*. It can consist of any or all of the letters sma (status, modify, append) except that if "m" is given, "s" must also be given. To specifically deny access to User_id*i*, "n", "", or null should be used for mode*i*.
3.	User_id*i*	is an access control name that must be of the form Person_id.Project_id.tag. If one or more of the components is missing, all entries that match User_id*i* are changed to mode*i*. (For a description of the matching strategy, refer to the set_acl command.) If all three components are present, the ACL entry with that name is changed to mode*i*, or one is added if none exists. If the last mode*i* has no User_id*i* following it, the user's name and project are assumed.
4.	control_arg	can be -ring *n* (or -rg *n*) to identify the ring number whose directory initial ACL should be set. It can appear anywhere on the line, except between a mode and its associated User_id, and affects the whole line. If present, it must be followed by *n* (where user's ring $\leq n \leq 7$). If the control argument is not given, the user's ring is assumed.

Examples

```
     set_iacl_dir listings sm * -ring 5
```

adds to the ring 5 directory initial ACL of the listings directory an entry with the mode sm for *.*.* (everyone) if that entry does not exist; otherwise it changes the mode of the *.*.* entry to sm.

```
     sid -wd sa Jones..
```

changes the mode of all entries with Person_id Jones in the directory initial ACL of the working directory to sa. If no such entries exist, an error message is printed.

Name: set_iacl_seg, sis

This command adds entries to a segment initial access control list (initial ACL) in a specified directory or modifies the access mode in an existing segment initial ACL entry. A segment initial ACL contains the ACL entries to be placed on segments created in the specified directory. For a discussion of initial ACLs see "Access Control" in Section III of the MPM Reference Guide.

Usage

```
     set_iacl_seg path mode1 User_id1 ... moden -User_idn- -control_arg-
```

where:

1.	path	specifies the directory in which the segment initial ACL should be changed. If it is -wd or -working_directory, the segment initial ACL for the working directory is changed. The star convention can be used.
2.	mode*i*	is the mode associated with User_id*i*. It can consist of any or all of the letters rew (read, execute, write). To specifically deny access to User_id*i*, "n", "", or null should be used for mode*i*.
3.	User_id*i*	is an access control name that must be of the form Person_id.Project_id.tag. If one or more of the components is missing, all ACL entries that match User_id*i* are changed to mode*i*. (For a description of the matching strategy, refer to the set_acl command.) If all three components are present, the ACL entry with that name is changed to mode*i*, or one is added if none exists.
4.	control_arg	can be -ring *n* (or -rg *n*) to identify the ring number whose segment initial ACL should be set. It can appear anywhere on the line except between a mode and its associated User_id, and affects the whole line. If present it must be followed by *n* (where user's ring $\leq n \leq 7$). If the control argument is not given, the user's ring is assumed.

Examples

```
     set_iacl_seg test rew *
```

adds to the segment initial ACL in the test directory an entry with mode rew for *.*.* (everyone) if that entry does not exist; otherwise it changes the mode of the *.*.* entry to rew.

```
     sis -wd re Jones.. -rg 5
```

changes the mode of all entries with Person_id Jones in the ring 5 segment initial ACL of the working directory to re. If no such entries exist, an error message is printed.

Name: set_search_rules, ssr

The set_search_rules command allows the user to set his dynamic linking search rules to suit his individual needs with only minor restrictions.

Usage

```
set_search_rules path
```

where path is the pathname of a segment containing the ASCII representation of the search rules.

Notes

Two types of search rules are permitted: absolute pathnames of directories to be searched and key words. The key words are:

1.	initiated_segments	check the already initiated segments
2.	referencing_dir	search the parent directory of the segment making the reference
3.	working_dir	search the working directory
4.	home_dir	search the home directory
5.	process_dir	search the process directory
6.	system_libraries	search the default system libraries

Currently, initiated_segments must be the first search rule. If the user decides not to put system_libraries in his search rules, then many standard commands cannot be found. There must be one search rule per line. A maximum of 21 rules is allowed. Leading and trailing blanks are allowed, but embedded blanks are not allowed.

See also the descriptions of the print_search_rules, add_search_rules, and delete_search_rules commands.

Name: set_tty, stty

The set_tty command is used to modify the terminal type associated with the user's terminal and/or the modes associated with terminal I/O. The type as specified by this command is used for determining character conversion and delay timings; it has no effect on communications line control.

Usage

```
set_tty -control_args-
```

where control_args may be chosen from the following control arguments:

-io_switch XX,
-is XX
: specifies that the command is to be applied to the I/O switch whose name is XX. If this control argument is omitted, the user_i/o switch is assumed.

-terminal_type XX,
-ttp XX
: causes the user's terminal type to be set to device type XX, where XX can be any one of the following:

1050	device similar to IBM Model 1050
2741	device similar to IBM Model 2741, EBCDIC codes
CORR2741, corr2741	device similar to IBM Model 2741 with correspondence keyboard and 015 typeball
TTY37, tty37	device similar to Teleytpe Model 37
TTY33, tty33	device similar to Teletype Model 33 or 35
TTY38, tty38	device similar to Teletype Model 38
TN300, tn300	device similar to GE TermiNet 300 or 1200
ARDS, ards	device similar to Adage, Inc. Advanced Remote Display Station (ARDS)
ASCII, ascii	device similar to a Computer Devices Inc. (CDI) Model 1030 or Texas Instruments (TI) Model 725, or a device with an unrecognized answerback, or a device without an answerback (these devices are collectively termed "ASCII" devices)

The default modes for the new terminal type are turned on.

-modes XX
: sets the modes for terminal I/O according to XX, which is a string of mode names separated by commas, each one optionally preceded by "^" to turn the specified mode off. For a list of valid mode names, see the description of the tty_ I/O module in the MPM Subroutines. Modes not specified in XX are left unchanged. See "Notes" below.

-reset
: turns off all modes that are not set in the default modes string for the current terminal type.

-tabs	specifies that the device has software-settable tabs, and that the tabs are to be set. This control argument currently has effect only for GE TermiNet 300-like devices.
-print	causes the terminal type and modes to be printed on the terminal. If any other control arguments are specified, the type and modes printed reflect the result of the command.

Notes

Invoking the set_tty command causes the system to perform the following steps in the specified order:

1. If the -terminal_type control argument is specified, set the specified device type and turn on the default modes for that type.
2. If the -reset control argument is specified, turn off all modes that are not set in the default modes string for the current terminal type.
3. If the -modes control argument is specified, turn on or off those modes explicitly specified.
4. If the -tabs control argument is specified, and the terminal has settable tabs, set the tabs.
5. If the -print control argument is specified, print the type and modes on the terminal.

Name: sort

The sort command provides a generalized file sorting capability, which is specialized for execution by user-supplied parameters. The basic function of the sort is to read one or more input files of unordered records, sort these records according to the values of one or more key fields, and write a single file of ordered (or "ranked") records.

For a detailed description of the sort command, refer to the Multics Sort/Merge Reference Manual, Order No. AW32.

This page intentionally left blank.

Name: sort_seg, ss

The sort_seg command orders the contents of a segment according to the ASCII collating sequence. Using the control arguments, the segment is broken down into separate sort units, which are strings or blocks of strings. A string may comprise one or more lines. These sort units are then sorted, and the ordered units either replace the original segment or are placed in a new segment.

Usage

```
sort_seg path -control_args-
```

1. path	specifies the pathname of the input segment to be sorted.
2. control_args	may be chosen from the following list of control arguments:
-segment path, -sm path	place the sorted units in a segment whose pathname is path. The use of this control argument is incompatible with the use of the -replace control argument.
-replace, -rp	replace the original contents of the input segment with the sorted units. This is the default mode of operation.
-unique, -uq	delete duplicate sort units from the sorted results. The default is to retain any duplicated units (see "Notes" below).
-delimiter XX, -dm XX	use XX concatenated with a newline character as the string delimiter. The characters XX may be any sequence of ASCII characters. The default is a single newline character (see "Examples" below).
-block n, -bk n	make the sort unit a block of n strings where n must be a positive integer. The default for n is 1 (see "Examples" below).
-descending, -dsc	make the sort in descending order, according to the ASCII collating sequence. The use of this control argument is incompatible with the use of the -ascending control argument.
-ascending, -asc	make the sort in ascending order, according to the ASCII collating sequence. This is the default mode of operation.

