MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

The Multiplexed Information and Computing Service: Programmers' Manual

PART II
REFERENCE GUIDE TO MULTICS

Revision: 15

Date: 11/30/73

FOREWORD

PLAN OF THE MULTICS PROGRAMMERS' MANUAL

November 30, 1972

The Multics Programmers' Manual (MPM) is the primary reference manual for user and subsystem programming on the Multics system. It is divided into three major parts:

Part I: Introduction to Multics

Part II: Reference Guide to Multics

Part III: Subsystem Writers' Guide to Multics

Part I is an introduction to the properties, concepts, and usage of the Multics system. Its four chapters are designed for reading continuity rather than for reference or completeness. Chapter 1 provides a broad overview. Chapter 2 goes into the concepts underlying Multics. Chapter 3 is a tutorial guide to the mechanics of using the system, with illustrative examples of terminal sessions. Chapter 4 provides a series of examples of programming in the Multics environment.

Part II is a self-contained comprehensive reference guide to the use of the Multics system for most users. In contrast to Part I, the Reference Guide is intended to document every detail and to permit rapid location of desired information, rather than to facilitate cover-to-cover reading.

Part II is organized into ten sections, of which the first eight systematically document the overall mechanics, conventions, and usage of the system. The last two sections of the Reference Guide are alphabetically organized lists of standard Multics commands and subroutines, respectively, giving details of the calling sequence and the usage of each.

Several cross-reference facilities help locate information in the Reference Guide:

- The table of contents, at the front of the manual, provides the name of each section and subsection and an alphabetically ordered list of command and subroutine names.

- A comprehensive index (of Part II only) lists items by subject.

- Reference Guide sections 1.1 and 2.1 provide lists of commands and subroutines, respectively, by functional category.

Part III is a reference guide for subsystem writers. It is of interest to compiler writers and writers of sophisticated subsystems. It documents user-accessible modules which allow a user to bypass standard Multics facilities. The interfaces thus documented are a level deeper into the system than those required by the casual user.

Examples of specialized subsystems for which construction would require reference to Part III are:

1) a subsystem which precisely imitates the command environment of some system other than Multics (e.g., an imitation of the Dartmouth Time-Sharing System);

2) a subsystem which is 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);

3) a subsystem which is protecting 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 which permits access only to program-defined aggregated information such as averages and correlations).

Each of the three parts of the MPM has its own table of contents and is updated separately, by adding and replacing individual sections. Each section is separately dated, both on the section itself, and in the appropriate table of contents. The title page and table of contents are replaced as part of each update, so one can quickly determine if his manual is properly up-to-date. The Multics on-line "message of the day" or local installation bulletins should provide notice of availability of new updates. In addition, the Multics command "help mpm" provides on-line information about known errors and the latest MPM update level.

In addition to this manual, users who will write programs for Multics will need a manual giving specific details of the language they will use; such manuals are currently available for PL/I, FORTRAN, and BASIC. A separate, specialized supplement to the MPM is also provided for users of graphic displays. The bibliography at the end of Part I, Chapter 1, describes these and other references in more detail.

Multics provides the ability for a local installation to develop an installation-maintained or author-maintained library of commands and subroutines which are tailored to local needs. The installation may also document these facilities in the same format as used in the MPM; the user can then interfile these locally provided write-ups in the command and subroutine sections of his MPM.

Finally, access to Multics requires authorization. The prospective user must negotiate with the administration of his local installation for permission to use the system. The installation may find it useful to provide the new user with a documentation kit describing available documents, telephone numbers, operational schedules, consulting services, and other local conventions.

CONTENTS

November 30, 1973

Section 9 Commands (03/20/72)

continued on next page

Section 9 Commands (continued)

continued on next page

Section 9 Commands (continued)

continued on next page

Section 9 Commands (continued)

continued on next page

Section 9 Commands (continued)

Section 10 Subroutines (03/24/72)

continued on next page

Section 10 Subroutines (continued)

continued on next page

Section 10 Subroutines (continued)

Reference Guide Index (11/30/73)

 (END)

THE MULTICS COMMAND REPERTOIRE

The following facilities are the ones considered to be part of Multics and are described in this manual. Detailed specifications on each of the commands mentioned below can be found, filed in alphabetical order by command name, in the MPM Reference Guide section, Commands. In addition, many commands have on-line descriptions that can be obtained with the help command.

The active functions available on Multics are described separately, by functional grouping, in several MPM Reference Guide write-ups: Alphabetical List of Active Functions, Logical Active Functions, Arithmatic Active Functions, Character String Active Functions, Segment Name Active Functions, Data and Time Active Functions, Question Asking Active Functions, and User Paremeter Active Functions. Active functions can be used in a command line in the manner described in the MPM Reference Guide section, The Command Language, under the heading Active Strings.

The user should also consult the list of items in the Author Maintained and/or Installation Maintained Library at his installation, since local language translators and other commands can substantially extend the standard command repertoire. Documentation of the Author Maintained and/or Installation Maintained Library is supplied by the local installation.

The command repertoire is organized by function into the following groups:

Storage System, Creation and Editing of Segments
Storage System, Segment Manipulation
Storage Segment, Directory Manipulation
Storage System, Access Control
Storage System, Formatted Output Facilities
Storage System, Address Space Control
Language Translators, Compilers, Assemblers and Interpreters
Object Segment Manipulation
Debugging and Performance Monitoring
I/O System Control
Command Typing and Control
Communication Among Users
Accounting
Control of Absentee Computations
Miscellaneous Tools

1) Storage System, Creation and Editing of Segments

adjust_bit_count	sets bit count of a segment to last nonzero character
basic_system	editor and runner of BASIC programs
compare_ascii	compares supposedly identical ASCII segments, reporting differences
create	creates an empty segment
edm	inexpensive, easy to learn editor
indent	indents a PL/I source segment to make it more readable
qedx	more sophisticated editor
sort_file	sorts ASCII segments line by line

2) Storage System, Segment Manipulation

addname	adds a name to a segment, directory, or link
adjust_bit_count	sets bit count of a segment to last nonzero character
archive	packs segments together to save physical storage breakage
archive_sort	sorts the contents of an archive segment by component segment name
compare	compares segments word by word, reporting differences
compare_ascii	compares supposedly identical ASCII segments, reporting differences
copy	copies a segment
create	creates an empty segment
delete	deletes a segment, but questions if segment is read only
deleteforce	deletes a segment without question
deletename	removes a name from a segment, directory, or link
file_output	directs typewriter output to a segment
fs_chname	an extra powerful rename command
link	creates a directory link to another segment
list, listnames, listotals	prints directory contents
move	moves segment to another directory
names	moves or copies names from one storage system entry to another
rename	renames a segment, link, or directory

reorder_archive	rearrages order of an archive segment
safety_sw_off	turns safety switch off for a segment or directory
safety_sw_on	turns safety switch on for a segment or directory
set_bit_count	sets a given bit count in a segment
status	prints everything known about an entry in a directory
truncate	used to truncate a segment to a specified length
unlink	removes a directory link

3) Storage System, Directory Manipulation

change_default_wdir	sets the default working directory
change_wdir	changes to a new working directory
createdir	creates a directory
delete_dir	destroys a directory
list, listnames, listotals	prints directory contents
print_default_wdir	prints default working directory name
print_wdir	prints current working directory name

4) Storage System, Access Control

delete_iacl_dir	removes an Initial ACL for new directories
delete_iacl_seg	removes an Initial ACL for new segments
deleteacl	removes an access control list (ACL) entry
deletecacl	removes a common access control list (CACL) entry
list_iacl_dir	prints an Initial ACL for new directories
list_iacl_seg	prints an Initial ACL for new segments
listacl	prints an ACL entry
listcacl	prints a CACL 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

setacl	adds (or changes) an ACL entry
setcacl	adds (or changes) a CACL entry

5) Storage System, Formatted Output Facilities

dprint	adds segment to the high speed line printer queue
dpunch	adds a segment to the card punch queue
dump_segment	selective octal dump of segment contents
mail	prints or sends mail
memo	allows users to set reminders for later printout
print	prints any ASCII segment
print_motd	prints the system's message of the day
runoff	formats a text segment according to internal control words
runoff_abs	invokes the runoff command as an absentee job

6) Storage System, Address Space Control

change_wdir	controls directory assumed when relative path names are given
initiate	adds a segment to the address space of a process
list_ref_names	prints all names by which segment is known to a process
print_search_rules	prints path of search for missing segments
print_wdir	prints name of current working directory
set_search_dirs set_search_rules	controls path of search for missing segments
terminate	removes a segment from process address space
where	prints full path name of a segment

7) Language Translators, Compilers, Assemblers, and Interpreters

alm	assembly language for Multics
alm_abs	invokes the ALM assembler as an absentee job
apl	invokes the APL interpreter
basic	translates and optionally runs Version 6 BASIC programs

basic_run	runs BASIC programs
basic_system	editor and runner of BASIC programs
fortran	standard FORTRAN IV compiler
fortran_abs	invokes the FORTRAN compiler as an absentee job
lisp	a LISP 1.5 interpreter and compiler
lisp_compiler	invokes a LISP compiler
pl1	the Multics PL/I compiler
pl1_abs	invokes the PL/I compiler as an absentee job
qedx	sophisticated editor, with macro facilities (a minor interpreter)
v2pl1_abs	invokes the Version II PL/I compiler as an absentee job
v5basic	translates and optionally runs Version 5 BASIC programs

8) Object Segment Manipulation

bind	packs two or more object segments into a single segment
display_component_name	prints name of bound component, gives offset
print_bind_map	prints information about a bound segment
print_link_info	prints list of entries and outbound links of an object segment

9) Debugging and Performance Monitoring

change_error_mode	adjusts length and contents of status messages
debug	symbolic source language debugger
dump_segment	selective octal dump of segment contents
hold	saves the stack for debugging or later restart
how_many_users	tells how many users are logged in
page_trace	prints list of pages recently demanded
print_linkage_usage	prints map of all current linkage
profile	prints information about execution of individual statement within a program
progress	prints information about the progress of a command as it is being executed

ready	prints a summary line of CPU time and paging usage
ready_off	suppresses the ready line following commands
ready_on	restores the ready line
reprint_error	allows retyping of earlier status messages
trace_stack	prints stack history
where	prints full path name of a segment

10) I/O System Control

cancel_daemon_request	cancels a previosuly submitted daemon request
close_file	closes open FORTRAN and PL/I files
console_output	restores typewriter output to the typewriter
dprint	adds a segment to the high speed printer queue
dpunch	adds a segment to the card punch queue
file_output	directs typewriter output to a segment
iocall	allows direct calls to I/O system entries
iomode	sets typewriter character conversion modes
line_length	sets typewriter carriage length
list_daemon_requests	prints list of daemon requests currently queued
print_attach_table	prints list of current I/O system stream attachments

11) Command Typing and Control

abbrev	allows user-specified abbreviations for command lines or parts of command lines
answer	answers questions normally asked of the user
do	executes a command line with arguments inserted
enter	enters an anonymous user into the system
exec_com	allows a segment to be treated as a list of commands to be executed
get_com_line, set_com_line	adjusts size of command line buffers

hold	saves the stack for debugging following an accident
login	enters user into the system
logout	exits user from the system
new_proc	creates a new process with a fresh address space
program_interrupt	signals a condition following an accident
release	releases a saved stack
start	restarts a computation following a quit or unexpected signal
walk_subtree	repeats a command in all directories below a given directory

12) Communication Among Users

change_wdir	changes base of operation to another directory
check_info_segs	checks information (and other) segments for changes
help	prints special information segments
link	inserts a directory link to another segment
mail	sends an ASCII message to another user
make_peruse_text	prepares segments for use by peruse_text
peruse_text	selectively prints structural information segments
set_search_dirs	sets path of search for missing segments
unlink	removes a directory link
where	prints full path name of a segment
who	prints list of users currently logged in

13) Accounting

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

14) Control of Absentee Computations

alm_abs	invokes the ALM assembler as an absentee job
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 as an absentee job
list_abs_request	prints list of absentee job requests currently queued
pl1_abs	invokes the Version I PL/I compiler as an absentee job
runoff_abs	invokes the runoff command as an absentee job
who	prints list of absentee jobs currently logged in

15) Miscellaneous Tools

archive	packs several segments together to save physical storage breakage
archive_sort	sorts the contents of an archive segment by component segment name
calc	a desk calculator
code	enciphers a segment, given a coding key
decam	another desk calculator
decode	deciphers a segment, given the proper coding key
print_motd	prints the system's message of the day
reorder_archive	rearranges order of an archive segment
who	prints list of users currently logged in

PROTOCOL FOR LOGGING IN

The first step in using Multics is to establish a connection between the user's terminal and the system, as described in the MPM Introduction, Chapter 3, under The Mechanics of Terminal Usage. After the user has established that connection, Multics responds:

```
Multics 20.6; MIT, Cambridge, Mass.
Load = 16.0 out of 40.0 units; users = 15
```

where the system, the site, the number of units and users currently logged in, and the maximum number of units currently permitted are, of course, variable.

Logging In

Once the user has obtained the "Load =" message (and if he still wishes to log in), he should type:

```
login person -project- -control_arguments-
```

If the project is omitted, the user's default project ID is used. (See the write-up of the login command for more information.) Multics responds:

```
Password:
```

The terminal prepares for the user's typing of his password by x'ing out the line following the password request. The user then types his password (eight characters or less); e.g.,

```
qwertyui
```

At this point, Multics prints one of the following messages:

1) person project logged in: date time from type terminal "x"

   There is a short interval during which the process is created for the user, followed by the printing of the message of the day, followed by a ready message (see Ready Messages below). If the user's password has been given incorrectly since its last correct use, a message to that effect is also printed.

2) System full. Please try again later.

   Acceding to the user's login request would cause the number of logged in users to exceed the current maximum. (Note that certain users have privileges such that they can log in despite the current maximum setting.)

3) Load control group full. Please try again later.

   The user's load control group has its maximum number of users already logged in.

4) Incorrect login word.

   The user has not specified a recognizable login request.

5) login: illegal argument "xxx".

   The user has specified an illegal parameter in his login request line.

6) Home directory missing.

   The user's home directory does not exist and cannot be created.

7) You are subject to preemption.

   This message is printed in conjunction with message 1 (above). It indicates that the user has been assigned secondary status and can be preempted if the system becomes full and a primary user from any load control group logs in.

8) You are protected from preemption until hhmm.

   This message is printed in conjunction with message 1 (above). It indicates that the user has been assigned primary status and cannot be preempted until the time given. Once this time arrives, the user retains primary status until another user from his load control group attempts to log in and finds the load control group full. The first user can then be demoted to secondary status or preempted, depending on whether or not the system is full.

9) Special session in progress.

A new version of Multics is being tested, or certain repair or administrative work is in progress which requires that users be temporarily restricted from accessing the system.

10) Login incorrect.

Some item in the system's verification of the user's attempt to access Multics was incorrect. I.e., the user's name or project was not registered (or was mistyped) or the password was incorrectly typed.

11) person.project already logged in from type terminal "x"

The user-project combination is already using Multics. A message notifying the already logged in user of the current login attempt is printed at his terminal.

If a user does not complete his login within a few minutes, his terminal hangs up.

User's Process Parameters

Several attributes of the user's process can be controlled by the user's project administrator. The project administrator can, in turn, allow the user to override some of these attributes by specifying control arguments in his login or enter line. (See the login and enter command write-ups.) The variable attributes and their values for most users are:

1) home directory

The project administrator can specify the path name of the user's home directory. The project administrator can also permit the user to override this specification by use of the -home_dir control argument to the login or enter commands. The usual value for a user's home directory path name is:

>user_dir_dir>project>person

and if the path name has this form, and the directory does not exist, the login or enter commands attempt to create the directory.

2) process overseer

The user's process overseer is the procedure which is called to initialize the user environment when a process is created. The project administrator can specify the path name of this procedure. The project administrator can also allow the user to override this specification by use of the -process_overseer control argument to the login or enter commands. The usual value for a user's process overseer is:

process_overseer_

If the user's process overseer cannot be located, the user process cannot be initialized.

3) initial and maximum ring number

The project administrator can specify the initial protection ring in which the user process begins execution, and the maximum ring in which the user process can execute, within limits set by the system administration. This specification cannot be overridden by the user. The usual values for these parameters are:

initial ring: 4
maximum ring: 7

4) preemption protection time

The project administrator can specify the minimum amount of time that a user can hold primary status, within limits set by the system administration. The user cannot override this specification. The usual value for most users for this parameter is 48 hours.

5) password

The user can change his password by use of the -change_password control argument to the login command. The command asks first for the current password to verify the identity of the user, then for the new password. On subsequent logins, the new password is required.

6) default project

The user can change his default project (for subsequent logins) to the project specified in the current login command line.

7) attributes

A number of one-bit flags which determine user privileges are referred to as the user attributes. The system administrator can allow project administrators to set certain of them for users on a project, and some of them can be overridden by the user. The attribute flags include:

a) brief

If on, the user does not receive messages associated with a successful login. If the user is using the standard process overseer, the message of the day is not printed. The project administrator can force this attribute to be on, or the user can set it to on by use of the -brief control argument to the login or enter commands.

b) nostartup

If on, the user can specify the -no_start_up control argument to the login or enter commands, and cause his start_up.ec to be bypassed. (See Start Up below.) Most users have this attribute set to on.

c) v_process_overseer

If on, the user can override the process overseer specification by use of the -process_overseer control argument to the login or enter commands. If off, the control argument has no effect. Most users have this attribute set to on.

d) v_home_dir

If on, the user can override the home directory specification by use of the -home_dir control argument to the login or enter commands. If off, the control argument has no effect. Most users have this attribute set to on.

e) preempting

If on, the user can preempt other primary users in his load control group whose grace has expired. The user can cause this attribute to be set to off by use of the -no_preempt control argument to the login or enter commands.

f) guaranteed_login

If on, the user is logged in, if at all possible, if he also specifies the -force control argument to the login or enter commands. This attribute flag is set to on only for system users for emergency repair applications.

g) multip

If on, the user can be logged in more than once simultaneously. Only system daemon users have this attribute.

h) nobump

If on, the user cannot be bumped by name by the system operator. Only system daemon users have this attribute.

i) nolist

If on, the user is not listed on the output of the who command. This attribute is not normally given to any user.

j) no_primary

If on, the user cannot have primary status.

k) no_secondary

If on, the user cannot have secondary status.

Start Up

Upon beginning execution in a new process, a user might desire his process to perform certain actions before coming to command level. In order to do this, the user can create an exec_com segment (see the MPM write-up of the exec_com command) that contains commands which are to be executed before his process attempts to read from his terminal. This segment must be named start_up.ec and must reside in the user's initial working directory.

If the segment start_up.ec exists in the user's initial working directory, the printing of the message of the day is suppressed. In addition, the command "exec_com start_up creationtype processtype" is executed as a command during process initialization. If the process is being created because the user has logged in, creationtype is the string "login". If the process is being created because of a new_proc command or a process termination, creationtype is the string "new_proc". If the process being created is interactive, then processtype is the string "interactive". If the process being created is an absentee process, processtype is the string "absentee".

This feature allows users to initialize their processes as they see fit. Some uses of this feature might be to suppress ready messages or change the search rules.

Ready Messages

The ready message is a printed response designed to provide timing information for the user. It is of the form

r time cpu mu pf

and is printed after every command or sequence of commands, and after a quit condition. time is the current time of day (24-hour notation), cpu is the amount of processor time charged to the process (in seconds, to the nearest millisecond), mu is the memory usage and pf is the number of page faults. The cpu, mu and pf figures are the values since the last calculation or, in the case of the first ready message for a new process, the time, memory usage, and page faults required to initialize the process.

Special Subsystem Use

Some projects may wish to arrange for a special process overseer procedure to be called when a user logs in, instead of the standard Multics procedure, in order to modify the system's appearance to the terminal user. A project administrator can specify a special process overseer for any user on his project. The MPM Subsystem Writers' Guide write-up for the process_overseer_ subroutine contains further information.

Anonymous Users

For some applications (like short courses), it is desirable to allow unregistered persons to log in as users of a particular project. Multics allows a project to register a special, anonymous user. Users who log in under this identity are given the access name anonymous.project.a, and any number of these users can log in at once. When used in conjunction with the special process overseer control argument described above, this facility allows a subsystem to simulate other time-sharing systems under Multics.

There are two varieties of anonymous user registrations and two corresponding login procedures. For anonymous users who have a password, the user types

```
enterp person project -control_arguments-
```

instead of the login line described above. No default project ID is possible. The system then asks

```
Password:
```

and continues with the normal login sequence. For anonymous users with no password (usually those with a special process overseer which checks for a password), the user types

```
enter person project -control_arguments-
```

The system skips requesting a password and goes directly to the logged in message. For both cases, the user's name as given on the login line is available to a special process overseer for any checks it wishes to make. (See the MPM command write-up for enter.)

Logging Out

When the user has finished his current terminal activity, he issues the command

```
logout
```

Multics disconnects the user's telephone line after printing

```
person project logged out date time
CPU usage ss sec, memory usage uu units
hangup
```

Emergency Logout

When a scheduled shutdown or system emergency occurs, the user may see the following message appear on his terminal:

```
Automatic logout
person project logged out date time
CPU usage ss sec, memory usage uu units
hangup
```

followed by the telephone line disconnection as for a normal logout command. If possible, a warning message is printed a few minutes before the automatic logout message appears.

Unscheduled Disconnections

If, for some reason, the user's telephone connection to Multics is broken, the Multics system recognizes this event and issues an automatic logout for the user (without printing on the user's terminal, needless to say). If he has more work to accomplish, he should dial into Multics again and attempt the standard login sequence.

Logout for Inactivity

If a user is inactive (that is, his process remains blocked) for more than the installation-specified maximum time limit, Multics logs the user out with the following message:

```
Maximum inactive time exceeded
person project logged out date time
CPU usage ss sec, memory usage uu units
hangup
```

Terminal Switch Settings

Switches must be set as indicated below for interaction with the computer.

2741:
LCL-COM switch on COM.
INHIBIT AUTO EOT switch (if present) on.
Power switch on.
The quit button is marked ATTN.
The end of line key is marked RETURN.

Datel:
LOCAL-REMOTE switch on REMOTE.
EOT-INH switch on INH.
Power switch on.
The quit button is marked ATTN.
The end of line key is marked RETURN.

1050:
System switch on Attend.
Printer 1 switch on SendRec.
Keyboard switch on.
System switch up.
Test switch off.
Power switch on.
Line Control Switch (located inside rear pedestal door) on.
The quit button is marked RESET LINE.
The end of line key is marked RETURN.

37:
No switches are accessible. Power is turned on by the DATA button.
The quit button is marked INTERRUPT.
The end of line key is marked LINE SPACE.

38:
Line button on
Both the CR and LF buttons must be pressed sequentially to generate the equivalent of a Multics new line (carriage return_line freed). Carriage return should be typed first, then line feed.
The quit button may be unmarked -- it is the second button from the bottom in the right hand colomn of five buttons.

35:
ON-LINE/OFF-LINE switch on ON-LINE.
HALF DUPLEX/FULL DUPLEX switch on FULL DUPLEX.
The quit button is marked BREAK.
The end of line key is marked LINE FEED.

33:
ON-LINE/OFF-LINE switch on ON-LINE.
HALF DUPLEX/FULL DUPLEX switch on FULL DUPLEX.
The quit button is marked BREAK.
The end of line key is marked LINE FEED.

ARDS:
Power switch (on back) on.
Press RESET button before dialing.
The quit button is marked:
a) when in read mode, the CONT and Q buttons pressed at the same time;
b) when in write mode, the WAIT button.

The end of line key is marked LF.

Terminet 300:
The ON LINE and READY buttons should be lit.
The INTERRUPT button should light up when the quit button is hit.
Transparency switch off.
Inhibit switch on Norm.
Rate switch set to 15.
Line Feed switch on 1.
Auto L.F. switch on.
The quit button is marked INTERRUPT.
The end of line key is marked RETURN.

Execuport 300:
The PWR switch (on back panel) should be on.
Mode switch on LINE.
DUPLEX switch on HALF. (Can be set to FULL to suppress printing of the password only.)
CHAR/SEC switch on 15. (Should be set to 10 if dialed into a 110 baud line.)
PARITY switch on EVEN.
QSL switch on LOWER.
The quit button is marked BRK.
Both the LF and CR buttons must be pressed sequentially to generate the equivalent of a Multics "new line" (carriage return-line feed). Either button can be pressed first.

Trendata 1000:
Power switch on (POWER indicator light turns on).
COMM button lighted (press switch to enter or leave communication mode).
LOCAL button unlighted (press switch to enter or leave local mode).
The quit button is marked ATTN.
The PROCEED light indicates that the keyboard is unlocked.
The ERROR CHECK light indicates that a character cannot be printed.
The UNLOCK button can be depressed to free the keyboard and turn on the PROCEED light.
The end of line key is marked RETURN.

Tektronix 4013: Power switch (under the keyboard) on (POWER indicator light turn on).
LINE/LOCAL switch to the LINE position.
TRANSMIT AND RECEIVE baud rates (on back of pedestal) set to 1200.
FULL DUPLEX/HALF DUPLEX switch (on back of pedestal) set to HALF DUPLEX, SUPERVISOR.
CARRIAGE RETURN switch (on back of pedestal) set to CR/LF.
The quit button is marked BREAK.
The end of line key is marked LF.

Teleterm 1030: (Multics version) Power switch on.
Duplex switch on half.
Parity switches set to on and even.
SPEED set to 10, 15, or 30 depending on whether a 110 baud, 150 baud, or 300 baud telephone number was dialed.

Note that when using Multics via a data network, the above switch settings may be superseded by local host conventions.

Note

See also the MPM Reference Guide section, Typing Conventions, for information on keyboard input and output conventions. In general, the conventions described there apply to logging in and out as well as all other typing.

 (END)

TYPING CONVENTIONS

Three categories of typing conventions are dealt with in this section: canonical form, erase and kill characters, and escape characters. Quits, which might reasonably be considered a typing convention, are discussed elsewhere. The button that can be pressed to cause a quit condition to be signalled is listed for each terminal type under Terminal Switch Settings in the MPM Reference Guide section, Protocol for Logging In. Handling of the condition is discussed under the quit condition in the MPM Reference Guide section, List of System Conditions and Default Handlers.

Canonical Form

The concept of a canonical representation of a printed line image described here has been used in at least two character oriented systems: in TYPSET on the IBM 7094 (as suggested by Earl Van Horn), and in the TITAN operating system on the ATLAS computer.

Characters are intended ultimately for human communication, and conventions about a printed line must be made with this in mind. A character stream is a representation of printed lines. In general, there are many possible character streams that represent the same line. In particular, on input, a typist can produce the same printed line twice with different sets of key strokes. For example, the line

```
start    lda  alpha,4     get first result.
```

could have been typed with either spaces or horizontal tabs separating the fields; one cannot tell by looking at the printed image. Since it is not possible for the individual to distinguish between several ways of typing a printed representation, no program should deliberately attempt to do so either.

For example, a program should be able to compare easily two character streams to see if they are the same, in the sense that they produce the same printed image. It follows that all character input to Multics must be converted into a standard (canonical) form. Similarly, all programs producing character output, including editors, must produce the canonical form of output stream.

Effectively, of all possible ASCII character strings, only certain of those strings are ever found within Multics. All of those strings that produce the equivalent printed effect on a

typewriter terminal are represented within Multics as one string, the canonical form for the printed image.

No restriction has been placed on the individual at his terminal; he is free to type a noncanonical character stream. This stream is automatically converted to the canonical form before it reaches his program. For the user who wants his program to receive raw or partially processed input from his terminal, the following escape mechanism is provided. The tw_ outer module (see the MPM write-up for the tw_ subroutine) supports the following applicable modes below through the changemode call:

¬can — no canonicalization of overstrikes.

¬esc — no canonicalization of escape characters.

¬erkl — no erase and kill processing.

rawi — the specified data is read from the terminal without any conversion or processing. This includes shift characters and undifferentiated upper and lower case characters.

Similarly, an I/O System Interface Module (IOSIM) is free to rework a canonical stream on output into a different form if, for example, the different form happens to print more rapidly or reliably on the device.

We assume that every IOSIM is able to determine unambiguously what precise physical motion of the device corresponds to the actual character stream coming from or going to it. In particular, the IOSIM must know the location of physical tab settings. This requirement places a constraint on devices with movable tab stops. When the tab stops are moved, the IOSIM must be informed of the new settings. Standard Multics software assumes that tab stops are at character positions 11, 21, 31, 41, etc.

The current Multics canonical form does not meet all of the objectives of the above discussion: it represents a compromise between whose objectives and that of convenience of typing aligned tabular information, which requires an ambiguous interpretation of the tab character. The following three statements describe the current Multics canonical form.

1) A message consists of strings of character positions separated by carriage motion.

2) Carriage motion consists of new line, tab, or space characters.

3) Character positions consist of a single graphic or an overstruck graphic. A character position with overstrikes contains the numerically (in ASCII sequence) smallest graphic, a backspace character, the next larger graphic, etc.

Thus, for the most part, the canonical stream differs little from the raw input stream it was derived from.

Examples of Canonical Form

Several illustrations of canonical form are shown below. The examples do not attempt to cover every conceivable variation or combination of characters, but, rather, illustrate the intent and the method. (In the examples, assume that the typist's terminal has horizontal tab stops set at 11, 21, 31, etc.)

```
Typist:          This is ordinary text.<NL>
Printed line:    This is ordinary text.
Canonical form: This is ordinary text. <NL>
```

For the case of simple, straight line input, the canonical form reduces to the original key strokes of the typist. Most input probably falls into this category.

```
Typist:          Here full<BS><BS><BS><BS>____ means that<NL>
Printed line:    Here full means that
Canonical form: Here _<BS>f_<BS>u_<BS>l_<BS>l means that<NL>
```

This is probably the most common example of canonical conversion, to insure that overstruck graphics are stored in a standard pattern.

```
Typist:          We see no prob <BS>lem<CR>__<NL>
Printed line:    We see no problem
Canonical form: _<bs>W_<bs>e see no problem
```

(Recall that the carriage return (CR) does not produce a line feed.) The most important property of the canonical form is that meanderings of the typist within a line are irrelevant. The typist need merely concern himself with the printed image.

Erase and Kill Characters

Experience has shown that even with sophisticated editor programs available, two minimal editing conventions at the earliest possible level are very useful for human input to a computer system. These two conventions give the typist two editing capabilities at the instant he is typing:

1) Ability to delete the last character or characters typed.

2) Ability to delete all of the current line typed up to this point.

(More complex editing capabilities are also available, but they fall in the domain of editing programs that can work with lines previously typed as well as the current input stream.) By framing these two editing conventions in the language of the canonical form, it is possible to preserve the ability to interpret unambiguously a typed line image despite the fact that editing was required.

The first editing convention is that one graphic, the number sign (#), is reserved as the erase character. When this character appears as the only graphic in a print position, it erases itself and the contents of the previous print position. When it is not the only graphic in a print position, it erases the print position in which it appears. If the erase character follows simple carriage motion, the carriage motion is erased. Several successive erase characters erase an equal number of print positions or simple carriage motions. Since erase processing occurs after the transformation to canonical form, there is no ambiguity as to which print position has been erased; the printed line image is always the guide. Whenever a print position is erased, any carriage motion on the two sides of the erased print position is combined into a single carriage motion.

The second editing convention reserves another graphic, the commercial at sign (@), as the kill character. When this character appears as the only graphic in a print position, the contents of that line up to and including the kill character are discarded. Again, since the kill processing occurs after the conversion to canonical form, there can be no ambiguity about which characters have been discarded. By convention, an overstruck erase character is processed before a kill character, and a kill character is processed before a non-overstruck erase character. Therefore, a kill character can only be erased by an overstruck erase character.

Examples of Erase and Kill Processing

```
Typist:          abcx#de<SP><BS>fzz##g<NL>
Printed line:    abcx#defzz##g
Canonical form:  abcx#defzz##g<NL>
Final input:     abcdefg<NL>
```

This represents the primary use of the erase character, correcting typing errors the moment they are noted. Note that the erroneous space between e and f was not erased, it was undone.

```
Typist:          This@The off<BS><BS><BS>___##n<BS>_<SP>state<NL>
Printed line:    This@The off##n state
Canonical form:  This@The _<BS>o_<BS>f_<BS>f##_<BS>n state<NL>
Final input:     The _<BS>o_<BS>n state<NL>
Printed appearance of final form:  The on state
```

Escape Characters

Contemporary terminal equipment often is not capable of representing all 128 of the ASCII code values. To keep full generality and flexibility in the future, standard software escape conventions are used for all terminal devices. On devices that have the revised ASCII set, the use of the escape mechanism is normally unnecessary. Each class of terminal device has a particular character assigned as the software escape character. When this character occurs in an input (or output) string to (or from) a terminal it always gives a special interpretation to the next one or more characters. The standard escape character is the left slant; this means that to input the code for it, an escape convention has to be used. Therefore, the left slant should be avoided in all Multics software. (It should be noted that the two standard erase characters, # and @, should also be avoided in all software.)

For simplicity, universal escape conventions have been established that are uniform over several terminal classes. For full flexibility, there is a mechanism for representing any arbitrary octal code in a character string. The universal escape conventions are:

\d1d2d3 for the octal code d1 d2 d3 where d1, d2, d3 are from zero to seven. \d2d3 is equivalent to \d1d2d3 if d1 is zero. \d3 is equivalent to \d1d2d3 if d1 and d2 are zero.

\<NL> for a local (i.e., concealed) use of the new line character that does not go into the computer-stored string on input and that is not in the computer-stored string on output.

\# for placing an erase character into the input string.

\@ for placing a kill character into the input string.

\\ for placing a left slant character into the input string.

One additional stylistic convention holds for all terminals: the solid vertical bar (|) and the broken vertical bar (¦) are equivalent representations of the graphic for ASCII code 174.

IBM 1050 and 2741 Terminals

Each type ball used requires a different set of escape conventions.

EBCDIC Type Ball (963) and EBCDIC Keypunches

The following non-ASCII EBCDIC graphics are considered to be stylized versions of ASCII characters:

¢	(cent sign)	for	\ (left slant, software escape)
'	(apostrophe)	for	´ (acute accent)
¬	(negation)	for	^ (circumflex)

In addition to the four universal escape conventions, the following are available for convenience.

¢'	for	`	(grave accent)
¢<	for	[	(left bracket)
¢>	for	]	(right bracket)
¢(	for	{	(left brace)
¢)	for	}	(right brace)
¢t	for	~	(tilde)

There are no currently implemented escapes for IBM 029 keypunches with EBCDIC codes.

In the case of keypunches, an end-of-card automatically generates a new line character. It is also convenient for input to have:

¢* for "skip reading the remainder of this card without the new line character"

¢/ for "new line and skip reading the remainder of this

card"
¢+ for "new line and keep reading this card"
¢H for horizontal tab

Note that one can use the multiple punch codes described in the MPM Reference Guide section, Punched Card Codes, when at an 029 keypunch instead of the above escape conventions. The two sets of conventions (escape and multiple punch) are interchangeable, even in the same card deck.

Model 33 and 35 Teletypes

Because these models do not have both upper and lower case, the following typing conventions are necessary to enable users to input the full ASCII character set:

1) The keys for letters A through Z input lower case letters a through z, unless preceded by the escape character \ (left slant). The left slant is shift-L on the keyboard, although it does not show on all keyboards. For example to input "Smith.ABC", type "\Smith.\A\B\C".

2) Numbers and punctuation map into themselves when possible. The underscore (_) is represented by the back arrow (←). The circumflex (^) is represented by the up arrow (↑). The acute accent (´) is represented by the apostrophe (').

3) The following other correspondences exist:

Character	type in	octal
backspace	\-	010
grave accent (`)	\'	140
left brace ({)	\(	173
vertical line (¦)	\!	174
right brace (})	\)	175
tilde (~)	\=	176

In normal I/O mode, characters are output according to these same rules. In edited I/O mode, the output drops the left slant and becomes much tidier, but ambiguous.

Model 37 and 38 Teletypes

There are no additional escape conventions for the model 37 and 38 teletypes since they use the full ASCII character set.

Execuport 300

The following non-ASCII characters on the keyboard are considered to be stylized versions of ASCII characters:

← (back arrow)	for	_	(underscore)
↑ (up arrow)	for	^	(circumflex)

THE COMMAND LANGUAGE

A system command is a program furnished by the system that performs some general, canned function for a user. Such programs are called commands because they carry out users' orders to the system (renaming a segment, compiling a program, etc.). In Multics, system commands are not a special class of program. Any user program that takes character strings as arguments can be invoked as a command. (See the MPM Reference Guide section, The Multics Programming Environment.) The command processor can be viewed as simply a mechanism for invoking programs by name.

It is clearly necessary to establish some conventions for the syntax of a command line. (A command line is defined as the sum of a command name plus any arguments to be passed on to the command when it is invoked. Note that, in Multics, the general rule is for system commands to accept arguments at invocation and not to acquire them by interrogation of the user during execution.) It should also be clear that it is not usually desirable to declare that the syntax be identical to the subroutine call of some particular programming language, since the engineering of human interfaces is different from that of subroutines. The conventions developed here are chosen for simplicity in the basic case, and for functional flexibility in the nonbasic cases. That is, various services such as nesting and iteration of commands are furnished, and the command language syntax allows these features to be specified by means of certain special delimiters in the command line; but if the services are not desired, the user need only type his commands according to the format discussed below under Simple Commands. For that matter, the command system is avoidable; some subsystems under Multics may choose to interact with their users in their own fashion, with their own conventions. Most users, however, deal with Multics directly through the command system.

Context

After successfully logging in to the system, the user is said to be at command level. Then and thereafter, from the point of view of the user, command level is the time after a ready message, at which point the system is available for new commands. That is, when a command has finished executing, it returns to the command processor (which called it in the first place), and the command processor prints a message on the user's terminal informing him that the system is ready for further orders. When the user is at command level, he issues commands in the syntax of the command language described herein. Note that because Multics typewriter input allows read-ahead, the user does not have to wait for a ready message before typing another command line. He

can, however, be interrupted in the middle of typing a line by the ready message. If this occurs, that line is lost and must be retyped. The typing of ready messages can be turned off and on using the commands ready_off and ready_on (see the MPM command write-ups).

Simple Commands

A command line specifies a function to perform plus, if appropriate, the arguments with which the function is to operate. Each command line element is currently treated as a character string. That is, individual commands are called with character string arguments, there currently being no conversion by the system of, say, numeric characters to binary representation.

There are two basic elements of a command line. The first is the command name. This is essentially a reference name, i.e., a (procedure) segment name, and, if appropriate, an entry point name. The command processor uses the user's search rules (see the MPM Reference Guide section, The System Libraries and Search Rules) to find the program whose name is the command name. However, the reference name can be preceded by a storage system hierarchy location specification; that is, it can be a path name. In this case, the search rules are superseded and the path name is used to find the program. Subsequent unqualified references to the same reference name are treated as if the path name were present (see example below). The second basic element is the argument. This is simply a character string designating, for instance, a segment. Depending on the particular command in question, there can be from zero to an arbitrary number of arguments. The element delimiter (or separator) in a command line is the space (or blank). The terminator of a command line can be either the semicolon (;) or the new line character. Note that more than one command can be issued on the same line of console input by using the semicolon between commands.

The general form of the simple command is

command_name arg1 ... argn

with each element separated from the preceding one by one or more spaces. For example, the rename command takes arguments in pairs; the first is the current path name of the segment to be renamed and the second is the desired new entry name. Thus,

rename square_root sqrt

causes the command processor to search for and invoke a procedure named rename at entry point rename with the character strings "square_root" and "sqrt" as arguments. There is no difference between invoking a command from the console and calling it in a procedure. For example, typing the command line "rename square_root sqrt" is equivalent to executing the following PL/I program:

```
x: proc;
call rename ("square_root", "sqrt");
end x;
```

As another example, suppose that one knows that an experimental version of the rename command resides in the directory >Smith_dir. If one types:

```
>Smith_dir>rename square_root sqrt
```

then the experimental version is invoked. That is, the program rename in the directory >Smith_dir is invoked with the character string arguments "square_root" and "sqrt". Subsequent unqualified references to rename invoke the one in >Smith_dir. In like manner, any program in the storage system hierarchy can be invoked.

Iteration

Sometimes the user wishes to repeat a command with one or more elements changed. The iteration facility of the command language is provided for economy of typing. The iteration set consists of one or more elements enclosed by parentheses (parentheses are reserved characters). Each element of the set, in turn, replaces the entire set in the command line. For example,

```
print (a b c).pl1
```

is equivalent to the three commands

```
print a.pl1; print b.pl1; print c.pl1
```

More than one iteration set can appear in a command. The corresponding element from each set is taken. For instance, the compound command

```
rename >Smith_dir>(Jones Doe Brown) (Day White Green)
```

would expand into the commands

```
rename >Smith_dir>Jones Day

rename >Smith_dir>Doe White

rename >Smith_dir>Brown Green
```

Nested iteration sets are also allowed. Evaluation of parentheses occurs from the outside in. The principal use of nested iteration sets is to reduce typing when subsets of an element are repeated. For example,

```
createdir >Smith_dir>(new>(first second) old>third)
```

would create three directories

```
>Smith_dir>new>first

>Smith_dir>new>second

>Smith_dir>old>third
```

Active Strings

The following hypothetical situation introduces the next feature of the command language. Suppose one wants to periodically delete the least recently used segment in a directory. After rejecting the notion of listing the directory and picking out the appropriate segment by hand, one would probably write a small program (say, oldest_segment) to perform the same task, have it print the name of the segment, and then delete the segment. Further thought leads one to the realization that it should be possible to give the results of oldest_segment directly to the delete command. The desire generalizes into the notion of active strings where an *active string* is defined to be an element of a command line which is immediately evaluated (executed) and the resultant value placed back into the command line. A program explicitly designed to be used in an active string is called an *active function*. An active function must return a varying character string as its value.

Since they are called from the user's terminal, active functions do not follow the standard library subroutine practice of returning a status code. Instead, upon detecting an error, they call the subroutine active_fnc_err_ which signals an error condition (see the MPM subroutine write-up for active_fnc_err_). Also, they are inherently more expensive than library subroutines

since they are designed to interact directly with a human user. For these reasons, they should not be called from a program.

The delimiters of an active string are the left bracket ([) and the right bracket (]). Thus, in terms of the hypothetical situation sketched above:

```
delete [oldest_segment]
```

performs the desired task where oldest_segment has been changed to return its value as a varying character string rather than printing its value on the terminal. The command processor scans the command line, discovers the active string, evaluates it, places the obtained value into the command line, discovers the terminator, evalutes what is left (which is "delete" as a command name and some character string as arguments), and then returns.

To afford fuller generality, active functions can have arguments and active strings can be nested. Suppose one had a random name generating routine called namer which took an arbitrary two-character string as a "seed" and returned a 32-character (or shorter) varying character string, then:

```
rename [oldest_segment] [namer xz]
```

would cause the least recently used segment to be renamed to whatever random name emerged from namer when the latter was invoked with xz as an argument. Now supposing one also had a random number generator called random for priming namer,

```
rename [oldest_segment] [namer [random]]
```

is also valid (provided random returns a varying character string value).

An implication of the use of active strings which deserves further emphasis is the fact that after being evaluated, the value is actually rescanned for active strings before being inserted into the command line. For example, if procedure alpha returned [beta] as its value, and x were some appropriate command, then

```
x [alpha]
```

would invoke x with whatever value procedure beta in turn returned.

In some cases, the user may not want the string returned by the active string to be rescanned for further active strings. To suppress this rescanning, one immediately precedes the active string with a vertical bar (|). Continuing the immediately preceding example,

```
x |[alpha]
```

would result in the invocation of x with [beta] as an argument. Note that all other scanning (e.g., for spaces) is performed on the returned string.

Iteration can, of course, be combined with the above features. For example, suppose the program "bill" returned the character string "arthur robert fred". Then the command line

```
print ([bill])
```

would expand into

```
print (arthur robert fred)
```

The command line finally obtained when all active strings have been processed is called the expanded command line. For example, in the case "print ([bill])" above, the expanded command line is "print (arthur robert fred)".

The maximum length of the expanded command line is, by default, 128 characters. This size can be changed using the set_com_line command (see the MPM command write-up). For the sake of efficiency, it is recommended that the size be left at 128 characters except when a larger size is temporarily needed (to accommodate a large returned string from some active function, for example).

For simplicity, all of the above examples have used active strings consisting of a single command line. In its most general form, an active string can consist of any number of legal command lines separated by semicolons. The value of the active string is the concatenation of the values of the command lines. For example, if the active string

```
[act_fnc1 x]
```

returns the value TURN and the active string

```
[act_fnc2 y z]
```

returns the value STYLE then the active string

```
[act_fnc1 x; act_fnc2 y z]
```

returns the value TURNSTYLE.

Note

The idea of active strings is taken from the TRAC® language designed by Calvin N. Mooers.

Concatenation

Another desirable feature of a command language is the ability to form basic elements (i.e., character strings) by concatenation with nonbasic elements (e.g., the values of active strings). For example, suppose one has written an active function called work which returns the character string representation of the path name of one's working directory. It should then be possible to perform a command (presumably from some other directory) such as

```
rename  [work]>square_root sqrt
```

and have the first argument to rename be the concatenation of the value of the work active function with the string ">square_root". In the Multics command language, this facility is furnished in precisely the manner shown. That is, the value of a delimited element of a command is concatenated with the string or delimited element adjacent to it when there is no space between the two.

Note that more than one delimited element can be concatenated. For example,

```
delete  [work]>([bill])
```

deletes the segments arthur, robert, and fred in the user's working directory where the active function "bill" is defined as above.

Note also that concatenation is permissible in either direction with regard to the delimited string and the non-delimited string. For example,

```
delete >project_dir>Doe>([bill])
```

deletes the segments arthur, robert, and fred in the directory >project_dir>Doe.

Concatenation was implicitly assumed in our description of iteration.

Reserved Characters and Quoted Strings

Note that in the context of the Multics command language, the following characters are reserved: space, quotation mark ("), semicolon (;), the new line character, left and right brackets ([and]), left and right parentheses ("(" and ")"), and the vertical bar (|) when adjacent to the left bracket. Occasionally, however, it is necessary to use a reserved character without its special meaning. For example, we might want to pass a semicolon as an argument to a command. The character quotation mark (") is reserved for this purpose. Reserved characters within a quoted string (i.e., a string of characters surrounded by quotation marks) are treated as ordinary characters. Thus,

```
rename ";" foo
```

causes a semicolon to be passed as an argument to the rename command. Also, since a quotation mark is a reserved character, it may be desirable to suppress its special meaning. For this purpose, two adjacent quotation marks within a quoted string are interpreted as a single quotation mark. For example,

```
delete "A""B"
```

causes the argument A"B to be passed to the delete command.