-field field_spec, -fl field_spec	specifies the field (or fields) when sorting within a sort unit. The field_spec string consists of positive integers separated by spaces: S_1_ L_1_ S_2_ L_2_ ... S_n_ L_n_ notice that the arguments must be given in pairs. The first argument of the pair (represented by "S") is the start position of the field in the sort unit (e.g., 1 if the field begins at the first character). The second argument (represented by "L") is the length of the field, in characters. The first pair of field specifications defines the primary sort field; the second pair defines the secondary sort field; and so forth. The use of this control argument is incompatible with the -all or -ordered_field control arguments (see "Notes" below).
-ordered_field field_spec, -ofl field_spec	specifies a sort with independent ordering of the fields, i.e., mixed ascending and descending fields. The field_spec arguments must be given in threes: S_1_ L_1_ O_1_ S_2_ L_2_ O_2_ ... S_n_ L_n_ O_n_ The first and second arguments are identical to those given with the -field control argument (i.e., positive integers specifying the start position and length of the sort field). The third argument (represented by "O") is either the string "asc" to indicate an ascending field or "dsc" to indicate a descending field. Use of this control argument is incompatible with the -ascending, -descending, or -field control arguments.
-all, -a	make the primary (and only) sort field the entire sort unit; i.e., the entire sort unit is considered when sorting. This is the default mode of operation.

Notes

If the sort_seg command is invoked without any control_args, the -replace, -ascending, -all, and -delimiter control arguments are assumed, and the default delimiter of a newline character is used. That is, the sort_seg command, when invoked with path as the only argument, will sort the lines of that segment in ascending ASCII collating sequence, replacing the original segment with the sorted result.

The start position of a sort field is calculated relative to the beginning of a sort unit. If the blocking factor is $\underline{n} = 1$, the start position is calculated relative to the beginning of a string. If the blocking factor is $\underline{n} > 1$, the start position is calculated relative to the beginning of the first string of a block. When calculating field specifications within a sort unit of $\underline{n} > 1$ strings (blocking factor $\underline{n} > 1$), string delimiters internal to the sort unit should not be considered (see "Examples" below).

Sort fields/units of unequal length are compared by assuming the shorter field/unit to be padded on the right with blanks, immediately following the rightmost character. The string delimiter is never considered when padding (see "Examples" below).

If characters are detected in the input segment following the final delimited sort unit, they are ignored for the purposes of sorting, but appear in the sorted output immediately following the final delimited sort unit. An error message specifies the location of the first nondelimited character.

A maximum of 262,143 units can be sorted. The sort is stable, i.e., duplicate units appear in the same order in the sorted segment as in the original segment.

The input segment is sorted using temporary segments in the process directory. If the -segment control argument is specified, and path is the pathname of an already existing segment, its contents are destroyed upon beginning the sort. If the sorted results are to replace the original contents of the input file, that replacement does not occur until the last possible moment.

The -unique control argument deletes duplicate sort units from the sorted results. The determination of whether or not a sort unit is to be deleted is independent of sort field specifications; i.e., given a number of nonidentical sort units that contain identical sort fields, all the units do appear in the sorted results.

Examples

Suppose a segment contains the following lines (where nl represents the ASCII newline character):

```
ABCDEFGHXYnl
ABCDEFXYnl
ABCDEFGHIJXYnl
ABCXYnl
```

The display below shows how the sort_seg command sorts the contents of this segment, according to the arguments specified in the first column (nl stands for the ASCII newline character and ƀ stands for the ASCII space character).

these arguments	define these sort units	sorted on these fields	giving these results
-dm XY	ABCDEFGH	ABCDEFGHƀƀ	ABCXYnl
	ABCDEF	ABCDEFƀƀƀƀ	ABCDEFXYnl
	ABCDEFGHIJ	ABCDEFGHIJ	ABCDEFGHXYnl
	ABC	ABCƀƀƀƀƀƀƀ	ABCDEFGHIJXYnl
-bk 2	ABCDEFGHABCDEF	ABCDEFGHABCDEF	ABCDEFGHXYnl
-dm XY	ABCDEFGHIJABC	ABCDEFGHIJABCƀ	ABCDEFXYnl
			ABCDEFGHIJXYnl
			ABCXYnl
-fl 6 4	ABCDEFGHXY	FGHX	ABCXYnl
	ABCDEFXY	FXYƀ	ABCDEFGHIJXYnl
	ABCDEFGHIJXY	FGHI	ABCDEFGHXYnl
	ABCXY	ƀƀƀƀ	ABCDEFXYnl
		first second	
-fl 1 4 7 2	ABCDEFGHXY	ABCD GH	ABCDEFGHXYnl
	ABCDEFXY	ABCD XY	ABCDEFGHIJXYnl
	ABCDEFGHIJXY	ABCD GH	ABCDEFXYnl
	ABCXY	ABCX ƀƀ	ABCXYnl
-dm Y	ABCDEFGHXABCDEFX	FGHX DE	ABCDEFGHIJXYnl
-bk 2	ABCDEFGHIJXABCX	FGHI DE	ABCXYnl
-fl 6 4 4 2			ABCDEFGHXYnl
			ABCDEFXYnl
		first second	
-ofl 6 4 dsc 3 3 asc	ABCDEFGHXY	FGHX CDE	ABCDEFXYnl
	ABCDEFXY	FXYƀ CDE	ABCDEFGHXYnl
	ABCDEFGHIJXY	FGHI CDE	ABCDEFGHIJXYnl
	ABCXY	ƀƀƀƀ CXY	ABCXYnl

This page intentionally left blank.

Name: start, sr

The start command is employed after the quit signal has been issued in order to resume execution of the user's process from the point of interruption. It can also be used to resume execution after an unclaimed signal, provided that the condition that caused the unclaimed signal either is innocuous or has been corrected. It restores the attachments of the user_input, user_output, and error_output I/O switches, and the mode of user_i/o to their values at the time of the interruption, unless the -no_restore control argument is given (see below).

Usage

```
     start -control_arg-
```

where control_arg can be -no_restore (or -nr). If present, it indicates that the standard I/O attachments should not be restored.

Notes

This command can be issued at any time after a quit signal as long as a release command has not been given.

If there is no suspended computation to restart, the command prints the message "start ignored."

Name: status, st

The status command prints selected detailed status information about the storage system entry specified. For a description of the attributes listed, see "Entry Attributes" in Section II of the MPM Reference Guide.

Usage

```
     status paths -control_args-
```

where:

1.	paths	are the pathnames of the segments, directories, multisegment files, or links for which status information is desired. The default pathname is the working directory, which can also be specified by -wd. The star convention can be used.
2.	control_args	are chosen from the following list of control arguments. The control arguments can appear anywhere on the line after the command name and are in effect for the whole line.

The control_args for segments, multisegment files, and directories are:

-all, -a	lists all relevant information returned by hcs_$status_long (see the hcs_$status_ subroutine described in the MPM Subroutines); i.e., the type of entry, names, unique identifier, date used, date modified, date branch modified, date dumped, author, bit count author (if different from author), device, bit count, records used, current blocks (for segments, if different from records used), maximum length in words (if type is segment), safety switch (if it is on), user's mode, ring brackets, access class (if it is not null), and copy switch (if it is on).
-date, -dt	lists all the dates on the entry: i.e., date used, date modified, date branch modified, date dumped.
-name, -nm	lists all the names on the entry.
-mode, -md	lists the user's effective mode, ring brackets, safety switch (if it is on), and access class (if it is not null).
-device, -dv	lists the logical volume name. For directories, prints the name of the logical volume on which segments, contained in the directory, reside.

-length, -ln	for segments: lists the bit count, the number of records used, the current blocks (if different from records used) and the maximum length in words; for multisegment files: lists the number of records used by the whole file, the sum of the bit counts of all components, and the number of components; for directories: lists the number of records used and the bit count.
-author, -at	lists the author of the entry and the bit count author (if different from author).
-type, -tp	lists the type of entry (segment, directory multisegment file, or link).

The control_args for links are:

-all, -a	lists all relevant information returned by hcs_$status_long; i.e., the pathname of the entry linked to, names, unique identifier, date link modified, date dumped, and the author of the link.
-date, -dt	lists all the dates: i.e., date link modified, date dumped, the pathname of the entry linked to.
-name, -nm	lists all the names on the link.
-author, -at	lists the author of the link.
-type, -tp	lists the type of entry (link) and the pathname of the entry linked to.

Notes

If no control argument is specified, the following information is printed for segments, multisegment files, and directories: names, type, date used, date modified, date branch modified, bit count, records used, user's mode, access class.

If no control argument is specified, the following information is printed for links: the pathname of the entry linked to, names, date link modified, date dumped. The -mode, -device, and -length control arguments are ignored for links.

Zero-valued dates (i.e., dates which have never been set) are not printed.

Directories that have been used to implement multisegment files are labeled as such.

Examples

To get all the status information on the segment named >user_dir_dir>Multics>Jones>working_file, type:

```
status >user_dir_dir>Multics>Jones>working_file   -all


names:    test_segment
          working_file
type:                       segment
unique id:                  764576046673
date used:                  01/27/75 1459.0 est Mon
date modified:              01/27/75 1459.0 est Mon
branch modified:            11/19/74 1542.6 est Tue
date dumped:                01/29/75 0305.4 est Tue
author:                     Hamilton.Multics.a
bit count author:           Jones.Multics.m
volume:                     public
bit count:                  292968
records used:               8
max length:                 261120
mode:                       rw
access class:               confidential
ring brackets:              4, 4, 4
safety sw:                  on
```

The current blocks are equal to the number of records used. The copy switch is off.

To get specific status information on entrynames with the first component of newtest in the current working directory, type:

```
status -type -mode -date newtest.*


     >user_dir_dir>Multics>Smith>newtest.pl1


type:                       segment
date used:                  01/26/75 2145.0 est Sun
date modified:              01/13/75 1630.0 est Mon
branch modified:            01/13/75 1626.7 est Mon
date dumped:                01/14/75 0305.4 est Tue
mode:                       rew
ring brackets:              4, 4, 4
```

The safety switch is off. The access class is null.

```
        >user_dir_dir>Multics>Smith>newtest.list

     names:                    newtest.list
     type:                     link
     links to:                 user_dir_dir>Multics>Smith>sub_dir>newtest.list
     date link modified:       01/26/74 2139.3 est Sun
```

To get status information about the directory named >user_dir_dir>Multics>Black>test, type:

```
     status >user_dir_dir>Multics>Black>test

     names:                          test
     type:                           directory
     date used:                      12/05/74  1606.6 est Thu
     date modified:                  12/05/74  1606.6 est Thu
     branch modified:                11/29/74  1957.2 est Fri
     bit count:                      0
     records used:                   1
     mode:                           sma
     access class:                   Sensitive,Research
```

Name: stop_cobol_run, scr

The stop_cobol_run command causes the termination of the current COBOL run unit. Refer to the run_cobol command for information concerning the run unit and the COBOL runtime environment.

Usage

```
     stop_cobol_run -control_arg-
```

where the control_arg may be -retain_data or -retd to leave the data segments associated with the programs comprising the run unit intact for debugging purposes. (See "Notes" below.)

Notes

The results of the stop_cobol_run command and the execution of the STOP RUN statement from within a COBOL program are identical. Stopping the run unit consists of cleaning up all files that have been opened during the execution of the current run unit, and ensuring that the next time a program that was a component of this run unit is invoked, its data is in its initial state.

To maintain the value of all data referenced in the run unit in its last used state, the -retain_data control argument should be used.

Refer to the related commands:

display_cobol_run_unit, dcr
cancel_cobol_program, ccp
run_cobol, rc

This page intentionally left blank.

Names: terminate, tm
terminate_segno, tms
terminate_refname, tmr
terminate_single_refname, tmsr

This command allows the user to remove a segment from his address space and esets links to the terminated segment. It is most useful when recompiling procedures so that the new version can be invoked with no linkage complications. Therefore the same operation is done automatically by the Multics compilers. The user can also call this command directly in order to test various versions of a procedure. A brief description of initiating and terminating segments appears in "Constructing and Interpreting Names" in Section I.

Usage

```
     terminate paths
```

where paths are the pathnames of segments to be terminated.

Entry: terminate_segno, tms

This entry allows termination by segment number rather than by pathname.

Usage

```
     terminate_segno  seg_nos
```

where seg_nos are the segment numbers (in octal) of segments to be terminated.

Entry: terminate_refname, tmr

This entry allows termination by reference name rather than by pathname. The segment itself is terminated, not merely the particular reference name specified.

Usage

```
     terminate_refname ref_names
```

where ref_names are the reference names of segments to be terminated.

Entry: terminate_single_refname, tmsr

This entry allows termination of a single reference name. Unless the specified reference name is the only one by which the segment is known, the segment itself is not terminated.

Usage

```
     terminate_single_refname ref_names
```

where ref_names are the reference names to be terminated.

Notes

Caution must be exercised when using these commands as the user can unintentionally terminate (within the user's process only) a segment of the command language interpreter or another critical piece of the environment. The usual result is termination of the user's process.

The star convention is not recognized in any of the above commands.

Name: trace

The trace command is a debugging tool that lets the user monitor all calls to a specified set of external procedures. The trace command modifies the standard Multics procedure call mechanism so that whenever control enters or leaves one of the procedures specified by the user, a debugging procedure is invoked. The user can request the following:

1. Print the arguments at entry, exit, or both.
2. Stop (by calling the command processor) at entry, exit, or both.
3. Change the frequency with which tracing messages are printed (e.g., every 100 calls, after the 2000th call, only if the recursion depth is less than five, etc.).
4. Execute a Multics command line at entry, exit, or both.
5. Meter the time spent in the various procedures being monitored.

Use of the trace command is subject to the following restrictions:

1. Only external procedures compiled by PL/I or FORTRAN can be traced.
2. Ring 0 or gate entries cannot be traced.
3. Incorrect execution results if the traced procedure looks back a fixed number of stack frames, e.g., cu_$arg_ptr cannot be traced.
4. Only 100 procedures can be traced at one time. Up to 16 locations can be watched at one time.
5. The procedure being traced and the trace package itself must share the same combined linkage segment.
6. A procedure in a bound segment can only be traced if its entry point is externally available.

Usage

```
trace -control_args- names
```

where:

1. names	is a pathname or reference name. The reference name or entry portion of a pathname is used in the trace table. (See "Notes" below.)

2. control_args may be chosen from the following:

-template, -tp	list the trace control template.
-first n, -ft n	start monitoring on the nth call (initial value = 1).
-last n, -lt n	stop monitoring after the nth call (initial value = 9999999999).
-every n, -ev n	monitor every nth call (initial value = 1).
-before n	call the command processor before calling the traced procedure every n times (initial value = 0: do not call).
-after n	call the command processor after calling the traced procedure every n times (initial value = 0: do not call).
-argument n, -ag n	print the arguments every nth time the procedure is entered (initial value = 0: do not print).
-depth n, -dh n	monitor to the maximum recursion depth of n (initial value = 0: no limit).
-in	print the arguments only on entry (initial value = yes).
-out	print the arguments only on exit (initial value = no).
-inout	print the arguments on both entry and exit (initial value = no).
-execute XX, -ex XX	execute the Multics command line specified by the string XX whenever the procedure is monitored (initial value = "": no command).
-meter on, -mt on	meter the time spent in the procedure (initial value = off: no metering). See "Metering" below.
-meter off, -mt off	stop metering the procedure.
-govern on, -gv on	limit the recursion level for a procedure (initial value = off). See "Recursion Limiting" below.
-govern off, -gv off	do not limit the recursion level for a procedure.
-return_value on, -rv on	print the return_value on exit (initial value = off). This control argument assumes the entry is a function.

-return_value off, -rv off	do not print the return_value on exit. This control argument assumes the entry is a function.
-status entryname, -st entryname	print the trace parameters and counters for the procedure specified by entryname. (See "Notes" below.)
-status *, -st *	print the procedures being monitored and their counters. (See "Notes" below.)
-reset entryname, -rs entryname	set the number of calls and recursion depth of the specified procedure to zero.
-off entryname	stop monitoring the specified procedure. The procedure remains in the trace table and calls continue to be counted.
-on entryname	resume monitoring the specified procedure. This control argument is used after the -off control argument.
-remove entryname, -rm entryname	remove the specified procedure from the trace table. Tracing can be removed at any time.
-brief, -bf	print a short form of the monitoring information.
-long, -lg	print the long form of the monitoring information. (For use after the -brief control argument to restore the long form.)
-watch XX, -wh XX	watch locations specified by the string XX and stop by calling the command processor when they change. (See "Watch Facility" below.)
-total, -tt	print and then clear the metering statistics.
-subtotal, -stt	print and do not clear the metering statistics.
-stop_proc path, -sp path	change the procedure that is called for stop requests from the command processor to the procedure specified by path. To reset the stop procedure, issue this control argument with no path argument.
-io_switch XX, -is XX	change the switch for output to the switch specified by XX. (See "Changing Output Switch" below.)

Notes

The procedure whose pathname is given in the command line is added to the trace table with the tracing parameters from the trace control template (TCT). If the procedure is already in the table, the counters are reset and the current parameters in TCT are used.

For control arguments that affect procedures being traced, the argument is an entryname or an asterisk (*). If an entryname is used, the control argument applies to that procedure. If an asterisk is used, the control argument is applied to all entries in the trace table. All control arguments that affect the TCT must have a number argument (indicated by n above).

Examples

The command:

```
trace -ag 1 -inout test
```

prints the arguments for test on entry and exit.

The command:

```
trace -ag 2 -in -depth 6 test
```

prints the arguments for test every second time test is entered up to a recursion depth of six, i.e., 2, 4, 6.

The command:

```
trace -govern on test
```

prints the arguments of test each time test is called with a new maximum recursion depth. The trace procedure calls the command processor every time the recursion depth is a multiple of 10.

The command:

```
trace -st * -tp
```

lists the procedures in the trace table and prints the values of the trace control template.