Notes

The command processor can be called from a program by using the procedure cu_$cp. See the write-up of the subroutine cu_ in the MPM. See also the MPM Reference Guide section, The Multics Programming Environment.

CONSTRUCTING AND INTERPRETING NAMES

The various types of names used on Multics are constructed and interpreted according to certain conventions. The names in question are user names, segment names, command names, subroutine names, I/O stream names and condition names.

User names are discussed in the MPM Reference Guide section, Access Control, since they are primarily used to specify access control information.

A segment can be named in two ways. Its location in the storage system hierarchy is specified by its path name. The name by which it is known in a process is its reference name. The star convention and equal convention provide shorthand methods of specifying segment names. Offset names allow specification of externally known locations in a segment.

Path Names

As described in the MPM Introduction Chapter 3, Beginner's Guide to The Use of Multics, each segment (or directory or link) in the Multics storage system has an entry in a superior directory. Any segment (or directory or link) can be found by following the appropriate entries from a designated directory through inferior directories until the desired segment (or directory or link) entry is reached. An absolute path name is just such a sequence of entry names starting from the root directory. A relative path name is a sequence relative to the current working directory. Path names, whether relative or absolute, are typically used as arguments to commands and subroutines.

An entry name is a string of 32 or fewer ASCII characters. Only the greater-than (>) and less-than (<) characters are prohibited in entry names, since they are used to form path names as described below. Several other characters are not recommended for entry names -- asterisk (*), question mark (?), percent sign (%), equals (=) and dollar sign ($) -- because standard commands attach special meanings to them. Each is explained below.

In general, entry names consist of the upper- and lower-case alphabetic characters, the digits, the underscore (_) and the period (.), and must have at least one nonblank character. The underscore is used to simulate a space for readability; e.g., a segment might be named new_seg. (Including a space in an entry name is permitted, but is cumbersome since the command language uses spaces to delimit command names and arguments.) The period is used to separate components of an

entry name, where a component is a logical part of the name. Several system conventions depend on components. For example, compilers on Multics expect the language name to be the last component of the name of a source segment to be compiled; e.g., square_root.pl1 for a PL/I source segment.

An absolute path name is formed from a sequence of entry names, each preceded by a greater-than character. The initial greater-than indicates that the entry name following it designates an entry in the root directory. Thus, an absolute path name has the form >first_dir>second_dir>third_dir>my_seg.

The directory first_dir is immediately inferior to the root, second_dir is an entry in first_dir, etc. A maximum of 16 levels of directories is allowed from the root to the final entry name. The number of characters in the path name cannot exceed 168. Each intermediate entry in the chain can be either a directory or a link to a directory. The final entry can be a directory, a segment or a link.

A relative path name looks like an absolute path name except that it does not contain a leading greater-than character, and can begin with less-than characters as explained below. It is interpreted by various commands to be a path name relative to the user's working directory. The simplest form of relative path name is the single name of an entry in the user's working directory. For example, the relative path name alpha refers to the entry alpha in the user's working directory. On a slightly more complex level, the relative path name sub_dir>beta refers to the entry beta in the directory sub_dir which is immediately inferior to the user's working directory.

The less-than character can be used at the front (left end) of a relative path name to indicate that the directory immediately superior to the working directory is where the following entry name is to be found. This principle can be extended so that several less-than characters cause the superior directory several levels higher than the working directory to be searched for the first entry name in the relative path name.

In the following examples, the user's working directory is

>dir1>dir2>dir3>dir4

A relative path name of

new_seg

would designate the segment with the absolute path name

```
>dir1>dir2>dir3>dir4>new_seg
```

A relative path name of

```
dir5>old_seg
```

would designate the segment

```
>dir1>dir2>dir3>dir4>dir5>old_seg
```

A relative path name of

```
<dir0>newer
```

would designate the segment

```
>dir1>dir2>dir3>dir0>newer
```

A relative path name of

```
<<<sample_dir>game_dir>chess
```

would designate the segment

```
>dir1>sample_dir>game_dir>chess
```

The Star Convention

Many commands that accept path names as input allow the final entry name in the path to be a star name. A star name is an entry name that identifies a group of entries in a single directory. Commands that accept star names perform their function for each directory entry identified by the star name.

A star name identifies all entries in a directory having an entry name that matches the star name. A special type of matching is performed in which some character strings of the star name are compared with corresponding strings of the entry name, while other character strings of the entry name are ignored. If the star name strings match the entry name strings, then the entry name matches the star name. Therefore, the entries identified by a star name all have similar names.

Under the star convention, the matching is performed according to the rules for constructing and interpreting star names listed below:

1) A star name is an entry name. Therefore, it is composed of a string of 32 or fewer ASCII printing graphics or spaces, none of which may be the less-than (<) or greater-than (>) character. Note that, unlike an entry name, a star name cannot contain control characters such as backspace, tab, or new line.

2) A star name is composed of one or more nonnull components. This means that a star name cannot begin or end with a period (.), and cannot contain two or more consecutive periods.

3) Each question mark (?) character appearing in a star name component is treated as a special character. The question mark matches any character that appears in the corresponding component and letter positions of the entry name.

4) Each asterisk (*) character (loosely referred to as a star) appearing in a star name component is treated as a special character. The asterisk matches any number of characters (including none) appearing in the corresponding component and letter positions of the entry name. Only one asterisk can appear in each star name component, except for a double star component as noted in the next rule.

5) A star name component consisting only of a double star (**) is treated as a special component. The double star component matches any number of components (including none) in the corresponding component position of the entry name. Only one double star component can appear in a star name.

Note that the rules above do not require that star names contain asterisks or question marks. Therefore, an entry name that does not contain either of these special characters can be used as a star name, as long as it does not contain any null components. Note too that the rules above impose no restrictions on the form of the entry names to be matched with the star name. Such names can contain null components that match only star name components of * or **.

The following examples illustrate some common forms for star names. The entry name

```
*.pl1
```

identifies all two-component entries in the user's working directory that have pl1 as their second component; the path name

```
sub_dir>my_prog.new.*
```

identifies all three-component entries in the directory sub_dir (which is immediately inferior to the user's working directory) that have my_prog.new as their first and second components; and

```
*
```

and

```
*.*
```

identify, respectively, all one-component and two-component entries in the working directory. The entry name

```
my_prog.**
```

identifies all entries with my_prog as the first (and possibly only) component;

```
*.**.my_seg
```

identifies all entries with two or more components of which the last is my_seg;

```
**.pl1
```

identifies all entries with pl1 as the last (and possibly only) component; and

```
**
```

identifies all entries in the user's working directory. The entry name

```
prog*.pl1
```

identifies all two-component entries whose first component begins with prog and has four or more characters, and whose second component is pl1;

```
*_data
```

identifies all one-component entries whose first component ends with _data and has five or more characters; and

```
interest_*_data.*.*
```

identifies all three-component entries whose first component begins with interest_, ends with _data, and has fourteen or more characters. Finally, the entry name

```
ad?
```

identifies all three-character one-component entries in the user's working directory which begin with ad;

```
!?????????????
```

identifies all fifteen character one-component entries beginning with ! (called unique names because such names are generated by the unique_chars_ subroutine and the unique active function); and

```
sub_dir>prog?.**.pl1
```

identifies all entries in the directory sub_dir (which is immediately inferior to the user's working directory) with two or more components, the first of which has five characters and begins with prog, and the last of which is pl1.

The Equal Convention

Some commands that accept pairs of path names as their arguments (e.g., the rename command) allow the final entry name of the first path to be a star name, and the final name of the second path to be an equal name. An equal name is an entry name containing special characters that represent one or more characters from the entry names identified by the star name. Commands that accept equal names provide a powerful mechanism for mapping certain character strings from the first path name into the second path name of a pair. Such a mechanism helps to reduce the typing required for the second path name, and it can be essential for mapping character strings from the entry names identified by the star name into the equal name, because these characters strings are not known when the command is issued.

Under the equal convention, the mapping of character strings from the star name into the equal name is performed according to the rules for constructing and interpreting equal

names given below:

1) An equal name is an entry name. Therefore, it is composed of a string of 32 or fewer ASCII printing graphics or spaces, none of which may be the less-than (<) or the greater-than (>) character. Note that, unlike an entry name, and equal name cannot contain control characters such as backspace, tab, or new line.

2) An equal name is composed of one or more nonnull components. This means that an equal name cannot begin or end with a period (.), and cannot contain two or more consecutive periods.

3) Each percent (%) character appearing in an equal name component is treated as a special character. The percent represents the character in the corresponding component and letter position of the entry name identified by the star name. An error occurs if the corresponding character does not exist.

4) Each equal sign (=) appearing in an equal name component is treated as a special character. The equal sign represents the corresponding component of the entry name identified by the star name. An error occurs if the corresponding component does not exist. An error also occurs if an equal sign appears in a component that also contains a percent character. Only one equal sign can appear in each equal name component, except for a double equal sign component, as noted in the next rule.

5) An equal name component consisting only of a double equal sign (==) is treated as a special component. The double equal sign component represents all components of the entry names identified by the star name that have no other corresponding components in the equal name. From this definition, it follows that if the double equal sign component represents (i.e., corresponds to) any components of the entry name identified by the star name, then the equal name necessarily has the same number of components as the entry name. Only one double equal sign component can appear in an equal name.

Note that the rules above do not require that equal names contain equal signs or percent characters. Therefore, an entry name that does not contain either of these special characters can be used as an equal name, as long as it does not contain any null

components. Note too that the rules above impose no restrictions on the form of the entry names identified by the star name. These names can contain null components. However, the rename and addname commands cannot be called with an entry name that contains null components, because these commands treat their arguments as either star names or equal names. The fs_chname command can be used to rename entries if names containing null components are accidentally created.

The following examples illustrate how equal names might be used in rename and addname commands. The command

```
rename  random.data_base  order.=
```

is equivalent to

```
rename  random.data_base  ordered.data_base
```

```
addname  world.data  =.statistics  =.census
```

is equivalent to

```
addname world.data  world.statistics world.census
```

The command

```
rename  random.data.base =.=
```

is equivalent to

```
rename  random.data.base  random.data
```

The star convention is used in the command

```
rename  *.data_base  =.data
```

to rename all two-component entry names with data_base as their second component to have, instead, a second component of data. The command

```
rename  alpha  beta.=.gamma
```

is in error because the first name of the pair does not contain a component corresponding to the equal sign in the second name. The command

```
rename  program.pl1  old_=.=
```

is equivalent to

```
rename  program.pl1  old_program.pl1
```

and

```
addname  data  first_=_set
```

is equivalent to

```
addname  data  first_data_set
```

The next rename command, which contains a double equal sign component,

```
rename  one.two.three  1.==
```

is equivalent to

```
rename  one.two.three  1.two.three
```

and

```
addname  one.two.three.four  1.==.4
```

is equivalent to

```
addname  one.two.three.four  1.two.three.4
```

Note that, in the two examples above, the first name has components that are represented by the double equal sign in the second name of each pair. As a result, the number of components represented by the equal name is the same as the number of components in the first name. On the other hand, in the command

```
addname  able  ==.baker.charlie
```

which is equivalent to

```
addname  able  baker.charlie
```

the double equal sign does not represent any component of the first name. Component able of the first name is represented in the equal name by baker. As a result, the equal name represents a greater number of components than there are in the first. The command

```
addname  **.ec  ==.absin
```

uses the star convention to add a name to each entry with a name whose last component is ec. The last component of this new name is absin, and the first components (if any) are the same as those of the name ending in ec. Finally, the command

```
rename  ???*.data  %%%.=
```

renames all two-component entry names that have a last component of data and a first component containing three or more characters to have a first component that has been truncated to the first three characters and the same last component. Note that the command

```
rename  *.data  %%%.=
```

may result in an error if the first component of any name matching *.data has fewer than three characters.

Reference Names

Procedures executing in a process need to refer by name to other segments known in that process. Such a name is a reference name. A reference name may be the same as an entry name of the segment, or may be different. For example, when a dynamic linkage fault occurs for a reference name, the linker searches (using search rules) for a segment which has an entry name identical to that reference name. A procedure call, an invocation of a command through the command processor, or a reference to an external data segment is of this type, as is a segment made known by the hcs_$make_ptr subroutine. Search rules (telling which directories to search for the entry name) may be specified by the user or may be system defaults. The default search rules are described in the MPM Reference Guide section, The System Libraries and Search Rules. Alternatively, the user may explicitly designate the reference name to be associated with a specified segment. The initiate command and the hcs_$initiate and hcs_$initiate_count subroutines perform this function. In this case, the reference name need not have any similarity to any entry name of the segment.

Since a reference name is associated only with segments made known in a process, the same reference name may be used in two different processes to refer to two different segments. Also, a reference name/segment binding exists only for the duration of the process in which it is specified. It is possible to break that binding by terminating the segment, thus causing all links

to that segment to be unsnapped and causing the segment to no longer be known in the process (by any reference name). Any reference name of a terminated segment can be used again in the process to refer to a different segment. (See the write-ups for the terminate and terminate_refname commands and the term_, hcs_$terminate_file, and hcs_$terminate_seg subroutines.)

Individual reference names can be unbound in a process without terminating the segment unless the reference name removed was the only one on the segment. (See the write-ups for the terminate_single_refname command and for the term_, hcs_$terminate_name, and hcs_$terminate_noname subroutines.) A user wishing to replace a system routine with one of this own by the same reference name might terminate that one reference name and initiate his routine by that same reference name:

```
terminate_single_refname sys_prog

initiate >my_dir>my_prog sys_prog
```

Thereafter, references to sys_prog would invoke his routine, my_prog, with one exception. Other system routines bound to sys_prog would continue to invoke the system routine since those links had been presnapped when the routines were bound together.

Note that the commands and the term_ subroutine unsnap dynamic links to any segment that has a linkage section, whereas the hcs_ entry points do not unsnap links.

Offset Names

Procedures frequently have more than one entry point, and data segments frequently have internal locations which are known externally by symbolic name. The names of the entry points and the internal locations are called offset names. Both designate symbolically an offset within the segment. The location specified may be refered to by the construction ref_name$offset_name where the dollar sign separates the reference name and offset name.

In many cases the entry point to a procedure has the same name as the segment itself (or the segment has several entry names corresponding to the names of its entry points). A shorthand notation allows the offset name to be assumed to be the same as the reference name. For example,

```
call square_root (n);
```

is interpreted to mean

```
call square_root$square_root (n);
```

and the command line

```
rename a b
```

is equivalent to

```
rename$rename a b
```

It is worthwhile to remember that if the user has renamed one of his procedure segments (perhaps to preserve an old copy) or created a storage system link to a segment using a different name, he must thereafter use the full reference name/offset name construction when referring to that segment as a procedure external data segment. For example, a PL/I subroutine compiled with subr_name as the label of its procedure statement, and then renamed new_name must be referred to as new_name$subr_name. It is also important to not that if a reference name/segment binding has been established in a process, then merely renaming the segment does not break the association in that process. To do this, the segment must be terminated.

Command, Subroutine, Condition and I/O Stream Names

These types of names all have some conventions in common.

1) Each is permitted to be not more than 32 characters in length.

2) All ASCII characters are legal in any position except as noted in points 3 and 4 below.

3) System subroutine names end in an underscore to prevent conflicts with subroutine names given by users. (I.e., the user may easily avoid conflicts by refraining from having an underscore as the last character of his subroutine names.)

4) Condition and I/O stream names which are part of the system should end in an underscore to help prevent conflicts with names given by users. A glance at the MPM Reference Guide sections, List of System Conditions and Default Handlers, and List of Names with Special Meanings, reveals many system condition and I/O stream names which do not observe this convention. These names were incorporated into the system before this convention was established.

COMMAND AND ACTIVE FUNCTION NAME ABBREVIATIONS

The following is a list of abbreviations for commands and active functions. The abbreviations are also listed immediately after the command or active function name in the individual write-ups. For example,

Name: abbrev, ab

Abbreviation	Command Name
aa	alm_abs
ab	abbrev
abc	adjust_bit_count
ac	archive
an	addname
as	archive_sort
bd	bind
br	basic_run
bs	basic_system
car	cancel_abs_request
cd	createdir
cdr	cancel_daemon_request
cdwd	change_default_wdir
cem	change_error_mode
cis	check_info_segs
co	console_output
cp	copy
cpa	compare_ascii
cr	create
cwd	change_wdir
da	deleteacl
db	debug
dc	deletecacl
dcm	decam
dcn	display_component_name
dd	delete_dir
df	deleteforce
did	delete_iacl_dir
dirs	directories
dis	delete_iacl_seg
dl	delete
dn	deletename
dp	dprint
dpn	dpunch
ds	dump_segment
e	enter
ear	enter_abs_request
ec	exec_com

ep	enterp
fa	fortran_abs
fl	format_line
fo	file_output
ft	fortran
gcl	get_com_line
gpn	get_pathname
gq	getquota
hd	hold
hmu	how_many_users
in	initiate
ind	indent
l	login
la	listacl
lar	list_abs_requests
lc	listcacl
lcp	lisp_compiler
ldr	list_daemon_requests
lid	list_iacl_dir
lis	list_iacl_seg
lk	link
ll	line_length
ln	listnames
lrn	list_ref_names
ls	list
lt	listotals
ml	mail
mpt	make_peruse_text
mq	movequota
mv	move
nondirs	nondirectories
nonsegs	nonsegments
pa	pl1_abs
pat	print_attach_table
pbm	print_bind_map
pdl	print_dartmouth_library
pdwd	print_default_wdir
pg	progress
pgt	page_trace
pi	program_interrupt
pli	print_link_info
plu	print_linkage_usage
pmotd	print_motd
pr	print
psr	print_search_rules
pt	peruse_text
pwd	print_wdir
qx	qedx

ra	reorder_archive
rq	reorder_archive
rdf	ready_off
rfa	runoff_abs
rf	runoff
rdn	ready_on
rdy	ready
re	reprint_error
rl	release
rn	rename
ru	resource_usage
sa	setacl
sbc	set_bit_count
sc	setcacl
scl	set_com_line
sdl	set_dartmouth_library
segs	segments
sf	sort_file
sid	set_iacl_dir
sis	set_iacl_seg
sr	start
ssd	set_search_directories
ssf	set_safety_sw_off
ssn	set_safety_sw_on
ssr	set_search_rules
st	status
tc	truncate
tm	terminate
tmr	terminate_refname
tms	terminate_segno
ts	trace_stack
ul	unlink
v2pa	v2pl1_abs
wh	where
ws	walk_subtree

ALPHABETICAL LIST OF ACTIVE FUNCTIONS

The active functions of interest to most Multics users are documented in this and seven other MPM Reference Guide Sections: 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 MPM Reference Guide section, The Command Language, describes the purpose of active functions and illustrates their use.

The following alphabetical list of the active functions available to Multics users includes the grouping to which each active function belongs; i.e., indicates which write-up contains its description.

Active Function	Group
and	logical
date	date and time
date_time	date and time
day	date and time
day_name	date and time
directories	segment name
directory	segment name
divide	arithmetic
entry	segment name
equal	logical
exists	logical
files	segment name
format_line	character string
get_pathname	segment name
greater	logical
home_dir	segment name
hour	date and time
index	character string
index_set	character string
length	character string
less	logical
links	segment name
long_date	date and time
minus	arithmetic
minute	date and time
mod	arithmetic
month	date and time
month_name	date and time
nondirectories	segment name
nonlinks	segment name
nonsegments	segment name

not	logical
or	logical
path	segment name
pd	segment name
plus	arithmetic
query	question asking
response	question asking
segments	segment name
string	character string
strip	segment name
strip_entry	segment name
substr	character string
suffix	segment name
time	date and time
times	arithmetic
unique	segment name
user	user parameter
wd	segment name
year	date and time

 (END)

LOGICAL ACTIVE FUNCTIONS

The active functions of interest to most Multics users are documented in this and seven other MPM Reference Guide sections: Alphabetical List of 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 MPM Reference Guide section, The Command Language, describes the purpose of active functions and illustrates their use. Note that in a command line an active function must be enclosed in square brackets. However, those brackets have been omitted from usage descriptions in this wirte-up. They are included in examples.

The following active functions 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. (See the MPM write-up for exec_com.)

Name: and

This active function returns the value "true" if both of its arguments are true. Otherwise, it returns the value "false".

Usage

and arg1 arg2

1) argi — 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.

Name: or

This active function returns the value "true" if either or both of its arguments are true. Otherwise, it returns the value "false".

Usage

or arg1 arg2

1) argi — 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.

Name: not

This active function returns either "true" or "false", whichever is the opposite of the value of its argument.

Usage

not arg

1) arg — is a character string. If arg = "true", then "false" is returned. If arg = "false", then "true" is returned. Otherwise an error diagnostic is issued.

Name: equal

This active function returns the value "true" if its two arguments are equal (i.e., identical). Otherwise, it returns the value "false".

Usage

equal arg1 arg2

1) argi — are any character strings.

Name: greater

This active function returns the value "true" if the value of the first argument is greater than the value of the second argument. Otherwise, it returns the value "false".

Usage

greater arg1 arg2

1) argi — are character strings. If both arg1 and arg2 are character string representations of single-precision fixed binary integers, then a numeric comparison is made. Otherwise, the comparison is made a character at a time, starting from the left, using the ASCII collating sequence.

Name: less

This active function returns the value "true" if the value of the first argument is less than the value of the second argument. Otherwise, it returns the value "false".

Usage

less arg1 arg2

1) argi — are character strings. The comparison is made as described above in the greater active function.

Name: exists

This active function checks for the existence of various types of items, depending on the value of a key.

Usage

exists key arg

1) key — can have any of the following values:

entry	returns "true" if an entry with path name arg exists; otherwise it returns "false".
branch	returns "true" if a branch with path name arg exists; otherwise it returns "false".
segment	returns "true" if a segment with path name arg exists; otherwise it returns "false".
directory	returns "true" if a directory with path name arg exists; otherwise it returns "false".
link	returns "true" if a link with path name arg exists; otherwise it returns "false".
non_null_link	returns "true" if a link with path name arg exists and points to an existing segment or directory; otherwise it returns "false".
argument	returns "true" if it has been passed an argument; otherwise it returns "false".

2) arg is the argument described for each possible value of key.

Example

The following example illustrates the use of one of the active functions described in this write-up. It involves the use of the &if control statement of the exec_com command. (See the MPM write-up for exec_com.)

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

&then &goto elsewhere

&else change_wdir [home_dir]
```

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

ARITHMETIC ACTIVE FUNCTIONS

The active functions of interest to most Multics users are documented in this and seven other MPM Reference Guide sections: Alphabetical List of Active Functions, Logical 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 MPM Reference Guide section, The Command Language, describes the purpose of active functions and illustrates their use. Note that in a command line an active function must be enclosed in square brackets. However, those brackets have been omitted from usage descriptions in this write-up. They are included in examples.

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

Name: plus

This active function returns a value that is the character string representation of the sum of its arguments.

Usage

plus arg1 ... argn

1) argi are character string representations of single-precision fixed binary integers.

Name: minus

This active function returns a value that is the characater string representation of the difference of its two arguments; i.e., its first argument minus its second argument.

Usage

minus arg1 arg2

1) argi are character string representations of single-precision fixed binary integers.

Name: times

This active function returns a value that is the character string representation of the product of its arguments.

Usage

times arg1 ... argn

1) argi are character string representations of single-precision fixed binary integers.

Name: divide

This active function returns a value that is the character string representation of the result of dividing its first argument by its second.

Usage

divide arg1 arg2

1) argi are character string representations of single-precision fixed binary integers.

Name: mod

This active function returns a value that is the character string representation of its first argument taken modulus its second argument.

Usage

mod arg1 arg2

1) argi are character string representations of single-precision fixed binary integers.

Name: min

This active function returns a value that is the character string representation of the numerical minimum of its arguments.

Usage

min arg1 ... argn

1) argi are character string representations of single-prevision fixed binary numbers.

Name: max

This active function returns a value that is the character string representation of the numerical maximum of its arguments.

Usage

max arg1 ... argn

1) argn are character string representations of single prevision fixed binary numbers.

Example

The following example illustrates the use of one of the active functions described in this write-up.

```
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.

 (END)*

CHARACTER STRING ACTIVE FUNCTIONS

The active functions of interest to most Multics users are documented in this and seven other MPM Reference Guide sections: Alphabetical List of Active Functions, Logical Active Functions, Arithmetic Active Functions, Segment Name Active Functions, Date and Time Active Functions, Question Asking Active Functions, and User Parameter Active Functions. The MPM Reference Guide section, The Command Language, describes the purpose of active functions and illustrates their use. Note that in a command line an active function must be enclosed in square brackets. However, those brackets have been omitted from usage descriptions in this write-up. They are included in examples.

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

Name: length

This active function returns a value that is the character string representation of the number of characters in a specified string

Usage

length arg

1) arg is a character string.

Name: index

This active function returns a value that is the character string representation of the character position in the first argument where a substring matching the second argument begins.

Usage

index arg1 arg2

1) argi are character strings.

Name: substr

This active function returns a value that is the character string representation of a substring of the first argument beginning at the character position specified by the second argument and having a length as specified by the third argument.

Usage

substr arg1 arg2 arg3

1) arg1 is a character string.

2) arg2, arg3 are character string representations of single-precision fixed binary integers. If arg3 is omitted, it is assumed to be the length of arg1 minus arg2, so that the value returned is the rest of arg1 beginning with the arg2-th character.

Name: index_set

This active function returns a value that is the character string representation of the numbers from 1 to the number specified, with a space between successive numbers. This active function is useful within an iteration in a command line.

Usage

index_set arg

1) arg is a character string representation of a single-precision fixed binary integer.

Name: string

This active function returns its input arguments, separated from one another by a single space, as a quoted character string. By using string, the user can treat the results of one or more active functions as a single character string.

Usage

string arg1 ... argn

1) argi are the optional input arguments that are to be returned as a single character string. If no input arguments are present, then 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: format_line, fl

This active function returns a formatted character string that is constructed from a control string and other optional input 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 arg1 ... argn

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 which are expanded into the appropriate characters and inserted into the return value. The MPM write-up of the ioa_ subroutine describes 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) argi — are optional input arguments that are substituted in the formatted return value, according to the ioa_ control string.

Name: search

This active function returns a value that is the character string representation of the character position in the first argument that contains a character matching any character in the second argument.

Usage

search arg1 ... arg2

1) arg1 is the character string to be searched.

2) arg2 is the set of characters searched for.

Name: verify

This active function returns a value that is the character string representation of the character position in the first argument that contains a character not matching any character in the second argument.

Usage

verify arg1 ... arg2

1) arg1 is the character string to be verified.

2) arg2 is the set of characters searched for.

Examples

The following examples illustrate the use of two of the active functions described in this write-up. One of the examples involves the use of the &if control statement of the exec_com command. (See the MPM write-up for exec_com.)

```
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. Note that the user's response has been underlined for the sake of clarity.

```
Is November 22, 1972 a good date? yes
Beginning execution.
```

SEGMENT NAME ACTIVE FUNCTIONS

The active functions of interest to most Multics users are documented in this and seven other MPM Reference Guide sections: Alphabetical List of Active Functions, Logical Active Functions, Arithmetic Active Functions, Character String Active Functions, Date and Time Active Functions, Question Asking Active Functions, and User Parameter Active Functions. The MPM Reference Guide section, The Command Language, describes the purpose of active functions and illustrates their use. Note that in a command line an active function must be enclosed in square brackets. However, those brackets have been omitted from usage descriptions in this write-up. They are included in examples.

The following active functions return a segment path name or entry name 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 entry name of a segment.

Name: home_dir

This active function returns the path name of the user's home directory (usually of the form >user_dir_dir>Project>Person) as obtained by a call to user_info_$homedir.

Usage

home_dir

Name: pd

This active function returns the path name of the process directory of the process in which it is invoked.

Usage

pd

Name: wd

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

Usage

wd

Name: get_pathname, gpn

This active function returns the absolute path name of the segment that is designated by the reference name or segment number specified. Reference names are discussed in the MPM Reference Guide section, Constructing and Interpreting Names.

Usage

get_pathname -control_arg- arg

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: path

This active function returns the absolute path name of the specified segment.

Usage

path arg

1) arg — is a segment name.

Name: directory

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

Usage

directory arg

1) arg — is a segment name.

Name: entry

This active function returns the entry name portion of the absolute path name of the specified segment.

Usage

entry arg

1) arg is a segment name.

Name strip

This active function returns the absolute path name of the specfied segment after having removed the last component of the entry name, if one exists or if it matches a specified character string.

Usage

strip arg1 -arg2-

1) arg1 is a segment name.

2) arg2 is an optional character string that, if present and if it matches the last component of the entry name portion of arg1, is removed from that entry name. If arg2 is not given, any last component is removed from the entry name portion of arg1, assuming arg1 has more than one component in its entry name.

Name: strip_entry, spe

This active function returns the entry name portion of the absolute path name returned by the strip active function.

Usage

strip_entry arg1 arg2

1) arg1 is a segment name.

2) arg2 is as described above under the strip active function.

Name: suffix

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

Usage

suffix arg

1) arg is a segment name.

Name: unique

This active function returns a unique character string as generated by unique_chars_ (see the MPM write-up).

Usage

unique

Name: files

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

Usage

files starname

1) starname is a path name for which the entryname portion (optionally) contains stars to be interpreted according to the star convention. (See the MPM Reference Guide section, Constructing and Interpreting Names.)

Name: segments, segs

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

Usage

segments starname

1) starname is as described above under the files active function.

Name: directories, dirs

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

Usage

directories starname

1) starname is as described above under the files active function.

Name: links

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

Usage

links starname

1) starname is as described above under the files active function.

Name: nonlinks, branches

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

Usage

nonlinks starname

1) starname is as described above under the files active function.

Name: nondirectories, nondirs

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

Usage

nondirectories starname

1) starname is as described above under the files active function.

Name: nonsegments, nonsegs

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

Usage

nonsegments starname

1) starname is as described above under the files active function.

Examples

The following examples illustrate the use of three of the active functions described in this write-up.

status [get_pathname refname]

This example invokes the status command with the path name 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.

dprint [segs *.pl1]

This example invokes the dprint command to have copies printed of all 2-component PL/I source segments in the working directory.

DATE AND TIME ACTIVE FUNCTIONS

The active functions of interest to most Multics users are documented in this and seven other MPM Reference Guide sections: Alphabetical List of Active Functions, Logical Active Functions, Arithmetic Active Functions, Character String Active Functions, Segment Name Active Functions, Question Asking Active Functions, and User Parameter Active Functions. The MPM Reference Guide section, The Command Language, describes the purpose of active functions and illustrates their use. Note that in a command line an active function must be enclosed in square brackets. However, those brackets have been omitted from usage descriptions in this write-up. They are included in examples.

The following active functions return information about dates and times.

Name: date

This active function returns a date abbreviation in the form "mm/dd/yy"; e.g., "02/20/73".

Usage

date arg1 ... argn

1) argi — are optional input arguments that determine the date and time for that information is returned. These arguments must be in a form acceptable to the convert_date_to_binary subroutine (see the MPM write-up). 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., "08/07/72 0945.7 est Mon.

Usage

date_time arg1 ... argn

1) argi — are as described above under the date active function.

Name: day

This active function returns the 1- or 2-digit number of a day of the month, from 1 to 31; e.g., "7".

Usage

day arg1 ... argn

1) argi are as described above under the date active function.

Name: day_name

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

Usage

day_name arg1 ... argn

1) argi are as described above under the date active function.

Name: hour

This active function returns the 1- or 2-digit number of an hour of the day, from 0 to 23; e.g., "9".

Usage

hour arg1 ... argn

1) argi 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, 1972".

Usage

long_date arg1 ... argn

1) argi are as described above under the date active function.

Name: minute

This active function returns the 1- or 2-digit number of a minute of the hour, from 0 to 59; e.g., "45".

Usage

minute arg1 ... argn

1) argi are as described above under the date active function.

Name: month

This active function returns the 1- or 2-digit number of a month of the year, from 1 to 12; e.g., "8".

Usage

month arg1 ... argn

1) argi 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 arg1 ... argn

1) argi are as described above under the date active function.

Name: time

This active function returns a 4-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 arg1 ... argn

1) argi are as described above under the date active function.

Name: year

This active function returns the 2-digit number of a year of the century; e.g., "73".

Usage

year arg1 ... argn

1) argi are as described above under the date active function.

Example

The following example illustrates the use of one of the active functions described in this write-up.

```
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 a string such as "2/1".

QUESTION ASKING ACTIVE FUNCTIONS

The active functions of interest to most Multics users are documented in this and seven other MPM Reference Guide sections: Alphabetical List of Active Functions, Logical Active Functions, Arithmetic Active Functions, Character String Active Functions, Segment Name Active Functions, Date and Time Active Functions, and User Parameter Active Functions. The MPM Reference Guide section, The Command Language, describes the purpose of active functions and illustrates their use. Note that in a command line an active function must be enclosed in square brackets. However, those brackets have been omitted from usage descriptions in this write-up. They are included in examples.

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".

Usage

query arg

1) arg is a question to be asked. 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

1) arg is a question to be asked the user.

Example

The following example illustrates the use of one of the active functions described in this write-up. It involves the use of the &if control statement of the exec_com command. (See the MPM write-up for exec_com.)

```
&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 active functions of interest to most Multics users are documented in this and seven other MPM Reference Guide sections: Alphabetical List of Active Functions, Logical Active Functions, Arithmetic Active Functions, Character String Active Functions, Segment Name Active Functions, Date and Time Active Functions, and Question Asking Active Functions. The MPM Reference Guide section, The Command Language, describes the purpose of active functions and illustrates their use. Note that in a command line an active function must be enclosed in square brackets. However, those brackets have been omitted from usage descriptions in this write-up. They are included in examples.

The following active function returns user parameters obtained from system data bases.

Name: user

This active function returns various user parameters.

Usage

user arg

1) arg — can have one of the following values:

name	returns the user name at log in time.
project	returns the user project ID.
login_date	returns the date at log in time. The date is of the form "mm/dd/yy".
login_time	returns the time of log in. 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 answer back 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"

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 according to MIT specifications. The modification prevents 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 log in. The usage is of the form "sss.t" with leading zeros suppressed.

log_time — returns the user connect time, in minutes, since log in. 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 path name 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 path name of his absentee output segment; otherwise, returns a null string.

Example

The following example illustrates the use of one of the active functions described in this write-up.

```
ioa_ [user login_time]
```

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

THE MULTICS SUBROUTINE REPERTOIRE

The following facilities are the ones considered to be part of Multics and are described in this manual. Detailed specifications on each of the subroutines mentioned below can be found, filed in alphabetical order by name, in the MPM Reference Guide section, Subroutines.

In addition, the user should consult the list of items in the Author Maintained and/or Installation Maintained Library at his installation, since local library procedures can substantially extend the standard subroutine repertoire. Documentation on the Author Maintained and/or Installation Maintained Library is supplied by the local installation.

The subroutine repertoire is organized by function into the following groups:

Storage System, Utility Routines
Storage System, Access Control and Rings of Protection
Storage System, Supervisor Entries for Manipulating Directories and Segments
Storage System, Supervisor Entries for Manipulating an Address Space
Clock and Timer Services
Subroutine Call and Argument Utilities
Command Environment Utility Procedures
I/O System Facilities
Error Handling Facilities
Routines to Convert Some Data Type To and From a Character String Representation
Object Segment Manipulation Routines
Miscellaneous Procedures

1) Storage System, Utility Routines

change_wdir_	changes the current working directory
check_star_name_	checks an entry name for correct construction as a star name
copy_acl_	copies access control list (ACL) from one segment to another
copy_names_	copies all names from one segment to another
copy_seg_	copies a segment
delete_	deletes a segment or unlinks a link
decode_entryname_	splits a procedure reference into reference name and offset name
get_equal_name_	implements the storage system equal convention
expand_path_	converts relative path name to absolute path name
get_default_wdir_	returns the default working directory
get_pdir_	returns path name of process directory
get_wdir_	returns path name of current working directory
match_star_name_	determines if an entry name matches a star name
move_names_	moves names from one segment to another
suffixed_name_	manipulates suffixes on storage system entry names
term_	removes a segment from the address space and also unsnaps any subroutine linkage to it

2) Storage System, Access Control and Rings of Protection

copy_acl_	copies ACL from one segment to another
cu_$level_get	obtains current ring validation level
cu_$level_set	sets the ring validation level
cv_acl_	formats a segment ACL entry for printing
cv_dir_acl_	formats a directory ACL entry for printing
cv_dir_mode_	converts directory access mode to bit form

cv_mode_	converts segment access mode to bit form
cv_userid_	converts process class identifier to normalized form
get_group_id_	returns access control name of current user
hcs_$add_acl_entries	adds or changes ACL entries on a segment
hcs_$add_dir_acl_entries	adds or changes ACL entries on a directory
hcs_$delete_acl_entries	deletes all or part of an ACL on a segment
hcs_$delete_dir_acl_entries	deletes all or part of an ACL on a directory
hcs_$fs_get_mode	returns access control mode for a given segment relative to the current validation level
hcs_$list_acl	returns all or part of an ACL on a segment
hcs_$list_dir_acl	returns all or part of an ACL on a directory
hcs_$replace_acl	replaces one ACL on a segment with another
hcs_$replace_dir_acl	replaces one ACL on a directory with another

3) Storage System, Supervisor Entries for Manipulating Directories and Segments

Note: some entries come in pairs. The name ending in "file" takes a segment name as an argument, while the name ending in "seg" takes a segment number instead.

hcs_$append_branch hcs_$append_branchx	creates a segment or a directory
hcs_$append_link	creates a directory link
hcs_$chname_file hcs_$chname_seg	adds, deletes, and changes names found in a directory
hcs_$delentry_file hcs_$delentry_seg	deletes a single entry in a directory
hcs_$fs_move_file hcs_$fs_move_seg	moves a segment from one directory to another
hcs_$make_seg	creates a new segment and then initiates it

hcs_$set_bc hcs_$set_bc_seg	sets the bit count of a segment
hcs_$star_	implements the storage system star convention
hcs_$status_	returns information about a given segment
hcs_$truncate_file hcs_$truncate_seg	truncates a segment to a given length

See also 1) Storage System, Utility Routines.

4) Storage System, Supervisor Entries for Manipulating an Address Space

hcs_$fs_get_mode	returns access control mode for a given segment relative to the current validation level
hcs_$fs_get_path_name	returns path name for a segment specified by segment number
hcs_$fs_get_ref_name	returns a reference name for a segment specified by segment number
hcs_$fs_get_seg_ptr	returns a segment number for a segment specified by a reference name
hcs_$initiate	maps a given segment into the address space of the current process
hcs_$initiate_count	same as hcs_$initiate but also returns the segment's bit count
hcs_$make_ptr	returns a pointer to a segment entry point, following search rules and link conventions
hcs_$terminate_file hcs_$terminate_seg	removes a segment from the address space of the current process
hcs_$terminate_name hcs_$terminate_noname	removes a reference name from the table which defines the address space

See also term_ and change_wdir_ under 1) Storage System, Utility Routines.

5) Clock and Timer Services

clock_	reads calendar clock
convert_date_to_binary_	converts ASCII string to binary time
cpu_time_and_paging_	returns CPU time used and paging activity for this process
date_time_	converts binary time to ASCII string

decode_clock_value_	returns information about binary time
timer_manager_	provides alarm clocks
total_cpu_time_	returns total CPU time used by this process

6) Subroutine Call and Argument Utilities

These are used in cases where standard PL/I language facilities are not adequate, such as a variable length calling sequence.

cu_$arg_count	returns number of arguments procedure was called with
cu_$arg_list_ptr	returns a pointer to current argument list
cu_$arg_ptr	returns a pointer to a specified argument in current argument list
cu_$arg_ptr_rel	returns a pointer to a specified argument in a specified argument list
cu_$gen_call	generates a subroutine call to a procedure where name and arguments are not known at compile time
cu_$ptr_call	generates a subroutine call to a procedure whose name is not known at compile time
cu_$stack_frame_ptr	returns a pointer to the current stack frame
cu_$stack_frame_size	returns current stack frame size
decode_descriptor_	used to interpret PL/I argument descriptors

See also the MPM Reference Guide section, Subroutine Calling Sequences.

7) Command Environment Utility Procedures

cu_$cl cu_$get_cl cu_$set_cl	controls which procedure is invoked to return to command level following quit or unclaimed signal
cu_$cp cu_$get_cp cu_$set_cp	controls which procedure is used as the command processor

See also the MPM Reference Guide section, The Command Language Environment.

8) I/O System Facilities

broadcast_	directs an output stream to several other streams
discard_output_	infinite sink for an output stream
file_	I/O interface to the Multics storage system
ioa_	produces formatted printed output
ios_	the complete Multics I/O system
ios_$read_ptr ios_$write_ptr	two special high speed entry points for character string I/O to and from the typewriter terminal
nstd_	nonstandard tape device interface module
plot_	produces two-dimensional graph for a display
read_list_	reads and converts free format typed input
syn	makes one stream name equivalent to another
tape_	Multics standard tape interface module
tw_	typewriter device interface module
user_info_$tty_data	returns information about the current terminal device
write_list_	automatically converts and formats output variables

See also the MPM Reference Guide section, Input and Output Facilities.

9) Error Handling Facilities

active_fnc_err_	handles errors encountered by active functions
com_err_	prints a standard status message for common errors
command_query_	handles questions generated by commands
condition_	establishes a procedure to handle a named condition
find_condition_info_	returns information about a condition
reversion_	discards a condition handling procedure
signal_	calls the handler of a named condition

unpack_system_code_	converts packed status code into standard status code

See also the MPM Reference Guide section, List of System Status Codes and Meanings.

10) Routines to Convert Some Data Type To and From a Character String Representation

convert_binary_integer_	converts binary integer to ASCII string
convert_date_to_binary_	converts ASCII string to binary clock reading
cv_bin_	converts binary integer to ASCII string
cv_dec_	converts ASCII decimal string to binary integer
cv_float_	converts ASCII floating point string to binary real
cv_oct_	converts ASCII octal string to binary integer
date_time_	convert clock reading to ASCII string

Note: some data type conversion can also be performed within the PL/I language.

See also 4) Storage System, Supervisor Entries for Manipulating an Address Space, for conversion from character string name to pointers in the address space.

11) Object Segment Manipulation Routines

adjust_bit_count_	sets bit count of a segment to last nonzero character
make_object_map_	used by a compiler to put finishing touches on a newly created object segment
object_info	returns structured information about an object segment
stu_	(Symbol Table Utility) used to retrieve information from a procedure symbol table

12) Miscellaneous Procedures

decipher_	decodes an encoded array of words
encipher_	encodes an array of words

get_process_id_	returns identification of current process
hcs_$block hcs_$wakeup ipc_	interprocess communication
parse_file_	parses ASCII text into tokens
random_	random number generator
unique_bits_	returns a bit string different from all other such strings
unique_chars_	converts a unique bit string to a unique character string which contains no vowels
user_info_	returns miscellaneous information about the current user

PROGRAMMING LANGUAGES

A number of programming languages are available on Multics: PL/I, FORTRAN, BASIC, ALM, APL, and LISP. ALM is an assembler, APL is an interactive interpreter, LISP is both an interactive interpreter and compiler, and the rest are compilers.

BASIC on Multics is the Dartmouth College BASIC language and the current compiler is a copy of the one developed by Dartmouth College. Multics PL/I is the proposed ANSI standard PL/I, Multics FORTRAN is the ANSI standard FORTRAN, and Multics APL is identical to the IBM APL. Multics PL/I uses the full ASCII character set.

PL/I can be regarded as the standard compiler language on Multics. The calling sequences, argument declarations, and standard data types in terms of which the system is documented are taken from PL/I. This is because the system itself is written largely in PL/I. In the same sense, ALM is the standard system assembler since areas of the system requiring assembly-language coding are written in ALM.

Users of other languages, however, should encounter no difficulties because they choose to work in another language. Each Multics translator is callable as a command and produces object code segments which are also callable as commands. A program written in one of the Multics languages can call other programs written in the same language by merely following that language's calling conventions. A more troublesome point is the ability to call programs written in another language since in some cases one language does not contain the mechanics required to construct calls or arguments to programs not written in the same language. (See Common Features below, number 6.)

Common Features

1) A Multics compiler or assembler is invoked by a command line of the form

```
language_name source_name
```

and takes as its source code a segment named "source_name.language_name". For example, the command line

```
pl1 square_root
```

would invoke the PL/I compiler to compile the source code contained in the segment square_root.pl1 in the user's working directory.

2) A source code segment must be prepared prior to invoking the translator. This is done using a Multics editor such as edm or qedx. (See the MPM write-ups for the edm and qedx commands.) It is at this point that the source code segment is given the appropriate name as described in 1) above. It can include instructions to insert the source code contained in other segments (known as include files) into this source segment at compilation time.

3) A Multics compiler or assembler produces an object segment in the user's working directory. The object segment contains the object code produced by translation of the source code as well as Multics standard linkage and symbol table information. The object segment has the same name as the source segment, minus the language_name component. Continuing the example from 1) above, the object segment square_root would be produced.

4) The production of compilation and assembly listings is at the user's option. If listings are produced they are put in the working directory with a name of source_name.list; e.g., square_root.list.

5) The user can control the verbosity of error messages printed on his terminal during a translation. See the MPM write-up for the translator to be used.

6) In general, programs coded in one Multics language are able to call programs coded in another Multics language. Any exceptions or restrictions (such as the types of arguments which may be passed) are noted in the documentation of the various languages.

7) Each Multics language has an MPM write-up describing the command which invokes the translator. It describes how the command is used, including the use of optional features such as production of listings and verbosity of error messages. This document refers, when necessary, to a complete language description usually found outside the MPM. The command write-up itself notes any differences between the standard description and the current version.

 (END)

SYSTEM PROGRAMMING STANDARDS

This section outlines many of the design and coding standards followed by Multics system programs. It is provided to give users some insights into what is considered to be good programming practice on Multics. The information presented below represents the accumulation of several years of experience in programming on Multics. It is hoped that it will aid users in their own programming efforts. As will be obvious, some of the standards apply only to modules of the system itself. On the other hand, those standards may suggest analogous procedures which would be appliable to other programming projects.

Coding Standards

1) All system subroutines must be pure, so that a single copy may be shared by all users. The Multics PL/I and FORTRAN compilers produce only pure subroutines.

2) All system subroutines must be written in the PL/I language. Explicit permission of the project management is required to use any other language. To aid others in understanding a program, the program listing should be well commented. This includes explaining the meaning of important variables.

3) Only subroutines documented as part of the Multics system (not including tools and the author-maintained library) may be called.

4) The names of all system programs that are not commands or active functions must end with an underscore (_). The names of all temporary segments and all I/O streams and condition names (other than PL/I defined condition names) used by system modules must also end in an underscore. This is to avoid naming conflicts with the user.

5) All variables used, including called subroutines, must be declared. This is done to increase program readability and reduce the confusion introduced by default or implicit declarations. For called subroutines, the parameter list must be fully declared, unless, of course, the subroutine accepts a variable number of arguments (e.g., a free format output subroutine). For readability, declarations should be collected together in a logical way (e.g., at the beginning of the subroutine or block for which they apply, or at the end) rather than being scattered throughout the program.

6) The use of pointers as arguments should be avoided when practical. Passing a data item as an argument rather than a

pointer to that item makes a program less error prone since the compiler can make checks for argument mismatch and since it is sometimes possible to perform run-time argument validation.

7) Special characters should be placed in the program directly. To lessen dependencies on the character code being used, the built-in function unspec should not be used for this purpose. For example,