Message Format

The message printed when control enters a procedure can appear in any one of several formats, depending on the setting of the brief switch and the status of the calling procedure. If the calling procedure is unbound or occurs in a bound segment containing a bindmap, the message takes the form:

```
Call 4.1 of alpha from beta|127, ap = 204|10746.
```

This is the fourth call of procedure alpha, which is at recursion level 1. The call comes from location 127 in component beta, and the argument list is at 204|10746. If the procedure making the call is in a bound segment that does not contain a bindmap, the message takes the form:

```
Call 4.1 of alpha from bound_gamma|437 (beta), ap = 204|10746.
```

The name in parentheses may not always be available and may be omitted in some cases. If the user has requested the brief output mode, the message is shortened to:

```
Call 4.1 of alpha.
```

When tracing is requested for a procedure, the parameters for that entry are taken from the trace control template (TCT). If the user does not alter the values in the TCT, the initial default values are used (see below). The initial values in the TCT specify that every call should be monitored.

Trace Control Template

As mentioned earlier, the trace table entry holds a number of parameters for each procedure to be traced. The values of the parameters are determined by the contents of the TCT at the time the table entry is filled in. These parameters are used in conjunction with N (the number of calls to the traced procedure in this process) and R (the current recursion depth) to control when and how the procedure should be monitored. The execution count (N) is set to 0 when tracing is first started and is incremented by 1 every time the traced procedure is called. The recursion depth (R) is set to 0 when tracing is first started and is incremented by 1 every time control enters the traced procedure and is decremented by 1 every time control leaves the traced procedure.

Let

D = the maximum recursion depth to be monitored (-depth)
F = the number of the first call to be monitored (-first)
L = the number of the last call to be monitored (-last)
E = how often monitoring should occur (-every)
B = the number of times the procedure will be called before trace stops at entry to the traced procedure (-before)

A = the number of times the procedure will be called before trace stops at exit from the traced procedure (-after)
AG = the number of times the procedure will be called before trace will print the arguments of the traced procedure (-argument)
I = a bit that is "1"b if the tracing procedure should print the arguments of the traced procedure when control goes into the traced procedure (-in)
O = a bit that is "1"b if the tracing procedure should print the arguments of the traced procedure when control goes out of the traced procedure (-out)

A call is monitored and the tracing procedure is called if, and only if,

```
F <= N <= L
R <= D
mod(N,E) = 0
```

If AG ^= 0, mod(N,abs(AG)) = 0, and I = "1"b, trace prints the values of the arguments (if any) being passed to the traced procedure. All of the arguments are listed when AG < 0. If AG < 0, the procedure is assumed to be a function and the value of the last argument is printed after the procedure returns.

If B ^= 0 and mod(N,B) = 0, the monitoring procedure prints "Stop" and calls the command processor (or a user-set procedure if the -stop_proc control argument was used). This call occurs before the procedure being traced has created its stack frame.

After control leaves the traced procedure, trace prints a line of the form:

```
Return N.R from alpha.
```

If AG ^= 0 and mod(N,abs(AG)) = 0, then all of the arguments of the traced procedure are printed if O = "1"b; otherwise, if AG < 0, the value of the last argument (assumed to be the value of the function) is printed.

Finally, trace calls the command processor. If the -stop_proc control argument was given, a procedure set by the user is called. This call occurs after the stack frame of the procedure being traced has been destroyed.

Metering

The trace command can be used to meter the execution of a specified set of procedures. If the metering feature is being used, trace does not call the debugging procedure when control enters a procedure being traced; instead, it determines the current time and the virtual CPU time used, and the number of page faults taken by the user's process before control enters and after control leaves the traced procedure. This information is used to compute the real time and CPU time used, and the number of page faults taken by the traced procedure on a local and global basis. The global CPU time is the time spent in the procedure including the time spent in any procedures that it calls. The local

CPU time does not include the time spent in any traced procedure called by the procedure, but it does include time spent in called procedures that are not being traced. The local and global versions of real time and page faults are calculated in a similar manner. Metering is only done when the first, last, every, and depth tracing conditions are satisfied.

The control argument:

-meter on, -mt on

sets the metering switch in the TCT; any procedures added to the trace table or that have their table entries updated after this argument is used are metered.

The control argument:

-meter off, -mt off

turns off the metering switch in the TCT; any procedures currently being metered continue to be metered.

The control argument:

-total

causes trace to print the metering statistics of all procedures in the trace table. The output gives the number of calls (#CALLS), global CPU time (GCPU), global real time (GREAL), global page waits (GPWS), local CPU time (LCPU), local real time (LREAL), local page waits (LPWS), and the usage percentage (%USAGE) based on local real time, of all the procedures being metered. The metering statistics are set to 0 after they are printed.

The control argument:

-subtotal, -stt

prints the same information as the -total control argument, but does not clear the statistics.

Recursion Limiting

The control argument:

-govern on, -gv on

sets a bit in the TCT that causes recursion limiting to be in effect for any procedure subsequently added to the trace table. When the governing feature is used, the depth control parameter is ignored and trace prints the call message only when the recursion depth of the traced procedure reaches a new, maximum depth. Each call message has a recursion depth one greater than the previous call message. In addition, trace calls the command processor (or a user-defined procedure if the -stop_proc control argument was used) whenever the recursion depth is a multiple of 10. Return messages are not printed. This feature enables the user to find and limit uncontrolled recursion; it can be very useful in finding the procedure(s) responsible for fatal process error.

The control argument:

-govern off, -gv off

turns off the governing switch in the TCT; any procedure currently being governed continues to be governed.

Watch Facility

The trace command has an optional watch facility in which trace watches the contents of a set of previously specified memory cells. The cells are checked at every entry to and every exit from every traced procedure. As long as the values in the locations being watched remain the same, no action is taken and no tracing messages are printed. The tracing message is printed as soon as trace finds that any of the locations being watched has had its value changed. This can be found either at entry to or exit from the traced procedure. When any value changes, the tracing message is preceded by lines that give the new values of all of the locations that have changed, and the command processor (or a user-set procedure if the -stop_proc control argument was used) is called even if the A or B conditions are not met. When execution continues, the locations that have changed are watched with the new value being used in subsequent checks. This feature can be very useful in determining which of the user's procedures has incorrectly modified a word of storage.

The control argument:

-watch XX, -wt XX

causes all procedures being traced to watch for a change in the current contents of the memory word(s) specified by the string XX. This string, specifying the location, can consist of a single address specification or a series of address specifications separated by blanks and surrounded by quotes. If an address specification does not contain a vertical bar (¦), it is taken to be an octal number giving a location in the stack; otherwise, it is taken to be a segment number and offset in octal in the standard form, e.g., segment_number¦offset.

The control argument:

-watch off, -wt off

turns off the watch facility.

The watch facility differs from other trace facilities in that there is a single table of locations being watched that is used by all procedures being traced. When the -watch control argument is processed, the new location(s) specified replace any locations currently in the watch table. There is no provision made for removing a single location from the watch table; the user must reissue a watch request that omits the location to be removed from the table.

Command Execution

The command execution facility of trace allows the user to specify a Multics command line to be executed whenever the trace debugging procedure is called. The trace procedure calls the command processor with the specified string after printing the tracing message, but before the stop request causes the command processor to be called.

The control argument:

-execute string

sets the execution string parameter in the TCT. Since string is a single argument, it must be enclosed in quotes if it contains any spaces. The execution parameter in the TCT is turned off if string has zero length (-execute ""). The following line:

```
trace -ex time test
```

will cause trace to execute the time command before and after test is called.

Changing Output Switch

All of the messages from the trace command that may be generated while actually monitoring procedures are normally written on the user_i/o switch so that trace can conveniently be used with procedures that change the attachment of the normal switch, user_output. The control argument:

```
-io_switch XX
```

causes trace to write further output on the switch specified by XX, which must already be attached and opened for stream_output.

Name: trace_stack, ts

The trace_stack command prints a detailed explanation of the current process stack history in reverse order (most recent frame first). For each stack frame, all available information about the procedure that established the frame (including, if possible, the source statement last executed), the arguments to that (the owning) procedure, and the condition handlers established in the frame are printed. For a description of stack frames, see "Standard Stack and Linkage Area Formats" in Section II of the MPM Subsystem Writers' Guide.

The trace_stack command is most useful after a fault or other error condition. If the command is invoked after such an error, the machine registers at the time of the fault are also printed, as well as an explanation of the fault. The source line in which it occurred can be given if the object segment is compiled with the -table option.

Usage

```
     trace_stack -control_args-
```

where control_args can be selected from the following:

-brief, -bf	suppress listing of arguments and handlers. This control argument cannot be specified if -long is also specified as a control argument.
-long, -lg	print octal dump of each stack frame.
-depth n, -dh n	dump only n frames.

Output Format

When trace_stack is invoked, it first searches backward through the stack for a stack frame containing saved machine conditions as the result of a signalled condition. If such a frame is found, tracing proceeds backward from that point; otherwise, a comment is printed and tracing begins with the stack frame preceding trace_stack.