```
declare nl (char(1) initial ("
");
```

declares "nl" to be a one-character string whose value is the new line character. The statement

```
unspec(nl) ="000001010"b;
```

should not be used.

8) Use of implicit conversion from one data type to another is prohibited, since it makes a program harder to understand. For example,

```
declare x fixed bin(18), y bit(18);

y=x;
```

should not be used. Instead one should write

```
y=bit(x,18);
```

9) Use of external static variables which do not contain a dollar sign (e.g., declare x external static) is prohibited since this data type is not efficiently implemented in the current Multics environment. External references of the form a$b are allowed. If the programmer needs to have an external data base which is shared among many subroutines, he may either create a segment by an appropriate storage system call and reference it using based structures or use the assembler to create a data segment by appropriate use of the segdef pseudooperation. The programmer wishing to do this should consult with a knowledgeable member of the Multics Development Group.

10) All variables should be of the automatic storage class unless there is a good reason for them to be internal static; i.e., they are static by nature. See also rule 11 below.

11) In PL/I programs, to avoid having to initialize variables whose values are constant every time the subroutine containing them is entered, and to avoid having copies of these variables made for every user of the subroutine, one should use internal static and initialize the variables using the initial attribute. The PL/I compiler will allocate space for these variables in the text section of the subroutine being compiled and will initialize them. Since the text section is pure, one copy of these variables will be used by all users of the subroutine. Unfortunately, if a variable of this type is passed as a argument to another subroutine, the compiler has no way of knowing whether or not that variable is to be changed by that subroutine and it, therefore, puts the variable into the linkage section. Therefore, if one has a large number of "constant" variables that are also passed as parameters, one should put them in the text portion of an assembly language program and initialize them using the appropriate data generating pseudooperations and reference them using either based structures or the "a$b" notation. This will assure that only one copy of these variables is used by all users of the subroutine. The programmer wishing more clarification of this point should consult with a knowledgeable member of the Multics Development Group.

12) Use of the PL/I allocate and free statements should be cleared in advance with project management, since there often exist more efficient ways to accomplish the same task. Subroutines that do perform allocations (or call subroutines which do) must establish a cleanup procedure to free the storage in the event that processing is aborted.

13) When possible, the PL/I on, revert and signal statements should be used instead of the condition_, reversion_ and signal_ subroutines since they are more efficient and make the program less system dependent.

14) The programmer should avoid writing PL/I functions with multiple entry points which return different data types unless there is a good reason to do so, since this generates extra code at each return statement.

Programming Style

1) The most common route through a program should be the most efficient. More exotic facilities which are inherently expensive should be separated from the simple facilities so that a casual user need not pay for the exotic each time he uses the simple.

2) System programs should, in general, use one of the three standard I/O streams: user_input, user_output, and error_output. Only special I/O service programs should issue I/O attach or detach calls for these streams. Commands should not, in general, provide optional off-line output. The file_output command is provided for this purpose.

3) All programs that are not commands or active functions should return a status code indicating successful completion or occurrence of an unexpected event, unless they are programs for which errors are unrecoverable or extremely rare; e.g., console output subroutines. This type of program should make use of the Multics signalling facility to signal that one-in-a-million error. In general, because of the higher overhead involved, programs should not make use of the Multics signalling facility for routine errors and status conditions. Subroutines which are directly called by the user must return only standard error_table_ codes. See the MPM Reference Guide section, Strategies for Handling Unusual Occurrences.

4) In most cases, programs that are not commands or active functions should not print error messages, but should allow a higher level subroutine to decide on the seriousness of errors and what to do about them. In general, it is wise to let the most qualified subroutine give the message. A good rule of thumb for determining the most qualified subroutine is to ask whether anything could be learned by reflecting the error to a higher level subroutine. If the answer is no, then the most qualified subroutine has been found.

5) All programs that are not commands, active functions or gates into a ring should assume they are called with the correct number and type of arguments and should not make checks. This is to avoid continually paying the cost of argument checking in programs which call the subroutines correctly. This does mean that the programmer must be careful to call subroutines correctly.

6) System programs should be prepared to execute properly even if they did not complete execution during a previous invocation

because of a quit or a fault. That is, they should either operate normally or warn the user of the consequences of continuing. For example, edm warns the user that, if he continues, the partically completed results of an earlier invocation will be lost.

7) System programs should never call a command if there is a subroutine which does almost the same thing. Commands are inherently more expensive since they are designed to interact directly with a human user.

8) System programs should not use a subroutine to do something which can be done reasonably easily in a few PL/I statements. The purpose of this rule is to avoid the proliferation of unnecessary system subroutines. The exceptions to this rule are input/output (see paragraph 1 under Error Handling and I/O below) and conversion from character to numeric data types. The reason for the latter exception is that this type of conversion is inherently more expensive than calling a specialized subroutine.

9) Calls to subroutines which require descriptors should be minimized when this does not conflict with program readability or degrade the user interface. This is because of the higher overhead involved in setting up argument lists with descriptors. For example, one should try to minimize the number of ioa_ calls in a program. This should not be interpreted to mean that one should remove all error messages from his program or make their output so terse as to be unreadable. It simply means that if, subject to the constraints mentioned above, it is possible to use one ioa_ call rather than two then the programmer should to so.

Data Base Management

Designing a program for a virtual memory environment requires a new outlook on program and data organization. Though the programmer is freed from the onerous task of allocating physical storage for his programs and data (e.g., storing intermediate results on secondary storage, overlaying parts of his programs with others to fit into core memory, etc.) he cannot ignore the issues of data management and program organization if he wants his program to be reasonably efficient. This is especially true for programs which manipulate large amounts of data. The attitude that an infinite virtual memory is available and if a program needs more room it can create another segment, may be all right for the casual user building a one-shot program

but not for the systems programmer. A major aim of the programmer should be to minimize the working set of his programs; i.e., his programs should create as few segments as is practical, reuse the ones they do create and should avoid unnecessary moving of data. The term working set is used loosely here to denote both the number of segments and the number of pages in the execution path of a program. In Multics it generally pays to spend CPU time (within reason) to save space. This principle should not, of course, be taken to an extreme. It does not mean, for instance, that one should not use a hash table. It is true that a hash table takes up more space than an equivalent linear list but a program will take fewer page faults referencing the former than searching the latter. In this case, the actual working set of the former is smaller even though its potential working set is larger. In all cases, the programmer must exercise his judgement as to the proper tradeoff between working set size and CPU usage, always avoiding the temptation to allow his working set to expand to infinity.

In addition to this basic principle, the following guidelines apply:

1) System programs must leave their data bases in a consistent state; e.g., a program which changes the contents of a segment should reset the bit count of that segment when it is finished. Programs should make any period of inconsistency as short as possible. They must also clean up after themselves; e.g., free storage should be released.

2) In order to assure consistent behavior, all standard translators must use the subroutine tssi_ to interface with the storage system. It might not make sense for nonstandard translators such as BASIC to use tssi_. Exceptions of this sort should be cleared in advance with the project management.

3) System programs should initiate the segments they access by a null reference name and should subsequently access those segments via a pointer. In general, segments initiated by a module should be terminated by that module (see point 4 below).

4) In general, the process directory should be used to hold temporary segments. If a program is not being entered recursively it should create temporary segments with intelligible names (e.g., containing the name of the creating program). It should clean up after itself before exiting by either truncating or deleting these temporaries. If the temporary segment can be reused the next time the program is

invoked it should be truncated; otherwise, it should be deleted. If a program is being entered recursively (e.g., one quits out of a command, issues a hold command, and reenters that command), it should create temporary segments whose names consist of a unique first component followed by one or more intelligible components. These segments should be deleted when the program exits. If, for some reason, a program cannot be made recursive it should detect the fact that it is being entered recursively, warn the user that partially completed work of an earlier invocation will be lost if he continues, then give him the option of continuing or exiting. Programs which create temporary segments should establish cleanup procedures to truncate or delete these segments if execution is abnormally terminated. As mentioned above, the names of temporary segments must end in an underscore.

5) Any system program which creates new segments (other than temporary segments) should put them into the user's current working directory unless the program explicitly makes provision for the user to provide a target directory. (The move and copy commands fall into this latter category.) The aim of this rule is to avoid messing up another directory, such as the directory from which a source segment was obtained.

6) System programs which create new segments must set access control lists according to the conventions enumerated below. If a segment is being replaced instead of being newly created, the command must leave the access control list as it was before the command acted. For instance, a translator finds that an object segment already exists with read and execute access for this user, and with other access for other users. The translator must obviously add write access to change the segment contents, but should restore the entire access control list to its former value when the translation is completed. The storage system interface subroutine tssi_ does this automatically for the translator writer. The access to be given to the user creating a segment is:

Segment Type	Access	Ring Brackets
directory segment	SAM	v,v
object segment	RE	v,v,v
data segment	RW	v,v,v

where v is the current validation level of the user. See the MPM Subsystem Writers' Guide section, Intraprocess Access

Control (Rings), for a discussion of validation level.

Additional Standards for Commands and Subsystems

Through the mechanism of the command processor any program -- system subroutine, system command, user subroutine -- can be invoked from the console. System commands are a special class of subroutines that are explicitly programmed with the console user in mind. They must check carefully for argument validity; they must warn the user of possible misunderstandings; they must be very reliable. They must, to the greatest possible extent, be a self-consistent set; i.e., the behavior of a command should be predictable from that of other commands.

For these reasons a number of additional standards are necessary for system commands and subsystems.

Naming Conventions

1) For ease of typing, all commands must have an abbreviated name consisting of the first letter of the first two or three syllables or first two or three words of its name (e.g., rename rn, unlink ul, print_attach_table pat).

2) All command names and abbreviations must be cleared in advance with the project management.

Programming Style and User Interface

1) If a command would also be useful as a subroutine, break it apart into a command which interfaces with the user (processes multiple arguments, handles the star and equals conventions, interprets control arguments, etc.) and a subroutine which does the work. This subroutine, like all subroutines, should return a status code rather than printing an error message. The outputing of error messages like all other user interface problems should be handled by the command.

2) Any command for which the star convention makes sense should use the star convention. Any command for which the equals convention makes sense should use the equals convention. See the MPM Reference Guide section, Contructing and Interpreting Names for a discussion of the star and equals conventions.

3) Characters which have special meanings to commands (e.g., "*", "=", ">" "<") should not be used in any context other than their standard one. For example, a command should not

interpret an argument of "*" as meaning that user wants to be logged out.

4) Commands should not be too powerful, that is, typing errors should not cause disastrous results. For example, with the old remove command

```
remove a>b
```

would delete the segment b in directory a, whereas

```
remove a> b
```

(i.e., one accidentally types a space before the b) would delete the directory a. To remedy this, there are now two commands: delete which deletes only nondirectory branches, and deletedir which deletes only directory branches.

5) Unless the purpose of a command is to produce some sort of output, it should produce no output during normal operation; i.e., it does not need to tell the user that it is doing its job. For example, if one enters the command

```
delete x y
```

the delete command produces output only if it has trouble deleting x or y. It does not type "deleting segment x", "deleting segment y". Commands which take a long time to execute (e.g., pl1) should print a short message when they are entered to indicated they are functioning. The general idea here is to reassure the user that he has not done something wrong. After more than a couple of seconds wait, the user, particularly a novice user, begins to worry that perhaps the computer is waiting for him.

6) Commands which take segment names as arguments should accept pathnames, not reference names, unless they explicitly deal with reference names (e.g., terminate_refname). The user who has a reference name he wishes to pass to a command may use the get_pathname active function to convert this reference name to a pathname (e.g.,

```
status [get_pathname x]
```

will cause the status command to be called with the pathname of the segment whose reference name is x). See the MPM Reference Guide section, Constructing and Interpreting Names

for a discussion of reference names.

7) Commands which interact with the typist should be prepared to handle the program_interrupt condition which is signalled by the program_interrupt command. Handling this condition correctly is quite tricky. See the MPM Reference Guide section, List of System Conditions and Default Handlers for details.

8) When a command which interacts with the typist produces an error message which the typist may not have expected, the command should normally follow the error message with a call to ios_$resetread (which discards all input read but not yet used) on the I/O stream from which it reads input so that the typist can modify his subsequent input.

9) We come now to a standard that is difficult to express with any degree of exactness. The phrase "commands should be designed with the user in mind" expresses the spirit of the standard. What follows is a series of examples designed to sensitize the reader to some of the issues involved in designing a command. Calling sequences should be logical (e.g., the user should not have to remember that % as a third argument to the xyz command causes all segments with a second component name fred to be deleted, whereas a ? in the same position suppresses this feature). Commands should allow the user to decide whether a protected segment should be deleted, rather than forcing him to make the segment deletable and to resubmit the delete request (or worse, delete the segment without warning). Judicious use of red console output is encouraged. It should be used to call attention to important or unusual occurrences. Remember, over-use destroys the whole purpose of red output -- a command which outputs everything in red may as well output everything in black. Canned messages printed by commands should not contain characters which come out as escape characters on IBM model 1050 and model 2741 consoles and on model 37 teletypes (e.g., "¢<segment¢> not found" is not an acceptable message).

Argument Handling

1) Commands, wherever possible, must accept path names (not just entry names) as arguments. The subroutine expand_path_ should be called to convert a relative path name into an absolute path name.

2) Commands which deal with segments whose names have a fixed suffix should not force the user to type that suffix.

Rather, they should append that suffix to their arguments if it is not given. For example, the command lines

```
pl1 x
```

and

```
pl1 x.pl1
```

should be equivalent.

3) Commands whose interface is simple (such as the delete and addname commands) should accept multiple arguments if it makes sense to do so.

4) All commands which accept a variable number of arguments should declare themselves as having no arguments (i.e., command_name: proc;) and should obtain their arguments using the procedure cu_$arg_ptr.

5) Commands must obey Multics control argument conventions as described in the MPM Reference Guide section, List of Command Control Arguments.

6) In general, for the convenience of the user, command arguments should be order independent unless the order dependency serves a useful purpose (as in the -ag control argument of the enter_abs_request command).

Error Handling and I/O

1) The input/output facilities of the PL/I language must not be used in system programs since they are more expensive than system-provided subroutines.

2) To read a line from the input stream user_input, use the subroutine ios_$read_ptr. To read a line with appropriate data type conversion (i.e., the user is typing in pointers, floating point numbers, etc.) use the subroutine read_list_.

3) Output lines fall into three distinct classes:

   a) unusual status messages
   b) questions
   c) everything else

Lines of type a) should be output using the subroutines com_err_ and active_fnc_err_ (active functions should use active_fnc_err_, all other modules should use com_err_). Lines of type b) should use the subroutine command_query_. These three subroutines are provided in order to centralize the processing of lines of type a) and b) so that changes in system conventions in this area may easily be made. For lines of type c) the subroutine ios_ should be used when it is necessary to format an output line; otherwise, use the subroutine ios_$write_ptr.

4) Commands should check for status codes which have special meaning to them and either print appropriate error messages or, if the error is easily recoverable, allow for user intervention using command_query_. All such messages must contain the name of the command which generated them, since otherwise the user would have no way of knowing which command generated a given message if he has issued several at once or was running an exec_com segment. Complex programs such as compilers may output diagnostics by standard output subroutines but should have at least one call to com_err_ to notify the system that an error has occurred.

 (END)*

CLOCK SERVICES

Two types of clocks are available on Multics: a real-time clock for the entire system and a process execution meter for each process. The real-time clock, a hardware calendar clock accessible via a special register on a memory controller, runs whenever the system is in operation. It contains a double-word integer that is incremented once per microsecond and represents the number of microseconds elapsed since January 1, 1901, 0000 hrs. GMT. An interrupt mechanism is associated with the calender clock so that a specified process can receive an interprocess wakeup when the calendar clock reaches a specified clock value. The specified clock value is regularly compared with the calendar clock value, and when the two are equal an interprocess wakeup is generated for the appropriate process.

A process execution meter is maintained as part of the state of each process. It counts the microseconds during which the process is running. This meter is used to record each process's usage of system resources as well as being available to the user to cause wakeups after a specified amount of time has passed. There are actually two such meters for each process; the more valuable one measures virtual time, which is the time during which the process is running decremented by the time it handles interrupts and page faults.

An interrupt mechanism associated with the virtual time meter allows a process to receive an interprocess wakeup when the meter is incremented beyond a specified value. This meter is compared to the specified value at regular intervals and when the value is exceeded an interprocess wakeup is generated for the running process.

The uses to which clocks are put are diverse. There are a number of general uses for which they can be used. Included are the following:

1) Resource monitoring and accounting;

2) Labeling data (e.g., storage system entries) with dates and times of interest;

3) Computing the date and time for output;

4) Generating a unique bit string;

5) Waking up a specified process at a specified time, perhaps causing a specified procedure to be called;

6) Interrupting a process after a specified amount of CPU time has elapsed.

Access to System Clocks

A number of commands and subroutines (all described in MPM write-ups) permit the user to inspect the real-time clock and the process execution meter. The clock_ subroutine reads the real-time clock and returns its current value as a fixed bin(71) quantity. This clock time can be converted to a more understandable form using either date_time_ (which returns a single character string) or decode_clock_value_ (which returns the various components of the time -- month, year, etc. -- as distinct variables. The convert_date_to_binary_ subroutine accepts a character string like that produced by date_time_ and returns a fixed bin(71) equivalent.

The value of the process execution meter is returned by both cpu_time_and_paging_ and total_cpu_time_. The resource_usage command prints a report of the resources used by the user from the beginning of the current month to the time of creation of the user's current process.

The status command and the hcs_$status_ subroutine both provide dates and times associated with storage system entries, such as the date and time modified and the date and time last used.

The unique_bits_ subroutine returns a bit string generated partly from the current real-time clock reading that is guaranteed to be unique among all bit strings so generated. The unique_chars_ subroutine converts this value into a character string that is also guaranteed to be unique among all character strings so generated.

Facilities For Timed Wakeups

The interprocess communication facility (described in the MPM Subsystem Writers' Guide under ipc_) allows a user to set up channels for sending interrupts (wakeups) to a specified process. The interrupt can merely cause that process to return from the blocked state to whatever it was previously doing, or can cause same other procedure to be called in that process as a result. One possible use of this facility is to wake up a process as the result of some clock activity. The timer_manager_ subroutine (described in an MPM subroutine write-up) provides the necessary interface. Using it, a user can specify an event channel for his own or another process, whether the process should merely be

wakened or a specified procedure should be called, and the nature of the clock activity that triggers the wakeup (i.e., real or CPU time). In specifying the time, the user can further specify absolute or relative time, and can use seconds or microseconds.

 (END)

THE STORAGE SYSTEM DIRECTORY HIERARCHY

The Multics storage system is used for supervisor segments as well as for user segments. Since a single directory hierarchy is used for both, it is frequently useful to know the relative arrangement of items within the hierarchy. This section outlines briefly the organization and contents of the directory hierarchy.

Generally, the user should never embed pathnames of supervisor segments in his programs. Most often, he will use these segments indirectly, by invoking some library procedure or command that knows both the pathnames and the format of the data stored there. The details provided here are intended as background information for understanding and, occasionally, debugging. The structure described here changes fairly often, so it should not be assumed when writing programs.

Figure 1 is a diagram of the top of the directory hierarchy. The name of the root directory is, by convention, omitted from pathnames. The diagram shows seven directories always found in the root. (Only the basic structure assumed by the Multics supervisor is illustrated. Additional segments and directories can also appear at all levels in the directory hierarchy.)

1) system_control_dir

   This directory is the storage location for most system accounting, authorization, and logging information. The table printed by the who command, the message of the day, and the absentee queue segments are the only generally accessible segments in this directory. Project administration tables are stored in a directory tree that starts in system_control_dir.

2) process_dir_dir

   This directory contains one directory for every process currently in the system. The name of an individual process directory is derived from the unique identification of the process. The process directory is used as a place to store all segments that are intended to have a lifetime no greater than that of the process that creates or uses them. Thus, if a compiler needs a scratch area, it can create a segment here. A program wishing to create or use a segment in the process directory does not normally need to know or construct the full pathname of the directory, since the most common means of creating a segment uses the process directory by default.

Five segments are normally placed in a process directory upon creation of the process. Others can be added by various commands or user programs. The five initial segments are:

pds	(Process Data Segment) A supervisor data base containing the state of the process with respect to the supervisor. (This segment is accessible only to the supervisor.)
kst	(Known Segment Table) A supervisor data base detailing the correspondence between segment numbers and segment names, as known in this process. (This segment is accessible only to the supervisor.)
pit	(Process Initialization Table) A driving table used to contain details on how the process should initialize itself. The name of the initial program to be called in this process is found in the pit.
stack_4	This segment is the stack used for PL/I automatic variables and for subroutine call and return operations. There is one stack segment for each active ring; the last character of the stack name is the ring number except in ring 0, where pds is used as a stack.
combined_linkage_4.00	This segment, managed by the linker, contains interprocedure links and PL/I internal static storage. If the total requirements for linkage and static storage of a process exceed the space available in a segment, additional segments are automatically created using the same name; the last two characters are a sequence number. In addition, each active ring except ring 0 has its own linkage; the character before the period in the segment name indicates the ring number.

Other segments commonly found in the process directory include the free storage area used to implement PL/I allocate and free statements, temporary storage areas of the editor commands, and segments used to contain the parse tree of the PL/I compiler. These segments are created only when needed by the various commands and subroutines using them.

3) daemon_dir_dir

This directory contains segments and directories used to support the various system daemon processes, such as automatic file backup and bulk (card and printer) input and output. Except for the queues of the I/O facilities, the contents of daemon_dir_dir are not generally accessible to users.

4) user_dir_dir

This directory is the base of a tree containing all of the personal segments of individual users. The immediate contents of user_dir_dir is a set of directories, one for each project that uses Multics. Contained in a project directory is usually one personal directory for each user working on that project.

5) system_library_standard

This directory contains the library of commands and subroutines provided as part of Multics. These procedures are documented in the Commands and Subroutine Calls sections of the MPM Reference Guide. Unless the user specifies otherwise, this directory is included in the list of directories to be searched when dynamic linking occurs.

6) system_library_languages

This directory contains the processors for all the programming languages supported on multics.

7) system_library_auth_maint

This directory is similar to system_library_standard except that it contains commands and subroutines provided by programmers of the local installation. It is distinct from system_library_standard so that it can be left out of the search path if desired.

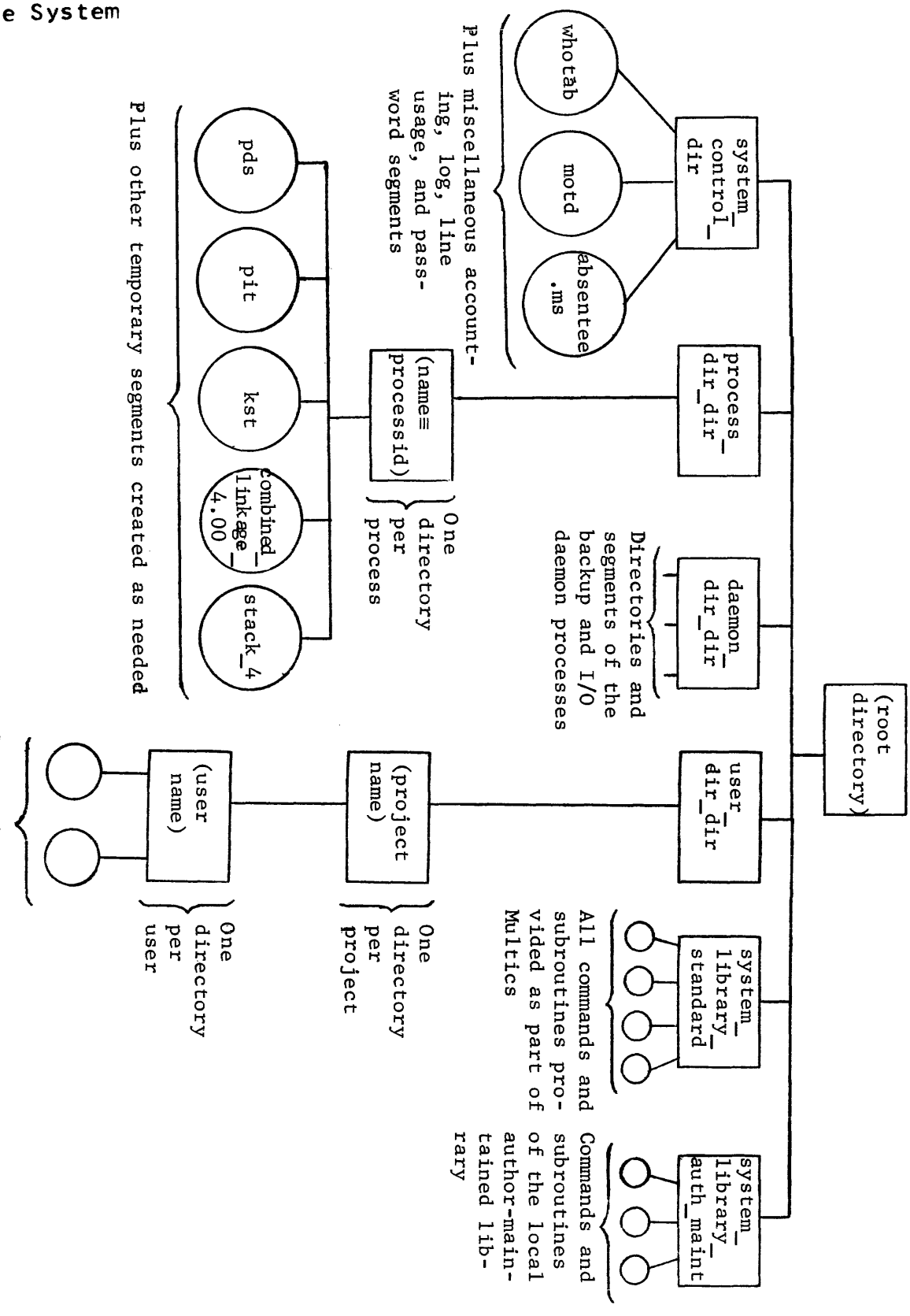


Figure 1: The Multics Directory Hierarchy

 (END)*

THE SYSTEM LIBRARIES AND SEARCH RULES

External references to procedures and data segments are bound at execution time by the dynamic linker. (External references include both subroutine calls and use of external static variables of the form alpha$beta.) The linker uses a standard set of rules in searching for the target of an external reference. The default search rules are given below. If appropriate, the user can specify his own set of search rules. See the write-ups of the set_search_dirs and the set_search_rules commands.

If neither of those commands have been used, the search for a segment with the same name as the external reference name proceeds as follows:

1) already initiated segments

   This is a list of names that have been previously referred to in this process. A reference name is associated with a segment by:

   a) use in a dynamically linked external reference from a program;

   b) a call (which may be made by the initiate command) to hcs_$initiate, hcs_$initiate_count or hcs_$make_seg with the "rname" argument a non-null string;

   c) a call to hcs_$make_ptr.

2) referencing directory

   This is the directory in which the calling or referencing procedure is contained as a segment (not a link).

3) working directory

   This is the directory that has been specified as the user's current working directory. The working directory can be changed with the change_wdir command or the change_wdir_ subroutine. The initial working directory is the user's home directory. (See the MPM write-up for user_info_.)

4) >system_library_standard

   This library contains the Standard Service System modules, the Development System modules, and the PL/1 System modules. It contains most system commands and subroutines.

5) >system_library_1

This library contains a small set of commands and subroutines that are reloaded every time the system is reinitialized. These include hcs_, ios_, cu_, clock_, com_err_, and error_table_. From the user's point of view, there is no reason for considering this library to be anything but a continuation of system_library_standard.

6) >system_library_languages

This library contains the language processors for the programming languages supported on Multics.

7) >system_library_auth_maint

This library contains the author-maintained and installation-maintained libraries. The author-maintained library consists of a collection of procedures contributed by users at a particular installation. It is maintained for the convenience of the local user community, as an aid in sharing of programs. Users of author-maintained procedures should be aware of two dangers:

a) there may have been little or no verification of the effectiveness or accuracy of the procedures;

b) no guarantee is made that the procedures will continue to be maintained as the system changes. Users of author-maintained language translators should be especially wary of this second danger.

The installation-maintained library contains procedures installed and maintained by the local installation. It differs from the author-maintained library in that verification of accuracy and effectiveness of the procedures has been performed by the installation, and the installation is committed to maintain the procedures.

Include Files

Several of the languages on Multics permit the inclusion of the contents of a separate segment into a specified place in a source segment at translation time. These separate segments are known as include files. The translator searches for include files as follows:

1) working directory

   See working directory (number 3 above) for a description of this directory.

2) >user_dir_dir>project_id>include

   This directory can be maintained if needed by a project for use by its members. The directory project_id in the path name is the name of the user's project.

3) >library_dir_dir>include

   This library of source code contains all include files used in system programs.

 (END)

SEGMENT, DIRECTORY AND LINK ATTRIBUTES

Every directory in the Multics storage system may contain three types of entries: segments, directories, and links. Each of these entries includes two types of information: contents (or data) and attributes. It is the contents of an entry that is the primary purpose for its extence. For a segment, the contents are the data of the segment. For a directory, the contents are the entries of the directory. For a link, the contents are the path name of the entry to which the link refers. The attributes of an entry contain information about the contents of the entry.

Each entry has a specific set of attributes. A list of the different types of attributes that may be associated with each type of entry is given below. Each member of the list contains the name of the attribute, a list of the types of entries that have this attribute, a description of the attribute, and a statement of whether or not the attribute may be modified. An attribute is explicitly modifiable if there is a storage system subroutine that will explicitly change the value of the attribute. An attribute is implicitly modifiable if something can be done to the entry which will change the value of the attribute. An example of the latter is changing the date/time modified of a segment by writing into a segment. A description of the type of access modes required to modify attributes is given in the MPM Reference Guide section Access Control.

access control list (segments, directories)

The access control list (ACL) specifies if and how different processes may access a segment or directory. ACLs are described in detail in the MPM Reference Guide section Access Control. An ACL may be explicitly modified.

author (segments, directories, links)

The author of an entry is the access identifier of the process that created the entry. The author is not modifiable.

bit count (segments)

The bit count of a segment specifies the length of the contents of the segment in units of bit, i.e., it delimits the last meaningful bit of information in the segment. The bit count is explicitly modifiable and since it is not maintained by the supervisor, should be maintained by all procedures that modify the length of the contents of the segment. Many system commands and subroutines will not

function properly unless the bit count is accurate. The bit count is maintained by system procedures that change the length of segments. The bit count is necessary because the system is unable to automatically maintain the length of a segment in units finer than the page size (this measure of length is called the current length, see below). In order to modify the bit count it is necessary only that the process have write access to the segment at the validation level, unlike all other segment attributes which require modify access in the directory containing the segment. This is to insure that any process that can modify the length of the segment can also modify its bit count.

copy switch (segments)

The copy switch provides a means by which many processes can simultaneously execute impure procedures. When a segment with the copy switch on is symbolically referenced, either by resolving a symbolic link reference or by initiating the segment, a pointer to a copy of the segment for this process is returned instead of a pointer to the segment itself. In this way each process will be using its own copy of the segment and will be able to modify the segment without affecting other processes. In the current implementation a new copy is generated each time a link is resolved to the segment by a different name or each time the segment is initiated by a different reference name. To avoid confusion, it is therefore recommended that a segment with the copy switch on have only a single name. The copy switch may be explicitly modified.

current length (segments, directories)

The current length specifies the length of the contents of the segment or directory, i.e., it delimits the last non-zero bit of data. It is accurate to units of page size. The current length is implicitly modified by storing data beyond the previous current length or by truncating the segment or directory.

date/time dumped (segments, directories, links)

This attribute specifies the last time a backup copy of this segment were made by the Multics backup procedures. The date/time dumped is implicitly updated by the Multics backup procedures when an entry is dumped.

date/time entry modified (segments, directories, links)

This attribute specifies the time any of the attributes of this entry were last modified. The date/time entry modified is implicitly updated when an entry is modified.

date/time modified (segments, directories)

This attribute specifies the time the contents of the entry were last modified. The date/time modified is implicitly updated when the data of an entry is modified. (For implementation reasons the date/time modified may not be precisely accurate but will normally be less than a few minutes after the correct time.)

date/time salvaged (directories)

This attribute specifies the time the directory last had to be salvaged. Salvaging implies that the directory had to be modified in order to eliminate an inconsistency in its contents so that the storage system can function properly. The date/time salvaged is implicitly modified when a directory is salvaged.

date/time used (segments, directories, links)

This attribute specifies the last time the contents of the entry were referenced. (For implementation reasons the date/time used may not be precisely accurate but will normally be less than a few minutes after the correct time.)

initial access control lists (directories)

The initial ACL specifies the default value for the ACL of a segment or directory newly created in the associated directory. A detailed description of initial ACLs is given in the MPM Reference Guide section Access Control. The initial ACLs may be explicitly modified. Only modify access in the directory at the validation level is necessary to modify an initial ACL. Unlike most directory attributes, no access on the superior directory is necessary.

maximum length (segments)

This attribute specifies the maximum length of the segment, i.e., the size beyond which the segment may not grow. The maximum length is accurate to units of page size. The maximum length may be explicitly modified.

multi-segment file indicator (directories)

A non-zero value for this attribute indicates that this directory is actually a multi-segment file. The value of the attribute is the number which is the entry name of the last segment in the multi-segment file. The multi-segment file indicator is implicitly modified by the multi-segment file primitives when the length of the file changes. It may also be explicitly modified.

names (segments, directories, links)

The names of an entry are the character strings used to identify and reference the entry. Each entry may have many names. The first name on an entry is termed the primary name and will always be listed first until it is deleted. Entry names may be explicitly modified.

quota (directories)

The quota of a directory is the number of records of storage that segments and directories inferior to this directory may occupy, excluding those subtrees that have their own quota. The quota may be explicitly modified. Only modify access in the directory at the validation level is necessary to modify the quota. Unlike most directory attributes, no access is needed in the containing directory.

records used (segments, directories)

The records used by a segment or directory is the number of records of secondary storage occupied by the contents of the segment or directory. The records used may be implicitly modified by changing the amount of storage occupied by the entry. This may be accomplished by changing its length.

ring brackets (segments, directories)

The ring brackets of a segment or directory partially specify access to that segment or directory. Ring brackets may be explicitly modified. A full description of ring brackts is found in the MPM Subsystem Writer's Supplement Reference Guide section Intraprocess Access Control (Rings).

safety switch (segments, directories)

If the safety switch of a segment or directory is on, that segment or directory can not be deleted. The safety switch may be explicitly modified.

secondary storage device identifier (segments, directories)

This attribute identifies the type of secondary storage device on which the contents of this segment or directory reside. The secondary storage device identifier is implicitly modified when the device upon which the contents of the entry reside is changed.

type (segments, directories, links)

The type of an entry indicates whether it is a segment, a directory, or a link. The type of an entry may not be modified.

unique identifier (segments, directories links)

Each entry in a Multics storage system hierarchy has a distinct unique identifier. This unique identifier may not be modified.

Warning

The information contained in this MPM section is not fully accurate until step 2 of directory reformatting is completed (probably about November, 1972). The following attributes will not exist until then: initial access control list, maximum length, safety switch, and (for directories) ring brackets.

 (END)

ACCESS CONTROL

Access control is the regulation of the right of a process (the active component of the system) to use or reference objects within the system. Examples of such objects are typewriters, printers, segments, and processes. This section discusses the regulation of the right of processes to use or reference certain objects within the Multics storage system, namely directories and segments.

This section is divided into two parts. The first part explains what rights may be granted or denied a process referencing a segment or directory. The second part describes how different access rights may be granted to different processes, i.e., interprocess access control.

A few sentences are in order about the use of this section. The access control mechanism represents an attempt to provide a general capabilty for controlling access in many different ways and yet keep the mechanism simple for common applications. This section is a comprehensive description of the full access control mechanism and most readers will find much if not all of the material of no interest to them. Users who do no sharing of segments, i.e. those who have segments which only they reference, need not know anything about access control because the system defaults automatically provide for this case. Even if the user makes use of programs of other users he need not know anything about access control because setting access is the responsibility of the other users. Only if the user wishes to share his segments with other users need he know anything about access control. In this case he should first read the MPM Introduction Chapter 3, Beginner's Guide to the Use of Multics. That chapter provides sufficient information about access control for most common applications. Only if that chapter is insufficient for the user's needs, should he then read this section.

Yet another facet of Access Control is described in the MPM Subsystem Writers' Supplement section Intraprocess Access Control (Rings). This part of access control differentiates between the access rights that a process may be granted in different states. It is called the ring mechanism, and is of use to the subsystem writer who wishes to write a protected subsystem.

Part 1: Access Modes

One does not simply want to regulate whether or not a process can reference a given object, but usually wants a finer control in order to regulate various ways in which a process may use an object. For different types of objects the means of

referencing may be different. For segments and directories these ways of referencing objects are termed modes of access or access modes. Since segments and directories are different types of objects, having different properties and different operations for referencing them, they have different modes.

Segment access modes determine the ways in which a process may reference the data of a segment. Directory access modes determine the ways in which a process may reference the attributes of directory entries. Each mode is labelled by a distinct, single character identifier that is used when specifying the mode to system commands.

The access modes for segments are:

execute (e) an executing procedure may transfer to this segment and words of this segment may then be interpreted as instructions and executed by a processor;

read (r) the process may execute instructions that cause data to be fetched (loaded) from the segment;

write (w) the process may execute instructions that cause data in the segment to be modified. *

The access modes for directories are:

status (s) the attributes of segments, directories and links contained in the directory and certain attributes of the directory itself may be obtained by the process (see the MPM Reference Guide section on Segment, Directory and Link Attributes for a definition of attributes);

modify (m) the attributes of existing segments, directories and links contained in the directory and certain attributes of the directory itself may be modified; and existing segments, directories, and links contained in the directory may be deleted;

append (a) new segments, directories and links may be created in the directory.

If a segment or directory is not accessible in any of the above modes then the process has no access to the segment.

* Until step 3 of directory reformatting has been completed (probably about February, 1973), the segment access mode append (a) should appear on segment ACLs that have write (w) access mode.

Part 2: Interprocess Access Control

In order to be able to grant different processes distinct access rights it is necessary to be able to distinguish different processes. For this purpose, each process has an associated access identifier. The access identifier is fixed for the life of the process. The identifier is a three component character string with the components separated by periods (.). The first component is the name of the person on whose behalf the process was created. The second component is the name of the project group of which the person named in the first component is a member. This person-project combination is termed a user. The same person may log into Multics under different projects and is considered to be two different users. The third component of the access identifier is the instance which is a single character used to distinguish different processes belonging to the same user. The access identifier must be less than 33 characters in length. The access identifier Jones.Faculty.a would be associated with a process created for Jones in the Faculty project. The "a" instance distinguishes the process from another process created for Jones.Faculty which might have an access identifier Jones.Faculty.b. All processes need not have distinct access identifiers. It is quite likely that several processes have the access identifier Jones.Faculty.a which simply means that all these processes have the same access rights to segments and directories in the storage system.

Access Control List

The rights that different process have when referencing a segment or directory are specified as an attribute of that segment or directory in the form of a list called the Access Control List (ACL). Each entry of the list specifies a set of processes (actually a set of access identifiers of processes) and the access modes that members of that set may use when referencing the segment or directory. The modes read, write, and execute may be specified in ACLs of segments and the modes status, modify, and append may be specified in ACLs of directories. On directory ACLs, modify mode may not appear without status mode. If some of these access modes are not granted in a ACL entry, then processes specified in the entry cannot access the segment or directory in the ungranted mode. For example, if the ACL of a segment contains an entry for a process and the modes specified are read and execute then the given process may execute instructions that fetch data from the segment, and transfer to and execute instructions in the segment, but it may not modify data in the segment.

The members of the set of processes associated with an ACL entry are specified by a character string called a process class identifier. The process class identifier is similar in appearance to an access identifier. In fact a string which is an access identifier may also be a process class identifier. Such a process class identifier identifies the class of processes whose access identifiers are the same as the process class identifier; e.g., the process class identifier Jones.Faculty.a identifies the class containing all processes with access identifier Jones.Faculty.a.

It is very useful to identify larger groups of processes than simply those with the same access identifier. This may be accomplished by replacing one or more of the three components of the process class identifier (i.e., the person name, project name, or instance) by the asterisk character (*). Such a character string identifies that class of processes whose access identifiers match the remaining components of the character string; i.e., those components of the string that are not the asterisk character. For example, the class identifier Jones.*.a identifies that class of processes with an access identifier containing Jones as the person identifier and "a" as the instance. Any project identifier in the access identifier will match. Therefore, processes with access identifiers Jones.Work.a, Jones.Lazy.a, and Jones.Faculty.a will be members of the class identified by Jones.*.a. Similarly, processes with access identifiers Jones.Lazy.a, Jones.Work.q, and Jones.Faculty.q are members of the class identified by Jones.*.*. The string *.*.* identifies the class of all processes.

Structure of an Access Control List

From the above discussion one can see that it is quite possible for a single process to be a member of more than one process class. This situation can lead to ambiguities on ACLs when more than one entry can apply to the same process. To eliminate this ambiguity and make ACLs more easily readable, four conventions are imposed on ACLs and their interpretation. First, no process class identifier may appear more than once on any ACL. Second, the ACL is ordered as explained below. Third, the entry that applies to a given process is the first entry on the list whose process class contains the given process. Finally, if no entry exists on the list for a given process then that process has no access to the segment or directory. These conventions assure that the access for every process is uniquely specified by the ACL.

In order to properly generate and modify ACLs it is necessary to have some understanding of how they are ordered. The ordering is done by leftmost specificity of components of process class identifiers. This can be easily explained by a simple ordering algorithm and an example. The entries to be ordered are first divided into two groups, those whose first (person) component are specific (i.e., are not asterisk) and those whose first component are asterisk. Those with specific first component are placed first on the ACL. Within these two groups a similar ordering is done by second (project) component again with the specific entries being first. This produces four groups. Finally, within each of these four groups a similar ordering is done on the third (instance) component to produce eight groups. The eight groups resulting will be in the following order:

1) class identifiers with no asterisks

2) class identifiers with an asterisk in the third component only

3) class identifiers with an asterisk in the second component only

4) class identifiers with asterisks in the second and third components only

5) class identifiers with an asterisk in the first component only

6) class identifiers with asterisks in the first and third components only

7) class identifiers with asterisks in the first and second components only

8) the class identifier *.*.*

Within each of these groups the ordering is unimportant because a process may belong to only one class in a group. The following is a validly ordered ACL:

Jones.Work.a	r	(1)
Smith.Lazy.*	rw	(2)
White.*.q	re	(3)
Black.*.*	rew	(4)
*.Faculty.m	no access	(5)
.Student.	re	(6)
.Lazy.	r	(7)
..b	rew	(8)
..*	r	(9)

In the above example a process with access identifier Smith.Lazy.h would be able to read and write the segment as derived from entry (2), a process with access identifier Jones.Lazy.h would be able only to read the segment as derived from entry (7), and a process with access identifier Smith.Faculty.q would be able to read the segment as derived from entry (9). Note that despite entry (9), which apparently grants read access to all processes, Smith.Faculty.m has no access since entry (5) is encountered first.

Maintenance of Access Control Lists

Both commands and subroutines are provided for the purpose of creating and modifying ACLs. The commands are listacl, setacl, and deleteacl (see the MPM write-ups for these commands). The subroutines are hcs_$add_acl_entries, hcs_$add_dir_acl_entries, hcs_$replace_acl, hcs_$replace_dir_acl, hcs_$delete_acl_entries, hcs_$delete_dir_acl_entries, hcs_$list_acl, and hcs_$list_dir_acl (see the MPM write-ups for these subroutines). The specific usage of each of these procedures is described in their command and subroutine write-ups. The commands and subroutines enforce the constraints mentioned above; i.e., they order the ACL and do not permit more than one entry with a given process class identifier to appear on the ACL.

Consider the example of a segment with an ACL containing the single entry:

```
Jones.*.*      r
```

A new entry is added for the process class *.Work.* resulting in the ACL:

```
Jones.*.*      r

*.Work.*       rw
```

This would superficially appear to given all members of the Work project the right to read and write the segment. In actuality it gives all members of the Work project the right to read and write the segment except for Jones (assuming Jones is a member of the Work project). Jones has only read access. If we truly wanted to give all members of the work project write access we would have to add another entry to produce:

```
Jones.Work.*     rw

Jones.*.*         r

*.Work.*         rw
```

The entry Jones.*.* is still useful for specifying access for Jones when he logs in on any project other than Work.

It is important to realize that placing a new entry on an ACL does not necessarily grant all members of that process class the specified access, for some members of that process class may also be members of process classes appearing earlier on the ACL. The user should, therefore, be aware of what an ACL currently contains before modifying it.

Special Entries on Access Control Lists

Several Multics system services are performed by special processes as opposed to being done in the user's process. These system service processes perform such functions as making backup copies of segments in the storage system and queued printing and punching of segments at users' requests. In order for these service processes to perform these functions they must have access to the segments to be serviced. In many cases the service processes normally service all segments in the storage system and, therefore, need access to most segments. These service

processes and only these service processes are members of a single project called SysDaemon. In order to assure that these service processes have access to the segments the storage system subroutines automatically place the ACL entry

.SysDaemon. rw

on the ACL of every segment, and the ACL entry

.SysDaemon. sma

on the ACL of every directory when the segment or directory is created or its ACL is entirely replaced. A user taking no special action with regard to any members of the SysDaemon project will, thereforce, have automatically granted the necessary access to all service processes so that they may perform their function.

Under special circumstances, some user may elect not to receive the service of a service process on some of his segments. To do this, the user simply denies access to his segments to that service process by modifying the ACL to contain an entry for that service process with null access. It is crucial that a user who elects not to receive such a system service be fully aware of the nature of the service and the consequences of his choice. For example, if the backup processes are not permitted access to a segment, backup copies of the segment cannot be made and the segment will not survive certain types of system failure.

Default Values for Access Control Lists

Many system commands and subroutines, e.g., create, create_dir, and hcs_$append_branch, add an entry for the creating process to the ACL of a newly created segment or directory. The storage system subroutines also automatically add the above mentioned service process entry to all newly created segments and directories. It is also useful to be able to specify a list of entries to be added to all newly created segments in addition to entries for the creating process and the service processes. This eliminates the need to expicitly modify an ACL each time a new segment or directory is created. This list of entries to be added to newly created segments or directories is called an initial access control list or initial ACL and is an attribute of a directory. Each directory has two sets of initial ACLs, one set for segments appended to the directory and one set for directories appended to the directory. Since each initial ACL is simply a list of ACL entries, it has the appearance of an ACL. When a segment or directory is created the service process ACL

entry is first placed on the ACL of the segment or directory. Then the appropriate initial ACL (i.e., either the one for segments or the one for directories) of the containing directory is merged with the ACL. The merging of two ACLs means that the entries are combined and sorted. If two entries on the resulting ACL contain the same process class identifier, then the entry that was originally on the ACL of the segment is deleted leaving the newly added entry. In this way the service process entry originally on the segment may be overridden by the initial ACL by placing an entry with process class identifier *.SysDaemon.* on the initial ACL. Finally, any entries specified in the call to append the segment (for most system commands this is simply one entry for the creating process) are merged into the ACL. Again these entries will override the service process and initial ACL entries if duplicate process class identifiers exist.

The default value for the initial ACLs of a newly created directory is empty, i.e., there are no entries in the initial ACLs.

Reference

Organick, E.I., The Multics System: An Examination of its Structure, Chapter 4, Access Control and Protection, M.I.T. Press, Cambridge, Mass. 1972

(END)

MULTI-SEGMENT FILES

Segments on Multics have a size limit which is suitable for a large class of applications. However, subsystems which make use of large data bases need ways of treating many segments as a single file. The approach that has been adopted in Multics is to put these segments into a directory and to set the directory's multi-segment file indicator (formerly called the bit count) to some nonzero value. Such a directory is called a multi-segment file (MSF).

The convention for MSFs is considered to be at a higher level than the file system and command system. That is, the file system and most of the command system do not recognize MSFs. To manipulate an MSF, the user must use the tools provided him by the subsystem which maintains the MSF. For instance, he cannot expect to edit an MSF containing ASCII information using a standard Multics editor. For user convenience, the list, status, and delete commands do recognize MSFs. In addition, the dprint command, if given an MSF, will assume that the multi-segment file indicator of the MSF is a count of the number of segments in the MSF and that if the MSF contains n+1 segments, then their names are the characters 0, 1, 2, ..., n. The MSFs produced by the file_ I/O System Interface Module (IOSIM) (see the MPM Command Section) are in this format, thus making it possible to dprint them.

(END)

BACKUP AND RETRIEVAL OF USER STORAGE

The Multics backup system provides insurance against the involuntary destruction of information maintained by the storage system. This insurance is obtained by preserving on magnetic tape recent copies of all nonexpendable segments known to the storage system, and reclaiming these copies when needed. In effect, the backup system augments the reliability of the on-line storage system.

It is the primary responsibility of the backup system to protect users against damage to their segments that can result from a system failure or error. As a secondary duty, the backup system also protects users against self-inflicted damage to their segments.

The backup system performs the following four functions:

1) segment backup copying

   the copying of segments from the Multics directory hierarchy onto tape.

2) retrieving

   the recovering (during normal Multics operation) of segments that have been copied onto tape.

3) reloading

   the recovering of the entire contents of on-line storage (generally following a system crash) in order to resume Multics operation.

4) salvaging

   the finding, reporting, and correcting of errors contained in the Multics directory hierarchy.

Users are normally concerned only with segment backup copying and retrieving, since reloading and salvaging are system functions performed automatically when the need arises.

A number of installation-determined parameters occur in the Multics backup system; in particular, the frequency of segment backup copying and the length of time for which tapes are kept are determined locally. Examples in the text are typical values and should not be relied on. The user should check with his local installation to find out these local values.

Segment Backup Copying

The segment backup copying mechanism searches out, selects, and copies onto tape, segments from the Multics directory hierarchy. (The term "dumping" has been used to indicate backup copying for historical reasons. It is a piece of jargon local to Multics and is avoided in this write-up as much as possible.) At the same time it produces a map indicating the segments included in each backup copy. The copying mechanism operates in three different modes corresponding to three different types of backup copies -- incremental, consolidated, and complete. These backup copies are distinguished by three different criteria used to select candidates for copying (as described below).

During each of the three types of backup copying, those portions of the hierarchy specified by a control segment are searched. In current practice, only two subdirectories of the root directory are not searched. One of these, >system_library_1 (which is logically a subset of >system_library_standard), is always reloaded from a Multics system tape and therefore does not require the services of backup. In >system_library_1 is contained the hardcore system plus that part of the command system needed during reloading. The other subdirectory, >process_dir_dir, contains only per-process information which is temporary in nature and hence also does not require the services of backup. All other sections of the hierarchy are included in the backup copying search path. See the MPM Reference Guide section, The Storage System Directory Hierarchy.

Two system processes are employed by the backup system for the purpose of segment backup copying, namely Backup.SysDaemon and Dumper.SysDaemon. The Backup process is used to produce incremental and consolidated backup copies, whereas the Dumper process is used to produce complete backup copies. Hence, complete backup copies can be produced concurrently with either incremental or consolidated backup copies.

Incremental Backup Copying

Incremental backup copying is the principal technique used to keep the backup system abreast of changes to on-line storage. It is the purpose of an incremental backup copier to discover modifications to on-line information not reflected in backup tape storage. The incremental backup copier locates and copies all segments in its search path that have been modified more recently than they have been copied. This criterion is easily determined by comparing the date/time modified (maintained by the storage

system) and the date/time dumped (maintained by the backup system) found in the entry for any given segment. Immediately after backup copying a segment, the incremental backup copier resets the date/time dumped for that segment. The net effect of the incremental backup copying scheme is to limit the amount of information that can be lost to those modifications that have occurred since the last backup copy.

Incremental backup copying is triggered periodically by a system timing mechanism. In order to restrict the maximum time span during which modifications to on-line storage can go unnoticed by the backup system, it follows that incremental backup copies should be produced frequently. On the other hand, because the backup daemon competes with ordinary users and exerts a considerable drain on system resources, it becomes economically desirable to lower the frequency of incremental backup copies. Therefore, the incremental backup copying rate at an installation is chosen as a compromise between these two considerations. A triggering interval of one hour might be chosen. This does not imply, however, that an incremental backup copy will necessarily finish within a single one-hour time interval. In fact, since the incremental backup copier normally enjoys no scheduling advantage, an incremental backup copy typically might require several one-hour intervals to complete during hours of heavy system load. A new backup copy begins immediately following the completion of such an overtime backup copy.

The backup system does not guarantee that segments are copied in a consistent state. In other words, it is possible that while the backup copier is copying a segment, another process might be writing into that same segment. Thus, an inconsistent copy of a segment might be produced. Note, however, that modifications that cause a segment copy to be inconsistent also cause another copy of the segment to be produced on the next pass of the incremental backup copier. Therefore, a consistent copy is eventually produced.

Consolidated Backup Copying

A consolidated backup copier locates and copies segments in its search path that have been modified after some specified time in the past. For example, an installation might choose to run a consolidated backup copier every midnight to copy all segments modified since the preceding midnight; i.e., since the preceding consolidated backup copy. When used in this manner, the consolidated backup copier is essentially imitating an incremental backup copier except, of course, that the interval

between consolidated copies is longer. Another possible practice is to backup copy all segments modified since the last complete backup copy (see below).

The original motivation behind consolidated backup copying was the need to consolidate the most recent copies of segments from a group of incremental tapes in order to reduce the amount of time needed to reload these tapes. An inefficient solution to this problem would be to actually merge a group of incremental tapes. Instead, the consolidated backup copy was invented to achieve the same goal. Notice that since a consolidated backup copier catches modifications that have accrued over a period of time encompassing many incremental backup copies, it effectively consolidates the most recent information from a group of incremental tapes. Furthermore, the consolidated backup copier also picks up segments that have been modified more recently than the last incremental backup copy. Hence, a consolidated backup copier performs the work of an incremental backup copier as well.

A second motive for consolidated backup copying might be simply expressed as "two copies are better than one". There is some question, however, as to when secondary copies should be produced. An alternate and somewhat more obvious strategy, for example, would be to have the incremental backup copier produce duplicate tapes. This method would guarantee that both primary and secondary copies were identical. Such a facility is, in fact, implemented in the backup copier, but is not used for incremental backup copying. The principal shortcoming of the duplicate tape strategy is that the primary and secondary copies are not produced in a sufficiently independent fashion. Therefore, both copies are, to a certain extent, susceptible to the same errors. This is particularly true of operational errors and, to a lesser degree, is true of hardware and software errors. Also, the duplicate tape strategy is comparatively expensive and voluminous.

Complete Backup Copying

A complete backup copier simply backup copies every segment in its search path without regard for modification time. Unlike modification-driven backup copying which attempts to keep the backup system up-to-date, complete backup copying is somewhat different in purpose and follows a more leisurely schedule. During a complete backup copy, the date/time dumped is not reset. Therefore, complete backup copying does not interact in any way with incremental or consolidated backup copying.

A complete backup copy establishes a checkpoint in time, essentially a snapshot of the entire Multics storage hierarchy. If it should ever become necessary to recover from scratch the contents of on-line storage, then the most recent complete backup copy marks a cutoff point beyond which no older backup tapes need be inspected.

Another purpose of complete backup copying involves the tape retention strategy. The high production rate of incremental and consolidated tapes makes the retention of these tapes for long periods of time impractical. Therefore, incremental and consolidated tapes are kept for some short time, perhaps three weeks. Complete backup copy tapes are retained for a longer time, perhaps six months, with the exception of one complete backup copy per month that might be held for a period of one year.

Retrieving

The segment retrieval mechanism used by the backup system consists of a group of programs known as the reloader/retriever. The reloader/retriever is used to recover segments from tapes produced by the backup copier. Retrieving, which occurs during normal Multics operation, is distinguished from reloading, which occurs prior to normal Multics operation.

When a user notices that a segment or directory has been lost or damaged, he can submit a request to the Multics operations staff for that segment or directory to be retrieved from a backup tape. The problem he faces, of course, is determining which backup copying operation produced the copy he wishes to retrieve. Usually the most recently produced copy is wanted. In the case of a damaged segment, however, the damaged version is likely to have been backup copied as well, and hence the most recent copy is not wanted. Hopefully a user knows approximately when his segment was lost or damaged. Also, he should remember if the segment has been recently modified. Using these two pieces of information he can make a reasonable guess as to which backup copy contains a suitable copy of a given segment.

Once a conjecture has been made as to which backup copy contains the desired copy, the conjecture can be verified by examining the corresponding backup copy map. The map indicates the tape reel on which the backup copy was written. A feature of the backup copy map which is sometimes helpful is the date/time last dumped for the segment, which effectively points to the next most recent backup copy of the segment.

The user can specify that a single segment, a directory without its subtree, or a directory including its subtree be retrieved. A directory for which the subtree is not retrieved contains only the links and access control information associated with the directory itself.

A user can also specify that the segment or directory be retrieved by a different path name. A single segment or a directory without a subtree can be relocated at any point in the storage system hierarchy. A directory subtree can be relocated at any point at the same level in the hierarchy (i.e., the number of > characters in the path name of the directory cannot change).

Normally the most recent copy of an entry on the specified tape is retrieved. However, the user can specify that the first occurrence is to be retrieved instead, presumably to retrieve a particular intermediate copy.

 (END)

USE OF THE INPUT AND OUTPUT FACILITIES

Multics provides various means for performing input and output, i.e., moving data in to or out of the Multics system. Users should note that input/output (I/O) in Multics is not the means by which the data stored in segments in the Multics storage system is referenced. Data in the Multics storage system is internal to Multics and referencing such data, therefore, does not require moving it in to or out of the system. Refer to the MPM Reference Guide Section on Using the Multics Storage System for information on referencing segments. Most I/O involves one of the many available peripheral devices such as typewriters, printers, tape drives, card readers and punches, graphic devices, etc. All I/O operations in Multics make use of the I/O system. The use of the Multics I/O system is described in the MPM Reference Guide Section on the Use of the Input and Output System. The I/O system provides a general means by which any I/O capable of being performed in Multics can be accomplished. However, for most common uses of I/O, procedures are provided which eliminate the need for the user to have a working knowledge of the I/O system. This section provides a brief introduction to some of these facilities and indicates where further information about each facility can be found. The MPM Reference Guide Section, Available Input and Output Facilities, provides a complete list of I/O facilities.

Translators

Multics provides several higher level languages which have built-in I/O facilities such as formatting and directing I/O to and from various devices (e.g., PL/I, FORTRAN). Users should consult the appropriate language manuals for a description of these facilities.

Console Input/Output

For simple reading from and writing to the user's console, the entries ios_$read_ptr and ios_$write_ptr are provided. See the MPM write-up on ios_.

Formatting

Two subroutines, ioa_ and write_list_, are provided for formatting output before it is printed on the console. The subroutine read_list_ is used to parse input read from the console. See the MPM Subroutine Calls Section describing these programs.

Redirecting Console Output to a Segment

The file_output command will cause all subsequent output normally printed on the user's console to be written instead to a segment in the file system. The console_output command causes such output to be directed again to the console. See the MPM Command Section describing these commands.

Printing of Segment Contents

The contents of segments that contain exclusively Multics ASCII characters may be printed on the console by invoking the "print" command. The contents of a segment which does not contain characters can be printed in octal using the command dump_segment or the Multics debugger, debug. See the MPM Command Section describing these commands.

Bulk Input/Output

In order to permit all users to make use of certain critical peripheral I/O devices in an efficient manner, Multics provides a service in which users may queue requests to input or output data using these devices. These devices are:

1) Printer - The contents of a segment containing Multics ASCII characters can be printed on a high speed printer using the dprint command. The dprint command queues the request for printing and, at some later time, the contents of the segment will be printed and made available to the user by the Multics installation. The printed output will consist of one or more pages of header indicating the path name of the segment and the user issuing the request, the contents of the segment, and one or more pages of information detailing the charges incurred in printing the contents of the segment. See the MPM command dprint.

2) Card Punch - The contents of a segment may be punched on cards using the dpunch command. In a manner similar to printing, the dpunch command queues the requests for punching and, at some later time, the contents of the segment will be punched and made available to the requester. The output deck will consist of several "flip" cards identifying the segment and requester, the contents of the segment in one of several possible formats, and cards indicating the end of a deck. For a description of the various types of card format, see the MPM Reference Guide Section on Bulk Input and Output. The use of the dpunch command is described in the MPM Command Section.

3) Card Reader - Information contained in a deck of cards may be read into the Multics system by submitting the deck to the Multics installation. The deck must consist of a control card which contains information about the deck and how it is to be located within the system, the cards containing the data in one of the Multics standard formats, and a special card indicating the end of the deck. The contents of the deck will be input at some later time to a segment in a special directory in the Multics system and a link to this segment placed as indicated on the control card. After the deck is read in, it will be made available to the submitter. The exact format of input decks is described in the MPM Reference Guide Section on Bulk Input and Output.

Graphics

Users having I/O devices with graphic I/O capabilities can make use of the Multics graphic system. The graphic system uses a general intermediate graphic structure language that permits programs to be written in a device-independent manner. For simple graphic applications, see the MPM subroutine plot_. Users wishing more sophisticated graphic capabilities should see the MPM Reference Guide Section on Graphics Support on Multics and obtain the MPM Graphics Users' Supplement.

(END)

USE OF THE INPUT AND OUTPUT SYSTEM

The primary purpose of the I/O system is to allow a process to communicate with external physical devices such as typewriters, on-line printers, tape drives, etc. In other words, I/O is the movement of data to and from the system, as opposed to the management of data within the system. An example of the latter is the storage system. On the Multics hardware, I/O communications are handled by the I/O channel controller which is fed instructions by the CPU. It is the function of the I/O system to translate high level I/O system calls into channel controller instructions. This is accomplished through various levels of programming both within and outside the hardcore supervisor. The final result of this programming is that the user is provided with a very flexible set of subroutines which allow him to easily communicate with devices. The flexibility is provided by the use of I/O stream names for addressing devices, the Attach Table for associating the stream name with a device, and a means by which the association can be changed. The user is able to perform I/O operations upon diverse devices by using a single set of simple operations. This is possible because the system is designed so that all I/O devices appear functionally identical at the user procedure level. Exceptions to the functional equivalence occur in devices upon which certain operations cannot be performed; for example, reading from a printer or writing to a card reader. The combination of stream names and functional equivalence of devices from the user level allows run-time specification and modification of the device or devices to which I/O is directed without modification of programs.

Interface Modules

The procedure responsible for coordinating the communication with a particular device is called an interface module. Each device has at least one associated interface module. The interface module is responsible for lending the appearance of functional equivalence of a device to the I/O system. It contains entry points whose calling sequences and functions are defined by system wide convention. These common operations are transformed by the interface module into calls to other I/O system subroutines in order to perform the actual I/O. It is the primary purpose of the interface module to create the common appearance of the device to higher levels of the I/O system in order that I/O calls directed to different devices can be made in a standard manner.

There are basically three types of interface modules:

1) Device Interface Module (DIM): This type of module coordinates communications with a particular external physical device such as a typewriter, printer, or tape drive.

2) Pseudo-Device Interface Module: This type of module permits a process to communicate with something other than a physical device as if it were a device. The best example is the Storage System Interface Module (SSIM) which allows a process to use a segment (or set of segments) in the storage system as an I/O device. Another use could be to create a pseudo-device to simulate an actual device for testing purposes.

3) Intermediate Interface Module: This type of module is intended to serve as an intermediary between a user program and another interface module and will be discussed in more detail later. It serves to modify an I/O communication before the DIM or pseudo-DIM is called. The modification might be the formatting of data or redirecting of the call to a different device or set of devices.

In order to perform its function, the interface module may call upon other routines for such things as code conversion or buffering. In some cases, device dependent code is placed in the Hardcore Ring supervisor for efficiency (e.g., the major portion of the typewriter DIM is in The Hardcore Ring). If an external device is involved, the interface module will have to send instructions to the hardware channel controller. A procedure in the supervisor has been provided for this purpose. The channel controller interface module is responsible for giving instructions (Data Control Words (DCW)) to the channel controller and is callable by User Ring procedures.

Streams and the I/O Switch

The user of the I/O system does not directly call an interface module. I/O calls are made by specifying a symbolic stream name. One of the arguments of each call to the I/O system is a stream name. Each stream name is associated with an interface module and a particular device. Thus, when a user wishes to obtain his input from or direct his output to a different device, he does not have to rewrite his programs; he simply changes the device with which the specified stream is associated, assuming the device is capable of the desired operations.

The association of stream names and interface modules and devices is kept in a data base called the Attach Table. The procedure which maintains and uses the Attach Table is called the I/O switch. All user calls to the I/O system are actually calls to the I/O switch. The switch finds the specified stream name in the Attach Table and calls the associated interface module at the appropriate entry. Two calls to the I/O switch are handled in a special manner. The attach call is used to initialize a stream and a device for communication. It also causes the I/O switch to create an association in the Attach Table between the specified stream name, interface module, and a particular device. The creation of such an association is initiated by use of the following call.

```
call attach ("user_i/o", "typewriter", "tty100");
```

This call causes the I/O switch to call the typewriter interface module, which has been specified by its name "typewriter" to prepare the specified typewriter, "tty100", for communications. If this is successful, the I/O switch creates an entry in the Attach Table relating the stream name "user_i/o" to "tty100" with the typewriter interface module being specified as the control program. All further I/O calls on the stream "user_i/o" will be forwarded to the typewriter interface module to be performed on "tty100". The detach call serves the opposite purpose. It causes the I/O switch to terminate communication with the device and remove the association from the Attach Table.

Intermediate Interface Modules

The attachments discussed so far associate a stream name with an interface module and a device. Another type of attachment associates a stream name with an interface module and another stream name. The interface modules involved in such attachments serve to intercept the I/O call made on the first stream and, after performing some processing, make another call to the I/O switch on the second stream. Such interface modules are effectively spliced into the logical flow of control and are, therefore, called intermediate interface modules. Such interface modules may be used for formatting data before it is actually passed on to a device. There are also some other important uses of intermediate interface modules.

The syn intermediate interface module simply passes an I/O call to a different stream. If the stream "user_output" is attached via the syn interface module to the stream "user_i/o", then all I/O calls to "user_output" will be directed by the syn

interface module to the stream "user_i/o". A call to either stream will have the same result and the streams are, therefore, synonymous. syn attachments are useful because they are much easier to change than device attachments because no device initializations or terminations are necessary. (Initialization and termination are those operations a DIM must perform to start or end communication with a physical device.)

The broadcast interface module is used for attaching a stream to many other streams. In doing so, a call on the first stream will result in calls on all the streams to which it is attached. This is a means by which I/O calls may fan out.

Figure 1 gives a block diagram of the I/O system showing the various types of modules which can exist.

Modes and Data Organization

Each attachment has associated with it certain attributes called modes. The precise interpretation of a mode is dependent upon the interface module being invoked, but some general statements about modes can be made.

The read and write modes specify that the read (input) and write (output) operations can be performed on this device. The user may wish to use a teletype only for writing and, therefore, may not wish to give it the read attribute. On devices such as a printer, the read attribute is meaningless; therefore, attempts to read from a printer will cause an error status to be reflected back to the user.

I/O data consists of a collection of elements where an element is the smallest indivisible unit of information. The type of an element is determined by its size, i.e., the number of bits it contains. Data operated on by a single I/O call to an interface module must consist of elements of the same type (size). Entries are provided to change the element size accepted by an interface module. An element could be a character or a bit or a word or a fixed-size logical record. The data element size is usually a function of the type of information being transferred and is not necessarily a function of the device. In all calls to the I/O system involving the reading or writing of data, the amount of data to be transferred is expressed as an integral number of elements.

There are two basic categories for data: physical and logical. The physical representation is the way the actual device accepts the data. The organization of logical data is

defined by the program. For example, magnetic tape requires that data be organized into physical records with gaps between the records. However, the user receives a linear stream of data as the logical representation.

Other modes are concerned with the synchronization of I/O calls and the actual I/O operations. The use of such features as read-ahead and write-behind means that the I/O operation may not be performed at the time of the actual I/O call. Various modes and calls are provided to give the user some control over the synchronization.

Status Reporting

It is necessary for the I/O system caller to know the status of his I/O call. This is especially true since the I/O operation may not be performed at the same time as the call. For this reason, a status argument is provided in each call. The I/O system returns essentially three pieces of information in the status argument. A code describing the nature of an error, if one occurs, is returned. Some bits are returned indicating the state of the transaction. The status argument also includes a unique identifier which enables the I/O system to identify the transaction if the caller wishes to request information about or modify the transaction at a later time. Status is described in more detail later in this document.

Usage

The various calls to the I/O system are described in the MPM write-up on ios_. In order to illustrate some of the uses to which the I/O system can be put, a few examples of its use follow. For ease of explanation, the calls shown in this section may not correspond exactly to the calls available on Multics; the reader should refer to other MPM sections for exact calling sequences.

A typical user performs I/O communications with a single device, a typewriter. To initialize this communication, calls are made to the I/O system at process initialization time. The following three calls are made:

```
call attach ("user_i/o", "typewriter", "tty100",
     "read, write", status);

call attach ("user_output", "syn", "user_i/o", "write",
     status);
```

```
call attach ("user_input", "syn", "user_i/o", "read",
     status);
```

The first call establishes that the device whose identifier is "tty100" will be controlled by the I/O interface module named "typewriter" for reading and writing. All subsequent operations for this device will be made by referencing the stream name "user_i/o". Information about the status of this I/O call is returned in "status". The second call establishes a synonym for "user_i/o" called "user_output". The "write" as the fourth argument, however, restricts this synonym to write calls only. Any write call to the stream "user_output" is now synonymous to a write call on the stream "user_i/o". Any read call on "user_output" will be an error that will be reflected in the status string of the read call. The third attach call establishes another synonym for "user_i/o" for reading only and is called "user_input".

The process may now communicate with the typewriter "tty100". To type something out on the typewriter, the process issues a write call:

```
call write ("user_output", workspace, nelements, status);
```

This call writes the specified number of elements (nelements) of data contained in workspace onto the stream "user_output". This results in a call to the stream "user_i/o". The typewriter interface module is invoked to transmit the information to "tty100". The sequence of calls is illustrated in Figure 2.

The process may also wish to receive input typed by the user on the typewriter. For this purpose, the following call is issued:

```
call read ("user_input", workspace, nelements, nactual,
     status);
```

This call reads into workspace the specified number of elements (nelements) from "tty100" following a similar sequence of calls as in the write request. Read calls have an additional feature in that they make use of read delimiters. The typewriter DIM will attempt to read the specified number of elements; however, if, in doing so, it encounters a read delimiter, reading ceases at that point. The actual number of characters read is returned as nactual. For example, if the process issued the call:

```
call read ("user_input", workspace, 10, nactual, status);
```

and the string typed on "tty100" by the user is abcdefghijklmn and the read delimiter is d, then the string abcd will be placed in workspace and nactual will be set to four. If, however, the read delimiter is m, then workspace will be set to contain abcdefghij and nactual set to ten since only ten elements (characters in this case) were requested in the read call. Delimiters and element types are established by various calls to the I/O system. The most commonly used read delimiter is the "new line" character.

Let us assume that the user wishes to direct his output to a segment in the storage system instead of to his typewriter. To do this, he may issue the following calls:

```
call attach ("segment_stream", "file_", "segment_name",
     "write", status);

call detach ("user_output", "user_i/o", status);

call attach ("user_output", "syn", "segment_stream",
     "write", status);
```

The first call initializes a segment whose name is "segment_name" as an I/O device via the SSIM. All write calls on the stream "segment_stream" will be directed to the segment "segment_name". However, we assume that the programs the user is about to invoke use the stream "user_output" for writing since this is the stream that normally is used for output. The detach call and the subsequent attach call serve to modify the synonymization of "user_output". "user_output" is now synonymous to "segment_stream" for writing and all write requests will now be directed to the SSIM for writing into the segment "segment_name". Note that if "user_output" had been attached directly to the typewriter, instead of indirectly through the syn interface module, it would have been necessary to terminate the actual device, i.e., tty100, to accomplish the detachment. By having the extra level of indirection, such changes are simply table modifications by the syn interface module. Now all output written to the stream "user_output" will appear in "segment_name" instead of on the typewriter. In order to revert back to the typewriter, the process issues the calls:

```
call detach ("user_output", "segment_stream", status);

call attach ("user_output", "syn", "user_i/o", "write",
     status);
```

Note that the segment remains attached via the stream "segment_stream" to allow the user to switch back to segment output more easily. When the user is sure he will no longer use the segment for output, he issues the call:

```
call detach ("segment_stream", "segment_name", status);
```

Note, also, that during this entire operation, the user's input stream was never affected, i.e., read calls directed to the stream "user_input" will obtain data from the typewriter.

Certain Multics commands and the Multics user environment expect the standard streams user_output, user_input, and error_output to be attached to other streams via the syn interface module. This assures that if any of these streams are detached and subsequently reattached, as is the case when the user quits and starts, that no information about the attachment is lost. Users should therefore attach these streams using only the syn interface module.

I/O Terms

Below are some of the more important terms used in describing the I/O system.

Attachment

An attachment is the association of one stream name with an interface module and either a device ID or another stream name. This association is established by an attach call. Subsequent to an attachment, data may be read or written by issuing a read or write call with the appropriate stream name.

Delimiters

There are two kinds of I/O delimiters meaningful to an I/O user on input. These are the break characters and the read delimiters which are established by means of the setdelim call. A break character is meaningful only to an interactive device and serves three functions: it delimits physical interrupts, canonicalization, and erase-and-kill processing. A break character is an interrupt delimiter in that it is recognized by the hardware channel controller and causes an immediate interrupt. A break character is an erase-and-kill delimiter in that its presence permits erase-and-kill processing to take place over all characters received since the preceding break character. A break character is a canonicalization delimiter in that its presence permits canonicalization to take place over all elements

received since the preceding canonicalization delimiter. For certain devices (e.g., typewriters), the "new line" character is the default break character. In addition, whether established as a break character or not, the "new line" character always delimits canonicalization and erase-and-kill processing. A read delimiter affects the amount of data transferred to the user program during a read call. A read call continues to transmit data to the caller until a read delimiter is recognized (and transferred) or until the specified amount of data has been transferred. The current break characters and read delimiters may be determined or modified using the getdelim or setdelim calls.

Element

An element is a linear array of bits. It is the smallest data entity referred to by an I/O call. The most frequent element sizes are 1 (bit), 9 (character), and 36 (word) bits. The current element size of a stream may be determined or modified using the getsize or setsize calls.

Modes

The modes of an attachment specify the setting of a collection of attributes relevant to the intended use of the specified stream name. Modes are expressed as a character string to the attach call or a changemode call. The string consists of key words, specifying individual modes, separated by commas. Modes need only be understandable by the interface module receiving the attach or changemode call. Some modes, however, have commonly accepted meanings:

1) read data may be input from this stream;

2) write data may be output to this stream.

The key words in a mode string may be preceded by the circumflex character which indicates the opposite of the mode. For example, the mode string "read,^write" indicates data may be input from this stream but no outputting may be done to this stream. Note that the mode string in the attach and changemode calls serves only to change the setting of the modes specified in the string and does not affect the setting of modes not specified in the string. In the attach call, the modes are changed from the default; in the changemode call, the modes are changed from the current setting.

Reference Pointers

Associated with each stream are five values called reference pointers. A reference pointer is the offset of an element of data that is of special interest. The five reference pointers and the elements to which they point are:

1) read — the next element to be read;

2) write — the next element to be written;

3) first — the first element of data;

4) last — the last element of data;

5) bound — the element beyond which data cannot be written.

The seek I/O call may be used to explicitly modify the values of these reference pointers. The read reference pointer is modified implicitly when a read or read_ptr I/O call inputs some data. The write reference pointer is modified implicitly when a write or write_ptr I/O call outputs some data. The last reference pointer is implicitly modified by a write or write_ptr I/O call which appends to the end of the data. The values of the reference pointers always obey the following relations:

$$first \leq read \leq last + 1 \leq bound + 1$$

$$first \leq write \leq last + 1 \leq bound + 1$$

For example, if one wanted to start reading the data of a device from the beginning one would issue the following seek call:

```
call seek (stream, "read", "first", 0, status);
```

If one wanted to append to the end of the data one would issue the call:

```
call seek (stream, "write", "last", 1, status);
```

If one wanted to erase the data one would issue the call:

```
call seek (stream, "last", "first", -1, status);
```

Note that the values of reference pointers are relative to one another. They do not have absolute values. As with I/O calls, all of the five reference pointers may not be implemented by each I/O System Interface Module (IOSIM). The MPM sections on IOSIMs

call requesting the read operation (read-ahead). If a stream is write asynchronous, the write operation may not complete until after the write call requesting the write operation has returned (write-behind). Synchronization modes may be established using the readsync and writesync calls.

Workspace

The buffer area in the user's address space, to or from which a request to transfer data is directed, is referred to as the workspace. A read call reads an integral number of elements into a specified workspace; a write call writes an integral number of elements from the workspace.

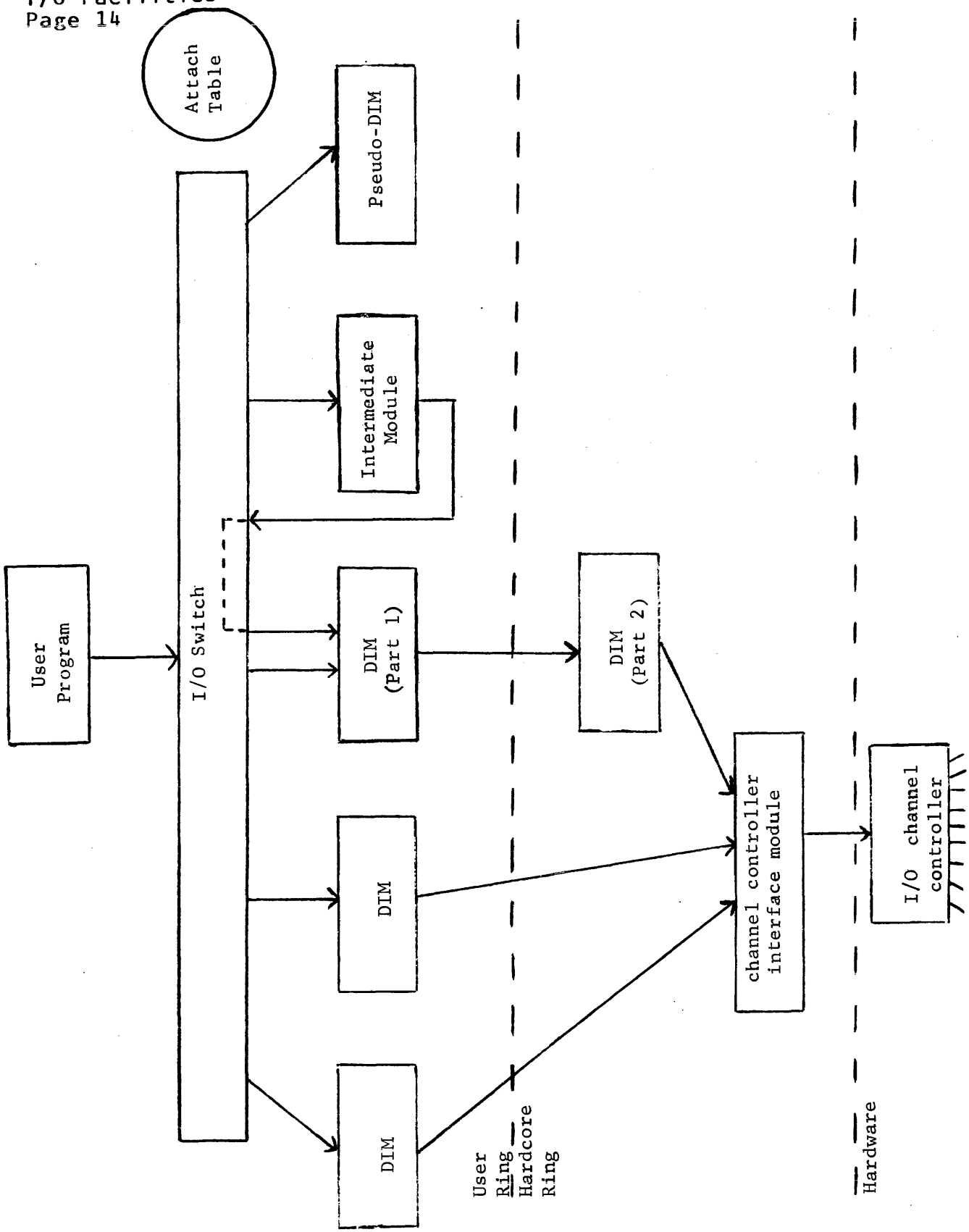


Figure 1: I/O Devices

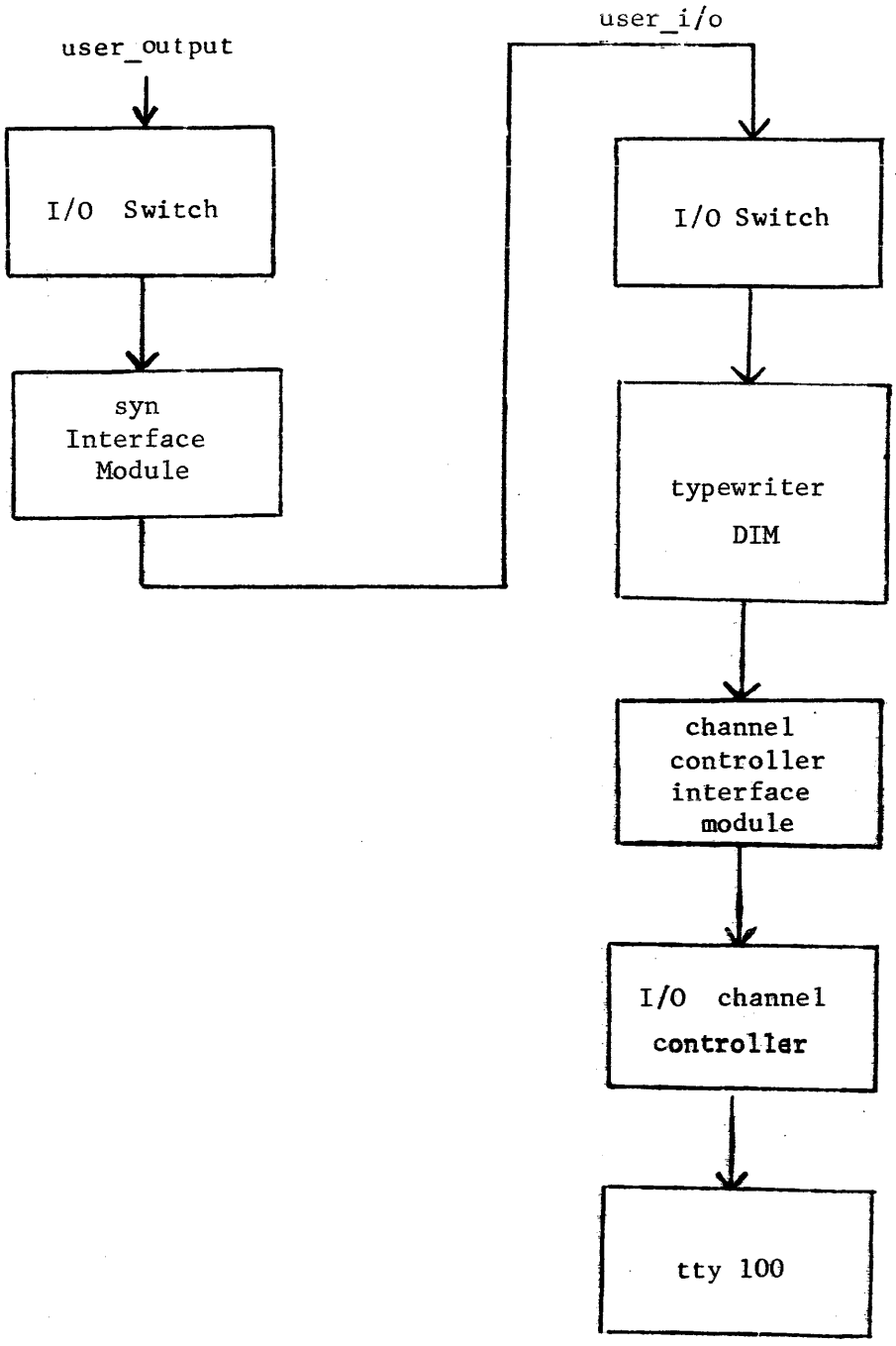


Figure 2

(END)*

AVAILABLE INPUT AND OUTPUT FACILITIES

The following is a list of commands and subroutines available to users that are commonly used to provide input and output functions. The list is organized by the function provided. The procedures listed as Miscellaneous I/O System Procedures and those subroutines categorized as Input/Output System Interface Modules (IOSIM) require some knowledge of the Multics I/O system in order to be used. (See the MPM Reference Guide Section on the Use of the Input and Output System.) The MPM section under which each module is described is given for easy reference. Note that the IOSIMs are categorized under the MPM Subroutine Calls Section.

Simple Input/Output

ios_$read_ptr	MPM subroutine
ios_$write_ptr	MPM subroutine

Formatted Input/Output

ioa_	MPM subroutine
read_list_	MPM subroutine
write_list_	MPM subroutine

Input/Output with Segments in the File System

console_output	MPM command (see the MPM command file_output)
debug	MPM command
dump_segment	MPM command
exec_com	MPM command
file_	MPM IOSIM
file_output	MPM command
print	MPM command

Translators

fortran	MPM command (see the FORTRAN Manual)
lisp	MPM command (see the LISP Manual)
pl1	MPM command (see the PL/1 Manual)

Bulk Input/Output

With the high speed printer:

dprint	MPM command

With punched cards:

dpunch	MPM command
Card Input	

With magnetic tape:

tape_	until general tape reel mounting
nstd_	facilities are available, these IOSIM's may be used only by special arrangement with operations.

See also the MPM Reference Guide Section on Bulk Input and Output.

Graphics

plot_	MPM subroutine

See also the MPM Reference Guide Section on Graphics Support on Multics and the MPM Graphics Users' Supplement.

Input/Output Devices

Cards:

dpunch	MPM command
Card Input	

See also the MPM Reference Guide Section on Bulk Input and Output.

Consoles (typewriters, teletypes, ARDS, etc.):

iomode	MPM command
line_length	MPM command
tw_	MPM IOSIM

Printer (high speed):

dprint	MPM command

Miscellaneous I/O System Procedures

broadcast_	MPM IOSIM
get_at_entry_	MPM subroutine
iocall	MPM command
ios_	MPM subroutine

print_attach_table MPM command
syn MPM IOSIM

See also the MPM Reference Guide Section on the Use of the Input and Output System.

(END)

BULK INPUT AND OUTPUT

The Multics system currently has provisions for three types of bulk I/O: high speed printed output, punched card input, and punched card output.

Printed Output

The dprint command (see the MPM Command Section) causes the contents of a Multics segment containing Multics ASCII characters to be printed on a high speed printer.

The printed output will be of the following form:

1) One or two header sheets containing the pathname of the segment printed, the identification of the requesting process, and a character string, if any, supplied by the requester in the dprint command.

2) The contents of the segment. Printed lines contain 136 character positions. If a line to be printed contains more than 136 character positions, it will be continued on the following line.

3) A sheet that summarizes the charges incurred in printing the contents of the segment.

Punched Card Input

Facilities are provided to read punched card decks into Multics segments. There are three types of card formats which can currently be input to Multics: Multics card codes, 7punch, and raw.

1) Multics card codes are defined in the MPM Reference Guide Section on Punched Card Codes. Essentially, they comprise a superset of the EBCDIC card punch codes, and are producible by 029 key punches. The 12 bit card codes are converted to 9 bit ASCII codes. (The escape conventions mentioned in the Punched Card Codes Section have not yet been implemented.)

2) 7punch decks are binary representations of existing segments, and the data portions of the cards are read in exactly as they were punched out.

3) Raw decks are simply read into Multics segments without any conversion, and without regard to format. That is, the 960 bits on each card are read into the segment, in column order. Any desired conversion may then be performed by the

user.

Note that "flip" cards (and other sorts of labelling cards from other systems) are not read correctly and should be removed from decks.

Procedure

Each deck must begin with an 029 key punch produced control card in the format described below, and end with a card which has a 5-7 multiple punch in column 1. The decks are submitted to Operations, and will, in general, be read in by the next day. Owing to protection considerations, segments will be created in a system directory rather than placed directly in the user's directory; a link to the input segment will be placed in the user's directory by the card reading program, through which the segment may be copied. In order that the link may be created, the user must not have removed append access for *.SysDaemon.* for the directory in question. (Note that SysDaemon is automatically granted access when directories are created.)

Note that segments must be copied from the system directory within a reasonable time, as the segments in that directory will be periodically deleted.

Control Card Format

TYPE DIRECTORY ENTRY ACCESS_NAME

1) TYPE is the deck type. Currently, the valid types are MCC (for Multics card codes decks), VIIPUNCH (for 7punched decks), and RAW (for unconverted decks).

2) DIRECTORY is the pathname of the directory into which a link to the input segment will be placed.

3) ENTRY is the desired entry name of the link to the input segment. Note that this entry name must not already exist, as the card reading program will not attempt to resolve name duplications.

4) ACCESS_NAME is the access control name which is to be given read access to the input segment. See the MPM Reference Guide Section on Naming Conventions for a description of access control names.

The items are to be separated by one or more blanks; all four items must be specified.

The format is shown in upper case because the 029 key punch prints that way. However, pathnames and access control names may contain both upper and lower case. Therefore, the card reading program will map all letters on the control card to lower case except those letters immediately following an escape character (¢). For example, *.¢MULTICS.* as an access name would be interpreted as *.Multics.*. This convention is established as an interim measure for ease in punching control cards, and will be superseded when the escapes of the Punched Card Codes Reference Guide Section have been implemented.

Example

Suppose user Doe, working on project Proj, wishes to read a FORTRAN source deck into a segment called alpha.fortran:

1) The control card, placed in front of the deck, is as follows:

```
MCC >UDD>¢PROJ>¢DOE>SUB XALPHA ¢DOE.¢PROJ.*
```

where MCC specifies Multics card code conversion, and XALPHA is chosen so that the eventual copy through the link need not be renamed.

2) The control card, deck, and end of file card (5-7 multiple punch in column 1) are submitted to Operations.

3) When the cards have been read, issue a copy command on the console for xalpha into alpha.fortran. If the cards were read in successfully, the copy will succeed. If not, xalpha will not be found; in this case, check with Operations to determine what went wrong.

Notes on Deck Size

Decks must not exceed the maximum length of a Multics segment. A good rule of thumb is to limit decks to single boxes of cards, although more precise counts can be made. For "raw" reading, the actual maximum is 2,456 cards. For Multics card codes, the actual maximum depends on the number of characters actually read, as trailing blanks on cards are ignored. Assuming all 80 columns are punched on each card, the maximum would be 3,276 cards. For 7punched decks, the length of the created segment depends on the length of the original segment. The typical 7punch card represents 22 words, but it may represent as many as 4,096 words if the original segment contained that many

consecutive words of identical contents.

Errors

The operator will return a note with the deck if any errors took place during the read. In general, the error should be corrected and the deck resubmitted. However, in certain cases it is possible to avoid rereading; when the only problem involves the input segment bit count or a name duplication in the user's directory, the uniquely named segment (in >daemon_dir_dir>cards) may be linked to directly.

Multics Card Output

The dpunch command (see the MPM Command Section) causes the contents of Multics segments to be punched. The segments may be punched under mcc, raw, or 7punch conversion modes.

The 7punch conversion mode essentially furnishes a binary representation of a segment, suitable for subsequent reloading. The 7punch format also provides sequencing and checksumming; hence, it is more secure than the raw mode, provided that the segment is being punched in order to serve as additional backup and not for use on any system other than Multics.

The Multics 7punch format is as follows:

Columns

Rows	1	2	3	4	5	6	7 - 72
1-3	7	w	s	c	c	c	d ... d
4-6	w	w	s	c	c	c	d ... d
7-9	w	t	s	c	c	c	d ... d
10-12	5	s	s	c	c	c	d ... d

1) 7 and 5 (octal) are 7punch format identifying codes.

2) wwww is the number of data words on the card, if less than 27(8); if greater, it is a replication count and indicates how many times the single data word on the card is to be replicated on reading back in.

3) t	is a last card code. It will be 0 on each card of the deck except the last card, where it will be 3. The bit count of the segment is punched as the last card for Multics decks.
4) sssss	is the sequence number of the card in the deck, starting from 0.
5) ccccccccccc	is the full word logical checksum of all bits on the card except the checksum itself.
6) dddd ... dddd	are the data words. On the last card, columns 7-9 contain the bit count (fixed binary(35)) and columns 10-80 are 0. Note that the word count is 0 on the last/bit count card.

Deck Structure

Labelling information is punched on "flip" cards. The structure of Multics produced decks is as follows:

Card 1:	A "flip" card containing two rows of X's
Card 2:	A "flip" card containing the heading (may be more than one card)
Card 3:	A "flip" card containing the date and time punched
Card 4:	A "flip" card containing the pathname of the segment punched (may be more than one card)
Cards 5-n:	The punched segment
Card n+1:	A "flip" card saying END OF DECK
Card n+2:	A "flip" card containing two rows of X's

(END)

GRAPHICS SUPPORT ON MULTICS

The basic purpose behind the development of the Multics Graphic System is to provide a terminal-independent general purpose graphic interface suitable for use by all graphic applications on Multics. A graphic program written at one type of graphic terminal should be operable at another type of terminal of similar capabilities without modification. A wide variety of graphic applications should be supported. The user should be able to write his program easily and naturally, and it should run in an efficient manner.

This first attempt at a graphic system for Multics naturally represents compromises with the above goals. The Multics Graphic System will change as better solutions to the problems involved are discovered. These changes will be upward compatible whenever possible, and every attempt will be made to support graphic programs written for previous versions of the graphic system.

The idea of terminal independence was adopted from the Multics typewriter I/O system, for the same reason that it is so important there. A wide variety of console types are connected to Multics (including graphic terminals used as video typewriters), and this console mix changes with time. To require a program to incorporate specific terminal dependencies would have severe disadvantages.

a) User Fragmentation. Only a limited amount of user program sharing could take place because a program written by a user could not be used by others working at different terminals. This kind of fragmentation would severely inhibit the development of on-line user communities.

b) Terminal Immobility. Being able to use only a fraction of the terminals connected to the utility is a considerable inconvenience. Even worse, the dependence of a subsystem on a specific terminal would grossly inhibit the transference to new and better types of terminals.

c) Certain modules provided by the utility would require a different version for each supported terminal type.

The net result would be to support only a limited number of different types of terminals, to inhibit the introduction of new types, and to retain an obsolete type long after the desirability of its removal had become apparent.

While these arguments for terminal independence now apply most strongly to typewriters, as more types of graphic terminals are added to Multics, they gain validity for graphics as well.

What is meant by terminal independence in graphics? There are many different types of graphic terminals ranging from storage tube terminals to refresh displays with internal processing capability. Even a plotter could be connected, though not as an interactive console. Each device has its own set of special features, such as dotted line or blinking capability. To create true device indifference by allowing the use only of those features common to all terminals is clearly an intolerable solution. Rather, as a compromise solution, an interface incorporating the union of all features of existing terminals is provided. This interface is extensible to include new features of terminals to be attached in the future.

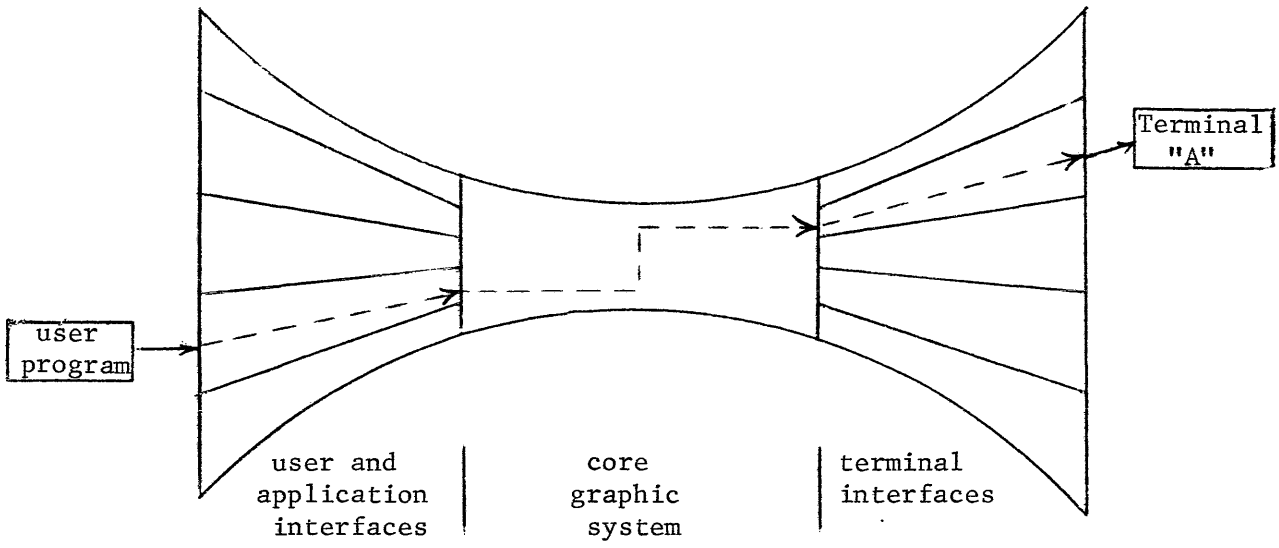
A user tailors his program to use the features of the terminal types he intends to use. When the program is run, the use of any unavailable feature is mapped into the most reasonable compromise feature of the terminal being operated. Thus, the user has a reasonable guarantee that a program coded in accord with the limitations of the system's terminal independence capability will generate a recognizable picture on practically any type of graphic terminal attached to Multics. However, it will not necessarily operate equally well at any terminal. For example, a program written to use the dynamic rotation and editing capabilities of a programmable display such as an IMLAC would operate rather poorly on a storage tube display terminal such as the ARDS.

The motivation to provide a general purpose graphic system is to avoid creating and maintaining a multiplicity of systems, each oriented towards a separate application. The extra overhead borne by the utility is obvious, but also important is the added burden of graphic users having to master the idiosyncrasies of entirely separate systems.

The design problem of any general purpose system is that generality is often opposite to ease and efficiency of use. A system intended to accept a wide variety of tasks may perform few of them well. The Multics Graphic System avoids this dilemma in a compromise fashion. It provides a sophisticated, picture-structure oriented programming interface which is suitable for direct use by a knowledgeable programmer. However, those desiring a more application oriented or simpler interface may use instead application modules that sit between a user program and the general graphic interface. The programming

interfaces of these application modules present a view of the graphic system more structured towards the user's needs. Currently, both a casual user module and a plotting module exist.

Conceptually, the Multics Graphic System can be viewed as a double-ended funnel implementing a switching function between a particular application interface and a particular terminal type.



A Conceptual View of the Organization
of the Multics Graphic System

The central portion is the core graphic system, through which everything is channeled. At the user end, the funnel expands to include a number of different application interface modules. The user is to select the most appropriate of these for use by his graphic program. At the system end, the funnel expands to include interface modules for all the different graphic terminals attached to the utility. The user exchanges graphic communications with his terminal through one of these.

More extensive graphic system information and graphic system module write-ups are available in the MPM Graphics Users' Supplement.

(END)*

WRITING AN I/O SYSTEM INTERFACE MODULE

An I/O System Interface Module (IOSIM) is responsible for coordinating all the activities of a particular device or pseudo-device. The interface module must functionally provide some or all of the services called for in the general specifications of the I/O system. These services are described in the MPM section for the subroutine ios_. In order that the I/O switch be able to call interface modules, these modules must have fixed calling sequences. These calling sequences contain basically the same information as the original call to the I/O switch (ios_). Since the same interface module may be used to control several devices simultaneously, it is necessary that the IOSIM maintain separate data about the status of each device. It is the job of the interface module to create space for and maintain the individual device data, but it is the responsibility of the I/O switch to keep track of the stream, device, and device data associations. Device data is kept in a region called the Stream Data Block (SDB) and contains information describing the status of each device and other information for use by both the interface module and the I/O switch. As an example of device status information, consider the typewriter interface module, which maintains the channel number of the typewriter being accessed and the event ID of the event which is signalled when an interrupt of interest comes from this typewriter. Information shared by both the interface module and the switch is kept at the beginning of the SDB in a fixed format (discussed later in this section). This information consists of the name of the interface module and a list of names of devices or streams upon which I/O operations are to be performed for calls to this stream. Each call by the switch to an interface module has as an argument a pointer to the SDB for the stream referred to in this call. This SDB pointer is passed to the interface module by the I/O switch in addition to the arguments received by the I/O switch in the original call to the I/O system.

Each interface module is called in a standard manner by the I/O switch. For the sake of efficiency, the calls to the interface module are directed through a transfer vector. Each interface module has a transfer vector. The transfer vector is an assembly language program consisting of a sequence of transfer instructions, each one transferring to a different entry in the corresponding interface module. The beginning of the transfer vector is identified by the external symbol whose name is formed by concatenating the name of the interface module and the string "module". The name of the transfer vector is the name of the interface module. The I/O switch calls a particular entry in an interface module by transferring to a particular location relative to the beginning of the transfer vector; therefore each

location in each transfer vector in the system corresponds to a particular entry in the interface module and is fixed by system-wide convention. A typical transfer vector is shown in Figure 1. Notice that some locations in the vector transfer to a small subroutine. This routine returns an error in the status argument indicating that this interface module does not contain this entry. Figure 2 shows a template transfer vector which shows the position of each interface module entry in the transfer vector. Note that since the name of the interface module is the name on the transfer vector, the name of the procedure the transfer vector transfers to must be distinct and not the name of the interface module. In Figure 1 this procedure is named typewriter_util_. There may, however, be several procedures involved, possibly each I/O system call being implemented by a separate program.

For example, the transfer vector for the typewriter_ interface module shown in Figure 1 contains the external symbol typewriter_module. When a call is made to this interface module, the I/O switch will transfer to an offset relative to the location corresponding to this external symbol. The specific offset depends upon the particular call being made.

Each interface module is responsible for returning the proper status string. (See the MPM Reference Guide Section on the Use of the Input and Output System.) The first 36 bits (first word) of the status string is an error code. A 0 error code indicates no error. The error code may be either a standard Multics error code (see the MPM Reference Guide Section on System Error Codes and Meanings) or a bit string representing the status of a physical device. The latter case is used by Device Interface Modules (DIM) in physical mode and is indicated by the first bit of the status string being one. The exact interpretation of the physical status bits is dependent on the interface module in question. The next 18 bits indicate the state of the transaction. One of these bits is the so called det bit. If the interface module turns the bit on, the I/O switch will delete this device attachment. Normally, only the detach entry to the interface module turns the det bit on. The final 18 bits of the status string is the transaction identification and is a unique string used to identify this transaction. The use of the transaction identification is not fully specifed yet and therefore should not be used.

The attach entry of an IOSIM must perform certain tasks. The requested attachment may be a multiple attachment, i.e., an attempt to attach a device or stream on a stream to which some other device or stream is already attached. This may be detected

by the interface module by looking at the SDB pointer passed to it by the I/O switch. If the SDB pointer is not a null pointer, then an SDB already exists for this stream, indicating a multiple attachment. If the interface module does not allow multiple attachments, an appropriate error code must be returned in the status string. If the interface module does allow multiple attachments, or the SDB pointer passed by the switch is null, then the interface module may proceed with the attachment. If a physical device is being attached, the interface module must call the I/O channel controller interface module to initialize the device. If a pseudo-device or intermediate interface module is involved, then whatever initialization that is appropriate must be performed. If no errors have occurred, the interface module must then update the information in the SDB. If this is the first attachment to this stream, then the interface module must allocate space for the SDB and return a pointer to the SDB to the switch by overwriting the null SDB pointer passed to it by the switch. One means of performing allocation is by using the PL/1 allocate statement. In order that both the interface module and the I/O switch may share certain data, it is necessary that this data be kept in a fixed format (see Figure 3) at the beginning of the SDB. The shared data includes the name of the interface module and a list of the devices or streams to which this stream is attached. A list of device or stream names is necessary to allow multiple attachments which permit an I/O call to one stream to fan out to many devices or streams (i.e., broadcasting). This shared information is most easily allocated and updated by the interface module, but is needed by the switch in order that it may inform the user as to the name of the interface module and the devices or streams to which a particular stream is attached. Once the information in the SDB has been updated, the information in the status string argument must be updated and the interface module may then return to the switch. If an uncorrectable error occurs at any point in the attachment, the interface module should return an appropriate error code in the status string. If the interface module operates in physical mode, it may choose to return the physical status of the device involved in place of an error code. It indicates this by turning on the first bit of the status string. If the attachment was not successful, and no other devices or streams are already attached to this stream, the SDB should be freed and the det bit of the status string should be turned on, indicating to the switch that the attachment should be destroyed.

For most of the other types of I/O calls, the actions taken depend upon the device being referenced and the nature of the call. As an example, consider a read call to a stream that is

attached to a typewriter. The interface module must reference the device data pointed to by the SDB pointer supplied by the switch to retrieve the logical channel of this device. The IOSIM then calls the channel controller module to read in the data (the issue of buffering has been ignored for this example). If nothing is read, the IOSIM calls ipc_$block to wait for more input. Here it is again necessary to reference the device data to get the event identifier of the event which indicates that more input has arrived. This event identifier is passed to ipc_$block so that the process is awakened only upon the occurrence of this event. Upon begin awakened, the interface module again calls the channel controller interface module to read the data and then fills in the status string appropriately.

In a detach call, the device must be terminated. For a physical device, this will require that the interface module call the channel controller module. In all cases, the interface module must update the status string and, if the detachment is successful and no other devices or streams are currently attached to this stream, deallocate the SDB and store a one in the det bit of the status string. Upon performing a detachment, the caller may not wish to completely terminate a device. For example, in detaching a tape, the caller may not wish to unload or even rewind the tape because it intends to continue using the tape. For this purpose, the detach call has a disposal argument. It is here that the caller may specify the extent to which the device is to be terminated. The default, i.e., a null string, is full termination. The effect of specifying any other disposal is dependent upon the interface module.

Calling sequences for each interface module entry are fixed by system convention. The arguments for each entry are the same as the arguments to the I/O switch for the same entry with the following exceptions:

1) The attach entry in the interface module contains an extra argument at the end of the argument list which is a pointer. On initial attachment on a stream, the interface module returns in this argument a pointer to the SDB it has allocated for this device. On all subsequent calls on this stream, the SDB pointer is passed back to the interface module.

2) All other entries to the interface module have the SDB pointer as their first argument. The SDB pointer argument replaces the first argument in the calling sequence for the same entry in the I/O switch.

Examples

Two entry statements in the typewriter outer module might be as follows:

```
typewriter_attach:  entry (stream, type, device, mode,
                    status, data_ptr);

typewriter_read:  entry (data_ptr, workspace, offset,
                  nelements, nactual, status);
```

```
"Interface Module Transfer Vector for the
"typewriter_ Interface Module

     entry       typewriter_module
typewriter_module:
     tra  *+1,6     go to proper transfer instruction

     tra  <typewriter_util_>|[typewriter_attach]
     tra  <typewriter_util_>|[typewriter_detach]
     tra  <typewriter_util_>|[typewriter_read]
     tra  <typewriter_util_>|[typewriter_write]
     tra  <typewriter_util_>|[typewriter_abort]
     tra  <typewriter_util_>|[typewriter_order]
     tra  <typewriter_util_>|[typewriter_resetread]
     tra  <typewriter_util_>|[typewriter_resetwrite]
     tra  <ios_>|[no_entry]  this entry not implemented
     tra  <ios_>|[no_entry]  this entry not implemented
     tra  <typewriter_util_>|[typewriter_setdelim]
     tra  <typewriter_util_>|[typewriter_getdelim]
     tra  <ios_>|[no_entry]  this entry not implemented
     tra  <ios_>|[no_entry]  this entry not implemented
     tra  <ios_>|[no_entry]  this entry not implemented
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this entry not implemented
     tra  <ios_>|[no_entry]  this entry not implemented
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused

     end
```

Figure 1: A Typical Transfer Vector

```
"Interface Module Transfer Vector for the
"xyz_ Interface Module

     entry       xyz_module
xyz_module:
     tra  *+1,6     go to proper transfer instruction

     tra  <xyz_util_>|[xyz_attach]
     tra  <xyz_util_>|[xyz_detach]
     tra  <xyz_util_>|[xyz_read]
     tra  <xyz_util_>|[xyz_write]
     tra  <xyz_util_>|[xyz_abort]
     tra  <xyz_util_>|[xyz_order]
     tra  <xyz_util_>|[xyz_resetread]
     tra  <xyz_util_>|[xyz_resetwrite]
     tra  <xyz_util_>|[xyz_setsize]
     tra  <xyz_util_>|[xyz_getsize]
     tra  <xyz_util_>|[xyz_setdelim]
     tra  <xyz_util_>|[xyz_getdelim]
     tra  <xyz_util_>|[xyz_seek]
     tra  <xyz_util_>|[xyz_tell]
     tra  <xyz_util_>|[xyz_changemode]
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <xyz_util_>|[xyz_readsync]
     tra  <xyz_util_>|[xyz_writesync]
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused
     tra  <ios_>|[no_entry]  this slot currently unused

     end
```

Figure 2: A Template Transfer Vector

```
declare 1 stream_data_block aligned based (sdb_pointer),
          2 outer_module_name char(32),
          2 device_name_list ptr,
          2 ---------;
```

1) outer_module_name — is the name of the interface module.

2) device_name_list — is a pointer to the threaded list of device names.

3) --------- — the remaining information for this stream is at the discretion of the interface module.

```
declare 1 device_name aligned based,
          2 next_ptr ptr,
          2 name_size fixed bin,
          2 name char (name_size) aligned;
```

1) next_ptr — is a pointer to the next entry. This is null if this is the last entry.

2) name_size — is the number of characters in the name.

3) name — is the name of the device or stream.

Figure 3: Declaration for a Stream Data Block

(END)*

ASCII CHARACTER SET

The Multics standard character set is the revised U.S. ASCII Standard (refer to USA Standards Institute, "USA Standard X3.4-1968"). The ASCII set consists of 128 seven bit characters. Internally these are stored right justified in four nine bit fields per word. The two high order bits in each field are expressly reserved for expansion of the character set; no system program shall use them. Any hardware device which is unable to accept or create the full character set should use established escape conventions for representing the entire set. It is emphasized that there are no meaningful subsets of the revised ASCII character set.

Included in the ASCII character set are 94 printing graphics, 33 control characters, and the space. Multics conventions assign precise interpretations to all the graphics, the space, and 11 of the control characters. One of these control characters is the "Enter Graphic Mode" character which is recognized only on a graphics device. The remaining 22 control characters are presently reserved. The graphics in the set are as follows:

Upper Case Alphabet	Lower Case Alphabet
ABCDEFGHIJKLMNOPQRSTUVWXYZ	abcdefghijklmnopqrstuvwxyz

Digits

0123456789

Special Characters

! exclamation point	; semicolon
" double quote	< less than
# number sign	= equals
$ dollar sign	> greater than
% percent	? question mark
& ampersand	@ commercial at
' acute accent	[left bracket
(left parenthesis	\ left slant
) right parenthesis	] right bracket
* asterisk	^ circumflex
+ plus	_ underline
, comma	` grave accent
- minus	{ left brace
. period	| vertical line
/ right slant	} right brace
: colon	~ tilde

Control Characters

The following conventions define the standard meaning of the ASCII control characters which are given precise interpretations in Multics. These conventions will be followed by all standard Device Interface Modules (DIM) and by all system software inside the I/O system interface. Since some devices have different interpretations for some characters, it is the responsibility of the appropriate DIM to perform the necessary translations.

The characters designated as "not used" are specifically reserved and may be assigned definitions at any time. Until defined, "not used" control characters will be output using the octal escape convention in normal output and not printed in edited mode. Users wishing to assign interpretations for a "not used" character must use a non-standard DIM.

If a device does not perform a function implied by a control character, its standard DIM will provide a reasonable interpretation for the character on output. This may be substituting one or more characters for the one, or printing an octal escape, or ignoring it.

The Multics standard control characters are:

BEL Sound an audible alarm.

BS Backspace. Move the carriage back one space. The backspace character implies overstrike rather than erase.

HT Horizontal tab. Move the carriage to the next horizontal tab stop. Multics standard tab stops are at 11, 21, 31... when the first column is numbered 1. This character is defined not to appear in a canonical string.

NL "New line". Move the carriage to the left end of the next line. This implies a carriage return plus a line feed. ASCII LF (octal 012) is used for this character.

VT Vertical tab. Move the carriage to the next vertical tab stop and to the left of the page. Standard tab stops are at lines 11, 21, 31... when the first line is numbered 1. This character is defined not to appear in a canonical string.

NP New page. Move the carriage to the top of the next page and to the left of the line. ASCII FF (octal 014) is used for this character.

CR Carriage return. Move the carriage to the left of the current line. This character is defined not to appear in a canonical string.

RRS Red ribbon shift. ASCII SO (octal 016) is used for this character.

BRS Black ribbon shift. ASCII SI (octal 017) is used for this character.

PAD Padding character. This is used to fill out words which contain fewer than four characters and which are not accompanied by character counts. This character is discarded when encountered in an output line and cannot appear in a canonical character string. ASCII DEL (octal 177) is used for this character.

Non-Standard Control Character

One control character is recognized under certain conditions by all DIMs because of its wide use outside Multics. This character is handled specially only when the DIM is printing in edited mode, i.e., it is ignoring unavailable control functions. This character is

NUL Null character. ASCII character NUL (octal 000) is used for this purpose. In normal mode, this character is printed with an octal escape sequence; in edited mode, it is treated exactly as PAD. This character cannot appear in a canonical character string.

Programmers are warned against using NUL as a routine padding character and using edited mode on output because all strings of zeros, including mistakenly uninitialized strings, will be discarded.

Graphic Control Characters

Character code 037 octal (ASCII US) is used to escape into graphic output mode when a graphic display terminal is in use; it is treated as "not used" and an octal escape is provided when an ordinary typewriter is in use. Details on Multics graphic output mode may be found in the Graphic Users' Supplement to the MPM.

ASCII Character Set
Standard Data Formats and Codes
Page 4

Not Used Characters

These characters are reserved for future use:

SOH	001	ACK	006	DC4	024	EM	031
STX	002	DLE	020	NAK	025	SUB	032
ETX	003	DC1	021	SYN	026	ESC	033
EOT	004	DC2	022	ETB	027	FS	034
ENQ	005	DC3	023	CAN	030	GS	035
						RS	036

Notes

The vertical line has two representations on current console devices. It may be represented as either a solid vertical line (|) or a broken vertical line (¦). These are represented identically internally by octal value 174.

ASCII Character Set on Multics

	0	1	2	3	4	5	6	7
000	(NUL)							BEL
010	BS	HT	NL	VT	NP	CR	RRS	BRS
020								
030								EGM
040	Space	!	"	#	$	%	&	'
050	(	)	*	+	,	-	.	/
060	0	1	2	3	4	5	6	7
070	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	[	\	]	^	_
140	`	a	b	c	d	e	f	g
150	h	i	j	k	l	m	n	o
160	p	q	r	s	t	u	v	w
170	x	y	z	{	¦	}	~	PAD

ASCII Character Set
Standard Data Formats and Codes
Page 6

Multics Definitions

NUL Null character (edited output mode only)
BEL Alarm
BS Backspace
HT Horizontal Tab
NL New Line (carriage return and line feed)
VT Vertical Tab
NP New Page (carriage return and form feed)
CR Carriage Return
RRS Red Ribbon Shift
BRS Black Ribbon Shift
EGM Enter Graphic Mode
PAD Padding Character

(END)*

PUNCHED CARD CODES

This write-up defines standard card punch codes to be used in representing ASCII characters for use with Multics. Since the card punch codes are based on the punch codes defined for the IBM EBCDIC standard, a correspondence between the EBCDIC and ASCII character sets is defined automatically.

Notes

The Multics standard card punch codes described here are not identical to the currently proposed ASCII punched card code. The proposed ASCII standard code is not supported by any currently available punched card equipment; until such support exists, it is not a practical standard for Multics work. The Multics standard card punch code described here is based on the widely available card handling equipment used with IBM System/360 computers. The six characters for which the Multics standard card code differs from the ASCII card code are noted in the table below.

EBCDIC and ASCII

The character set used for symbolic source programs and input/output on Multics is the American National Standard Code for Information Interchange, X3.4-1968, known as ASCII. This set is described in the MPM Reference Guide section, ASCII Character Set.

Similarly, the character set used for input/output with some devices from a System/360 computer is the IBM standard, known as EBCDIC. This set is described on page 150.3 of the IBM Systems Reference Library Manual "IBM System/360 Principles of Operation", A22-6821-7.

EBCDIC is an eight-bit code for which graphics* have been assigned to 88 code values, controls to 51 code values, and card codes to all 256 possible values. ASCII is a seven-bit code with graphics assigned to 94 code values and controls to 34.

---

* By way of terminology, the characters are divided into two groups, named graphics and controls (including the space). The graphics are further divided into alphabetic (upper and lower case), numeric, and special subgroups.

Although there are 85 graphics in common between EBCDIC and ASCII, there is no practical algorithm by which one can deduce an EBCDIC code value from the ASCII code value (or vice versa), short of a complete table look-up. That is to say, the numerical values of the two codes are more or less completely unrelated.

Graphic Correspondence

On the other hand, since there are so many common graphics, one can define a correspondence between at least the graphic parts of the two codes, and thereby establish conventions for communication between computers using the codes. Simultaneously, a card punch code for ASCII is defined, as mentioned above, that has the practical advantage of equipment available in quantity using these card codes. Table 1 provides this correspondence as used on Multics.

In interpreting Table 1, it is helpful to observe that the correspondence between ASCII Code Value in column one and ASCII Meaning in column two is firmly defined by the ASCII standard. Similarly, correspondence among Corresponding EBCDIC Meaning in column three, EBCDIC Code Value in column four, and EBCDIC/Multics Punch Code in column five is firmly defined by the IBM standard. This table provides a correspondence between the first two columns on the one hand, and the last three on the other hand, based on graphic similarities and other suggestions, as noted.

The graphic correspondence in Table 1 is derived as follows: 85 ASCII graphic characters correspond directly with identical EBCDIC graphics. Three ASCII graphics are made to correspond with the three remaining EBCDIC graphics as follows:

ASCII	EBCDIC
acute accent	apostrophe
left slant	cent sign
circumflex	negation

Thus all 88 EBCDIC graphics have an equivalent ASCII graphic. The remaining six ASCII graphics, namely:

left and right square brackets
left and right braces
grave accent
overline (tilde)

have no EBCDIC graphic equivalent. In Table 1 they are made to correspond to unassigned EBCDIC codes which, nevertheless, have well-defined card punch code equivalents. Where possible, the unassigned EBCDIC codes chosen result in the same punch card representation as in the proposed ASCII standard card code. Thus a majority of the Multics standard card codes do, in fact, agree with the proposed standard.

The programmer faced with the problem of representing ASCII data in the EBCDIC environment must make some arbitrary decisions if he needs to obtain graphic representation of these six characters. One appropriate technique is that the suggested "illegal" code be used wherever EBCDIC code representation is required (e.g., in cards or in core memory), but, when printing readable output, the illegal codes be printed as escapes or overstrikes.

For example, choosing the cent sign as an escape character, one has the following graphic representation borrowed from Multics conventions.

ASCII Graphic	EBCDIC Escape Representation
left brace	¢(
right brace	¢)
tilde	¢t
left accent	¢'
left bracket	¢<
right bracket	¢>
left slant	¢134

The last escape is required in order to insure unambiguous meaning of the cent sign as an escape character.

Alternatively, one can propose a series of overstrike graphics which are more suggestive of the ASCII graphics being represented. For example,

ASCII Graphic	EBCDIC Overstrike Representation
left brace	¢ (left parenthesis over minus sign)
right brace	¢ (right parenthesis over minus sign)
left bracket	¢ (left parenthesis over equals sign)
right bracket	¢ (right parenthesis over equals sign)
grave accent	¢ (apostrophe over minus sign)
tilde	¢ (double quote over negation sign)

These two alternatives suggested for printing readable output in an EBCDIC environment are mirrored in the Multics card input conventions (based on card punching with EBCDIC equipment). Either the multicolumn escape sequences described above or the single column multiple punch codes (with meaningless graphics printed on the card, of course) can be used to represent these characters.

Control Character Correspondence

The 34 ASCII control characters and 51 EBCDIC control characters match in 33 cases. The remainder have no correspondence that can be expected to work in most cases.

As a result, the programmer transforming character data from one environment to another must study the precise meaning of the control codes in the new environment. For example, some EBCDIC control codes might logically transform into ASCII hardware escape sequences for some hardware devices. Other controls might not be imitable in the new environment and might instead be printed with graphic escape sequences, or possibly ignored.

Since, in general, it is awkward to hand punch the card codes which correspond to the controls, it should be noted that for Multics input, control codes can be punched as graphic octal escape sequences. Also, the end of a card is interpreted as a new line character.

Eight-Bit Environment

In the System/360 Manual, "Principles of Operation", there is published a code table labeled USASCII-eight. This table purports to show how the seven-bit ASCII code is represented in an eight-bit environment. It is obtained by taking ASCII, interchanging bits 6 and 7, and duplicating bit 7 as bit 8. This method of representing ASCII in an eight-bit environment is not an ANSI standard but rather an IBM suggestion (resulting from a nine-track tape design problem) which has no official sanction. The ANSI standard way of representing ASCII code in an eight-bit environment is by setting bit 8 to zero. (See, for example, USAS X3.22-1967, paragraph 6.4.3, describing nine-track tape standards.) In any case, the Multics standard for representation of ASCII codes in the nine-bit environment of the Honeywell 6180 is seven-bit codes right adjusted in nine-bit fields, with leading zeros.

Bibliography

1) "USA Standard Code for Information Interchange", USAS X3.4-1968, American National Standards Institute, October 10, 1968.

2) IBM SRL, "IBM System/360 Principles of Operation", form A22-6821-7, September 1968, pp. 149-150.3.

3) "Proposed American Standard: Twelve-Row Punched-Card Code for Information Interchange", Communications of the ACM, June 1966, pp. 450-459, obsolete.

4) "USA Standard Recorded Magnetic Tape for Information Interchange (800 CPI, NRZI)", USAS X3.22-1967, American National Standards Institute, November 13, 1967.

5) "Proposed American Standard: Twelve-Row Punched-Card Code", ASA document X3.2/532, not yet published. Latest proposed ANSI standard card code. (Replaces published reference 3), above.)

Table 1: Correspondence Between ASCII Characters and EBCDIC Characters

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
000	(NUL)	NUL	00	9-12-0-8-1	
001	(SOH)	SOH	01	9-12-1	
002	(STX)	STX	02	9-12-2	
003	(ETX)	ETX	03	9-12-3	
004	(EOT)	EOT	37	9-7	
005	(ENQ)	ENQ	2D	9-0-8-5	
006	(ACK)	ACK	2E	9-0-8-6	
007	BEL	BEL	2F	9-0-8-7	
010	BS	BS	16	9-11-6	
011	HT	HT	05	9-12-5	
012	NL(LF)	NL	15	9-11-5	(Note 1)
013	VT	VT	0B	9-12-8-3	
014	NP(FF)	FF	0C	9-12-8-4	
015	(CR)	CR	0D	9-12-8-5	
016	RRS(SO)	SO	0E	9-12-8-6	
017	BRS(SI)	SI	0F	9-12-8-7	
020	(DLE)	DLE	10	12-11-9-8-1	
021	(DC1)	DC1	11	9-11-1	

ASCII code values are in octal; EBCDIC code values are in hexadecimal.

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
022	HLF(DC2)	DC2	12	9-11-2	
023	(DC3)	TM	13	9-11-3	(Note 3)
024	HLR(DC4)	DC4	3C	9-8-4	
025	(NAK)	NAK	3D	9-8-5	
026	(SYN)	SYN	32	9-2	
027	(ETB)	ETB	26	9-0-6	
030	(CAN)	CAN	18	9-11-8	
031	(EM)	-	19	9-11-8-1	
032	(SUB)	SUB	3F	9-8-7	
033	(ESC)	ESC	27	9-0-7	
034	(FS)	IFS	1C	9-11-8-4	
035	(GS)	IGS	1D	9-11-8-5	
036	(RS)	IRS	1E	9-11-8-6	
037	(US)	IUS	1F	9-11-8-7	
040	Space	Space	40	(No punches)	
041	!	!	5A	11-8-2	(Note 1)
042	"	"	7F	8-7	
043	#	#	7B	8-3	
044	$	$	5B	11-8-3	

ASCII code values are in octal; EBCDIC code values are in hexadecimal.

Punched Card Codes
Standard Data Formats and Codes
Page 8

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
045	%	%	6C	0-8-4	
046	&	&	50	12	
047	'	'	7D	8-5	Maps ASCII accute accent into EBCDIC apostrophe
050	(	(	4D	12-8-5	
051	)	)	5D	11-8-5	
052	*	*	5C	11-8-4	
053	+	+	4E	12-8-6	
054	,	,	6B	0-8-3	
055	-	-	60	11	
056	.	.	4B	12-8-3	
057	/	/	61	0-1	
060	0	0	F0	0	
061	1	1	F1	1	
062	2	2	F2	2	
063	3	3	F3	3	
064	4	4	F4	4	
065	5	5	F5	5	
066	6	6	F6	6	

ASCII code values are in octal; EBCDIC code values are in hexadecimal.

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
067	7	7	F7	7	
070	8	8	F8	8	
071	9	9	F9	9	
072	:	:	7A	8-2	
073	;	;	5E	11-8-6	
074	<	<	4C	12-8-4	
075	=	=	7E	8-6	
076	>	>	6E	0-8-6	
077	?	?	6F	0-8-7	
100	@	@	7C	8-4	
101	A	A	C1	12-1	
102	B	B	C2	12-2	
103	C	C	C3	12-3	
104	D	D	C4	12-4	
105	E	E	C5	12-5	
106	F	F	C6	12-6	
107	G	G	C7	12-7	
110	H	H	C8	12-8	
111	I	I	C9	12-9	

ASCII code values are in octal; EBCDIC code values are in hexadecimal.

Punched Card Codes
Standard Data Formats and Codes
Page 10

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
112	J	J	D1	11-1	
113	K	K	D2	11-2	
114	L	L	D3	11-3	
115	M	M	D4	11-4	
116	N	N	D5	11-5	
117	O	O	D6	11-6	
120	P	P	D7	11-7	
121	Q	Q	D8	11-8	
122	R	R	D9	11-9	
123	S	S	E2	0-2	
124	T	T	E3	0-3	
125	U	U	E4	0-4	
126	V	V	E5	0-5	
127	W	W	E6	0-6	
130	X	X	E7	0-7	
131	Y	Y	E8	0-8	
132	Z	Z	E9	0-9	
133	[	None	8D	12-0-8-5	May be punched as ¢<. (Notes 1,2)

ASCII code values are in octal; EBCDIC code values are in hexadecimal.

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
134	\	¢	4A	12-8-2	Maps ASCII left slant into EBCDIC cent sign. Used as an escape. (Note 1)
135	]	None	9D	12-11-8-5	May be punched as ¢>. (Notes 1,2)
136	^	¬	5F	11-8-7	Maps ASCII circumflex into EBCDIC negation.
137	_	-	6D	0-8-5	
140	`	None	79	8-1	May be punched as ¢'. (Note 2)
141	a	a	81	12-0-1	
142	b	b	82	12-0-2	
143	c	c	83	12-0-3	
144	d	d	84	12-0-4	
145	e	e	85	12-0-5	
146	f	f	86	12-0-6	
147	g	g	87	12-0-7	
150	h	h	88	12-0-8	
151	i	i	89	12-0-9	

ASCII code values are in octal; EBCDIC code values are in hexadecimal.

Punched Card Codes
Standard Data Formats and Codes
Page 12

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
152	j	j	91	12-11-1	
153	k	k	92	12-11-2	
154	l	l	93	12-11-3	
155	m	m	94	12-11-4	
156	n	n	95	12-11-5	
157	o	o	96	12-11-6	
160	p	p	97	12-11-7	
161	q	q	98	12-11-8	
162	r	r	99	12-11-9	
163	s	s	A2	11-0-2	
164	t	t	A3	11-0-3	
165	u	u	A4	11-0-4	
166	v	v	A5	11-0-5	
167	w	w	A6	11-0-6	
170	x	x	A7	11-0-7	
171	y	y	A8	11-0-8	
172	z	z	A9	11-0-9	
173	{	None	C0	12-0	May be punched as ¢(. (Note 2)
174	¦	|	4F	12-8-7	(Note 1)

ASCII code values are in octal; EBCDIC code values are in hexadecimal.

ASCII Code Value	ASCII Meaning	Corresponding EBCDIC Meaning	EBCDIC Code Value	EBCDIC/ Multics Punch Code	Comments
175	}	None	D0	11-0	May be punched as ¢). (Note 2)
176	~	None	A1	11-0-1	May be punched as ¢t. (Note 2)
177	PAD(DEL)	DEL	07	12-7-9	

Note 1: In the punched card code proposed for ASCII in reference 5), a different card code is used for this character.

Note 2: This graphic does not appear in (or map into any graphic which appears in) the EBCDIC set; it is assigned to an otherwise illegal EBCDIC code value/card code combination.

Note 3: In some applications, the ASCII meaning of this control character may not correspond to the EBCDIC meaning of the corresponding control character.

Where the Multics meaning of a control character differs from the ASCII meaning, the ASCII meaning is given in parentheses.

Table 2: Summary of Extensions to EBCDIC to Obtain Multics Standard Codes

ASCII Character	Unassigned EBCDIC Card Code Chosen	
open bracket	12-0-8-5	
left slant	12-8-2	
close bracket	12-11-8-5	
grave accent	8-1	*
open brace	12-0	*
close brace	11-0	*
overline/tilde	11-0-1	*
acute accent	8-5	*
circumflex	11-8-7	*

* Same as the ASCII choice for this graphic.

Table 3: Summary of Differences Between Multics Standard Card Codes and Proposed ASCII Standard Card Codes

ASCII Character	Multics Standard Card Code	ASCII Standard Card Code
new line	11-9-5	0-9-5
exclamation point	11-8-2	12-8-7
open bracket	12-0-8-5	12-8-2
left slant	12-8-2	0-8-2
close bracket	12-11-8-5	11-8-2
vertical line	12-8-7	12-11

MULTICS STANDARD MAGNETIC TAPE FORMAT

This write-up describes the standard physical format to be used on 7-track and 9-track magnetic tapes on Multics. Any magnetic tape not written in the standard format described here is not a Multics standard tape.

Standard Tape Format

The first record on the tape following the beginning of tape (BOT) mark is the tape label record. Following the tape record is an end of file (EOF) mark. Subsequent reels of a multireel sequence also have a tape label followed by EOF. (An EOF mark is the standard sequence of bits on a tape that is recognized as an EOF by the hardware.)

Following the tape label and its associated EOF are the data records. An EOF is written after every 128 data records with the objective of increasing the reliability and efficiency of reading and positioning within a logical tape. Records that are repeated because of transmission, parity, or other data alerts, are not included in the count of 128 records. The first record following the EOF has a physical record count of 0 mod 128.

An end of reel (EOR) sequence is written at the end of recorded data. An EOR sequence is:

EOF mark

EOR record

EOF mark

EOF mark

Standard Record Format

Each physical record consists of a 1024-word (36864-bit) data space enclosed by an 8-word header and an 8-word trailer. The total record length is then 1040 words (37440 bits). The header and trailer are each 288 bits. This physical record requires 4680 frames on 9-track tape and 6240 frames on 7-track tape. This is approximately 5.85 inches on 9-track tape at 800 bpi and 7.8 inches on 7-track tape at 800 bpi, not including interrecord gaps. (Record gaps on 9-track tapes are approximately 0.6 inches and on 7-track tapes are approximately 0.75 inches, at 800 bpi.)

For 1600 bpi 9-track tape, the record length is approximately 2.925 inches (with an interrecord gap of approximately 0.5 inches).

Physical Record Header

The following is the format of the physical record header:

Word 1: Constant with octal representation 670314355245.

Words 2 and 3: Multics standard unique identifier (70 bits, left justified). Each record has a different unique identifier.

Word 4: Bits 0-17: the number of this physical record in this physical file, beginning with record 0.

Bits 18-35: the number of this physical file on this physical reel, beginning with file 0.

Word 5: Bits 0-17: the number of data bits in the data space, not including padding.

Bits 18-35: the total number of bits in the data space. (This should be a constant equal to 36864.)

Word 6: Flags indicating the type of record. Bits are assigned considering the leftmost bit to be bit 0 and the rightmost bit to be bit 35. Word 6 also contains a count of the rewrite attempt, if any.

Bit	Meaning if Bit is 1
0	This is an administrative record (one of bits 1 through 13 is 1).
1	This is a label record.
2	This is an end of reel (EOR) record.
3-13	Reserved.
14	One or more of bits 15-26 are set.

15 This record is a rewritten record.

16 This record contains padding.

17 This record was written following a hardware end of tape (EOT) condition.

18 This record was written synchronously; that is control did not return to the caller until the record was written out.

19 The logical tape continues on another reel (defined only for an end of reel record).

20-26 Reserved.

27-35 If bits 14 and 15 are 1, this quantity indicates the number of the attempt to rewrite this record. If bit 15 is 0, this quantity must be 0.

Word 7: Contains the checksum of the header and trailer excluding word 7; i.e., excluding the checksum word. (See the MPM Reference Guide section, Standard Checksum, for a description of standard checksum computation.)

Word 8: Constant with octal representation 512556146073.

Physical Record Trailer

The following is the format of the trailer:

Word 1: Constant with octal representation 107463422532.

Words 2 and 3: Standard Multics unique identifier (duplicate of header).

Word 4: Total cumulative number of data bits for this logical tape (not including padding and administrative records).

Word 5: Padding bit pattern (described below).

Word 6: Bits 0-11: reel sequence number (multireel number), beginning with reel 0.

Bits 12-35: physical file number, beginning with physical file 0 of reel 0.

Word 7: The number of the physical record for this logical tape, beginning with record 0.

Word 8: Constant with octal representation 265221631704.

Note: The octal constants listed above were chosen to form elements of a single-error-correcting code whether read as 8-bit tape characters (9-track tape) or as 6-bit tape characters (7-track tape).

Administrative Records

The standard tape format includes two types of administrative records: 1) a tape label record; or 2) an EOR record.

The administrative records are of standard length: 8-word header, 1024-word data area, and 8-word trailer.

The tape label record is written in the standard record format. The data space of the tape label record contains:

Words 1-8: 32-character ASCII installation code. This identifies the installation that labelled the tape.

Words 9-16: 32-character ASCII reel identification. This is the reel identification by which the operator stores and retrieves the tape.

The remaining words are a padding pattern.

The end of reel record contains only padding bits in its data space. The standard record header of the EOR record contains the information that identifies it as an EOR record (word 6, bits 0 and 2 are 1).

Density and Parity

Both 9-track and 7-track standard tapes are recorded in binary mode with odd ones having lateral parity. Standard densities are 800 frames per inch (bpi) (recorded in NRZI mode) and 1600 bpi (recorded in PE mode).

Data Padding

The padding bit pattern is used to fill administrative records and the last data record of a reel sequence.

Write Error Recovery

Multics standard tape error recovery procedures differ from the past standard technique in that no attempt is made to backspace the tape on write errors. If a data alert occurs while writing a record, the record is rewritten. If an error occurs while rewriting the record, that record is again rewritten. As many attempts as desired can be made to write the record. No backspace record operation is performed.

The above write error recovery procedure is to be applied to both administrative records and data records.

Compatibility Consideration

Software shall be capable of reading Multics Standard tapes that are written with records with less than 1024 words in their data space. In particular, a previous Multics standard tape format specified a 256-word (9216-bit) data space in a tape record.

 (END)

MULTICS STANDARD DATA TYPE FORMATS

This section describes the representation of Multics standard data types. See the MPM Reference Guide section Subroutine Calling Sequences for a discussion of data descriptors. In the following discussion let p be the declared precision of an arithmetic datum. Let n be the declared length of a string datum, and let k be the declared size of an area datum.

Any scaling factor declared for a fixed-point datum is not stored with the datum. The scaling factor is applied to the value of the datum when the value participates in a computation or conversion.

Real Fixed-Point Binary Short (descriptor type 1)

A real, fixed-point, binary, unpacked datum of precision $0<p<36$ is represented as a 2's complement, binary integer stored in a 36-bit word.

A real, fixed-point, binary, packed datum of precision $0<p<36$ is represented as a 2's complement, binary integer stored in a string of p+1 bits.

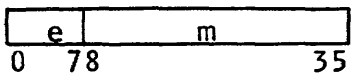
Real Fixed-Point Binary Long (descriptor type 2)

A real, fixed-point, binary, unpacked datum of precision $35<p<72$ is represented as a 2's complement, binary integer stored in a pair of 36-bit words the first of which has an even address.

A real, fixed-point, binary, packed datum of precision $35<p<72$ is represented as a 2's complement, binary integer stored in a string of p+1 bits.

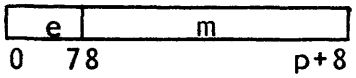
Real Floating-Point Binary Short (descriptor type 3)

A real, floating-point, binary, unpacked datum of precision $0<p<28$ is represented as a 2's complement, binary fraction m and a 2's complement, binary integer exponent e stored in a 36-bit word of the form:



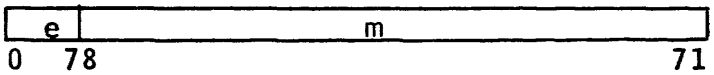
The value 0 is represented by m=0 and e=-128. For all other values, m satisfies $1/2 \leq |m| < 1$.

A real, floating-point, binary, packed datum of precision 0<p<28 is represented as a 2's complement, binary fraction m and a 2's complement, binary, integer exponent e stored in a string of p+9 bits.



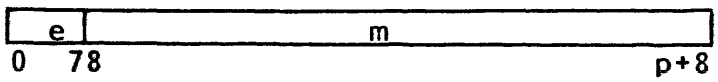
The value 0 is represented by m=0 and e=-128. For all other values, m satisfies $1/2 \leq |m| < 1$.

A real, floating-point, binary, unpacked datum of precision 27<p<64 is represented as a 2's complement, binary fraction m and a 2's complement, binary, integer exponent e stored in a pair of 36-bit words the first of which has an even address.



The value 0 is represented by m=0 and e=-128. For all other values, m satifies $1/2 \leq |m| < 1$.

A real, floating-point, binary, packed datum of precision 27<p<64 is represented as a 2's complement, binary fraction m and a 2's complement, binary, integer exponent e stored in a string of p+9 bits.



The value 0 is represented as m=0 and e=-128. For all other values, m satisfies $1/2 \leq |m| < 1$.

Complex Fixed-Point Binary Short (descriptor type 5)

A complex, fixed-point, binary, unpacked datum of precision 0<p<36 is represented as a pair of 2's complement, binary integers stored in a pair of 36-bit words the first of which has an even address. The first integer is the real part of the complex value and the second integer is the imaginary part of the complex value.

A complex, fixed-point, binary, packed datum of precision 0<p<36 is represented as a pair 2's complement, binary integers stored in a string of 2(p+1) bits. The first p+1 bits contain the integer representation of the real part and the second p+1 bits contain the integer representation of

the imaginary part.

Complex Fixed-Point Binary Long (descriptor type 6)

A complex, fixed-point, binary, unpacked datum of precision 35<p%< IS REPRESENTED AS A PAIR OF <'s complement, binary integers stored in 4 consecutive 36-bit words the first of which has an even address. The first two words contain the integer representation of the real part and the last two words contain the integer representation of the imaginary part.

A complex, fixed-point, binary, packed datum of precision 35<p<72 is represented as a pair of 2's complement, binary integers stored in a string of 2(p+1) bits. The first p+1 bits contain the integer representation of the real part and the last p+1 bits contain the integer representation of the imaginary part.

Complex Floating-Point Binary Short (descriptor type 7)

A complex, floating-point, binary, unpacked datum of precision 0<p<28 is represented as a pair of real, floating-point, binary, unpacked data stored in two 36-bit words the first of which has an even address. The first word contains the real part of the complex value and the second word contains the imaginary part of the complex value.

A complex, floating-point, binary, packed datum of precision 0<p<28 is represented as a pair of real, floating-point, binary, packed data stored in a string of 2(p+9) bits. The first p+9 bits contain the real part of the complex value and the last p+9 bits contain the imaginary part of the complex value.

Complex Floating-Point Binary Long (descriptor type 8)

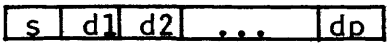
A complex, floating-point, binary, unpacked datum of precision 27<p<64 is represented as a pair of real, floating-point, binary, unpacked data stored in 4 consecutive 36-bit words the first of which has an even address. The first two words contain the real part of the complex value and the last two words contain the imaginary part of the complex value.

A complex, floating-point, binary, packed datum of precision 27<p<64 is represented as a pair of real, floating-point, binary, packed data stored in 2(p+9) bits. The first p+9 bits contain the real part of the complex value and the last p+9 bits contain the imaginary part of the complex value.

Real Fixed-Point Decimal (descriptor type 9)

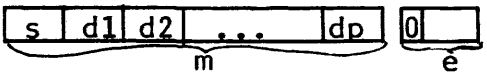
A real, fixed-point, decimal datum (packed or unpacked) of precision p is represented as a signed, decimal integer stored as a string of p+1 characters. The leftmost character is either a "+" or a "-", and all other characters are from the set "0123456789".

An unpacked, decimal datum is aligned on a word boundary and occupies an integral number of words, some bytes of which may be unused.



Real Floating-Point Decimal (descriptor type 10)

A real, floating-point, decimal datum (packed or unpacked) of precision p is represented as a signed, decimal integer m and a 2's complement, binary, integer exponent e stored as a string of characters of the form:



The exponent e is right justified within the last 9-bit character and the unused bit is zero.

An unpacked, decimal datum is aligned on a word boundary and occupies an integral number of words, some bytes of which may be unused.

Complex Fixed-Point Decimal (descriptor type 11)

A complex, fixed-point, decimal datum (packed or unpacked) of precision p is represented as a pair of real, fixed point, packed, decimal data of precision p. The first represents the real part of the complex value, and the second represents the imaginary part of the complex value.

An unpacked, complex, decimal datum is aligned on a word boundary and occupies an integral number of bytes, some of which may be unused.

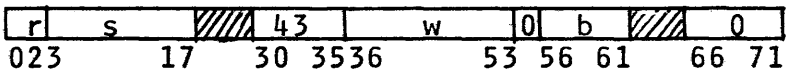
Complex Floating-Point Decimal (descriptor type 12)

A complex, floating-point, decimal datum (packed or unpacked) of precision p is represented by a pair of real, floating-point, packed, decimal data of precision p. The first represents the real part of the complex value and the last represents the imaginary part of the complex value.

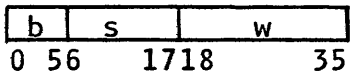
An unpacked, complex, decimal datum is aligned on a word boundary and occupies an integral number of bytes, some of which may be unused.

Pointer (descriptor type 13)

An unpacked pointer datum is represented by a ring number r, a segment number s, a word offset w, and a bit offset b, stored in a pair of 36-bit words the first of which has an even address.

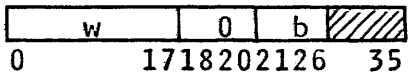


A packed pointer datum is represented by a segment number s, a word offset w, and a bit offset b, stored as a string of 36-bits.



Offset (descriptor type 14)

An offset datum (packed or unpacked) is represented by a word offset w, and a bit offset b, stored in a single 36-bit word.



Label (descriptor type 15)

A label datum (packed or unpacked) is represented by a pair of unpacked pointers. The first pointer identifies a statement within a procedure and the second pointer identifies a stack frame of an activation of the block immediately containing the statement identified by the first pointer.

Entry (descriptor type 16)

An entry datum (packed or unpacked) is represented by a pair of unpacked pointers. The first pointer identifies an entry to a procedure and the second identifies a stack frame of an activation of the block immediately containing the procedure whose entry is identified by the first pointer. If the first pointer identifies an entry to an external procedure, the second pointer is null.

Structure (descriptor type 17)

A structure is an ordered sequence of scalar data. A packed structure contains only packed data, whereas an unpacked structure contains either packed or unpacked data or both. An unpacked structure contains at least one unpacked datum.

A structure is aligned on a storage boundary that is the most stringent boundary required by any of its components. A packed structure that is not a member of a structure is aligned on a word boundary.

An unpacked member of a structure is aligned on a word or double word boundary depending on its data type and occupies an integral number of words.

A packed member of a structure is aligned on the first unused bit following the previous member, except that up to 8 bits may be unused in order to insure that decimal arithmetic or non-varying string datum is aligned on a 9-bit byte boundary.

An unpacked structure occupies an integral number of words.

Area (descriptor type 18)

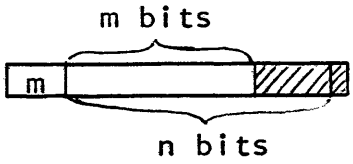
An area datum (packed or unpacked) whose declared size is k occupies k words of storage, the first of which has an even address. The content of these k words is not yet defined as a Multics standard.

Bit-String (descriptor type 19)

A bit-string (packed or unpacked) whose length is n occupies n consecutive bits. The leftmost is bit 1 and the rightmost is bit n. An unpacked bit-string is aligned on a word boundary and occupies an integral number of words. Some bits of the last word may be unused.

Varying Bit-String (descriptor type 20)

A varying bit-string (packed or unpacked) whose maximum length is n is represented by a real, fixed-point, binary short, unaligned integer followed by a nonvarying bit-string of length n.



The length of the current value is m. A varying bit-string is aligned on a word boundary and occupies an integral number of words, the last of which may contain unused bits.

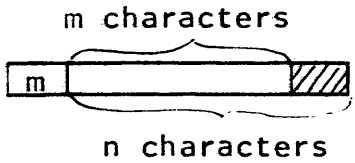
Character-String (descriptor type 21)

A character-string (packed or unpacked) whose length is n occupies n consecutive 9-bit bytes. Each byte contains a single 7-bit ASCII character right justified within the byte. The two unused bits must be zero.

An unpacked character-string is aligned on a word boundary and occupies an integral number of words, the last of which may contain unused bytes.

Varying Character-String (descriptor type 22)

A varying character-string (packed or unpacked) whose maximum length is n is represented by a real, fixed-point, binary, short, unaligned integer followed by a nonvarying character-string of length n.



The length of the current value is m.

A varying character-string is aligned on a word boundary and occupies an integral number of words the last of which may contain unused bytes.

File (descriptor type 23)

A file datum (packed or unpacked) is represented by a pair of unpacked pointers the second of which points to a file-state block and the first of which points to a bit-string. Neither the form of the file-state block nor the form of the bit-string are defined as Multics standards.

Arrays

An array is an n-dimensional, ordered collection of scalars or structures, all of which have identical attributes. The elements of an array are stored in row major order. (When accessed sequentially the rightmost subscript varies most rapidly).

Summary of Data Descriptor Types

1	real fixed-point binary short
2	real fixed-point binary long
3	real floating-point binary short
5	complex fixed-point binary short
6	complex fixed-point binary long
7	complex floating-point binary short
8	complex floating-point binary long
9	real fixed-point decimal
10	real floating-point decimal
11	complex fixed-point decimal
12	complex floating-point decimal
13	pointer
14	offset
15	label
16	entry
17	structure
18	area
19	bit-string
20	varying bit-string
21	character-string
22	varying character-string
23	file

(END)*

STANDARD SEGMENT FORMATS

The Multics storage system does not make any restrictions or assumptions about the nature or format of the data stored in segments in the hierarchy. It does provide four length attributes for each segment giving some information about the contents: bit count, current length, maximum length, and number of records occupied. For a discussion of these and other attributes see the MPM Reference Guide section Segment, Directory and Link Attributes.

The system commands and subroutines which deal with the contents of segments, however, expect them to be in one of a small number of formats:

1) Object (procedure) segments for execution as machine instructions or as external-reference data (i.e., of the form alpha$beta), with special characteristics to satisfy the pure procedure and dynamic linking requirements of Multics.

2) Archive segments for combining other segments (of any format) into a single segment, thus reducing the number of segments in a directory and eliminating the wasted bits at the end of the last record of each segment.

3) ASCII character string segments for editing, printing, and as input to various Multics commands (e.g., the standard Multics language translators).

4) Data segments peculiar to individual commands or subsystems. These have no standard format.

Object Segments

A Multics object segment contains an array of machine-executable and relocatable (to permit binding) words divided into four sections.

1) The text section contains the pure executable part of the object program: instructions, read-only constants and relative pointers into the other sections.

2) The definition section contains non-executable read-only symbolic information to be used in dynamic linking and symbolic debugging.

3) The linkage section contains impure data: links which are snapped at execution time, and internal storage which exists beyond a single invocation of the program.

4) The symbol section contains any pure data which does not belong in the other three sections. In particular, it contains a map of all other sections of the object segment and the symbol tree (a description of source language variables).

For a full description of object segments, see the MPM Reference Guide section Standard Object Segments.

Archive Segments

A segment in archive format consists of the individual component segments in linear juxtaposition, each component preceded by a header giving the name of the component, its length (in bits), and the time of creation of the component, as well as certain constant information used to verify that the segment is in fact correctly archived. All of the per-component header information is maintained as ASCII characters, so that if all the component segments are ASCII character string segments, the entire segment may be printed. Each component segment will be separated from the preceding one by a new page, since the first character in each header is the ASCII "new page" character. The name of the component will appear roughly in the center of the first line on the page, the actual contents beginning three lines later.

The terminal component of the name of an archive segment is ".archive". Using the archive command, components may be added, deleted or replaced in an archive segment, and a table of the contents of the archive segment may be obtained. Archive segments are also used as input to the Multics binder (see the MPM Reference Guide section Standard Object Segments), the component segments of the archive being the object segments to be bound together. Note that archiving is similar to binding, but that an archive segment may not be properly executed as a procedure since no relocation is done on the data inside, and internal addressing references are thus incorrect.

ASCII Character String Segments

Segments which are to be edited, printed, or used as input for standard translators and other commands are composed of strings of characters of the ASCII character set. Each 7-bit character is right justified in a 9-bit field, with the two high-order bits set to 0 and reserved for compatibility and future USASCII extensions. There are four characters per machine word, accessed sequentially from left to right, the first character of each word occupying the leftmost nine bits. The

segment's bit count is set by the creator of the segment to indicate 9 times the number of characters in the segment.

ASCII segments to be used as input to system commands frequently have very rigid format requirements. In these cases, the command write-ups adequately describe the format. Language translator formats are usually described in a separate (non-MPM) document which is referenced in the write-up of the command which invokes the translator. The translators which require ASCII character string input segments are pl1, fortran, alm, and basic. Other commands which require such input are bind, enter_abs_request, exec_com, help, peruse_text, runoff and set_search_rules.

Many ASCII commands produce ASCII segments intended for printing. While these segments have a definite format, that format is prepared by the command, and the user need not know it in detail. These commands are the language translators, bind, the absentee facility, runoff and mail.

Most of the ASCII input and output segments discussed above have reserved suffixes on their names. These suffixes are listed in the MPM Reference Guide section List of Names with Special Meanings.

Other

The user may encounter some other segments which fit into none of the above categories. Some of these are:

1) The breaks segment produced by the debug command to record information about breakpoints.

2) The saved environment produced by the lisp and apl commands which may be reentered by a later invocation of lisp or apl.

3) A message segment (with last component name .ms) which is an Administrative Ring segment and has its own managing subroutine. It is not accessible to the user.

4) The user profile which is maintained by the abbrev and check_info_segs commands.

(END)

STRATEGIES FOR HANDLING UNUSUAL OCCURRENCES

During its execution, a procedure may encounter a set of circumstances which prohibit it from continuing in the normal manner. Examples of such circumstances are an attempt to divide by zero and being unable to find a necessary segment in the storage system. Clearly, whether or not a particular set of circumstances, such as those given above, prohibits a procedure from continuing in a normal manner is dependent upon the procedure in question. Circumstances that are abnormal for one procedure may be quite normal when encountered in a different procedure. If it is unable to continue in the normal manner, a procedure will want to notify its caller or others of its ancestors that an unusual occurrence has taken place. This section, and the immediately following sections, describe means by which procedures can handle such occurrences and notify their ancestors of such occurrences. The means used to handle and send notification of unusual occurrences depends on many things, such as the significance of the effect of the occurrences, the expected frequency of the occurrences, the nature of the environment in which the program is executing, the nature of the occurrences, the ability to modify the circumstances, etc.

The discussion of the methods of unusual occurrence reporting described below will enable users to understand how to handle unusual occurrences reported by system procedures. Also, the discussion will enable users to better select an appropiate means for handling and reporting unusual occurrences that may arise during the execution of their own procedures.

Printed Messages

The type of unusual occurrence reporting that most Multics users first encounter is a message printed on their terminals. Since, in some sense, the caller of a command is the user himself, printing a message on the user's terminal is the means by which a command can report an unusual occurrence to its caller. There are essentially two general types of printed messages used to report unusual occurrences: statements and questions. A statement is simply a description of the unusual occurrence that informs the user that the occurrence has been encountered. If the user wishes to take action to rectify the circumstances of the occurrence, he must subsequently issue commands to do so. A question gives a description of the occurrence, but also requests an immediate response from the user in the form of a character string entered at the terminal. In this way, the user must immediately specify one of several courses of action that the command will take with respect to the occurrence.

Most Multics system commands generate such printed messages in a standard format. For statements, this format consists of the name of the command printing the statement, a description of the unusual occurrence causing the message to be printed, and more detailed information about the occurrence, if appropriate. For questions, the format consists of the name of the command asking the question, followed by the question itself. The question contains sufficient descriptive information about the unusual occurrence so that the user can supply an intelligent reply. Two procedures, com_err_ and command_query_ (see the MPM subroutine write-ups), are provided to help report unusual occurrences through printed statements and questions. Their use is strongly encouraged because they provide many facilities other than simple formatting.

Status Codes

For passing descriptions of unusual occurrences between procedures, the character string is too cumbersome and inefficient. For this purpose, a coded description of the unusual occurrence, called the status code, is used. The status code is either a short bit string or arithmetic number which takes on a different value for each possible unusual occurrence. If the status code is a bit string, usually each bit refers to the occurrence of some circumstance, as in Multics I/O system status. If the status code is an arithmetic number, then each different value corresponds to an unusual occurrence or set of unusual occurrences, as in the case of the Multics storage system codes. The status codes are passed from a calling procedure to the called procedure as an argument. The called procedure assigns the appropriate value to the argument at some point during its execution. When the called procedure returns to the calling procedure, the calling procedure examines the status code to determine what unusual occurrence has been encountered, if any, and then takes special action, if desired. Note that the status code is a means by which a called procedure may report an unusual occurrence only to its immediate ancestor. However, the first ancestor may, in turn, reflect the status code to its immediate ancestor, and so on.

Multics provides a means by which status codes may be generated and interpreted. The status codes generated by this facility are fixed binary(35) (one word) arithmetic numbers whose scope is a single process. The actual values of the codes are generated dynamically when referenced symbolically from a program, and may be interpreted (i.e., converted to a character string description) by calling com_err_. By using these dynamically generated status codes rather than status codes with

fixed, preassigned values, one avoids the problem of conflict between several separately compiled subsystems which may all use the same status code to represent different occurrences. In the dynamic scheme, all status codes are guaranteed to be unique within a process. Note that status codes cannot be used in a process other than the one generating them because they will not have the same interpretation in another process.

In order to have a status code generated, a Multics standard status code segment must exist. (A description of how to generate a standard status code segment is given in the MPM Subsystem Writers' Supplement (SWS).) This segment contains an externally defined symbol corresponding to each status code to be generated in the segment, as well as space for the code itself, and the character string interpretation of the code. When the status code segment is first referenced in a process, the system generates a new value for each status code defined in the segment, and stores it into the segment. (Actually, it is stored into the linkage section of the status code segment, so that a different status code may be generated for each process.) From then on, all references to that external symbol will be referencing the generated status code. com_err_, when given such a status code, is able to locate and return the associated character string interpretation.

When a program wishes to reference a status code, it must do so symbolically. If, for example, a program wished to return a status code appearing in the status code segment "mistake" and having the external symbol "bad_argument", then the following PL/I statements would be needed:

```
declare mistake$bad_argument fixed bin(35) external;

return (mistake$bad_argument);
```

If a program wanted to examine a status code for a particular value to determine if it should take some distinct action, it would contain statements like:

```
declare mistake$bad_argument fixed bin(35) external;

if status_code = mistake$bad_argument then do;
```

Note that all references to the status code are symbolic and that the mechanism of generating the status code is automatic and not visible to the program or programmer at all.

Most Multics system procedures use standard status codes. A list containing the symbolic names, character string interpretations, and meaning of the status codes returned by system procedures is given in the MPM Reference Guide section, List of System Status Codes and Meanings.

Important Note

Some of the documents describing MPM subroutines declare their status code argument (usually called "code") as fixed bin or fixed bin(17). This is a vestige of an earlier form of status codes. Users should declare all status codes returned by MPM subroutines as fixed bin(35), although any of the three declarations will generally work correctly.

Conditions

Status codes enable an ancestor procedure to take action on an unusual occurrence only after the procedure encountering the occurrence has returned. It is sometimes necessary for an ancestor procedure to gain control immediately upon encountering an unusual occurrence, so that it may decide what action to take. If the ancestor procedure decides to take corrective action, it may then continue execution from the point of the occurrence. This is the purpose of the Multics condition mechanism as described, in detail, in the MPM Reference Guide section, The Multics Condition Mechanism.

The Multics system invokes the condition mechanism upon encountering certain unusual occurrences during the execution of a process. The Multics standard user enviroment acts upon these system generated occurrences, as well as user program generated occurrences, if the user programs do not do so themselves. A list of occurrences which cause the system to invoke the condition mechanism, and the action taken by the Multics standard user environment if it is invoked to act upon these occurrences, is given in the MPM Reference Guide section, List of System Conditions and Default Handlers.

Faults

There is a class of unusual occurrences that are detected by the Multics hardware processor. These occurrences are called faults, and are a subset of the set of occurrences that cause the system to invoke the condition mechanism. They are, therefore, included in the List of System Conditions and Default Handlers in the MPM Reference Guide.

(END)

THE MULTICS CONDITION MECHANISM

The condition mechanism is a facility of the Multics system that notifies a program of an exceptional condition detected during its execution. A condition is a state of the executing process. Each condition that is detected is identified by a condition name. For example, division by zero is a condition identified by the condition name, zerodivide. An attempt by a user to exceed his storage allocation limit is a condition identified by the name, record_quota_overflow.

A condition can be detected by the system or by a user program. When a condition is detected, it is signalled. A signal causes a block activation of the most recently established on unit for the condition. Thus, by establishing an on unit, a program arranges with the system to receive control when conditions of interest to it are detected and signalled.

An on unit can be a begin block or independent statement, or it can be a procedure entry. A program (an activation of a procedure block or begin block) can establish a begin block or an independent statement as an on unit for a particular condition by executing an PL/I on statement that names that condition. A program can establish a procedure entry as an on unit by calling the condition_ subroutine with the condition name as an argument. (See the MPM subroutine write-up.)

When an on unit is activated, it can take any action to handle a condition. Typically, the on unit might try to rectify the circumstances that caused the condition and then restart execution of the interrupted program at the point where the condition was detected; or it might abort execution of the program by performing a nonlocal transfer to a location within the interrupted program or to one of its callers.

All of the on units established by a block activation are reverted when that block activation terminates by returning to its caller or when it is aborted by a nonlocal transfer. An on unit for a particular condition can be explicitly reverted. If the on unit is a begin block or an independent statement, it can be reverted by executing a PL/I revert statement, or by executing another on statement, naming the condition. If it is a procedure entry, it can be reverted by calling the reversion_ subroutine, or by calling the condition_ subroutine again, with the condition name as an argument. (See the MPM subroutine write-up for reversion_.) Therefore, each block activation can have no more than one on unit established for each condition at any given time; however, there can be as many on units established for a particular condition as there are block activations. Signalling

a condition causes a block activation of the most recently established on unit for that condition. Normally, this is the only on unit that is activated, even though other on units for the condition were established by preceding block activations.

The effect of this scheme is that, once a block activation has established an on unit for a condition, any occurrence of the condition activates that on unit. This remains true only until the block activation is terminated or until the on unit is reverted and while no descendant block activation establishes an on unit for the condition.

The general philosophy of establishing on units is that procedures that can take action when a condition is detected should establish an on unit for that condition. Of those block activations that have established an on unit for the condition, the most recently established on unit is activated since the most recent is probably the best qualified to handle the condition.

The conditions detected and signalled by the system are listed in the MPM Reference Guide section, List of System Conditions and Default On Unit Actions. Methods of signalling conditions from user programs are discussed later in this write-up.

An Example of the Condition Mechanism

The example below is presented to illustrate the mechanism discussed above. It is not meant to illustrate typical or recommended use of the condition mechanism.

```
Example:  proc;

          declare Sub1 external entry;
          declare Sub2 external entry;
          declare c fixed bin;
          declare wrong_way condition;

          on wrong_way begin;                    (1)
            .
            .
            .
          end;

          call Sub1;                             (2)

          c = 2;                                 (3)
```

```
        call Sub2;                                  (4)

        end Example;


Sub1:   proc;

        declare a fixed bin;
        declare wrong_way condition;

        a = 0;                                      (S1)

        on wrong_way begin;                         (S2)
          .
          .
          .
        end;

        a = 1;                                      (S3)

        end Sub1;


Sub2:   proc;

        declare b fixed bin;
        declare wrong_way condition;

        b = 1;                                      (S4)

        on wrong_way begin;                         (S5)
          .
          .
          .
        end;

        b = 2;                                      (S6)

        revert wrong_way;                           (S7)

        b = 3;                                      (S8)

        end Sub2;
```

In the above example, if procedure Example is called, the executable statements are executed in the order, (1), (2), (S1), (S2), (S3), (3), (4), (S4), (S5), (S6), (S7), (S8), under normal

circumstances. However, if the wrong_way condition is detected and signalled during the execution of (S1), then the on unit established for wrong_way by Example is activated because Sub1 has not established an on unit for the wrong_way condition at this time. If the on unit simply corrects the circumstances that caused the wrong_way condition and returns, then execution resumes in (S1) from the point of interruption. If condition wrong_way is detected and signalled during the execution of statement (S3), then the on unit estalished in Sub1 is activated because Sub1 has established the most recent on unit for wrong_way. If wrong_way is signalled during (3), the on unit established by Example is activated because the block activation for Sub1 has been terminated and its on unit is no longer established. If wrong_way is signalled during (S8), the on unit established in Example is activated because Sub2 explicitly reverted the on unit it had previously established, making Example's on unit the most recently estalished wrong_way on unit.

An On Unit Activated by All Conditions

The above description inidcates how on units can be established for specific conditions. It is sometimes desirable to handle any and all conditions that occur. To do this, a block activation can establish an on unit for the any_other condition. When a particular condition is signalled, the any_other on unit established by the block activation is activated if no specific on unit for the condition was established by the block activation, and if no on unit for that condition or the any_other condition was established by a more recent block activation. In other words, when a condition is signalled, each block activation, starting with the most recent, is inspected for an on unit established for that specific condition and, if none is found, for an established any_other on unit. The first such specific or any_other on unit found is the one that is activated. Note that, as with on units for specific conditions, only one any_other on unit can be established by a given block activation. Establishing a second any_other on unit simply overwrites the first.

As a summary, the flow diagram of Figure 1 illustrates the algorithm used by the condition mechanism to determine which on unit to activate when a condition is signalled. The action taken when no on unit for a condition can be found is described later in this write-up.

Obtaining Additional Information About a Condition

An on unit might (in fact, probably does) need information about the circumstances under which it was activated. The find_condition_info_ subroutine (described in an MPM subroutine write-up) makes such information available to an on unit. The information might include machine conditions (i.e., the processor state) or other information describing the condition in question. The information that is available when system-detected conditions are signalled is listed in the MPM Reference Guide section, List of System Conditions and Default On Unit Actions.

Interaction with the Multics Ring Structure

The condition mechanism interacts with the Multics ring structure. The above description of how an on unit is selected for activation applies only to block activations within a single ring. When a condition is signalled in a particular ring, the algorithm of Figure 1 is followed for the block activations in that ring. If no on unit for the condition is found in that ring, then the ring is abandoned and the same condition is signalled in the higher ring that called the abandoned ring. This process is repeated until all existing rings have been abandoned, indicating that this process has not established an on unit for the condition being signalled, in which case the process is terminated.

Signalling Conditions in a User Program

A user program can signal a condition by executing a PL/I signal statement that names that condition. Alternately, it can call the signal_ subroutine with the condition name as an argument. (See the MPM subroutine write-up for signal_.) If the on unit activated by the signal returns, the user program should retry the operation that was interrupted by the condition.

Differences Between PL/I and Multics Condition Mechanisms

The PL/I language on, revert, and signal statements are very similar in purpose and function to the Multics condition_, reversion_, and signal_ subroutines and for many applications can be used interchangeably. However, there are important differences between them as noted below, and the user should not interchange them blindly. The signal_ subroutine can be called with arguments that describe the particular circumstances under which a condition is being signalled. (See the MPM subroutine write-up for signal_.) The PL/I signal statement does not accept

arguments. Therefore, a block activation must call signal_ if it wants to pass descriptive arguments when signalling a condition.

Executing a PL/I on statement to establish an on unit is equivalent to calling the condition_ subroutine. An on unit established for a condition by either method is activated when the condition is signalled, either by the execution of a PL/I signal statement or by a call to the signal_ subroutine. However, an on unit established by calling condition_ must be a procedure entry; therefore, it can accept the descriptive arguments passed with the signal by signal_. An on unit established by executing a PL/I on statement must be a begin block or an independent statement; it can refer to the arguments passed by signal_ only by calling the find_condition_info_ subroutine.

A PL/I on statement and a call to the condition_ subroutine must not be executed by the same block activation in order to establish an on unit for a given condition. Also, a PL/I revert statement can only revert on units established by an on statement, but cannot revert on units established by condition_. Similarly, the reversion_ subroutine can only revert on units established by condition_, but cannot revert on units established by an on statement.

In PL/I Version 2, when calls to condition_ or reversion_ appear within the scope of an internal procedure, the no_quick_blocks option must be specified in the procedure statement of that procedure. The no_quick_blocks option is a nonstandard feature of the Multics PL/I language; therefore, programs using it might require conversion when being transferred to other systems.

Action Taken by the Default On Unit

Some conditions are routinely handled by the system's default on unit (in the absence of a user-supplied on unit) by printing a message on the user's terminal to alert him that the condition has occurred and returning to command level. These conditions are denoted by "Default action: prints a message and returns to command level." in the MPM Reference Guide section, List of System Conditions and Default On Unit Actions.

In many cases, the subroutine that is executing when a condition is detected is a system or PL/I support subroutine that is of little interest to the user. In such cases, the user needs to know the location at that the most recent non-support subroutine was executing before the condition was detected. To

fill this need, the default on unit actually hunts through the block activations that precede the support subroutine until it finds the first non-support subroutine; it then indicates that the condition was detected while executing at a location within that non-support subroutine.

Machine Conditions

As described above, information is available that describes the state of the processor at the time a hardware condition (fault) was raised. It has the following declaration:

```
declare  1 mc based (mc_ptr) aligned,
         2 prs (0:7) ptr,
        (2 regs,
          3 x (0:7) bit(18),
          3 a bit(36),
          3 q bit(36),
          3 e bit(8),
          3 pad bit(64),
         2 scu (0:7) bit(36),
         2 pad1 bit(108),
         2 errcode fixed bin(35),
         2 pad2 bit(7 2),
         2 ring bit(18),
         2 fault_time bit(54),
         2 pad3 (0:7) bit(36) ) unaligned;
```

1) prs — is the contents of the 8 pointer registers at the time the condition occurred.

2) regs — is the contents of the other registers at the time the condition occurred.

   a) x — is the contents of the 8 index registers.

   b) a — is the accumulator contents.

   c) q — is the q-register contents.

   d) e — is the exponent register contents.

3) scu — is the stored control unit, expanded below.

4) errcode — is the fault error code. Refer to the MPM Reference Guide section, List of Sysyem Status Codes and Meanings.

5) ring is the ring in which the condition occurred.

6) fault_time is the time the condition occurred.

The stored control unit is declared as follows:

```
declare 1 scu aligned,

          /* WORD (0) */

         (2 ppr,
           3 prr bit(3),
           3 psr bit(15),
           3 p bit(1),
          2 pad4 bit(17),

          /* WORD (1) */

          2 pad5 bit(35),
          2 fi_flag bit(1),

          /* WORD (2) */

          2 tpr,
          3 trr bit(3),
          3 tsr bit(15),
          2 pad6 ibt(18),

          /* WORD (3) */

          2 pad7 bit(30),
          2 tpr_tbr bit(6),

          /* WORD (4) */

          2 ilc bit(18),
          2 ir,
          3 zero bit(1),
          3 neg bit(1),
          3 carry bit(1),
          3 ovfl bit(1),
          3 eovf bit(1),
          3 eufl bit(1),
          3 oflm bit(1),
          3 tro bit(1),
          3 par bit(1),
          3 parm bit(1),
          3 bm bit(1),
```

```
                3 tru bit(1),
                3 mif bit(1),
                3 abs bit(1),
                3 pad bit(4),

                /* WORD (5) */

                2 ca bit(18),
                2 pad8 bit(18),

                /* WORD (6) */

                2 even_inst bit(36),

                /* WORD (7) */

                2 odd_inst bit(36);
```

1) ppr — is the procedure pointer register contents.

   a) prr — is the ring number portion of ppr.

   b) psr — is the segment number portion of ppr.

   c) p — is the procedure privileged bit.

2) fi_flag — equals "1"b after a fault, "0"b after an interrupt.

3) tpr — is the temporary pointer register contents.

   a) trr — is the ring number portion of ptr.

   b) tsr — is the segment number portion of tpr.

4) tpr_tbr — is the bit offset portion of tpr.

5) ilc — is the instruction counter contents.

6) ir — is the contents of indicator registers.

   a) zero — zero indicator.

   b) neg — negative indicator.

   c) carry — carry indicator.

d) ovfl overflow indicator.

e) eovf exponent overflow.

f) eufl exponent underflow.

g) oflm overflow mask.

h) tro tally runout.

i) par parity error.

j) parm parity mask.

k) bm bar mode.

l) tru truncation mode.

m) mif multiword instruction mode.

n) abs absolute mode.

7) ca is the computed address.

8) even_inst the instruction causing the fault is stored here.

9) odd_inst the next sequential instruction is stored here if ilc (see above) is even.

Information Header Format

A standard header is required at the beginning of each information structure provided to an on unit. This information is particular to the condition in question and varies among conditions except for the header. The format of that header is:

```
declare 1 info_structure aligned,
        2 length fixed bin,
        2 version fixed bin,
        2 action_flags aligned,
          3 cant_restart bit(1) unaligned,
          3 default_restart bit(1) unaligned,
          3 pad bit(34) unaligned,
        2 info_string char(256) var,
        2 status_code fixed bin(35),
```

1) length is the length of the structure in words.

2) version — is the version number of this structure.

3) action_flags — indicates appropriate behavior for a handler:

   cant_restart — if "1"b, a handler should never attempt to return to the signalling procedure.

   default_restart — if "1"b, the computation can resume with no further action on the handler's part except a return.

4) info_string — is a printabel message about the condition.

5) status_code — if nonzero, is a code interpretable by com_err_ further defining the condition.

If neither action flag is set, restarting is possible, but its success depends on action taken by the handler.

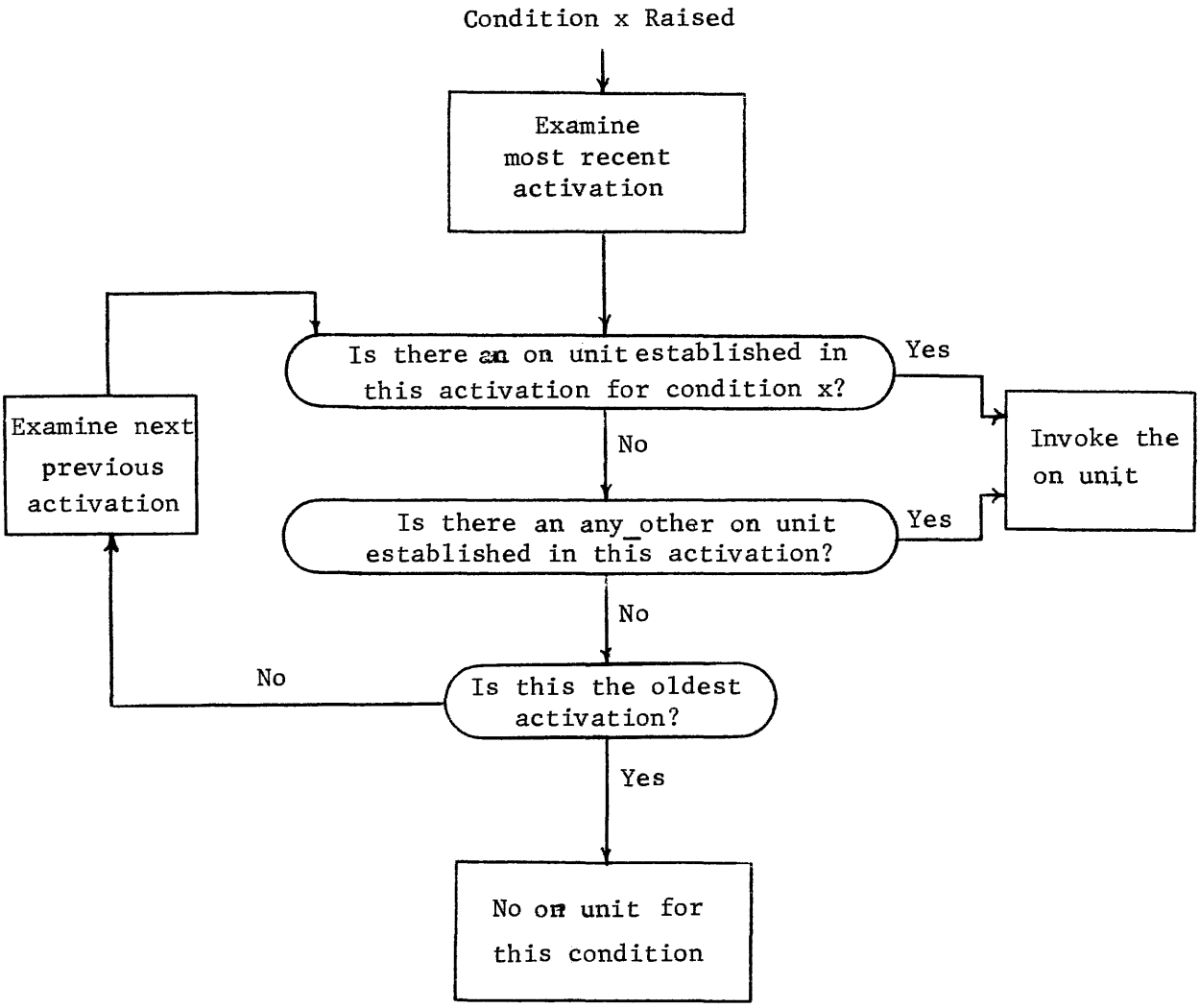


Figure 1: Simplified algorithm for determining which on unit to invoke when condition x is raised.

NONLOCAL TRANSFERS AND CLEANUP PROCEDURES

Many languages provide the ability to perform nonlocal transfers. In Multics, this is a facility by which the currently executing procedure activation may transfer to a location in an earlier existing procedure activation and, as a consequence, abort all activations descendant from the earlier activation. Programmers of certain types of procedures may wish to have these procedures establish a set of code to be executed if an activation of one or more of these procedures is aborted in this manner. An example of such a procedure is a program that references static data that must be reset so that the procedure can be reentered. This function of executing predefined code when an activation is aborted by a nonlocal transfer is termed cleaning up. A procedure or entry that contains the code for cleaning up is termed a cleanup procedure.

A procedure may establish a cleanup procedure by calling the MPM subroutine establish_cleanup_proc_. Having a cleanup procedure established will cause the specified cleanup procedure to be invoked if the establishing block activation is aborted by a nonlocal transfer. The establishment of a cleanup procedure may be reverted by calling the MPM subroutine revert_cleanup_proc_. If a procedure activation is terminated either normally by a return or abnormally by a nonlocal transfer, any established cleanup procedure is automatically reverted. In the latter case of an abnormal termination, the cleanup procedure is automatically reverted when it is invoked.

In PL/I Version 2, when calls to establish_cleanup_proc_ or revert_cleanup_proc_ appear within the scope of a begin block or internal procedure of a procedure, the no_quick_blocks option must be specified in the procedure statement of that procedure. The no_quick_blocks option is a nonstandard feature of the Multics PL/I language and, therefore, programs using it may not be transferable to other systems.

(END)

LIST OF SYSTEM STATUS CODES AND MEANINGS

Status codes report unusual occurences encountered by procedures during execution. The codes are returned by Multics System commands and subroutines. Printed messages which correspond to these status codes appear on printed output in the format consisting of the name of the command printing the statement, a description of the unusual occurrence causing the message to be printed, and more detailed information when appropriate.

To test for the return of a particular system-defined status code, the following approach can be taken in order to avoid compiling particular numeric values, which might change, into programs:

```
declare error_table_$entry

if code = error_table_$entry then ...
```

where

1) code — is a status code (fixed binary(35)) returned from a Multics system command or subroutine.

2) entry — is an error_table_ entry taken from the list below.

See also the MPM Reference Guide section, Strategies for Handling Unusual Occurences, and the subroutine write-up for com_err_. com_err_ reflects to printed output the occurrence and interpretation of any of the status codes.

The first part of this write-up contains an alphabetic list of printed messages. Each message is followed by the name of the entry for the status code in error_table_. The error_table_ entry name is followed by a more extensive interpretation of the status code.

This version of error_table_ is accurate through Multics system 20.12

A call that must be in a sequence of calls was out of sequence.
(error_table_$out_of_sequence)
Meaning: The procedure called required another call to have been made prior to this call.

A logical error has occurred in initial connection.
(error_table_$net_icp_error)
Meaning: Network only. A Network connection management program has detected an error in the execution of Network protocol to establish a connection to a foreign host. Re-try the connection attempt; if the problem persists, it may be a sign that either the foreign host or Multics is not properly following Network protocol.

ACL/CACL is empty.
(error_table_$empty_acl)
Meaning: The ACL of a directory or segment is empty.

Allocation could not be performed.
(error_table_$notalloc)
Meaning: This operation required an allocation in an area that did not contain enough space to perform the allocation.

An event channel is being used in an incorrect ring.
(error_table_$wrong_channel_ring)
Meaning: A channel name was supplied that does not correspond to a channel in the current ring.

An initial connection is already in progress from this socket.
(error_table_$net_already_icp)
Meaning: Network only. The user's process has requested that a Network connection be established from a local socket that is already involved in a connection attempt. If possible, the user should direct his program to use a different socket, or, if appropriate, close the already-established connection if it is no longer desired.

Append permission missing.
(error_table_$no_append)
Meaning: Append permission is missing for an operation that requires it.

Argument ignored.
(error_table_$arg_ignored)
Meaning: An argument was found that was not expected and was ignored.

Argument is not an ITS pointer.
(error_table_$bad_ptr)
Meaning: One of the argument pointers, in the argument list used in a cross-ring call, is not in the correct format.

Argument size too small.
(error_table_$smallarg)
Meaning: The argument size is too small (in length).

Argument too long.
(error_table_$bigarg)

Meaning: An entry name argument greater than 32 characters or a path name argument greater than 168 characters was passed to a program.

Attachment loop.
(error_table_$att_loop)
Meaning: The attempted attachment would result in the given stream being attached to itself, either directly or indirectly, through intermediate outer modules.

Attempt to access beyond end of segment.
(error_table_$boundviol)
Meaning: An attempt was made to access beyond the maximum length of the segment.

Attempt to attach to an invalid device.
(error_table_$invalid_device)
Meaning: The device specified in this I/O system attach call is not of a type handled by the IOSIM to which the attach was directed.

Attempt to change first pointer.
(error_table_$change_first)
Meaning: The type of device associated with the given stream name does not permit the value of the first reference pointer to be changed.

Attempt to convert directory or link to multisegment file.
(error_table_$bad_ms_convert)
Meaning: An unsuccessful attempt was made to convert a directory or link to a multisegment file.

Attempt to delete segment whose safety switch is on.
(error_table_$safety_sw_on)
Meaning: The user attempted to delete a segment, directory, or directory subtree for which the safety switch was on (preventing deletions).

Attempt to execute in data segment.
(error_table_$execute_data)
Meaning: The user has attempted to transfer to a segment to which he does not have execute access.

Attempt to manipulate last or bound pointers for device that was not attached as writeable.
(error_table_$invalid_seek_last_bound)
Meaning: Changing the position of the last or bound reference pointers of a device that cannot be written is nonsensical and is therefore not allowed.

Attempt to read or move read pointer on device which was not attached as readable.
(error_table_$invalid_read)
Meaning: Changing the position of the read reference pointer of a device that cannot be read is nonsensical and is therefore not allowed. From tape_: an attempt was made

to change the mode from write to read during a seek request.

Attempt to set delimiters for device while element size is too large to support search.

(error_table_$invalid_setdelim)

Meaning: The type of device associated with the given stream does not support read delimiters or break characters with the current element size.

Attempt to set max length of a segment less than its current length.

(error_table_$invalid_max_length)

Meaning: The user attempted to set the maximum length of a segment to a value less than its current length.

Attempt to specify the same segment as both old and new.

(error_table_$sameseg)

Meaning: There was an attempt to use a single segment (possibly specified twice) with an operation that requires two different segments (e.g., copying).

Attempt to unlock a lock that was not locked.

(error_table_$lock_not_locked)

Meaning: An attempt was made to unlock a lock that was not locked.

Attempt to unlock a lock which was locked by another process.

(error_table_$locked_by_other_process)

Meaning: An attempt was made to unlock a lock that was locked by another existing process.

Attempt to write or move write pointer on device which was not attached as writeable.

(error_table_$invalid_write)

Meaning: Changing the position of the write reference pointer of a device that cannot be written is nonsensical and is therefore not allowed. From tape_: an attempt was made to change the mode from read to write during a seek request.

Bad class code in definition.

(error_table_$bad_class_def)

Meaning: An object segment containing nonstandard information was referenced.

Bad definitions pointer in linkage.

(error_table_$no_defs)

Meaning: The linkage section has been illegally modified.

Bad gate for entry referenced.

(error_table_$bad_arg_type)

Meaning: A bad argument specification was found in the gate validation information.

Bad mode specification for ACL/CACL.

(error_table_$bad_acl_mode)

Meaning: The user specified an illegal mode or combination of modes in the process of setting an ACL.

Bad socket gender involved in this request.
(error_table_$net_bad_gender)
Meaning: Network only. A Network socket connection is either read-only or write-only; an attempt has been made to perform the opposite operation on a socket.

Bad syntax in pathname.
(error_table_$badpath)
Meaning: A syntax error of the following form was used in a path name: 1) a less than character (<) following a non-less than character (e.g., <a< or <><); 2) two sucessive greater than characters (e.g., >>); or 3) a greater than character immediately following a less than character (e.g., <>).

Brackets do not balance.
(error_table_$unbalanced_brackets)
Meaning: The brackets in a command line do not balance.

Communications with this foreign host not enabled.
(error_table_$net_fhost_inactive)
Meaning: Network only. The user has requested a connection to a foreign Network host with whom Multics does not routinely communicate. If communication with this host is desired, the user should consult the local installation management or Network technical liason.

Connection not completed within specified time interval.
(error_table_$net_timeout)
Meaning: Network only. The user's process has attempted to establish a Network connection to a foreign host, but that host has not responded within a reasonable period of time. Most likely, the foreign host is overloaded, or is about to cease all Network communication for some reason.

Could not create dartmouth job core.
(error_table_$no_job_core)
Meaning: The Dartmouth subsystem encountered a system error while attempting to create temporary segments in the process directory.

Current processid does not match stored value.
(error_table_$bad_processid)
Meanings: The user has tried to use a socket that belongs to some other process.

Dartmouth job aborted.
(error_table_$dart_abort)
Meaning: The Dartmouth subsystem cound not find a Dartmouth command (e.g., basic) in the library; or the Dartmouth subsystem encountered a system error while attempting to creat temporary segments in the process directory.

Data not in expected format.
(error_table_$improper_data_format)

Meaning: From tape_: the tape to be read is not in Multics standard tape format.

Directory or link found in multisegment file.
(error_table_$bad_ms_file)
Meaning: A directory or link was found in a multisegment file. Only segments are permitted as components of a multisegment file.

Directory pathname too long.
(error_table_$dirlong)
Meaning: A specified directory name is greater than 168 characters in length.

Duplicate entry name in bound segment.
(error_table_$dup_ent_name)
Meaning: Two or more components containing entry points with the same name were bound together, and the user referred to bound_seg_name$entry_name so that the appropriate entry point cannot be determined.

Entry is not a branch.
(error_table_$not_a_branch)
Meaning: A storage system entry that is not a branch was used in a context where a branch was expected.

Entry is not a directory.
(error_table_$notadir)
Meaning: A storage system entry that is not a directory was used in a context where a directory was expected.

Entry name too long.
(error_table_$entlong)
Meaning: The specified entry name in a directory is greater than 32 characters in length (perhaps after the addition of a suffix component, e.g.; .pl1, .archive, etc.).

Entry not found.
(error_table_$noentry)
Meaning: The branch specified by the path name does not exist.

Equals convention makes entry name too long.
(error_table_$longeql)
Meaning: The user supplied a name with the equals convention which, when expanded, becomes greater than 32 characters in length.

Error in internal ioat information.
(error_table_$ioat_err)
Meaning: System error. Contact Multics operations.

Error zeroing entry in the linkage offset table.
(error_table_$loterr)
Meaning: The linkage section is in an improper format.

Expanded command line is too large.
(error_table_$command_line_overflow)
Meaning: The evaluation of active functions has overflowed

the available space for command line expansion. See the command set_com_line.

Expected argument descriptor missing.
(error_table_$nodescr)
Meaning: The expected argument descriptor is missing.

Expected argument missing.
(error_table_$noarg)
Meaning: An argument expected by a program was not passed to that program.

External symbol not found.
(error_table_$no_ext_sym)
Meaning: The entry name was not found on the segment being referenced.

Fatal error. Translation aborted.
(error_table_$translation_aborted)
Meaning: The translator has become internally inconsistent, probably not due to the source program, and is reverting directly to command level.

Foreign IMP is down.
(error_table_$net_fimp_down)
Meaning: Network only. The user has attempted to connect to, or has had an open connection to a network host whose Interface Message Processor has gone down. Communication with that host is not possible at this time.

Foreign host is down.
(error_table_$net_fhost_down)
Meaning: Network only. The user has attempted to connect to a Network host which is not presently communicating with the Network, or has had an open Network connection to a foreign host that has just ceased communicating with the Network.

IO device failed to become unassigned.
(error_table_$io_still_assnd)
Meaning: A call to detach an I/O device failed for some reason. From tape_: a different reel ID was specified on detaching the stream than on attaching it.

IO device not currently assigned.
(error_table_$dev_nt_assnd)
Meaning: A call was made to perform I/O using an illegal device.

Illegal entry name.
(error_table_$badstar)
Meaning: A syntax error of the following form appeared in an entry name utilizing the star convention: 1) a null component; 2) a less than (<) or greater than (>) character; 3) a non-printing ASCII character (e.g., tab or new line); 4) more than one component of "**"; or 5) a

component which is not "**" but contains more than one * character.

Illegal entry point name in make_ptr call.
(error_table_$bad_entry_point_name)
Meaning: There is an illegal entry point name in the hcs_$make_ptr call.

Illegal host number or id.
(error_table_$net_bad_host)
Meaning: Network only. The user has specified a Network host name unknown to Multics, or has given a host number which was either negative or greater than the largest host number known to Multics. The user should check the spelling of the name or number.

Illegal initialization info passed with create-if-not-found link.
(error_table_$bad_link_target_init_info)
Meaning: There is an unrecognizable action code in the initialization information for a link target to be created when first linked to.

Illegal self reference type.
(error_table_$bad_self_ref)
Meaning: The object segment is in an invalid format.

Illegal type code in type pair block.
(error_table_$bad_link_type)
Meaning: The object segment is in an invalid format.

Illegal use of equals convention.
(error_table_$badequal)
Meaning: There was no letter or component in the entry name that corresponds to a % or an = in the equal name.

Improper access to given argument.
(error_table_$bad_arg_acc)
Meaning: An argument to which the process does not have access was passed on a cross-ring call.

Improper mode specification for this device.
(error_table_$bad_mode)
Meaning: Either the given I/O mode specification was of illegal syntax or the type of device associated with the given stream does not support one or more of the modes. From tape_: The mode specification on the attach call was other than r or w.

Improper syntax in command name.
(error_table_$bad_command_name)
Meaning: Command name is of the form a$, $a, or $.

Inconsistent combination of control arguments.
(error_table_$inconsistent)
Meaning: The command invocation contained an inconsistent combination of control arguments.

Incorrect access on entry.
(error_table_$moderr)

Meaning: The user attempted to reference a segment to which he has insufficient access for the reference, but has sufficient access to know of its existence.

Incorrect access to directory containing entry.
(error_table_$incorrect_access)
Meaning: The user had incorrect access to the directory containing the entry he wished to access; i.e., the access mode required for the operation was not present.

Incorrect detachable medium label.
(error_table_$bad_label)
Meaning: there is an inconsistency or error in the label on this detachable volume.

Indicated device assigned to another process.
(error_table_$already_assigned)
Meaning: The indicated device is already assigned to another process.

Infinite recursion.
(error_table_$recursion_error)
Meaning: Recursive include segments were encountered while expanding a source segment.

Initial connection socket is in an improper state.
(error_table_$net_ipc_bad_state)
Meaning: Network only. The user's process had made a call to a Network connection management program (such as net_icp_) to establish a socket connection to a foreign host, and some anomalous event has occurred which has left the user's local (Multics) socket in an improper state. The connection attempt should be re-tried; if the problem persists, it should be reported.

Insufficient access to return any information.
(error_table_$no_info)
Meaning: The user had insufficient access to the specified entry or to its superior directory to return any information about the entry.

Internal index out of bounds.
(error_table_$bad_index)
Meaning: The device index given does not correspond to a device owned by this process.

Invalid backspace_read order call.
(error_table_$invalid_backspace_read)
Meaning: The backspace_read request attempts to set the read reference pointer of an I/O system stream to the element after the previous read delimiter; however, no read delimiters currently exist for the given stream.

Invalid element size.
(error_table_$invalid_elsize)
Meaning: The element size specified in this I/O system call

is not valid for the type of device associated with the specified stream.

Invalid mode specified for ACL.
(error_table_$invalid_mode)
Meaning: The user attempted to set m (modify) permission on a directory without setting s (status) permission.

Invalid move of quota would change terminal quota to non terminal.
(error_table_$invalid_move_quota)
Meaning: An attempt was made to move all assigned quota to a superior directory when the quota is still used in an inferior directory.

Invalid project for gate access control list.
(error_table_$invalid_project_for_gate)
Meaning: Access to gates can only be set for users in the same project.

Invalid volume identifier.
(error_table_$bad_volid or error_table_$bad_tapeid)
Meaning: The specified detachable volume name is incorrect or does not match the volume name stored in the volume label.

Ioname already attached and active.
(error_table_$ionmat)
Meaning: An attempt has been made to attach an I/O device to a stream to which no more devices can be attached.

Ioname not active.
(error_table_$ioname_not_active)
Meaning: The stream name specified in a call to the I/O system is not attached to any device. Either the specified stream name or one of the stream names to which it is attached through on intermediate outer module should be attached to a device.

Ioname not found.
(error_table_$ioname_not_found)
Meaning: No stream with the stream name given in the call to the I/O system exists.

Linkage section not found.
(error_table_$no_linkage)
Meaning: Either a data segment has been called or a bad object segment referenced.

Looping searching definitions.
(error_table_$defs_loop)
Meaning. There were too many definitions found in the definitions search. The user should try reducing the number of entry names in the segment. It is also possible that the linkage information has been illegally modified to produce a circular list.

Maximum number of arguments for this command exceeded.

(error_table_$too_many_args)
Meaning: A command which has a maximum on some argument set was called with more arguments of that type than are allowed.

Mismatched iteration sets.
(error_table_$mismatched_iter)
Meaning: The command line contains more than one iteration element and the number of iterations specified by one is not the same as specified by another.

Missing entry in outer module.
(error_table_$missent)
Meaning: The I/O system call is not implemented for the type of device associated with the stream specified in the call.

Mount request could not be honored.
(error_table_$bad_mount_request)
Meaning: Some error has occurred while attempting to perform this mount request.

Mount request pending
(error_table_$mount_pending)
Meaning: The volume requested is currently being mounted as requested; i.e., this is a redundant request.

Multics IMP is down.
(error_table_$imp_down)
Meaning: Network only. The Interface Message Processor that connects Multics to the Network is not operating. All connections with other Network hosts have been broken, and no new connections can be established until the IMP resumes communication with Multics.

Name already on entry.
(error_table_$segnamedup)
Meaning: An attempt was made to add a name to a storage system entry when the name was already on that entry.

Name duplication.
(error_table_$namedup)
Meaning: An attempt was made to add a name to a storage system entry when the name was already on some other entry in the same directory.

Name list exceeds maximum size.
(error_table_$too_many_names)
Meaning: From get_library_source: The user specified too many source segment names or system names.

Name not found.
(error_table_$oldnamerr)
Meaning: An attempt was made to delete from a storage system entry a name that was not on that entry.

Negative number of elements supplied to data transmission entry.

(error_table_$negative_nelem)
Meaning: A negative number of elements specified in an I/O system call is not permitted.

Negative offset supplied to data transmission entry.
(error_table_$negative_offset)
Meaning: Negative offsets to I/O system read or write calls are not permitted.

Network connection closed by foreign host.
(error_table_$net_socket_closed)
Meaning: Network only. A previously open Network connection to a foreign host has been disconnected by that host; communication over that connection has been terminated.

Network control program not in operation.
(error_table_$net_not_up)
Meaning: Network only. The user has attempted to establish a Network connection to a foreign host, but Multics is not presently communicating with the Network.

New offset for pointer computed by seek entry is negative.
(error_table_$new_offset_negative)
Meaning: The value of a reference pointer relative to the first reference pointer can never be negative. The specified seek call would have resulted in such a negative offset.

No PRPH card was found for the requested device.
(error_table_$no_prph_card)
Meaning: No PRPH (peripheral) card was found in the configuration deck for the requested device.

No bases supplied in force call.
(error_table_$force_bases)
Meaning: Insufficient information was supplied to complete an Indirect Through Base (ITB) link.

No device currently available for attachment.
(error_table_$no_device)
Meaning: No free device is currently available for attachment by this process.

No interrupt was received on the designated IO channel.
(error_table_$no_io_interrupt)
Meaning: The DCW list active switch in the printer DCM had not been turned off in 100,000 tests of its status.

No linkage offset table in this ring.
(error_table_$nolot)
Meaning.: There is no linkage offset table in this ring.

No room available for device status block.
(error_table_$no_room_for_dsb)
Meaning: The attachment could not be performed because space for the necessary data was not available. This problem might be corrected by detaching another stream

associated with the same type of device.

No wired structure could be allocated for this device request.
(error_table_$no_wired_structure)
Meaning: All available wired structures for this device type were in use when the device attachment was attempted.

No/bad linkage info in the lot for this segment.
(error_table_$nolinkag)
Meaning: The system could not find the linkage information for a segment.

Not enough room in stack to complete processing.
(error_table_$stack_overflow)
Meaning: The request specified by the user requires too many recursive calls to be processed; e.g., by the command processor.

Not in proper ring bracket to perform desired operation.
(error_table_$not_in_proper_bracket)
Meaning: The ring brackets, independently of the access modes, prevent the desired access to a segment.

Null bracket set encountered.
(error_table_$null_brackets)
Meaning: An active string of the form [ ] was encountered.

Obsolete object segment format.
(error_table_$oldobj)
Meaning: The object segment being referenced is in an obsolete format.

Odd number of arguments.
(error_table_$odd_no_of_args)
Meaning: An attempt was made to call a command that requires pairs of arguments with an odd number of arguments.

Parentheses do not balance.
(error_table_$unbalanced_parentheses)
Meaning: The parentheses in a command line do not balance.

Pathname too long.
(error_table_$pathlong)
Meaning: Total path name is longer than 168 characters.

Physical end of device encountered.
(error_table_$device_end)
Meaning: The physical end of the device (e.g., magnetic tape) was encountered.

Pointer name passed to seek or tell not currently implemented by it.
(error_table_$unimplemented_ptrname)
Meaning: The pointer name passed to the seek or tell call is not currently implemented by it.

Procedure called improperly.
(error_table_$badcall)
Meaning: The procedure was called improperly.

Procedure was not invoked as an active function.
(error_table_$not_act_fnc)
Meaning: A procedure that is intended to be used as an active function was invoked as a normal command.

Process lacks permission to alter device status.
(error_table_$io_no_permission)
Meaning: The user attempted to do something to an I/O device that does not belong to him.

Process not attached to indicated device.
(error_table_$not_attached)
Meaning: The process is not attached to the indicated device.

Quotes do not balance.
(error_table_$unbalanced_quotes)
Meaning: The quotes in a command line do not balance.

Record quota overflow.
(error_table_$rqover)
Meaning: An attempt was made to use more records than the user is permitted by the storage system.

Relevant data terminated improperly.
(error_table_$data_improperly_terminated)
Meaning: From tape_: the end of readable data was reached on read, but the tape had not been properly detached when data was written.

Request for connection refused by foreign host.
(error_table_$net_rfc_refused)
Meaning: Network only. The user has requested a Network connection to a particular socket at a foreign host, and the foreign host has refused that connection request. It may be that the foreign host is not offering the desired Network service at this time, or that no process on that foreign host is listening for connection requests on that socket.

Request for connect received from improper foreign socket.
(error_table_$net_bad_connect)
Meaning: Network only. The user's process was waiting to complete a connection to a foreign Network host, but the connection established was not the expected one. The connection attempt was therefore abandoned. The user should try again.

Request is inconsistent with current state of device.
(error_table_$invalid_state)
Meaning: The operation requested is not possible because the specified device is in a state inconsistent with that operation.

Request is inconsistent with state of socket.
(error_table_$net_invalid_state)
Meaning: Network only. Some operation could not be performed on the user's Network connection, because the

present state of the connection does not allow it. For example, it is not possible to read or write on a Network socket that is not connected to a foreign Network host.

Request not recognized.
(error_table_$request_not_recognized)
Meaning: The user requested a program to perform an action that it was not prepared to perform.

Requested tape backspace unsuccessful.
(error_table_$no_backspace)
Meaning: From nstd_: An unsuccessful backspace to retry writing the record after a bad transmission occurred.

Requested volume is already mounted.
(error_table_$redundant_mount)
Meaning: The tape reel or other volume of detachable nature is already mounted on the drive requested.

Requested volume is not yet mounted.
(error_table_$mount_not_ready)
Meaning: The tape real or other volume of detachable nature is still being mounted on the drive requested.

Ring brackets input to directory control are invalid.
(error_table_$bad_ring_brackets)
Meaning: The ring brackets to be added to an ACL are inconsistent or illegal.

Segment already known to process.
(error_table_$segknown)
Meaning: The segment is already known to the process and the information returned by the call should be assumed to be valid.

Segment is not bound.
(error_table_$not_bound)
Meaning: The referenced segment is not a bound segment.

Segment not found.
(error_table_$seg_not_found)
Meaning: The segment was not found using the user's search rules.

Segment not known to process.
(error_table_$seg_unknown)
Meaning: The user has attempted to terminate a segment that is not known to this process.

Segment not of type specified.
(error_table_$not_seg_type)
Meaning: The segment specified as an argument was not of the type expected; e.g., a segment specified to the mail command as a mailbox was actually a directory.

Some directory in path specified does not exist.
(error_table_$no_dir)
Meaning: The user specified a path name containing a

directory that does not exist.

Specified buffer size too large.
(error_table_$buffer_big)
Meaning: From nstd_: too large a read/write request was passed to the Hardcore Ring tape DCM.

Specified control argument is not implemented by this command.
(error_table_$badopt)
Meaning: A control argument was used that is not applicable to this particular command or, perhaps, a valid control argument was misspelled.

Specified offset out of bounds for this device.
(error_table_$dev_offset_out_of_bounds)
Meaning: From tape_: A nonzero offset was passed to the seek request.

Specified socket not found in network data base.
(error_table_$net_socket_not_found)
Meaning: Network only. An attempt has been made to perform an operation on a Network socket connection of that the Multics Network Control Program has no record. A now-invalid socket identifier may have been retained from a previous invocation of some program in the user's process.

Status permission missing on directory containing entry.
(error_table_$no_s_permission)
Meaning: The user attempted to access an entry in a manner requiring s (status) permission on the superior directory when he did not have permission to that directory.

Strings are not equal.
(error_table_$strings_not_equal)
Meaning: Character or bit strings that should be equal were not equal.

Supplied area too small for this request.
(error_table_$area_too_small)
Meaning: The supplied area is too small for this request.

Supplied identifier already exists in data base.
(error_table_$id_already_exists)
Meaning: An identifier that must only appear once in a data base has appeared more than once.

Supplied machine conditions are not restartable.
(error_table_$no_restart)
Meaning: The user attempted to restart (using the start command) after a fault that cannot be restarted because the machine conditions are invalid.

Symbol segment not found.
(error_table_$no_sym_seg)
Meaning: The symbol segment was not found.

Syntax error in ascii segment.
(error_table_$badsyntax)
Meaning: There was a syntax error in an ASCII segment.

System service process not currently available.
(error_table_$nosys)
Meaning: The requested service normally provided by the system cannot be used at this time.

The NCP could not find a free table entry for this request.
(error_table_$net_table_space)
Meaning: Network only. A system-wide data base maintained by the Network Control Program is full, and no more Network connections can be established at this time. If this problem persists, it should be reported so that increasing Network traffic can be accommodated.

The access name specified has an illegal syntax.
(error_table_$bad_name)
Meaning: An access control name was encountered that was not of the form "person_id.project_id.tag."

The directory hash table is full.
(error_table_$full_hashtbl)
Meaning: A user has tried to add two many names within a directory. (This will not be restricted in the future and the status code will no longer be returned.)

The directory is the ROOT.
(error_table_$root)
Meaning: The root has no branch and therefore, the requested operation does not work for this directory.

The equal name specified had illegal syntax.
(error_table_$bad_equal_name)
Meaning: An equal name: 1) had more than one == component; 2) contained a < or >character; 3) had an illegal character (e.g., tab or new line); 4) had more than one = in a component that was not == ; or 5) had a null component.

The event channel specified is not a valid channel.
(error_table_$invalid_channel)
Meaning: A channel name was supplied to ipc_ that does not correspond to an existing event channel.

The event channel table was full.
(error_table_$ect_full)
Meaning: No more event channels can be created in the current ring unless some existing channels are deleted.

The event channel table was in an inconsistent state.
(error_table_$inconsistent_ect)
Meaning: There is an inconsistency in the data bases of ipc_, probably resulting from interrupting ipc_ while it was running. A new process is recommended.

The initial connection has not yet been completed.
(error_table_$net_ipc_not_concluded)
Meaning: Network only. The user's process has made a call to a Network connection management program (such as

net_ipc_) to gain control over a socket connection on which a connection attempt has been initiated but has not yet been completed. The user's process should wait for a wakeup from the connection management program before making this call.

The lock could not be set in the given time.
(error_table_$lock_wait_time_exceeded)
Meaning: An attempt was made to lock a lock already locked by another process and the lock was not relinquished to the current process in the specified amount of time.

The lock was already locked by this process.
(error_table_$locked_by_this_process)
Meaning: An attempt was made to lock a lock that was already locked by the current process.

The lock was locked by a process that no longer exists, therefore the lock was reset.
(error_table_$invalid_lock_reset)
Meaning: An attempt was made to lock a lock that was already locked by another process; however, the other process does not currently exist so the lock was forcibly locked for the current process.

The maximum depth in the storage system hierarchy has been exceeded.
(error_table_$max_depth_exceeded)
Meaning: The maximum depth of the storage system hierarchy is 16 levels. An attempt was made to create an entry at a deeper level.

The name specified contains non-ascii characters.
(error_table_$invalid_ascii)
Meaning: The user specified an access control name which contains non-ASCII characters.

The name was not found.
(error_table_$name_not_found)
Meaning: The entry name specified by the user was not found in the directory.

The normal/ge-xtnd switch is set incorrectly on the printer controller.
(error_table_$print_mode_switch)
Meaning: The switch on the printer controller governing the normal or ge-xtnd mode setting is set incorrectly.

The operation would leave no names on entry.
(error_table_$nonamerr)
Meaning: An attempt has been made to delete the last name on a storage system entry.

The process's limit for this device type is exceeded.
(error_table_$device_limit_exceeded)
Meaning: From tape_: the caller already has attached to his process the maximum number of drives allowed to be assigned at any one time.

The reference name count is greater than the number of reference names.
(error_table_$refname_count_too_big)
Meaning: The reference name count is greater than the number of reference names on an initiated segment.

The requested action was not performed.
(error_table_$action_not_performed)
Meaning: The user requested some action but either 1) had previously directed the program to perform only certain other actions, or 2) directed the program to not perform the action when questioned about it.

The rest of the tape is blank.
(error_table_$blank_tape)
Meaning: From tape_: an attempt was made to read a blank tape (returned only by the attach request).

The same fault will occur again if restart is attempted.
(error_table_$useless_restart)
Meaning: The user attempted to restart (using the start command) after a fault that should not be restarted because it will recur immediately.

The specified detachable volume has not been registered.
useless_restart
The same faule occur again if restart is attempted.
(error_table_$unregistered_volume)
Meaning: The tape reel or other detachable volume has not been registered with Multics Operations, and therefore cannot be user

The star convention is not implemented by this procedure.
(error_table_$nostars)
Meaning: This procedure does not allow any special characters in name arguments.

The stream is attached to more than one device.
(error_table_$multiple_io_attachment)
Meaning: The specified stream was associated with more than one device when the attempt was made to get information about it. Therefore, not all the associations could be returned.

There are too many links to get to a branch.
(error_table_$toomanylinks)
Meaning: The number of links traversed to refer to a branch has exceeded the system limit (currently 10).

There is an inconsistency in arguments to the file system.
(error_table_$argerr)
Meaning: The arguments given were incorrect because of type, number, and/or format.

There is an internal inconsistency in the segment.
(error_table_$bad_segment)

Meaning: From object_info_: the segment is not an object segment. From message segment facility: the message segment was being salvaged or was left in an inconsistent state (perhaps due to a system crash).

There is no initial connection in progress from this socket.
(error_table_$net_no_icp)
Meaning: Network only. The user's process has made a call to a Network connection management program (such as net_icp_) to gain control over a socket connection without having previously called this program to initiate that connection. This may indicate a programming error in the user-level Network interface program being used.

There is no more room in the KST.
(error_table_$nrmkst)
Meaning: There is no more room in the KST to allocate KST entries. A solution for this problem is to terminate segments, terminate reference names, or to cause a new process to be created.

There is no room to make requested allocations.
(error_table_$noalloc)
Meaning: The user-specified area for return arguments is not large enough, or the user attempted to add an entry to a directory that had no room for additional entries.

There was an attempt to create a copy without correct access.
(error_table_$invalid_copy)
Meaning: An attempt has been made to initiate a directory with the copy switch on.

There was an attempt to delete a non-empty directory.
(error_table_$fulldir)
Meaning: The user attempted to delete a directory that contains branches and/or links.

There was an attempt to make a directory unknown that has inferior segments.
(error_table_$infcnt_non_zero)
Meaning: There was an attempt to make a directory unknown that has inferior segments known.

There was an attempt to move segment to non-zero length entry.
(error_table_$clnzero)
Meaning: The user called hcs_$fs_move_file or hcs_$fs_move_seg and specified a nonzero length segment as the entry to move to and did not specify the truncate switch.

There was an attempt to use an invalid segment number.
(error_table_$invalidsegno)
Meaning: The user attempted to use a pointer that contained a segment number that does not reference any segment known to his process.

This operation is not allowed for a directory.

(error_table_$dirseg)
Meaning: The attempted operation is illegal when performed on a directory.

This operation is not allowed for a segment.
(error_table_$nondirseg)
Meaning: The attempted operation is illegal when performed on a segment.

This procedure does not implement the requested version.
(error_table_$unimplemented_version)
Meaning: The version number on a data structure is unknown to the system module attempting to manipulate the data, indicating that the data might not be in the expected format.

Too many "<" 's in pathname.
(error_table_$lesserr)
Meaning: The user supplied a relative path name that contains more less than characters than his current working directory is deep in the hierarchy.

Too many buffers specified.
(error_table_$too_many_buffers)
Meaning: Not enough wired core exists to transfer the requested data.

Too many read delimiters specified.
(error_table_$too_many_read_delimiters)
Meaning: The type of device associated with the given stream does not support the given number of read delimiters or break characters at the current element size. It may be possible to specify fewer read delimiters or break characters.

Too many search rules.
(error_table_$too_many_sr)
Meaning: The number of search rules to be used exceeds the system limit.

Translation failed.
(error_table_$translation_failed)
Meaning: The translation was not able to produce a usable object segment because of errors in the source segment.

Typename not found.
(error_table_$typename_not_found)
Meaning: The device type specified in a call to the I/O system is unknown.

Unable to convert character date/time to binary.
(error_table_$date_conversion_error)
Meaning: Illegal syntax or conflicting specifications were used in the input string that specifies the date/time.

Unable to create a copy.
(error_table_$no_create_copy)

Meaning: A copy of the desired segment could not be created.

Unable to initiate the copy.
(error_table_$copy_not_init)
Meaning: The copy of the desired segment cound not be initiated.

Unable to make original segment known.
(error_table_$no_makeknown)
Meaning: A segment with a copy switch on could not be initiated and, hence, no copy could be made.

Unable to move segment because of type, access or quota.
(error_table_$no_move)
Meaning: The segment could not be moved because of type. access or quota.

Unable to process a search rule string.
(error_table_$bad_string)
Meaning: The syntax in a search rule string is unacceptable.

Unable to set the bit count on the copy.
(error_table_$no_set_btcnt)
Meaning: The bit count on the copy of the desired segment could not be set.

Undefined order request.
(error_table_$undefined_order_request)
Meaning: The request specified in this I/O system order call does not exist for the type of device associated with the given stream name.

Unrecognizable ptrname on seek or tell call.
(error_table_$undefined_ptrname)
Meaning: An invalid reference pointer name was given in an I/O system seek or tell call.

Unrecoverable data-transmission error on physical device.
(error_table_$device_parity)
Meaning: From tape_: the program was physically unable to finish writing the tape or unable to read the desired tape record. If another read request is made, processing begins with the next logical record.

Use of star convention resulted in no match.
(error_table_$nomatch)
Meaning: The use of the star convention resulted in no match during the requested directory search.

User name not on access control list for branch.
(error_table_$user_not_found)
Meaning: A storage system subroutine for deleting or listing ACL entries could not find a name that was requested.

Wrong number of arguments supplied.
(error_table_$wrong_no_of_args)

Meaning: A program was not passed the correct number of arguments.

Zero length segment.
(error_table_$zero_length_seg)
Meaning: The bit count of the segment indicates that it is of zero length.

The following is an alphabetic cross-reference by error_table_ entry name to printed messages listed in the first part of this write-up. The error_table_$ prefix has been removed from these entry names to facilitate perusing.

action_not_performed
The requested action was not performed.
already_assigned
Indicated device assigned to another process.
area_too_small
Supplied area too small for this request.
arg_ignored
Argument ignored.
argerr
There is an inconsistency in arguments to the file system.
att_loop
Attachment loop.
bad_acl_mode
Bad mode specification for ACL/CACL.
bad_arg_acc
Improper access to given argument.
bad_arg_type
Bad gate for entry referenced.
bad_class_def
Bad class code in definition.
bad_command_name
Improper syntax in command name.
bad_entry_point_name
Illegal entry point name in make_ptr call.
bad_equal_name
The equal name specified had illegal syntax
bad_index
Internal index out of bounds.
bad_label
Incorrect detachable medium label.
bad_link_target_init_info
Illegal initialization info passed with create-if-not-found link.
bad_link_type
Illegal type code in type pair block.
bad_mode
Improper mode specification for this device.
bad_mount_request
Mount request could not be honored.
bad_ms_convert
Attempt to convert directory or link to multisegment file.
bad_ms_file

Directory or link found in multisegment file.

bad_name
The access name specified has an illegal syntax.

bad_processid
Current processid does not match stored value.

bad_ptr
Argument is not an ITS pointer.

bad_ring_brackets
Ring brackets input to directory control are invalid.

bad_segment
There is an internal inconsistency in the segment.

bad_self_ref
Illegal self reference type.

bad_string
Unable to process a search rule string.

bad_tapeid
Invalid volume identifier.

bad_volid
Invalid volume identifier.

badcall
Procedure called improperly.

badequal
Illegal use of equals convention.

badopt
Specified control argument is not implemented by this command.

badpath
Bad syntax in pathname.

badstar
Illegal entry name.

badsyntax
Syntax error in ascii segment.

bigarg
Argument too long.

blank_tape
The rest of the tape is blank.

boundviol
Attempt to access beyond end of segment.

buffer_big
Specified buffer size too large.

change_first
Attempt to change first pointer.

clnzero
There was an attempt to move segment to non-zero length entry.

command_line_overflow
Expanded command line is too large.

copy_not_init
     Unable to initiate the copy.
dart_abort
     Dartmouth job aborted.
data_improperly_terminated
     Relevant data terminated improperly.
date_conversion_error
     Unable to convert character date/time to binary.
defs_loop
     Looping searching definitions.
dev_nt_assnd
     IO device not currently assigned.
dev_offset_out_of_bounds
     Specified offset out of bounds for this device.
device_end
     Physical end of device encountered.
device_limit_exceeded
     The process's limit for this device type is exceeded.
device_parity
     Unrecoverable data-transmission error on physical device.
dirlong
     Directory pathname too long.
dirseg
     This operation is not allowed for a directory.
dup_ent_name
     Duplicate entry name in bound segment.
ect_full
     The event channel table was full.
empty_acl
     ACL/CACL is empty.
entlong
     Entry name too long.
execute_data
     Attempt to execute in data segment.
force_bases
     No bases supplied in force call.
full_hashtbl
     The directory hash table is full.
fulldir
     There was an attempt to delete a non-empty directory.
id_already_exists
     Supplied identifier already exists in data base.
imp_down
     Multics IMP is down.
improper_data_format
     Data not in expected format.
inconsistent
     Inconsistent combination of control arguments.

inconsistent_ect
The event channel table was in an inconsistent state.

incorrect_access
Incorrect access to directory containing entry.

infcnt_non_zero
There was an attempt to make a directory unknown that has inferior segments.

invalid_ascii
The name specified contains non-ascii characters.

invalid_backspace_read
Invalid backspace_read order call.

invalid_channel
The event channel specified is not a valid channel.

invalid_copy
There was an attempt to create a copy without correct access.

invalid_device
Attempt to attach to an invalid device.

invalid_elsize
Invalid element size.

invalid_lock_reset
The lock was locked by a process that no longer exists, therefore the lock was reset.

invalid_max_length
Attempt to set max length of a segment less than its current length.

invalid_mode
Invalid mode specified for ACL.

invalid_move_quota
Invalid move of quota would change terminal quota to non terminal.

invalid_project_for_gate
Invalid project for gate access control list.

invalid_read
Attempt to read or move read pointer on device which was not attached as readable.

invalid_seek_last_bound
Attempt to manipulate last or bound pointers for device that was not attached as writeable.

invalid_setdelim
Attempt to set delimiters for device while element size is too large to support search.

Invalid_state
Request is inconsistent with current state of device.

invalid_write
Attempt to write or move write pointer on device which was not attached as writeable.

invalidsegno
     There was an attempt to use an invalid segment number.
io_no_permission
     Process lacks permission to alter device status.
io_still_assnd
     IO device failed to become unassigned.
ioat_err
     Error in internal ioat information.
ioname_not_active
     Ioname not active.
ioname_not_found
     Ioname not found.
ionmat
     Ioname already attached and active.
lesserr
     Too many "<" 's in pathname.
lock_not_locked
     Attempt to unlock a lock that was not locked.
lock_wait_time_exceeded
     The lock could not be set in the given time.
locked_by_other_process
     Attempt to unlock a lock which was locked by another process.
locked_by_this_process
     The lock was already locked by this process.
longeql
     Equals convention makes entry name too long.
loterr
     Error zeroing entry in the linkage offset table.
max_depth_exceeded
     The maximum depth in the storage system hierarchy has been exceeded.
mismatched_iter
     Mismatched iteration sets.
missent
     Missing entry in outer module.
moderr
     Incorrect access on entry.
mount_not_ready
     Requested volume is not yet mounted.
mount_pending
     Mount request pending.
multiple_io_attachment
     The stream is attached to more than one device.
name_not_found
     The name was not found.
namedup
     Name duplication.

negative_nelem
Negative number of elements supplied to data transmission entry.

negative_offset
Negative offset supplied to data transmission entry.

net_already_icp
An initial connection is already in progress from this socket.

net_bad_connect
Request for connection received from improper foreign socket.

net_bad_gender
Bad socket gender involved in this request.

net_bad_host
Illegal host number or id.

net_fhost_down
Foreign host is down.

net_fhost_inactive
Communications with this foreign host not enabled.

net_fimp_down
Foreign IMP is down.

net_ipc_bad_state
Initial connection socket is in an improper state.

net_ipc_error
A logical error has occurred in initial connection.

net_icp_not_concluded
The initial connection has not been completed.

net_invalid_state
Request is inconsistent with state of socket.

net_no_icp
There is no initial connection in progress from this socket.

net_not_up
Network control program not in operation.

net_rfc_refused
Request for connection refused by foreign host.

net_socket_closed
Netword connection closed by foreign host.

net_socket_not_found
Specified socket not found in network data base.

net_table_space
The NCP could not find a free table entry for this request.

net_timeout
Connection not completed within specified time interval.

new_offset_negative
New offset for pointer computed by seek entry is negative.

no_append
Append permission missing.

no_backspace
Requested tape backspace unsuccessful.
no_create_copy
Unable to create a copy.
no_defs
Bad definitions pointer in linkage.
no_device
No device currently available for attachment.
no_dir
Some directory in path specified does not exist.
no_ext_sym
External symbol not found.
no_info
Insufficient access to return any information.
no_io_interrupt
No interrupt was received on the designated IO channel.
no_job_core
Could not create dartmouth job core.
no_linkage
Linkage section not found.
no_makeknown
Unable to make original segment known.
no_move
Unable to move segment because of type, access or quota.
no_prph_card
No PRPH card was found for the requested device.
no_restart
Supplied machine conditions are not restartable.
no_room_for_dsb
No room available for device status block.
no_s_permission
Status permission missing on directory containing entry.
no_set_btcnt
Unable to set the bit count on the copy.
no_sym_seg
Symbol segment not found.
no_wired_structure
No wired structure could be allocated for this device request.
noalloc
There is no room to make requested allocations.
noarg
Expected argument missing.
nodescr
Expected argument descriptor missing.
noentry
Entry not found.
nolinkag

No/bad linkage info in the lot for this segment.

nolot
No linkage offset table in this ring.

nomatch
Use of star convention resulted in no match.

nonamerr
The operation would leave no names on entry.

nondirseg
This operation is not allowed for a segment.

nostars
The star convention is not implemented by this procedure.

nosys
System service process not currently available.

not_a_branch
Entry is not a branch.

not_act_fnc
Procedure was not invoked as an active function.

not_attached
Process not attached to indicated device.

not_bound
Segment is not bound.

not_in_proper_bracket
Not in proper ring bracket to perform desired operation.

not_seg_type
Segment not of type specified.

notadir
Entry is not a directory.

notalloc
Allocation could not be performed.

nrmkst
There is no more room in the KST.

null_brackets
Null bracket set encountered.

odd_no_of_args
Odd number of arguments.

oldnamerr
Name not found.

oldobj
Obsolete object segment format.

out_of_sequence
A call that must be in a sequence of calls was out of sequence.

pathlong
Pathname too long.

print_mode_switch
The normal/ge-xtnd switch is set incorrectly on the printer controller.

recursion_error
Infinite recursion.
redundant_mount
Requested volume is already mounted.
refname_count_too_big
The reference name count is greater than the number of reference names.
request_not_recognized
Request not recognized.
root
The directory is the ROOT.
rqover
Record quota overflow.
safety_sw_on
Attempt to delete segment whose safety switch is on.
sameseg
Attempt to specify the same segment as both old and new.
seg_not_found
Segment not found.
seg_unknown
Segment not known to process.
segknown
Segment already known to process.
segnamedup
Name already on entry.
smallarg
Argument size too small.
stack_overflow
Not enough room in stack to complete processing.
strings_not_equal
Strings are not equal.
too_many_args
Maximum number of arguments for this command exceeded.
too_many_buffers
Too many buffers specified.
too_many_names
Name list exceeds maximum size.
too_many_read_delimiters
Too many read delimiters specified.
too_many_sr
Too many search rules.
toomanylinks
There are too many links to get to a branch.
translation_aborted
Fatal error. Translation aborted.
translation_failed
Translation failed.
typename_not_found

Typename not found.

unbalanced_brackets
Brackets do not balance.

unbalanced_parentheses
Parentheses do not balance.

unbalanced_quotes
Quotes do not balance.

undefined_order_request
Undefined order request.

undefined_ptrname
Unrecognizable ptrname on seek or tell call.

unimplemented_ptrname
Pointer name passed to seek or tell not currently implemented by it.

unimplemented_version
This procedure does not implement the requested version.

unregistered_volume
The specified detachable volume has not been registered.

useless_restart
The same fault will occur again if restart is attempted.

user_not_found
User name not on access control list for branch.

wrong_channel_ring
An event channel is being used in an incorrect ring.

wrong_no_of_args
Wrong number of arguments supplied.

zero_length_seg
Zero length segment.

LIST OF SYSTEM CONDITIONS AND DEFAULT ON UNIT ACTIONS

System conditions are signalled to report certain unusual occurrences encountered by system procedures. The MPM Reference Guide section, The Multics Condition Mechanism, describes the signalling and handling of conditions in general. See also the MPM Reference Guide section, Strategies for Handling Unusual Occurrences.

This section lists the conditions signalled by system procedures, and the default actions taken for each. The default on unit is invoked if no other user or system on unit has been established for the condition. The conditions are listed in alphabetical order by name.

When present, the parenthetical type designator at the right margin on the same line with the name indicates that the condition is either:

1) defined by the PL/I language; or

2) due to a hardware fault or an error encountered while processing a hardware fault (indicating that a processor state description is available).

Otherwise, the condition is neither of these.

Four items follow for each condition:

1) cause — is the reason the condition is signalled;

2) default action — is a brief description of the action taken by the default on unit;

3) restrictions — indicate when the user should not attempt to handle the condition and note when restarting after an occurrence of the condition is inappropriate;

4) data structure — is the PL/I declaration of the data that can be pointed to by info_ptr, the fourth argument available to a condition handler. (See the MPM Subsystem Writers' Guide section, Multics Condition Mechanism Arguments, for details.) Unless otherwise specified, it is not generally useful for the handler to change the values of variables in the data structure.

The PL/I Condition Data Structure

Most of the PL/I conditions have the data structure described below. Only the items associated with a particular instance of a condition are filled in. The relevant information should be obtained from the PL/I defined ondata functions. Users should not refer to this structure (beyond the header) since it is primarily an implementation vehicle for the ondata functions.

For brevity, the data structure item of PL/I conditions that use this data structure is listed as "the standard PL/I data structure".