If a machine-conditions frame is found, trace_stack repeats the system error message describing the fault. Unless the -brief control argument is specified, trace_stack also prints the source line and faulting instruction and a listing of the machine registers at the time the error occurred.

The command then performs a backward trace of the stack, for n frames if the -depth n argument was specified, or else until the beginning of the stack is reached.

For each stack frame, trace_stack prints the offset of the frame, the condition name if an error occurred in the frame, and the identification of the procedure that established the frame. If the procedure is a component of a bound segment, the bound segment name and the offset of the procedure within the bound segment are also printed.

The trace_stack command then attempts to locate and print the source line associated with the last instruction executed in the procedure that owns the frame (that is, either a call forward or a line that encountered an error). The source line can be printed only if the procedure has a symbol table (that is, if it was compiled with the -table option) and if the source for the procedure is available in the user's working directory. If the source line cannot be printed, trace_stack prints a comment explaining why.

Next, trace_stack prints the machine instruction last executed by the procedure that owns the current frame. If the machine instruction is a call to a PL/I operator, trace_stack also prints the name of the operator. If the instruction is a procedure call, trace_stack suppresses the octal printout of the machine instruction and prints the name of the procedure being called.

Unless the -brief control argument is specified, trace_stack lists the arguments supplied to the procedure that owns the current frame and also lists any enabled condition, default, and clean-up handlers established in the frame.

If the -long control argument is specified, trace_stack then prints an octal dump of the stack frame, with eight words per line.

Example

After a fault that reenters the user environment and reaches command level, the user invokes the trace_stack command.

For example, after quitting out of the list command, the following process history might appear:

```
list

Segments=8, Records=3

rew  0  mailbox
r w
QUIT

trace_stack


quit in ipc_$block|166
(>system_library_1>bound_command_loop_|166)
  No symbol table for ipc_
    166   000007600004    tze     7,ic        175   600104235100
```

```
Machine registers at time of fault

pr0        241¦3216              bound_sss_wired_$operator_table¦3216
pr1        57¦560                pds¦560
pr2        140¦302               hcs_.link¦302
pr3        100¦0                 tc_data¦0
pr4        236¦2100              combined_linkage_4.00¦2100
                                   (linkage for bound_command_loop_¦0)
pr5        57¦436                pds¦436
pr6        214¦1500              stack_4¦1500
pr7        214¦0                 stack_4¦0

x0    266   x1 777775   x2      0   x3 600000
x4      0   x5      4   x6   2604   x7     52
a 000000000004  q 000000000004  e 000
Indicators: Zero Carry

Control unit:  4 00235 050040  0000200000 66
               4 00235 000100  000000 0220 00
               000166 500200   000163 0005 00
Instruction:   400010116100     cmpq    pr4¦10
Next:          400010116100     cmpq    pr4¦10
Ring:          4

Backward trace of stack from 214¦1500


1500 quit ipc_$block¦166 (bound_command_loop_¦166)
  No symbol table for ipc_
   166   000007600004     tze     7,ic       175   600104235100
         ARG  1: 236¦2512 combined_linkage_4.00¦2512
                    (linkage for bound_command_loop_¦412)
         ARG  2: 214¦1434 stack_4¦1434
         ARG  3: 000000000000


1320       tw_write_$tw_read_¦167 (bound_command_loop_¦465)
  No symbol table for tw_write_
   465   000623700100     tsx0     pr0¦623
         ARG  1: 236¦2470 combined_linkage_4.00¦2470
                    (linkage for bound_command_loop_¦370)
         ARG  2: 214¦501 stack_4¦501 ( -> "list")
         Warning: arg 3 precision mismatch: 21 supplied, 17 expected.
         ARG  3: 0
         Warning: arg 4 precision mismatch: 21 supplied, 17 expected.
         ARG  4: 132
         Warning: arg 5 precision mismatch: 21 supplied, 17 expected.
         ARG  5: 0
         Warning: arg 6 type mismatch: Structure supplied, Bit expected.
         Warning: arg 6 length mismatch: 2 supplied, 72 expected.
         ARG  6: (Structure at 214¦1146) 000000000000


660        tty_attach$tty_get_line¦2350 (bound_iox_¦2504)
  No symbol table for tty_attach
  2504   000622700100     tsx0     pr0¦622
         ARG  1: 236¦4746 combined_linkage_4.00¦4746
                    (linkage for bound_iox_¦312)
         ARG  2: 214¦501 stack_4¦501 ( -> "list")
         ARG  3: 000000000204
         ARG  4: 000000000000
         ARG  5: 000000000000
```

```
360       listen_$listen_¦433 (bound_command_loop_¦2717)
  No symbol table for listen_
  2717   000617700100     tsx0    pr0¦617 call_var
          ARG  1: "exec_com >user_dir_dir>MPM>Thomas>start_up login
          interactive" on "cleanup"  call listen_¦231
          (bound_command_loop_¦2515)


160       process_overseer_$process_overseer_¦136
            (bound_command_loop_¦22340)
  No symbol table for process_overseer_
 22340   000622700100     tsx0    pr0¦622
          Argument list header invalid.
          on "any_other"
          call standard_default_handler_$standard_default_handler_¦3
          (bound_command_loop_¦7137)


60        user_init_admin_$user_init_admin_¦36 (bound_command_loop_¦23036)
  No symbol table for user_init_admin_
 23036   700036670120     tsp4    pr7¦36,* alm_call
          No arguments.

End of trace.
```

Name: truncate, tc

This command truncates a segment to a specified length and resets the bit count accordingly. It sets the bit count author to be the user who invoked the command. The segment can be specified by pathname or segment number.

Usage

```
truncate -control_arg- seg_no length
```

1.	control_arg	if present, must be -name or -nm, indicating that the following seg_no is in fact a pathname, although it might look like a number.
2.	seg_no	is either a pathname or an octal segment number. A pathname that happens to be an octal number should be preceded by the control argument -name or -nm.
3.	length	is an octal integer indicating the length of the segment in words after truncation. If no length argument is provided, zero is assumed.

Notes

The user must have write access on the segment to be truncated.

If the segment is already shorter than the specified length, its length is unchanged, but the bit count is set to the specified length.

This command should not be used on segments that are (or are components of) structured files.

Example

```
truncate alpha 50
```

truncates segment alpha to 50 words; i.e., all words from word 50 (octal) on are zero. The bit count of the segment is set to the truncated length.

This page intentionally left blank.

Name: unassign_resource, ur

The unassign_resource command unassigns one or more resources that have been assigned to the user's process by the Resource Control Package (RCP).

Usage

```
     unassign_resource resources -control_args-
```

where:

1. resources — specifies the resources to be unassigned from the user's process. Currently, the only resource managed by RCP are devices. If a device is attached, it is automatically detached. A user may unassign all devices assigned to the process by specifying the keyword "all". A user may unassign one device by specifying its name.

2. control_args — may be chosen from the following:

   -comment XX, -com XX — is a comment string that is displayed to the operator when the resource is unassigned. This comment is displayed only once, even if several resources are being unassigned. (See the assign_resource command for details about comment strings.)

   -admin, -am — forces an unassignment. This control argument should be specified by highly privileged users who want to unassign a resource that is assigned to some other process.

Example

In the example that follows the user unassigns a tape previously assigned by the assign_resource command.

```
     unassign_resource tape_03
```

Name: unlink, ul

The unlink command deletes the specified link entry. For a discussion of links see "Segment, Directory, and Link Attributes" in Section III of the MPM Reference Guide.

Usage

```
     unlink paths
```

where paths specify storage system link entries to be deleted.

Notes

The user must have modify access in the directory containing the link.

The star convention can be used.

The delete, delete_force, and delete_dir commands can be used to delete segment and directory entries.

Name: vfile_adjust, vfa

The vfile_adjust command is used to adjust structured files left in an inconsistent state by an interrupted opening, or unstructured files in any state. For unstructured files a control argument must specify the desired adjustment. Otherwise, no control arguments are allowed. A sequential or blocked file is adjusted by truncation after the last complete record. An indexed file is adjusted by finishing the interrupted operation.

Usage

```
     vfile_adjust path -control_arg-
```

where:

1. path	is the pathname of the file to be adjusted.
2. control_arg	must be specified only for unstructured files and is selected from the following:
-set_nl	if the last nonzero byte in the file is not a newline character, a newline character is appended. The bit count of the file's last nonempty segment is then set to the file's last nonzero byte (which is now sure to be a newline character).
-use_nl	the file is truncated after the last newline character.
-set_bc	the bit count of the file's last nonempty segment is set to the last nonzero byte in that segment. Any components beyond it are deleted.
-use_bc -n-	the file is truncated to the byte specified by the bit count of multisegment file component n. If n is not given, it is taken to be the last nonempty component.

Notes

See the description of the vfile_ I/O module (described in the MPM Subroutines) for further details. The adjust_bit_count command used with the character control argument is equivalent to vfile_adjust used with the -set_bc control argument, except that the latter only operates on a file that appears to be unstructured.

Name: vfile_status, vfs

The vfile_status command prints the apparent type (unstructured, sequential, blocked, or indexed) and length of files. For structured files, information about the state of the file (if busy) and the file version (unless current) is printed. The maximum record length is printed for blocked files. For indexed files, the following statistics are printed:

1. The number of records in the file, including zero length records
2. The number of nonnull records in the file, if different from the above
3. The total length of the records (bytes)
4. The number of blocks in the free space list for records
5. The height of the index tree (equal to zero for empty files)
6. The number of nodes (each 1K words, page aligned) in the index tree
7. The total length of all keys (bytes)
8. The number of keys (if different from record count)
9. The number of duplicate keys (if nonzero)
10. The total length of duplicate keys (if any)

Usage

```
vfile_status path
```

where path is the pathname of the segment or multisegment file of interest. If the entryname portion of the pathname denotes a directory, it is ignored. If no files are found for the given pathname, a message to that effect is printed. If the entry is a link, the information returned pertains to the entry to which the link points. The star convention is permitted.

Notes

Additional information may be obtained through the status command.

Examples

Assume that the file foo is in the user's working directory. The command:

```
     vfile_status  foo
```

might produce the following output:

```
     type:   unstructured
     bytes:   4993
```

if the file is unstructured,

or

```
     type:   sequential
     records:   603
```

if the file is sequential,

or

```
     type:   blocked
     records:   1200
     max recl:   7 bytes
```

if the file is blocked,

or

```
     type:   indexed
     records:   397
     state:   locked by this process
     action:   write in progress
     record bytes:   3970
     free blocks:   1
     index height:   2
     nodes:   3
     key bytes:   3176
```

if the file is indexed and a write operation has been interrupted in the user's process.

This page intentionally left blank.

Name: walk_subtree, ws

The walk_subtree command is used to execute a given command line in a given directory (called the starting node) and in directories inferior to the starting node. The command prints the pathname of every directory in which the command line is executed. Control arguments are provided to modify the behavior of the command (see "Usage" below). See "The Storage System Directory Hierarchy" in Section III of the MPM Reference Guide for a description of the Multics directory structure.

Usage

```
walk_subtree path command_line -control_args-
```

where:

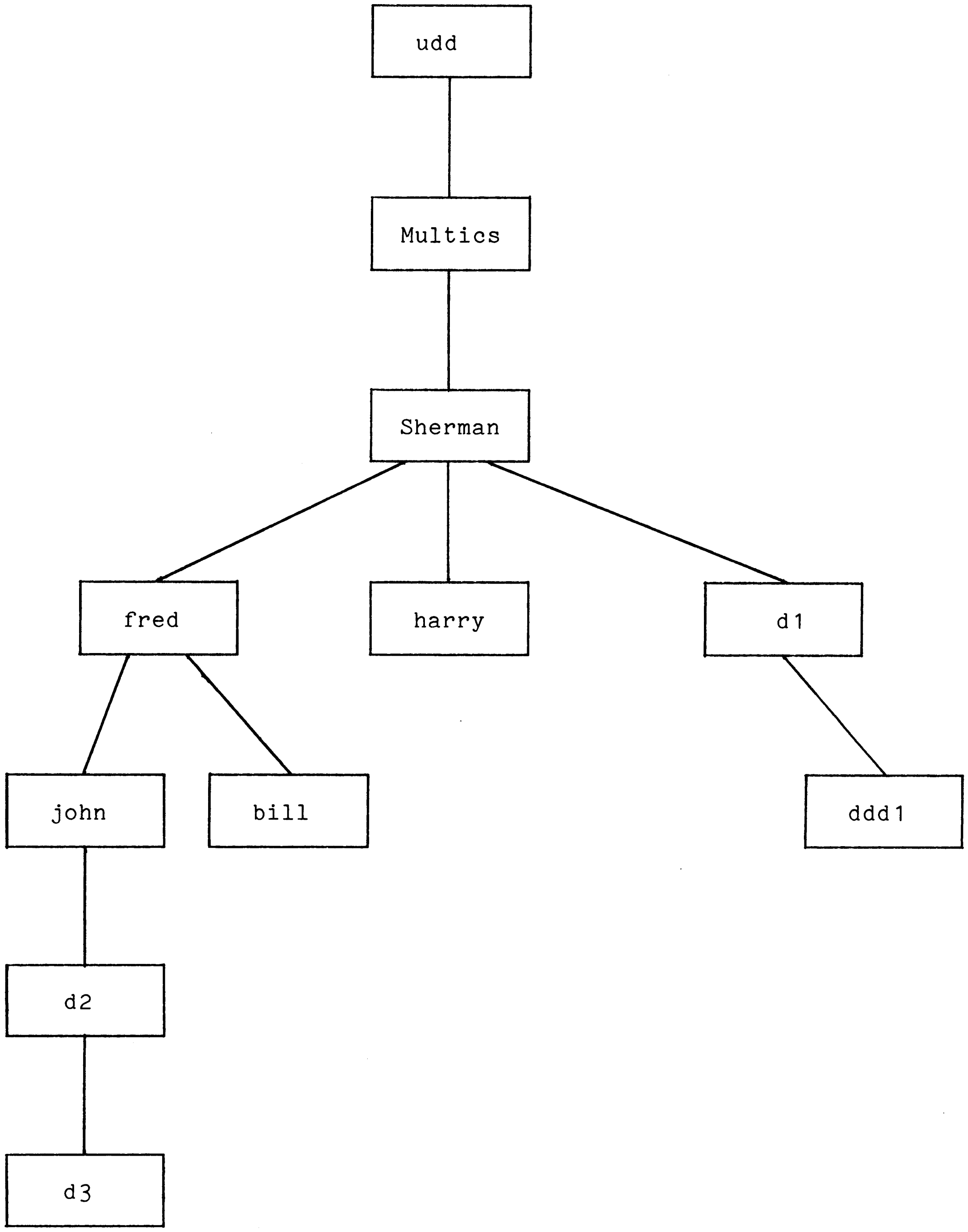
1. path	is the starting node. This must be the first argument. A path of -wd specifies the working directory.
2. command_line	is the command line to be executed. The entire command line is taken to be a single argument. Therefore, a multiple-word command line should be typed as a quoted string.
3. control_args	are chosen from the following list of control arguments. These control arguments can appear in any order following the command line.
-first n, -ft n	makes n the first level in the storage system hierarchy at which the command line is to be executed where, by definition, the starting node is level 1. The default is -first 1.
-last n, -lt n	makes n the last level in the storage system hierarchy at which the command line is to be executed. The default is -last 99999, i.e., all levels.
-brief, -bf	suppresses printing of the names of the directories in which the command line is executed.
-bottom_up, -bu	causes execution of the command line to commence at the last level and to proceed upward through the storage system hierarchy until the first level is reached. In the default mode, execution begins at the highest (first) level and proceeds downward to the lowest (last) level.

Notes

The walk_subtree command has a program_interrupt handler. If the user quits out of the command and immediately types pi (or program_interrupt), his working directory is changed back to what it was before the walk_subtree command was invoked.

Examples

Assume the following directory structure:



Assume that the user's current working directory is Sherman. Then, the command:

```
walk_subtree -wd "list *.pl1"
```

lists all the segments having a two-component name with a second component of pl1 in the directory Sherman and all of its subdirectories.

```
walk_subtree >udd>Multics>Sherman "list *.pl1; dl *.pl1"
```

lists and then deletes all the segments with a second component name of pl1 in the directory Sherman and all of its subdirectories.

```
walk_subtree -wd "list *.pl1" -first 3
```

executes the command line "list *.pl1" in the directories john, bill, ddd1, d2, and d3.

```
walk_subtree >udd>Multics>Sherman "list *.pl1" -last 2
```

executes the command line "list *.pl1" in the directories Sherman, fred, harry, d1.

```
walk_subtree >udd>Multics>Sherman "list *.pl1" -last 4 -first 3
```

executes the command line "list *.pl1" in the directories john, bill, ddd1, and d2.

```
walk_subtree fred "set_acl -wd rew Jones.Faculty.*"
```

executes the given command line first in the directory fred, then in john and in bill, then in d2, then in d3.

```
walk_subtree fred "delete_acl -wd Jones" -bottom_up
```

executes the given command line first in the directory d3, then in d2, then in john and in bill, then in fred. The -bottom_up control argument is essential in this case for the delete_acl command to succeed. If access were deleted first from a superior directory, the user might not have sufficient access to delete ACL entries on the inferior directories.

Name: where, wh

The where command uses the standard search rules to search for a given reference name of a segment. (For a discussion of search rules, see "Search Rules" in Section IV of the MPM Reference Guide.) The command prints out the full pathname of that segment, using its primary name. If the segment is not in the search path, an error message is printed.

Usage

```
     where ref_names -control_arg-
```

where:

1.	ref_names	are segment reference names. Each name can be a maximum of 32 characters.
2.	control_arg	can be -all or -a to list the pathnames of all segments with ref_name*i* that can be found using the current search rules, the user's effective access to each segment, and the name of the search rule used to find each segment.

Note

The primary name of a storage system entry is the name that is first in the list of names on that entry.

Example

If a user has his own copy of a standard command, such as cwd, in his working directory, and that copy has been initiated, the command:

```
     where cwd -all
```

prints three lines:

```
     >udd>Project_id>Person_id>wd>cwd (re) search rule "initiated_segments"
     >udd>Project_id>Person_id>wd>cwd (re) search rule "wd"
     >sss>cwd (re) search rule "system_library_standard"
```

Name: who

The who command lists the number, identification, and status of all users of the system. The command prints out a header and lists the name and project of each user. The header consists of the system name, the total number of users, the current system load, the maximum load, the current number of absentee users, and the maximum number of absentee users. (See the description of the how_many_users command to print only the header.)

Usage

```
who -control_args- -optional_args-
```

where:

1.	control_arg*i*	can be chosen from the following list of control arguments:
	-long, -lg	prints the date and time logged in, the terminal identification and the load units of each user, in addition to his name and project. The header includes installation identification and the time the system was brought up. If available, the time of the next scheduled shutdown, the time when service will resume after the shutdown, and the time of the previous shutdown are printed.
	-project, -pj	sorts the output by the Project_id of each user.
	-name, -nm	sorts the output by the name (Person_id) of each user.
	-absentee, -as	lists only absentee users.
	-brief, -bf	suppresses the printing of the header.
2.	optional_args	can be selected from the following list:
	Person_id	lists only users with the name Person_id.
	.Project_id	lists only users with the project name Project_id.
	Person_id.Project_id	lists only users with the name Person_id and the project name Project_id.

Notes

Absentee users are denoted in the list by an asterisk (*) following Person_id.Project_id.

If the who command is specified with no arguments, the system responds with a two-line header followed by a list of interactive users sorted according to login time. (See "Examples" below.)

If the -project or -name control arguments are omitted, the output is sorted on login time. Both arguments may not be used at once, as the sort is performed on one key at a time.

If an optional_arg is specified, the header is suppressed even if the -long control argument is specified.

It is possible for a user to prevent his name from being listed; to do this, the user should first contact his project administrator.

Examples

To print the default information, type:

```
who

Multics 2.0, load 4.0/100.00; 4 users
Absentee users  1/2

IO.SysDaemon
Jones.Faculty
Doe.Work
Smith.Student*
```

To print long information for absentee users on the Student project (with no header), type:

```
who -absentee -long .Student

Absentee users = 1/2

10/21/74  0050.2 none 1.0  Smith.Student*
```

To print brief information for all users, type:

```
who -brief

IO.SysDaemon
Jones.Faculty
Doe.Work
Smith.Student*
```

SECTION IV

ACCESS TO THE SYSTEM

This section describes the requests interpreted by the answering service. These requests can only be issued from a terminal connected to the answering service; that is, one that has just dialed up or one that has been returned to the answering service after a session terminated with a "logout -hold" command. (For more information on gaining access to the system, see "Protocol for Logging In" in Section I of the MPM Reference Guide.)

For clarity, this section identifies two categories of answering service requests: preaccess and access. The preaccess requests are necessary because certain terminals do not have an answerback. By convention, Multics uses a terminal answerback to identify the particular type of device being used. The device type is used by the system to interpret all input/output. Therefore, for input to be understood by Multics and output understood by the user, these requests must be given before the access requests. The access requests connect the terminal to a process. This process may exist already (e.g., dial) or be created in response to the request (e.g., login).

The same conventions described in "Command Descriptions" of Section III also apply to the requests in this section.

Name: dial, d

The dial request is used to connect an additional terminal to an existing process. It is a request to the answering service to perform the connection and to notify the user's process of the new terminal connection.

When the dial request is invoked, the answering service searches for a logged-in process that is accepting dial connections using the dial_id specified by the user. If no such process is found, the message "Dial line not active." is printed, and the user may try again, with a different dial_id. If a process is found, a one-line message verifying the connection is printed. All further messages printed on the terminal are from the user process itself.

This request is administratively restricted.

Usage

```
dial dial_id Person_id.Project_id
```

where:

1.	dial_id	is the identifying keyword that uniquely specifies a logged-in process that is accepting dial connections. This keyword is supplied by that process when it informs the answering service that it is accepting dialed terminals.
2.	Person_id.Project_id	is the Person_id and Project_id of the process that the user wishes to connect to.

Notes

None of the arguments are optional; all must be supplied in the correct order.

If the user process terminates or logs out, a message is printed on the terminal, and control of the terminal is returned to the answering service.

Users who wish to accept dialed terminals must be registered with the dialok attribute, as must their project. The dialok attribute is normally assigned by the project administrator.

Names: enter, e
enterp, ep

These requests are used by anonymous users to gain access to Multics. Either one is actually a request to the answering service to create a process for the anonymous user. *

Anonymous users who are not to supply a password use the enter (e) request. Anonymous users who are to supply a password use the enterp (ep) request.

Usage

```
     enter -anonymous_name- Project_id -control_args-
```

where:

1. anonymous_name	is an optional identifier that is not checked by the system, but is passed to the user's process overseer as if it were a person identifier. If anonymous_name is not specified, it is assumed to be the same as the project identifier.
2. Project_id	is the identification of the user's project.
3. control_args	can be chosen from the following list of control arguments:
-brief, -bf	suppress messages associated with a successful login. If the user is using the standard process overseer, the message of the day is not printed.
-home_dir path, -hd path	set the user's home directory to the pathname specified, if the user's project administrator allows him to specify his home directory.
-process_overseer path, -po path	set the user's process overseer to the path specified, where path is the pathname of the procedure. This can only be done if the user's project administrator allows him to specify his process overseer.
-no_print_off, -npf	overtype a string of characters to provide a black area for the user to type his password.
-print_off, -pf	do not overtype an area for the password.
-no_preempt, -np	refuse to log the user in, if he can only be logged in by preempting some other user in his load control group.

-no_start_up, -ns	instruct the standard process overseer not to execute the start_up.ec segment if the user has one and the project administrator allows the user to avoid it.
* -force	log the user in if at all possible if he has the guaranteed login attributes.

Note

See "Protocol for Logging In" in Section I of the MPM Reference Guide for an explanation of the responses to the enter and enterp requests.

Name: login, l

The login request is used to gain access to the system. It is a request to the answering service to start the user identification and process creation procedures.

The login request asks for a password from the user (and attempts to ensure either that the password does not appear at all on the user's terminal or that it is thoroughly hidden in a string of cover-up characters). The password is a string of one to eight letters and/or digits associated with the Person_id.

After the user responds with his password, the answering service looks up the Person_id, the Project_id, and the password in its tables and verifies that the Person_id is valid, that the Project_id is valid, that the user is a legal user of the project, and that the password given matches the registered password. If these tests succeed, and if the user is not already logged in, the load control mechanism is consulted to determine if allowing the user to log in would overload the system.

If the user is permitted to log in, a process is created for the user, and the terminal is placed under control of that process.

Many control arguments are available for tailoring various attributes of the new process to the user's needs.

Usage

```
login Person_id -Project_id- -control_args-
```

where:

1. Person_id	is the user's registered personal identifier. This argument must be supplied. The user's personal identifier may be replaced by his registered "login alias" if he has one. Aliases, like personal identifiers, are registered by the system administration and are unique at the installation. The login alias is translated into the user's personal identifier during the login process, and there is no difference between a user process created by supplying a personal identifier and one created by supplying an alias.
2. Project_id	is the identification of the user's project. If this argument is not supplied, the default project associated with the Person_id is used. See the -change_default_project control argument below for changing the default project to the Project_id specified by this argument.

3. control_args	can be selected from the following:
-brief, -bf	suppress messages associated with a successful login. If the standard process overseer is being used, then the message of the day is not printed.
-home_dir path, -hd path	set the user's home directory to the path specified, if the user's project administrator allows him to specify his home directory.
-process_overseer path, -po path	set the user's process overseer to the procedure given by the path specified, if the user's project administrator allows him to specify his process overseer.
-no_print_off, -npf	cause the system to overtype a string of characters to provide a black area for the user to type his password.
-print_off, -pf	suppress overtyping for the password.
-no_preempt, -np	refuse to log the user in if he can only be logged in by preempting some other user in his load control group.
-no_start_up, -ns	instruct the standard process overseer not to execute the user's start_up.ec segment, if he has one, and if the project administrator allows him to avoid it.
-force	log the user in if at all possible, provided the user has the guaranteed login attribute. Only system users who perform emergency repair functions have the necessary attribute.
-change_password, -cpw	change the user's password to a newly given password. The login request asks for the old password before it requests the new one. It requests the new one twice, to verify the spelling. If it is not typed the same both times, the login and the password change are refused. If the old password is correct, the new password replaces the old for subsequent logins, and the message "password changed" is printed at the user's terminal. The user should <u>not</u> type the new password as part of the control argument.
-subsystem path, -ss path	create the user's process using the prelinked subsystem in the directory specified by path.

-generate_password, -gpw	change the user's password to a new password, generated for the user by the system. The login request asks for the old password first. Then, a new password is generated and typed on the user's terminal. The user is asked to retype the new password, to verify that he has seen it. If the user types the new password correctly, it replaces the old password for subsequent logins, and the message "password changed" is printed at the user's terminal. If the user mistypes the new password, the login and password change are refused.
-change_default_project, -cdp	change the user's default project to be the Project_id specified in this login request line (see the description of the Project_id argument above). The default Project_id is changed for subsequent logins, and the message "default project changed" is printed at the user's terminal. If the -cdp control argument is given without a Project_id argument, an error message is printed.
-authorization XX, -auth XX	set the authorization of the process to that specified by XX, where XX is a character string composed of level and category names for the desired authorization, separated by commas. The XX character string may not contain any embedded blank or tab characters. (The short names for each level and category are guaranteed to not contain any blanks or tabs, and can be used whenever the corresponding long names do contain blanks or tabs.) The XX character string must represent an authorization that is less than or equal to the maximum authorization of Person_id on the project Project_id. If this control argument is omitted, the user's registered default login authorization is used. (See "Access Control" in the MPM Reference Guide for more information about process authorizations.)
-change_default_auth, -cda	change the user's registered default login authorization to the authorization specified by the -authorization control argument. If the authorization given by the user is valid, the default authorization is changed for subsequent logins, and the message "default authorization changed" is printed at the terminal. If the -cda control argument is given without the -auth argument, an error message is printed.
-ring n	set the user's initial ring to be ring n, if this ring number is greater than or equal to the user's registered initial ring and less than his registered maximum ring.

-no_warning, -nw	suppress even urgent system warning and emergency messages from the operator, both at login and during the user's session. Use of this argument is recommended only for users who are using a remote computer to simulate a terminal, or are typing out long memoranda, when the process output should not be interrupted by even the most serious messages.
-outer_module p, -om p	attach the user's terminal via the outer module named p rather than the user's registered outer module, if the user has the privilege of specifying his outer module.
-terminal_type XX, -ttp XX	set the user's terminal type to XX, where XX is a string acceptable to the -terminal_type control argument of the set_tty command. (See the description of the set_tty command for a complete explanation of possible devices.) This control argument overrides the default terminal type.
-modes XX	set the I/O modes associated with the user's terminal to XX, where the string XX consists of modes acceptable to the tty_ I/O module. (See the tty_ I/O module description in the MPM Subroutines for a complete explanation of possible modes.) The XX string is usually a list of modes separated by commas; the XX string must not contain blanks. (See "Examples" below.)

Notes

Several parameters of the user's process, as noted above, can be controlled by the user's project administrator. The project administrator can allow the user to override some of these attributes by specifying control arguments in his login line.

If the project administrator does not allow the user to specify the -home_dir, -process_overseer, or -ring control arguments or if he does allow one or more of these control arguments and they are incorrectly specified by the user, these control arguments are ignored and the default values are used.

Examples

In the examples below, the lines typed by the user are preceded by an exclamation mark (!) and the user's password is shown even though in most cases the system either prints a string of cover-up characters to "hide" the password or temporarily turns off the printing mechanism of the user's terminal.

Probably the most common form of the login request is to specify just the Person_id and the Project_id (and then the password) as:

```
!   login Jones Demo
    Password:
!   mypass
```

To set (or change) the default project to Demo, type:

```
!   login Jones Demo -cdp
    Password:
!   mypass
    Default project changed.
```

To set the tabs and crecho I/O modes so the terminal uses tabs rather than spaces where appropriate on output and echoes a carriage return when a line feed is typed (assuming the user has a default project), type:

```
!   login Jones -modes tabs,crecho
    Password:
!   mypass
```

To change the password from mypass to newpass (assuming the user has a default project), type:

```
!   login Jones -cpw
    Password:
!   mypass
    New Password:
!   newpass
    New Password Again:
!   newpass
    Password changed.
```

Name: MAP

The MAP request tells the system that the user is attempting to gain access from a terminal whose keyboard generates only uppercase characters. This request must be invoked before the access requests (e.g., login) can be successfully issued.

Once the request has been issued, the system changes the translation tables used by the terminal control software so that all uppercase alphabetic characters are translated to lowercase. The user still needs to use the special escape conventions to represent the ASCII graphics that are not on the uppercase-only terminal keyboard. Uppercase alphabetic characters also require the escape conventions. (See "Escape Characters" in Section V of the MPM Reference Guide.) After the map request is given, the user may log in normally.

This request must be used for 150- and 300-baud terminals if their keyboards can transmit only uppercase characters; for any other terminal type, it is ignored.

Usage

```
MAP
```

Note

For further reference, see "Protocol for Logging In" in Section I of the MPM Reference Guide.

Example

The following example shows a user invoking the MAP request. The lines typed by the user are preceded by an exclamation mark (!).

```
!   MAP
!   LOGIN \JONES \DEMO
    PASSWORD:
!   MYPASS
```

Names: 963
029

The 963 and 029 preaccess requests tell the system whether the user is attempting to gain access from a device similar to an EBCDIC or Correspondence code IBM Model 2741. These requests must be invoked before the access requests (e.g., login) can be successfully issued.

If the user attempts to log in from a device similar to an EBCDIC or Correspondence code IBM Model 2741, the system returns a "Type 'help' for instructions" message accompanied by a partially readable line. For example,

```
cidu #63 cqn U:XVOXK Type 029 for Correspondence code.

   or

Type 963 for EBCDIC. Ula; z17 qis Fiss;nairp;rf; fip;-
```

The user should respond to this message by typing the specified request.

Once the request has been issued, the system changes the translation tables used by the terminal control software so that all input/output is readable. The user may then log in normally.

These requests are valid for 134-baud devices similar to an IBM Model 2741 only; for any other terminal type, they are ignored.

The names (963, 029) of the requests are actually the standard part numbers of the usual typeballs for EBCDIC and Correspondence code IBM Model 2741s, respectively.

Usage

```
963

   or

029
```

Note

For further reference, see "Protocol for Logging In" in Section I of the MPM Reference Guide.

Name: hello

The hello preaccess request repeats the greeting message that is printed whenever a terminal is first connected to the system. The request is particularly useful after a 963 or 029 request since the greeting message is then printed in the proper code.

Name: slave

The slave preaccess request changes the service type of the channel from login to slave for the duration of the connection. This enables a privileged process to request the answering service to assign the channel to it, and then attach it. Refer to the description of the dial_manager_ subroutine in the MPM Subsystem Writers' Guide for an explanation of the mechanism for requesting channels from the answering service.

INDEX

HONEYWELL INFORMATION SYSTEMS

Publications Remarks Form*

TITLE:	SERIES 60 (LEVEL 68) MULTICS PROGRAMMERS' MANUAL COMMANDS AND ACTIVE FUNCTIONS	ORDER No.:	AG92, REV. 1
		DATED:	JANUARY 1975

ERRORS IN PUBLICATION:

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION:

(Please Print)

FROM: NAME ______________________ DATE: ____________

COMPANY ______________________

TITLE ______________________

____

____

*Your comments will be promptly investigated by appropriate technical personnel, action will be taken as required, and you will receive a written reply. If you do not require a written reply, please check here. ☐

CUT ALONG LINE

CUT ALONG LINE

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WELLESLEY HILLS,
MASS. 02181

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

POSTAGE WILL BE PAID BY:

HONEYWELL INFORMATION SYSTEMS
60 WALNUT STREET
WELLESLEY HILLS, MASS. 02181

ATTN: PUBLICATIONS, MS 050

FOLD ALONG LINE

Honeywell

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	MULTICS PROGRAMMERS' MANUAL COMMANDS AND ACTIVE FUNCTIONS ADDENDUM A	**ORDER NO.**	AG92A, REV. 1
		DATED	SEPTEMBER 1975

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

FIRST CLASS
PERMIT NO. 39531
WELLESLEY HILLS,
MA 02181

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

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
60 WALNUT STREET
WELLESLEY HILLS, MA 02181

ATTENTION: PUBLICATIONS, MS 050

FOLD ALONG LINE

Honeywell

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

TITLE	SERIES 60 (LEVEL 66) MULTICS PROGRAMMERS' MANUAL COMMANDS AND ACTIVE FUNCTIONS ADDENDUM B	ORDER NO.	AG92B, REV. 0
		DATED	FEBRUARY 1976

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

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

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

TITLE	SERIES 60 (LEVEL 68) MULTICS PROGRAMMERS' MANUAL COMMANDS AND ACTIVE FUNCTIONS ADDENDUM C	ORDER NO.	AG92C, REV. 1
		DATED	JULY 1976

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

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