```
declare 1 info aligned,
          2 length fixed bin,
          2 version fixed bin,
          2 action_flags aligned,
            3 cant_restart bit(1),
            3 default_restart bit(1),
            3 pad bit(34),
          2 info_string char(256) var,
          2 status_code fixed bin(35),
          2 id char(8) init ("pliocond"),
          2 content_flags aligned,
           (3 v1sw,
            3 oncode_sw,
            3 onfile_sw,
            3 file_ptr_sw,
            3 onsource_sw,
            3 onchar_sw,
            3 onkey_sw,
            3 onfield_sw) bit(1) unaligned,
          2 oncode fixed bin(35),
          2 onfile char(32) aligned,
          2 file ptr ptr,
          2 onsource char(256) var,
          2 oncharindex fixed bin,
          2 onkey_onfield char(256) var;
```

1) length — is the length in words of this structure.

2) version — is the version number of this structure.

3) action_flags — indicates appropriate behavior for a handler:

   cant_restart — if "1"b, a handler should never attempt to return to the signaling procedure.

default_restart if "1"b, the computation can resume with no further action on the handler's part except a return.

4) info_string is a printable message about the condition.

5) status_code is the status code, if any, that caused the condition to be signalled.

6) id identifies this structure as belonging to a PL/I condition.

7) v1sw if "1"b, indicates that the condition was raised by a version 1 PL/I procedure.

8) oncode_sw if "1"b, indicates that the structure contains a valid oncode.

9) onfile_sw if "1"b, indicates that a file name has been copied into the structure.

10) file_ptr_sw if "1"b, indicates that there is a file associated with this condition.

11) onsource_sw if "1"b, indicates that there is a valid onsource string for this condition.

12) onchar_sw if "1"b, indicates that there is a valid onchar index in this structure.

13) onkey_sw if "1"b, indicates that there is a valid onkey string in this structure.

14) onfield_sw if "1"b, indicates that there is a valid onfield string in this structure.

15) oncode is the condition's oncode if oncode_sw = "1"b.

16) onfile is the onfile string if onfile_sw = "1"b;

17) file_ptr is a pointer to a file value if file_ptr_sw = "1"b.

18) onsource is the onsource string if onsource_sw = "1"b.

19) oncharindex is character offset in onsource of the offending character if onchar_sw = "1"b.

20) onkey_onfield is the onkey string if onkey_sw = "1"b and is the onfield string if onfield_sw = "1"b.

Shorthand Notation

One default action description occurs frequently. For brevity, it is listed as

"prints a message and returns to command level"

to mean

"an error message is printed on the stream "error_output", and the user is placed at command level with a higher level stack frame than before the condition was signalled".

Thus his stack is intact and the history of the error is preserved. The user can hold the stack for further debugging activities, or he can release it. (See the MPM write-ups for the debug, hold, release and start commands.)

System Conditions

active_function_error

Cause: the user incorrectly used an active function in a command line. The procedure active_fnc_err_ signals this condition. See the MPM Reference Guide section, The Command Language.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure:

```
declare 1 active_function_error_info aligned,
          2 length fixed bin,
          2 version fixed bin,
          2 action_flags aligned,
            3 cant_restart bit(1) unaligned,
            3 default_restart bit(1) unaligned,
            3 pad bit(34) unaligned,
```

```
2 info_string char(256) var,
2 status_code fixed bin(35),
2 name_ptr ptr,
2 name_lth fixed bin,
2 errmsg_ptr ptr,
2 errmsg_lth fixed bin,
2 max_errmsg_lth fixed bin,
2 print_sw bit(1);
```

1) length — is the length is words of this structure.

2) version — is the version number of this structure.

3) action_flags — indicates appropriate behavior for a handler:

   cant_restart — if "1"b, a handler should never attempt to return to the signalling procedure.

   default_restart — if "1"b, the computation can resume with no further action on the handler's part except a return.

   pad — is currently ignored.

4) info_string — is a printable message about the condition.

5) status_code — is the status code being reported by active_fnc_err_.

6) name_ptr — is a pointer to a character string containing the name of the procedure which called active_fnc_err_.

7) name_lth — is the length of the name of the procedure which called active_fnc_err_.

8) errmsg_ptr — is a pointer to a character string containing the error message prepared by active_fnc_err_. A handler might wish to alter that message.

9) errmsg_lth — is the significant length of the error message prepared by active_fnc_err_. This datum can be changed by the

handler.

10) max_errmsg_lth is the size of the character string containing the error message prepared by active_fnc_err_.

11) print_sw if "1"b, the error message will be printed by active_fnc_err_ if and when the handler returns control to it. This datum can be changed by the handler.

alrm (hardware)

Cause: a real-time alarm occurred a specified length of time after a call by the user to timer_manager_$alarm_call (to set the alarm). See the MPM write-up for timer_manager_.

Default action: the handler looks up the alarm that is expected at the time this one occurred, and calls the appropriate user-specified procedure. When (if) this procedure returns, the user's process is returned to the point at which it was interrupted.

Restrictions: the user should not attempt to handle this condition.

Data structure: none.

Note: any_other handlers should pass this on.

area (PL/I)

Cause: the user attempted to either 1) allocate storage in an area that had insufficient space remaining to generate the storage needed; or 2) assign one area to another, and the second had insufficient space to hold the storage allocated in the first.

Default action: prints a message on the "error_output" stream and signals the error condition. Upon a normal return, the attempted allocation is retried in case the user has freed some storage from an area in the interim.

Restrictions: none.

Data structure: none.

bad_outward_call (hardware)

Cause: the user attempted to make an illegal call to an outer ring.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

command_error

Cause: the user incorrectly used a command (such as giving it bad arguments), or a command encountered a situation that prevented it from completing its operation normally. The procedure com_err_ signals this condition.

Default action: returns to com_err_, which then prints a formatted message on the stream "error_output". Other more sophisticated handlers could reformat the error message to the individual user's taste, or take some special action depending on the particular condition in question.

Restrictions: none.

Data structure:

```
declare 1 command_error_info aligned,
          2 length fixed bin,
          2 version fixed bin init(2),
          2 action_flags aligned,
            3 cant_restart bit(1) unaligned,
            3 default_restart bit(1) unaligned,
            3 pad bit(34) unaligned,
          2 info_string char(256) var,
          2 status_code fixed bin(35),
          2 name_ptr ptr,
          2 name_lth fixed bin,
          2 errmess_ptr ptr,
          2 errmess_lth fixed bin,
          2 max_errmess_lth fixed bin init(256),
```

```
        2 print_sw bit(1) init("1"b);
```

1) length — is the length in words of this structure.

2) version — is the version number of this structure.

3) action_flags — indicates appropriate behavior for a handler:

   cant_restart — if "1"b, a handler should never attempt to return to the signalling procedure.

   default_restart — if "1"b, the computation can resume with no further action on the handler's part except a return.

   pad — is currently ignored.

4) info_string — is a printable message about the condition.

5) status_code — is the status code being reported by com_err_.

6) name_ptr — is a pointer to a character string containing the name of the procedure which called com_err_.

7) name_lth — is the length of the name of the procedure which called com_err_.

8) errmess_ptr — is a pointer to a character string containing the error message prepared by com_err_. A handler might wish to alter that message.

9) errmess_lth — is the significant length of the error message prepared by com_err_. This datum can be changed by the handler.

10) max_errmess_lth — is the size of the character string containing the error message prepared by com_err_.

11) print_sw — if "1"b, the error message is printed by com_err_. This datum can be set by the handler.

command_query_error

Cause: the user specified a handler for the command_question condition that did not return a "yes" or "no" answer when the data structure element indicated that a "yes" or "no" answer was required. The procedure command_query_ signals this condition.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

command_question

Cause: a command is asking a question of the user. The procedure command_query_ signals this condition.

Default action: returns to command_query_, which then prints the question on the stream "user_output". Other more sophisticated handlers could supply a preset answer, modify the question or suppress its printing. See the data structure below for details.

Restrictions: none.

Data structure:

```
declare 1 command_question_info,
          2 length fixed bin,
          2 version fixed bin init(2),
          2 action_flags aligned,
            3 cant_restart bit(1) unaligned,
            3 default_restart bit(1) unaligned,
            3 pad bit(34) unaligned,
          2 info_string char(256) var,
          2 status_code fixed bin(35),
          2 query_code fixed bin(35),
          2 question_sw bit(1) init ("1"b) unaligned,
          2 yes_or_no_sw bit(1) unaligned,
          2 preset_sw bit(1) init("0"b) unaligned,
          2 answer_sw bit(1) init("1"b) unaligned,
          2 name_ptr ptr,
          2 name_lth fixed bin,
```

```
        2 question_ptr ptr,
        2 question_lth fixed bin,
        2 max_question_lth fixed bin,
        2 answer_ptr ptr,
        2 answer_lth fixed bin,
        2 max_answer_lth fixed bin;
```

1) length — is the length in words of this structure.

2) version — is the version number of this structure.

3) action_flags — indicates appropriate behavior for a handler:

   cant_restart — if "1"b, a handler should never attempt to return to the signalling procedure.

   default_restart — if "1"b, the computation can resume with no further action on the handler's part except a return.

   pad — is currently ignored.

4) info_string — is a printable message about the condition.

5) status_code — is the status code that prompted the call to command_query_.

6) pad — is currently ignored. (A value of zero is always passed to the handler.)

7) question_sw — if "1"b, command_query_ prints the question. This datum can be set by the handler.

8) yes_or_no_sw — if "1"b, indicates that command_query_ expects the preset answer (if any) returned by the handler to be either "yes" or "no". In this case, if the handler returns any other string, command_query_ signals the command_query_error condition.

9) preset_sw — if "1"b, the handler is returning in the character string pointed to by

answer_ptr a preset answer to command_query_. In that case, command_query_ returns the preset answer to its caller. That is, it does not attempt to obtain an interactive response by reading from the stream "user_input". This datum can be changed by the handler.

10) answer_sw — if "1"b, command_query_ prints the preset answer (if any). This datum can be changed by the handler.

11) name_ptr — is a pointer to a character string containing the name of the procedure which called command_query_.

12) name_lth — is the length of the name pointed to by name_ptr.

13) question_ptr — is a pointer to a character string containing the question prepared by command_query_. A handler might wish to alter that question.

14) question_lth — is the significant length of the question pointed to by question_ptr. This datum can be changed by the handler.

15) max_question_lth — is the size of the character string pointed to by question_ptr.

16) answer_ptr — is a pointer to a character string that can be used by the handler to return a preset answer.

17) answer_lth — is the significant length of the preset answer pointed to by answer_ptr. This datum can be changed by the handler.

18) max_answer_lth — is the size of the character string pointed to by answer_ptr.

Notes: a preset answer is treated exactly as if it had been read from the stream "user_input"; that is, leading and trailing blanks and the terminal new line character (if any)

are removed.

If the yes_or_no_sw is on and a preset answer is returned that is not "yes" or "no", command_query_ signals the condition command_query_error.

conversion (PL/I)

Cause: a PL/I conversion or runtime-I/O routine attempted an illegal conversion from character string representation to some other representation. Possible illegal conversions are a character other than 0 or 1 being converted to bit string, and non-numeric characters where only numeric characters are permitted in a conversion to arithmetic data.

Default action: prints a message on the "error_output" stream and signals the error condition. Upon a normal return, the conversion is attempted again, using the value of the PL/I onsource pseudovariable as the input character string.

Restrictions: none.

Data structure: the standard PL/I data structure.

Note: the user can establish a handler that uses the onchar and onsource builtin functions to alter the invalid character string.

cput (hardware)

Cause: a CPU-time interrupt occurred after a user-specified amount of CPU time had passed following a call to timer_manager_$cpu_call. (See the MPM write-up for timer_manager_.)

Default action: the handler looks up the CPU time interrupt that is expected at this time and calls the appropriate user-specified procedure. When (if) this procedure returns, the process is returned to the point at which it was interrupted.

Restrictions: the user should not attempt to handle this condition.

Data structure: none.

Note: any_other handlers should pass this on.

cross_ring_transfer (hardware)

Cause: the user attempted to cross ring boundaries using a transfer instruction. A CALL or RTCD instruction must be used to cross ring boundaries.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

derail (hardware)

Cause: the user attempted to execute a derail instruction on the 6180.

Default action: prints a message and returns to command level.

Restrictions: usually none. However, some subsystems (e.g., Dartmouth) use it for special purposes. When operating within such subsystems, the user should not attempt to handle the condition.

Data Structure: none.

endfile (f) (PL/I)

Cause: a PL/I get or read statement attempted to read past the end of data on the file f.

Default action: prints a message on the "error_output" stream and signals the error condition. Upon return from any handler, control passes to the PL/I statement following the statement in which the condition was raised.

Restrictions: none.

Data structure: the standard PL/I data structure.

endpage (f) (PL/I)

Cause: PL/I inserted the last "new line" character of the current page into the output stream of the file f. I.e., if page_size is the number of lines normally placed on a page, then the (page_size)th "new line" character is the last one.

Default action: begins the next page on the file f and returns.

Restrictions: none.

Data structure: the standard PL/I data structure.

Note: the handler can begin a new page via a PL/I statement of the form

```
put file (f) page ... (... "title"...) ...;
```

or can simply return, permitting the number of lines on the current page to exceed the number normally occurring.

error (PL/I)

Cause: some other (more specific) PL/I condition occurred, and its handler signalled the error condition. Alternatively, some PL/I runtime subroutine (e.g., one in the mathematical library) enountered one of a variety of errors.

Default action: prints a message and returns to command level.

Restrictions: if the error condition is not merely an echo of another PL/I condition, then restarting (i.e., returning control to the signaller) is usually undefined.

Restarting from other PL/I conditions is discussed under the individual conditions.

Data structure: the standard PL/I data structure.

fault_tag_1, fault_tag_3 (hardware)

Cause: the user attempted an indirect reference through a word pair containing either a fault tag 1 or a fault tag 3 modifier.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

finish

Cause: the user's process is being terminated by a logout (either voluntary or involuntary) or by a new_proc command.

Default action: closes all open files and returns.

Restrictions: if the process is terminating because of a bump or resource limit stop, there is only a small grace period before the process is actually killed. If a user-supplied handler does not return, the process continues to run but in some cases a subsequent process termination is fatal.

Data structure: none.

Note: any_other handles should pass this on.

fixedoverflow (hardware)

Cause: the result of a binary fixed-point operation exceeded 71 bits, or the result of a decimal fixed-point operation exceeded 63 digits.

Default action: prints a message on the "error_output" stream and signals the error condition.

Restrictions: a return to the point where the signal occurred is prohibited since continued execution from this point is undefined.

Data structure: none.

gate_error

Cause: the user attempted an inward wall crossing through a gate segment with the wrong number of arguments.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

illegal_modifier (hardware)

Cause: an illegal modifier appeared on an indirect word.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

Note: this error caused the op_not_complete condition to be signalled on the 645.

illegal_opcode (hardware)

Cause: the user attempted to execute an illegal operation code. In a machine language program this could be a simple programmer error. It could also be a compiler error or a hardware error.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

illegal_procedure (hardware)

Cause: the user attempted to execute a privileged instruction, or tried to execute an instruction in an illegal way.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

illegal_ring_order (hardware)

Cause: ring brackets on a segment are in the wrong order; i.e., not in ascending order.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

illegal_return

Cause: an attempt was made to restore the control unit with illegal information.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

io_error

Cause: an I/O procedure which does not return an I/O system status code received such a code from an inferior I/O procedure. The first procedure (e.g., ioa_) reflects the error by signalling this condition.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure:

```
declare 1 io_error_info aligned,
          2 length fixed bin,
          2 version fixed bin init (0),
          2 action_flags aligned,
            3 cant_restart bit(1) unaligned,
            3 default_restart bit(1) unaligned,
            3 pad bit(34) unaligned,
          2 info_string_char(256) var,
          2 stream char(32),
          2 status bit(72);
```

1) length — is the length in words of this structure.

2) version — is always 0 in this case.

3) action_flags — indicates appropriate behavior for a handler:

   cant_restart — if "1"b, a handler should never attempt to return to the signalling procedure.

   default_restart — if "1"b, the computation can resume with no further action on the handler's part except a return.

   pad — is currently ignored.

4) info_string — is a printable message about the condition.

5) status_code — is the unexpected status code received by an I/O procedure.

6) stream — is the name of the stream on which the I/O operation was performed.

7) status — is the I/O system status code describing the error.

Cause: the user called an ioa_ entry with illegal arguments. The possible incorrect calls are:

1) failed to provide a stream name for
ioa_$ioa_stream
ioa_$ioa_stream_nnl

2) failed to provide a correct character string descriptor for
ioa_$rs
ioa_$rsnnl
ioa_$rsnpnnl

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

key (f) (PL/I)

Cause: the user attempted to specify an invalid key in a PL/I record-I/O statement on the file f. Two possible illegal uses are 1) a keyed search failed to find the designated key; and 2) on output, the designated key duplicates a pre-existing key.

Default action: prints a message on the "error_output" stream and signals the error condition. Upon return from any handler, control passes to the PL/I statement following the statement in which the condition was raised.

Restrictions: none.

Data structure: the standard PL/I data structure.

Note: the handler can obtain the value of the invalid key by use of the onkey builtin function. The invalid key cannot, however, be corrected in the handler.

linkage_error (hardware)

Cause: the user's process encountered a fault tag 2 in a word pair. It then attempted to reference the external

entry specified by the word pair and failed because either the segment was not found or the entry point did not exist in that segment.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

lockup (hardware)

Cause: a pending interrupt has not been allowed for too long. This can be caused by a looping instruction pair, an infinite indirection chain, or a bar mode interrupt inhibit bit on for too long.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

Note: this condition was signalled only because of an infinite indirection chain on the 645.

loop_wait_error

Cause: a procedure operating in or calling into the Hardcore Ring called pxss$loop_wait with bad or illegal arguments.

Default action: prints a message and returns to command level, after leaving the Hardcore Ring (i.e., the default handler operates only in outer rings).

Restrictions: the user should not attempt to handle this condition.

Data structure: none.

message_segment_error

Cause: an absentee queue was found by the message segment facilty to be in an inconsistent state, or a crawlout from the Administrative Ring occurred in the message segment facility.

Default action: prints a message and returns to command level.

Restrictions: since the message segment facility is used only for system services such as absentee queues, the user should not attempt to handle this condition.

Data structure: none.

mme1, mme2, mme3, mme4 (hardware)

Cause: the user attempted to execute the 6180 instruction mme_n_, where _n_ is 1, 2, 3 or 4.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

Note: the debug command uses the mme2 condition to implement breakpoints. This the user will encounter problems if he attempts to set breakpoints in a program that handles the mme2 condition.

name (f) (PL/I)

Cause: an invalid identifier occurred in a PL/I get data statement on the file f.

Default action: prints a message on the stream "error_output" and signals the error condition. Upon return from any handler, the invalid identifier and its associated value field are skipped.

Restrictions: none.

Data structure: the standard PL/I data structure.

no_execute_permission (hardware)

Cause: the user attempted to execute a segment to which he did not have execute permission.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

no_read_permission (hardware)

Cause: the user attempted to read from a segment to which he did not have read permission.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

no_write_permission (hardware)

Cause: the user attempted to write into a segment to which he did not have write permission.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

not_a_gate (hardware)

Cause: the user attempted to call into a gate segment beyond its call limiter; i.e., beyond the upper bound of the transfer vector in a gate.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

not_in_call_bracket (hardware)

Cause: the user attempted to call a segment from a ring not within the segment's call bracket.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

not_in_execute_bracket (hardware)

Cause: the user attempted to execute a segment from a ring not within the segment's execute bracket.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

not_in_read_bracket (hardware)

Cause: the user attempted to read a segment from a ring not within the segment's read bracket.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

not_in_write_bracket (hardware)

Cause: the user attempted to write into a segment from a ring not within the segment's write bracket.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

op_not_complete (hardware)

Cause: 1) the processor failed to access memory within approximately 2 ms after its previous memory access.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

Note: upon return to the signalling procedure, the processor attempts to continue execution at the point where the op not complete was detected. The processor usually continues execution correctly but the machine state might be such that continued execution is at the user's risk.

out_of_bounds (hardware)

Cause: the user attempted to refer to a location beyond the end of the segment specified.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

overflow (hardware)

Cause: the result of a floating-point computation had an exponent exceeding 127.

Default action: prints a message on the "error_output" stream and signals the error condition.

Restrictions: returning to the point where the signal occurred is not allowed since continued execution from this point is undefined.

Data structure: none.

page_fault_error (hardware)

Cause: the normal paging mechanism of the Multics supervisor could not bring a referenced page into memory because the storage system device containing the page could not be read due to a hardware error that could not be corrected by the error condition mechanism.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

parity (hardware)

Cause: the process attempted to refer to a location in memory that has incorrect parity, or to use a RCCL (read clock) instruction on a memory port that does not have a clock. The first is a hardware error; the second is a hardware error or incorrect hardware configuration.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

program_interrupt

Cause: the user issued the program_interrupt (pi) command for the express purpose of signalling this condition. The condition is used by several commands to return to their internal request level (waiting for the next request) after the previous request has been aborted by the user pressing his interrupt (quit) button.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

Note: any_other handlers should pass this on.

quit

Cause: an interactive user has requested a quit; for example, by pressing the quit button on his terminal.

Default action: prints "QUIT" on the terminal, aborts any pending terminal I/O activity, reverts the standard I/O attachments to their default settings, and establishes a new command level, saving the current stack history.

Restrictions: none. But, in general, the user's programs should not handle the quit condition since this condition is normally intended to bring the process back to command level. In addition, a program with a quit handler is more difficult to debug since a bug in the quit handler might make it impossible to interrupt the execution of the program. Certain subsystems can, for various reasons, still choose to make use of the quit condition; but most programs should, instead, use the program_interrupt condition as described eariler in this section.

Data structure: none.

Notes: The standard I/O attachments are described under Usage in the MPM Reference Guide section Use of the Input and Output System. any_other handlers should pass this on.

record (f) (PL/I)

Cause: a PL/I read statement on the file f read a record of a size different from the variable provided to receive it.

Default action: prints a message on the "error_output" stream and signals the error condition. Upon return from any handler data is copied from the record to the variable by a simple bit-string copy as though both were the length of the shorter.

Restrictions: none.

Data structure: the standard PL/I data structure.

record_quota_overflow (hardware)

Cause: the user attempted to increase the number of records taken up by the segments inferior to a directory to a number greater than the secondary storage quota for that directory.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

seg_fault_error (hardware)

Cause: the user attempted to use a pointer with an illegal segment number. This situation arises when 1) a segment was deleted or terminated after the pointer was initialized; 2) the pointer was not initialized in the current process; or 3) the user had null access to the segment.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

simfault_nnnnnn (hardware)

Cause: the user attempted to use a null pointer; i.e., a pointer with a segment number of -1 (2's complement) and an offset of nnnnnn. The offset is mapped into the 6-character string nnnnnn that makes up part of the condition name.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

Note: If a user references through a null pointer with no offset modification, the condition simfault_000001 is signalled.

size (PL/I)

Cause: some value was converted to fixed-point with a loss of one or more high-order bits or digits.

Default action: prints a message on the "error_output" stream and signals the error condition.

Restrictions: returning control to the point where the signal occurred is not allowed since the results of continued execution are undefined.

Data structure: the standard PL/I data structure.

stack

Cause: The user attempted to make a reference within the last four pages of the stack.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

Note: any_other handlers should pass this on.

storage (PL/I)

Cause: the PL/I "system storage" has insufficient space for an attempted allocation.

Default action: prints a message on the "error_output" stream and signals the error condition. Upon a normal return the allocation is retried.

Restrictions: none.

Data structure: none.

store (hardware)

Cause: an out_of_bounds error occurred while operating in bar mode, or the user referred to a non-existant memory (e.g., by attempting to read a clock on the memory).

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure: none.

stringrange (PL/I)

Cause: the substr pseudovariable or builtin function specified a substring that is not in fact contained in the string specified.

Default action: prints a message on the "error_output" stream and signals the error condition.

Restrictions: returning control to the point where the signal occurred is not allowed since the results of continued execution are undefined.

Data structure: the standard PL/I data structure.

stringsize (PL/I)

Cause: a string value was assigned to a string variable shorter than the value.

Default action: returns to the point where the condition was signalled, causing a truncated copy of the string value to be assigned to the string variable.

Restrictions: none.

Data structure: the standard PL/I data structure.

subscriptrange (PL/I)

Cause: the value of a subscript lies outside the range of values declared for the bounds of the dimension to which it applies.

Default action: prints a message on the "error_output" stream and signals the error condition.

Restrictions: returning control to the point where the signal occurred is not allowed since the results of continued execution are undefined.

Data structure: the standard PL/I data structure.

timer_manager_err

Cause: the event channel on which timer_manager_ would go to sleep could not be created, or ipc_$block returned a non-zero status code when timer_manager_ went to sleep on it. Either internal static storage for timer_manager_ has been destroyed or the system is about to crash. This condition is also signalled if timer_manager_ is called in a ring other than that in which the process was created, indicating a programming error in the calling procedure.

Default action: prints a message and returns to command level.

Restrictions: the user should only attempt to handle this in a handler for otherwise unclaimed signals.

Data structure: none.

transmit (f) (PL/I)

Cause: a value was incorrectly transmitted between storage and the data set corresponding to the file f. In the case of list-directed input, the condition is signalled after each assignment by the get statement of a value that might have been in error due to the bad input line.

Default action: prints a message on the "error_output" stream and signals the error condition. Upon return from any handler, the program continues from the point of detection as though the transmission had been correct.

Restrictions: none.

Data structure: the standard PL/I data structure.

truncation (hardware)

Cause: the user executed an extended instruction set instruction to move string data with the truncation bit set, and the target string was not large enough to contain the source string, or bit strings were being combined to the left or right (also EIS instructions) and there was not enough room to hold the combined string.

Default action: prints a message and return to command level.

Restrictions: none.

Data structure: none.

undefinedfile (f) (PL/I)

Cause: an attempt to open the PL/I file f failed.

Default action: prints a message on the "error_output" stream and signals the error condition.

Restrictions: none.

Data structure: the standard PL/I data structure.

underflow (hardware)

Cause: the result of a floating-point computation had an exponent less than -128.

Default action: prints a message on the "error_output" stream and returns.

Restrictions: none.

Data structure: none.

Note: before the underflow condition is signalled the floating-point value in question is set to zero.

unwinder_error

Cause: the user attempted to perform a non-local transfer to an invalid location.

Default action: prints a message and returns to command level.

Restrictions: none.

Data structure:

```
declare invalid_label label;
```

1) invalid_label is the invalid label to which this transfer was attempted.

zerodivide (PL/I)

Cause: the user attempted to divide by zero.

Default handling: prints a message on the "error_output" stream and signals the error condition.

Restrictions: returning control to the point where the signal occurred is not allowed since the results of continued execution are undefined.

Data structure: the standard PL/I data structure.

 (END)

THE LIMITED SERVICE SYSTEM

A small set of Multics commands, all of which are noted for their light resource usage, compose what is known as the Limited Service System (LSS). A user of the LSS is prevented by the system from using commands outside this set; in addition, his rate of CPU usage is tightly controlled.

The LSS provides a way for a project to allow its members (or a subset of its members) access to computing services with a ceiling on the amount of money they will spend. For example, a professor could leave a terminal logged in all day, available to anyone in his class, and still be sure that only a fixed amount of money would be spent each day.

To register an LSS user, the project administrator need only specify the LSS process overseer as that user's process overseer.

The LSS is sufficiently modular that pieces of it (for instance, the piece that controls which commands are accessible) may be used in other subsystems. This means that subsystem writers can easily tailor their own limited systems with their own list of commands and/or governing parameters. Documentation of the subroutines necessary to do this are published in the MPM Subsystem Writers' Supplement (SWS).

The following commands are currently allowed in the LSS:

addname
basic
basic_system
calc
decam
delete
deletename
edm
help
list
listnames
listotals
logout
print
program_interrupt
ready
ready_off
ready_on
rename
start

(END)

THE MULTICS DARTMOUTH SYSTEM

The Multics Dartmouth System provides a user with a closed subsystem which duplicates, as closely as possible, the Dartmouth Time-Sharing System (DTSS) as implemented at Dartmouth College. Both the command language and the access control mechanism have been simulated in Multics. The language processors and text editors of the subsystem are actual Dartmouth object modules, running in an environment which duplicates that of an HIS-635 computer on which DTSS is normally run.

Most commands operate identically to their counterparts at Dartmouth. Therefore, a user should refer to the documentation published by Dartmouth for detailed information.

To use the Multics Dartmouth System, a user's process overseer should be set to the Dartmouth process overseer. This operation must be done by the user's project administrator. To assure the security of the DTSS access control mechanism, a Multics Dartmouth user will not be able to reference his Dartmouth directory, except through the Multics Dartmouth System.

Segments created under the Multics Dartmouth System are not compatible with other Multics segments. The end of the line convention under DTSS is the ASCII carriage return-new line. Under the regular Multics system, it is just "new line".

The Multics Dartmouth System uses, unchanged, modules of the real DTSS. Errors detected in these modules should be referred to Dartmouth College.

The following commands are supported with differences as noted:

Command	Notes
BUILD	No change.
BYE	Executes the Multics logout command.
CATALOG	Only the options ALL, NFILES, and NHEADER are implemented.
COMPILE	No change.
EDIT	The EXPLAIN option is not implemented.
GOODBYE	is identical to BYE.

HELLO	Executes the Multics command "logout hold".
IGNORE	No change.
LENGTH	No change.
LIST	No change.
NEW	No change.
OLD	See Notes on passwords. If the syntax: <USERNUMBER>:<SEGMENTNAME> is used, the segment is assumed to be in >user_dir_dir>Project>USERNUMBER>SEGMENTNAME.
RENAME	No change.
REPLACE	See Notes on passwords.
RUN	No change.
SAVE	See Notes on passwords.
SCRATCH	No change.
SORT	No change.
STRINGEDIT	No change.
SYSTEM	The following systems are implemented: ALGOL FORTRAN BASIC LISP CHECKERS MIX
TEST	Contact the system administrator to use the TEST command.
TEXTEDIT	No change.
TTY	No change.
UNSAVE	See Notes on passwords.
USERS	Prints the number of users currently on Multics.

XTEST See TEST.

The following commands are not implemented:

APPEND	JOIN
BACKGROUND	KEYBOARD
BILLS	LINK
DIRECT	NFRIDEN
EXPLAIN	NPARITY
FRIDEN	PARITY
FULLDUPLEX	PUNCH
HALFDUPLEX	TAPE

The following systems are not implemented:

9MAP	GMAP
ALGOL68	LAFFF
DEBUGGER	TRAC
GEFORT	

Notes

At Dartmouth, a user may type:

<COMMAND> <NAME>,<PASSWORD>

and DTSS will overstrike the password for security. This can be done at Dartmouth because the user terminates his command line with the ASCII character carriage return. Multics terminals normally terminate input lines with the "new line" character, thus precluding overstriking the line. Multics Dartmouth will accept this syntax but will not attempt to overstrike the line. However, both DTSS and Multics Dartmouth will accept the following syntax:

<COMMAND> <NAME>,

Following this command, the user is asked to enter a password. In this manner, the overstrikes may be typed first and password security maintained. This second method is recommended to the user.

The erase character on Multics Dartmouth is the number sign (#). The kill character is the commercial at sign (@). A quit condition may be signalled only by pressing the appropriate key on the terminal. (See the MPM Reference Guide section on the Protocol for Logging In for the quit button's marking on various

terminals.) The quit will be followed by the messages STOP and READY, at which point the monitor will again be listening for commands.

Segment names in Multics Dartmouth should not contain the characters greater than (>) or less than (<).

To enter a commercial at sign or a number sign into actual text, the user should precede the character with a backslash (\@ or \#). To enter a backslash into his actual text, the user should type two backslashes (\\). (Note: the backslash character is a cent sign (¢) on a 2741, 1050, and Datel 30 terminal.)

DTSS FORTRAN and LISP require all input to be in upper case letters. DTSS MIX maps all lower case letters into upper case. All other systems will accept upper and lower case letters interchangeably.

A large percentage of the programs in the DTSS Program Library are available to users of the Multics Dartmouth System (and users of the basic and basic_run commands). These are programs written in BASIC and ALGOL and cover a large range of applications. See TM010 (described below) for full documentation.

Documentation

The Multics Dartmouth user should consult the following documents for detailed information:

TM002	Dartmouth EDIT
TM003	Dartmouth String Editor
TM004	Text Users Manual
TM006	Double Precision in Dartmouth BASIC
TM010	User's Guide to the DTSS Program Library
TM013	MIX User's Reference Manual
TM015	System FORTRAN Reference Manual
TM016	Accurate Matrix Inversion in BASIC
TM017	LISP System Reference Manual for DTSS on GE635

TM021 Dartmouth ALGOL for the DTSS

TM022 User's Guide to DTSS

BASIC, Fifth Edition

The above documents are all published by the Dartmouth College Kiewit Computation Center, Hanover, New Hampshire.

In addition, the user should reference the manual FORTRAN Language, published by General Electric for the Mark II Time-Sharing Service, manual number 802209.

The current version of the Dartmouth System is a proprietary program of Dartmouth College. It has been made available to users of the M.I.T. Information Processing Center with the permission of Dartmouth College. The Dartmouth System may not be used at other computer installations without permission of Dartmouth College.

(END)*

LIST OF NAMES WITH SPECIAL MEANINGS

The following names are reserved for special purposes within Multics. The user should not use them with a different meaning. See also the following MPM Reference Guide sections for other names with special meanings: List of System Conditions and Default Handlers, List of Names in the System Libraries, and Obsolete Procedures.

Reserved I/O Stream Names

By convention, the following I/O stream names are reserved. Those maintained by the standard environment are:

user_i/o	is the stream attached to the user's terminal or absentee input and output segments.
user_input	is the stream attached to user_i/o and devoted expressly to read calls.
user_output	is the stream attached to user_i/o and devoted expressly to write calls.
error_output	is the stream attached to user_i/o and devoted expressly to write calls under error conditions.

Those maintained by system commands or subroutines are:

exec_com_stream_N	is the stream attached by the exec_com command using the attach command line. N is a unique sequence number assigned by exec_com. user_input is attached to this stream through the syn interface module.
file_output_stream_	is the stream attached by the file_output command. user_output is attached to this stream through the syn interface module.
graphic_input	is the stream used for graphics input.
graphic_output	is the stream used for graphics output.

Reserved Segment Names

By convention, the following segment names are reserved. Those maintained in the home directory are:

mailbox	is the segment used by the mail command.
start_up.ec	is the exec_com invoked at the beginning of a process in the standard environment.
username.breaks	is the break segment used by the debug command. (username is the name derived from the login command.)
username.con_msgs	is the segment used by the message facility (see the MPM write-up of the send_message command). (username is the name derived from the login command.)
username.memo	is the segment used by the memo command. (username is the name derived from the login command.)
username.motd	is the segment used by the print_motd command. (username is the name derived from the print_motd command.)
username.profile	is the segment used by the abbrev command. (username is the name derived from the login command.)

Those maintained in the process directory are:

combined_linkage_N.jk	is the user's linkage segment for ring number N (1<=N<=7). jk is a two digit sequence number. This segment also contains internal static storage.
drum_temp1_ drum_temp2_	is temporary storage used by alm, edm, and qedx.
kst	(Known Segment Table) is a Hardcore Ring data segment.
pds	(Process Data Segment) is a Hardcore Ring data segment.

pit	is the user's Process Initialization Table. It should only be referenced through the subroutine user_info_ (see the MPM subroutine write-up for user_info_).
stack_N	is the user's automatic storage area for ring number N (1<=N<=7).
system_free_N_	is the free storage area used by system commands for ring number N (1<=N<=7).

In general, users should not create segments whose names end in a trailing underscore (_). These names are reserved for system subroutines and may cause errors if they are in the user's search path. (See the MPM Reference Guide section, The System Libraries and Search Rules.)

Reserved Segment Name Suffixes

Suffixes are used as in the following example: If one is creating a PL/I source program to be named xyz, he would create a source language segment named xyz.pl1. The PL/I compiler, by convention, translates this segment, producing the segment xyz.list, containing a printable listing, and the segment xyz, containing the object program.

By convention, the following segment name suffixes are reserved. The language translator source segment suffixes are:

Language Translator	Source Segment	Include Files
PL/I compiler	.pl1	.incl.pl1
FORTRAN compiler	.fortran	.incl.fortran
ALM assembler	.alm	.incl.alm
BASIC compiler	.basic	

The listing segment suffix is:

.list	is the suffix on printed output listing segments produced by compilers, the assembler, and the binder.

Other special suffixes are:

.absin	is the input segment suffix for an absentee process.
.absout	is the default output segment suffix for an absentee process.
.apl	is the suffix on the segment containing a saved workspace from the apl command.
.archive	is the suffix on the segment created by the archive command.
.bind	is the suffix on the input control segment for the binder.
.dobj	is the suffix on the output segment produced from the -compile control argument to the BASIC compiler.
.ec	is the suffix on the input segment to the exec_com command.
.info	is the suffix on a segment, in >documentation>info_segments, for use with the help command.
.lisp	is the suffix on the segment containing a saved environment from the lisp command.
.ms	is the suffix on an Administrative Ring message segment.
.pt	is the segment suffix for use with the peruse_text command.
.runoff	is the input segment suffix to the runoff command.
.runout	is the output segment suffix from the runoff command.

Reserved Object Segment Entry Point

By convention, the following entry point definition in object segments is reserved.

symbol_table is the entry point definition which provides the address of the symbol table produced by the pl1 or fortran commands.

Since this is a reserved entry point, no user-created program can use this name. A statement of the form

```
symbol_table: procedure...
```

is illegal if it is the external procedure block.

LIST OF NAMES IN THE SYSTEM LIBRARIES

The Multics system libraries, >system_library_standard and >system_library_1, are described in the MPM Reference Guide section, The System Libraries and Search Rules. They contain the system commands and subroutines described in the MPM commands and MPM Subroutines sections. They also contain a number of other procedures not intended to be called directly by users, but included in these directories for various reasons. For example, old commands and subroutines that are being phased out (but are still available for an interim period) are left in these directories for the convenience of users who are converting to the replacements for these procedures. Similarly, if the name of a command or subroutine is changed, both names appear on the segment for a time, but users should call it only by the new name.

In addition, the libraries contain entries that are internal interfaces of the Multics supervisor or command system, and are not intended to be called by the user. They are user-accessible primarily to ease the job of checkout of new system commands. User programs should not be coded with calls to these procedures as such calls would produce undesirable dependence one internal system organization or hardware configuration. This set of entries also represents a collection of names that should not be chosen for user-written subroutines, since if the user-written subroutine is lost, a call to it could wind up in the system subroutine of the same name.

This write-up lists the names of those entries in >system_library_standard and >system_library_1 that the user should avoid. Several types of names are excluded from the list to make it more compact. The types of names excluded are:

1) all system command names and their abbreviations. A list of command names can be found in the MPM Reference Guide table of contents; a list of abbreviations can be found in the MPM Reference Guide section, Command Name Abbreviations.

2) all system subroutine names. A list of subroutine names can be found in the MPM Reference Guide table of contents.

3) old versions of recently updated commands and subroutines (and possibly other procedures). These entry names have an additional component of "1" or "2"; e.g., an old version of the compare command might have the entry name compare.1.

4) all names ending in an underscore. System subroutine names are guaranteed to end in an underscore as described in the

MPM Reference Guide section, Constructing and Interpreting Names. Users can avoid conflicts by adopting some other convention.

5) all names with more than one component. Users should not encounter conflicts with these names since procedure names should be of only one component. Typically, multi-component names are used in situations where only the user's working directory is searched.

6) all bound segments. The primary entry name of a bound segment in the system library always has the character string "bound_" as part of the first component of the name. Other entry names on the bound segment are unaffected (in this list) by the application of this rule.

7) all separate linkage section segments. These segments are recognizable by the presence of "link" as the last component of the entry name. Each one has a corresponding text segment without the "link" component in its entry name.

8) all unique names. These segments all have 15-character names with an exclamation point as the first character.

active_all_rings_data
admin_mode_exit
asr
backup_util
changewdir
db_print
delete_search_rules
disassemble
dsr
file_util
fscodedinfo
global
ibm
imf_state
install
lot_maintainer
lsrb
name_table
ncp_test
phd
pom
pp_off
pp_size
print_pdt
add_search_rules
ame
backup_load
caller
check_object
db_regs
deletedir
dissassemble
file
fl
gb
gr_print
iload
ind
interpret_bind_map
lset_ring_brackets
moveb
nc
netcall
pl1_operators
pp_mode
pp_on
print_object_map
printhomedir

printwdir
pur
reload
retrieve
sar
setquota
signal
spe
stacq
sys_info
terminate_reference
translator_ec_ec
unique_chars
vlpa
proj_usage_report
read_convert
rename_ns
ring_zero_cleanup
sethomedir
shd
slt
sq
submit_abs_request
tdsm
translator_absin_absin
unique_bits
unwinder
vlpl1_abs

 (END)

OBSOLETE PROCEDURES

The following procedures are obsolete subroutines or writearounds which remain in the system so that early versions of system commands will continue to work. New programs should not be written to call any of these entry points. Old programs which use them should be modified to use the new procedure or technique indicated.

acm_	(Use timer_manager_)
bindarchive	(Use bind)
bsys	(Use basic_system)
check_fs_errcode_	(Use convert_status_code_, documented in the MPM Subsystem Writers' Guide)
decode_object_	(Use object_info_)
default_handler_	(Establish on unit for any_other condition)
deletedir	(Use delete_dir)
equal_	(Use get_equal_name_)
establish_cleanup_proc_	(Establish on unit for cleanup condition)
global	(Use walk_subtree)
hcs_$acl_add	(Use hcs_$add_acl_entries, hcs_$add_dir_acl_entries, hcs_$delete_acl_entries, hcs_$delete_dir_acl_entries, hcs_$list_acl, hcs_$list_dir_acl, hcs_$replace_acl, or hcs_$replace_dir_acl)
hcs_$chname	(Use hcs_$chname_file or hcs_$chname_seg)
hcs_$fs_get_brackets	(Use hcs_$get_ring_brackets or hcs_$get_dir_ring_brackets, documented in the MPM Subsystem Writers' Guide)
hcs_$fs_search_get_wdir	(Use get_wdir_)
hcs_$fs_search_set_wdir	(Use change_wdir_)
hcs_$get_dbrs	
hcs_$get_usage_values	(Use cpu_time_and_paging_)
hcs_$proc_info	(Use get_process_id_, get_group_id_, get_pdir_, or get_ring_)
hcs_$set_timer	(Use timer_manager_)
hcs_$usage_values	(Use cpu_time_and_paging_)
make_obj_map_	(Use make_object_map_)
moveb	(Use move)
move_	(Use PL/1 array substitution)
ms_	(Use cu_)

print_object_map	(Use print_link_info with the -ln control argument)
printhomedir	(Use print_default_wdir)
probe	(Use debug, dump_segment, list_ref_names, or trace_stack)
revert_cleanup_proc_	(Revert on unit for cleanup conition)
sethomedir	(Use change_default_wdir)
set_search_directories	(Use set_search_dirs)
submit_abs_request	(Use enter_abs_request)
ti_	(Use tssi_, documented in the MPM Subsystem Writers' Guide)
tio_	(Use ios_)

Pointers on Converting to New Interfaces

From time to time, as procedures become obsolete, the following pages will be updated to supply information useful for converting old programs to work with new interfaces.

Name: decode_object_

This subroutine was used to obtain pointers to the components of a segment in object format. The subroutine object_info_ now provides much more complete information about an object segment. Therefore, decode_object_ is considered obsolete and will eventually be removed from the system.

Old Method

```
declare decode_object_ entry (ptr, fixed, fixed, ptr, fixed,
        fixed);

call decode_object_ (segp, bc, i, q, len, bits);
```

1) segp is a pointer to the object segment. (Input)

2) bc is the bit count of the object segment pointed to by p. (Input)

3) i indicates the desired component (standard assignments: 1 = text, 2 = link, 3 = symbol). (Input)

4) q is a pointer to the desired component, null if the component does not exist or the segment is not an object segment. (Output)

5) len is the number of words occupied by the ith component. (Output)

6) lits is the bit count of the ith component. (Output)

Current Method

```
declare object_info_$brief entry (ptr, fixed bin(24), ptr,
        fixed bin(35));

call object_info_$brief (segp, bc, infop, code);
```

1) segp is as above.

2) bc is as above.

3) infop is a pointer to an info structure in which the object information is returned. (Input)

4) code is a standard Multics status code. (Output)

Two other entries, $display and $long have identical calling sequences.

The structure of the info segment is described in the MPM write-up of the object_info_ subroutine.

Name: move_

The function of this procedure (to rapidly copy a block of data from one place to another) is now implemented at least as efficiently by the PL/1 compiler. Therefore move_ is considered obsolete and will eventually be removed from the system. Procedures calling it should be modified to use in-line code as described below.

Old Method

```
declare move_ entry (ptr, ptr, fixed bin);

call move_ (fromp, top, wcnt);
```

1) fromp is a pointer to the start of the data to be copied. (Input)

2) top is a pointer to the start of the block where data is to be copied to. (Input)

3) wcnt is the number of words to be copied. (Input)

Current Method

```
declare block (wcnt) fixed bin (35) based;
.
.
.
top -> block = fromp -> block;
```

where fromp, top, and wcnt are as described above.

Name: ti_

The ti_ subroutine provided an interface between translators and the storage system. The new subroutine, tssi_ (described in the MPM Subsystem Writers' Supplement), provides the same functions (setting up output segments, finishing them and cleaning up after an interrupt) for multi_segment files as well as for single segments. Note that only translator writers have need for this facility.

Old Methods for Segments

To set up an output segment:

```
declare ti_$getseg entry (char(*) aligned, char(*) aligned,
        ptr, fixed bin(35), fixed bin);

call ti_$getseg (dname, ename, segp, aclinfo, code);
```

1) dname is the name of the directory in which the segment resides. (Input)

2) ename is the name of the segment. (Input)

3) segp is the pointer to the segment. (Output)

4) aclinfo is coded information about where to find the segment's previous ACL saved. (Output)

5) code is a standard Multics status code. (Output)

To finish an output segment and give it "re" access:

```
declare ti_$finobj entry (ptr, fixed bin(35), fixed bin(35),
        fixed bin);

call ti_$finobj (segp, bitcnt, aclinfo, code);
```

1) segp as above. (Input)

2) bitcnt is the bit count of the output segment. (Input)

3) aclinfo as above. (Input)

4) code as above. (Output)

To finish an output segment and give it "rwa" access:

```
declare ti_$findata entry (ptr, fixed bin(35), fixed bin(35),
        fixed bin);

call ti_$findata (segp, bitcnt, aclinfo, code);
```

Arguments are as for ti_$finobj.

To clean up after an interrupt:

```
declare ti_$clean_up entry (fixed bin(35));

call ti_$clean_up (aclinfo);
```

The argument is as above.

New Methods for Segments

The entry tssi_$get_segment is equivalent to ti_$getseg, except that the fourth argument, aclinfo, is a standard pointer datum rather than coded information.

The entry tssi_$finish_segment performs the functions of both ti_$finobj and ti_$findata. It has an additional argument (in the third position) which specifes the access to be placed on the segment. Again, the aclinfo argument is a standard pointer datum.

The entry tssi_$clean_up_segment is equivalent to ti_$clean_up, again with aclinfo a standard pointer datum.

Multisegment Files

Entries for handling multisegment files do not exist in ti_. See the MPM Subsystem Writers' Supplement write-up of tssi_ for their usage.

 (END)

STANDARD CHECKSUM

This write-up describes a technique for computing a full word checksum on the Honeywell 6180 computer. This technique is the Multics standard technique.

Algorithm

Checksums are computed using the "awca" instruction followed by an "alr 1" instruction. Upon completion of checksum computation, two "awca 0,dl" instructions are executed to include all carries in the checksum.

A typical checksum computation scheme follows:

```
        ldi     =o004000,dl     inhibit overflow fault
        sti     indics          save indicators
        lda     0,dl            initialize "a" to zero
        eaxl    0               count locations in xl

loop:   ldi     indics          restore indicators
        awca    word,1          add with carry to checksum
        sti     indics          save indicators (they get
                                clobbered by cmpxl)
        alr     1               rotate "a" left
        eaxl    1,1             count 1 location and
        cmpxl   size,du         check for completion
        tnc     loop            loop

        ldi     indics          restore indicators
        awca    0,dl            add in carry, if any
        awca    0,dl            in case carry generated by
                                last instruction
        sta     cksum           save the checksum
```

 (END)

HARDWARE FEATURES TO AVOID

This write-up documents a number of restrictions on usage of the 6180 that should be observed when writing programs to operate in the Multics environment. Some of these restrictions are enforced by the Multics supervisor; others, while not enforced, should be followed to minimize the effect of potential supervisor or hardware changes. The Multics system uses these features, but does so in a controlled way. All instances of their use are localized in a very few procedure segments to minimize the effects of changes.

Consult the 6180 processor manual for descriptions of the instructions and modifiers listed below.

Hard-to-Interrupt Instructions and Modifiers

The 6180 processor has in its repertoire a number of instructions and special modifiers with the property that an instruction, once begun:

1) might not be able to complete execution because of a missing page or pending interrupt;

2) cannot be scrapped and restarted from the beginning because a core location or register has already been modified.

Such instructions must be interruptable in mid-execution in such a way that they can be continued at a later time; elaborate special-purpose hardware has been provided to snapshot the entire processor state including internal registers when an interrupt or fault occurs.

The cost of providing this interruptability is quit high for two reasons.

1) The special-purpose hardware is not needed for any other function.

2) Every interrupt and fault (not just those occuring during execution of a hard-to-interrupt instruction) must be started and ended with a pair of relatively long instructions requiring 11 microseconds to save the snapshot in core, and restore it to the processor, repectively. In addition, following every fault or interrupt on which control was returned to the user, the store machine conditions must be checked for validity by a procedure that performs about a dozen tests.

Since it is unlikely that these continually paid costs are paid back by the time saved in occasional use of these instructions, a hardware change to force a compatibliity fault when they are used would allow removal of both of the above costs, and it would also permit addition of an interpreter procedure that simulates the effect desired but using more easily interruptible instructions. Thus it is unadvisable to utilize any of the instructions or modifiers in question, so as to make as simple as possible any future hardware change along this line.

The following instructions and modifiers are included in the above discussion.

1) Instructions:

XED Execute double

RPT Repeat

RPD Repeat double

RPL Repeat link

2) Modifiers that change the indirect word:

CI

DI

AD

SD

ID

DIC

IDC

SC

SCR

Extended Instruction Set

The 6180 has in its repertoire a set of single-word and multiple-word instructions for bit and character string manipulation and for decimal arithmetic. At present they do not work dependably and, thus, should be avoided. The following instructions make up the extended instruction set.

MLR	Move Alphanumeric Left to Right
MRL	Move Alphanumeric Right to Left
MVE	Move Alphanumeric Edited
CMPC	Compare Alphanumeric Character String
SCD	Scan Character Double
SCDR	Scan Character Double in Reverse
TCT	Test Character and Translate
TCTR	Test Character and Translate in Reverse
SCM	Scan with Mask
SCMR	Scan with Mask in Reverse
MVN	Move Numeric
CMPN	Compare Numeric
MVNE	Move Numeric Edited
AD3D	Add Using 3 Decimal Operands
AD2D	Add Using 2 Decimal Operands
SB3D	Subtract Using 3 Decimal Operands
SB2D	Subtract Using 2 Decimal Operands
MP3D	Multiply Using 3 Decimal Operands
MP2D	Multiply Using 2 Decimal Operands
DV3D	Divide Using 3 Decimal Operands
DV2D	Divide Using 2 Decimal Operands
CSL	Combined Bit Strings Left
CSR	Combined Bit Strings Right
SZTL	Set Zero and Truncation Indicators with Bit Strings Left
SZTR	Set Zero and Truncation Indicators with Bit Strings Right
CMPB	Compare Bit Strings
DTB	Decimal to Binary Convert
BTD	Binary to Decimal Convert
LARn	Load Address Register n
LAREG	Load Address Registers
SARn	Store Address Register n
SAREG	Store Address Registers
AWD	Add Word Displacement to Specified AR
A9BD	Add 9-Bit Character Displacement to Specified AR
A6BD	Add 6-Bit Character Displacement to Specified AR
A4BD	Add 4-Bit Character Displacement to Specified AR
ABD	Add Bit Displacement to Specified AR
SWD	Subtract Word Displacement from Specified AR
S9BD	Subtract 9-Bit Character Displacement from Specified AR

S6BD	Subtract 6-Bit Character Displacement from Specified AR
S4BD	Subtract 4-Bit Character Displacement from Specified AR
SBD	Subtract Bit Displacement from Specified AR
AARn	Alphanumeric Descriptor to ARn
NARn	Numeric Descriptor to ARn
ARAn	ARn to Alphanumeric Descriptor
ARNn	ARn to Numeric Descriptor
LPL	Load Pointers and Lengths
SPL	Store Pointers and Lengths

 (END)

COMMANDS AND ACTIVE FUNCTIONS

This section contains, in alphabetic order, descriptions of all standard Multics commands. The user of this section will also want to refer to the Reference Guide section on the Command Language Environment, which contains

1. A guide to the commands, organized by function.
2. An alphabetic list of command name abbreviations.
3. A description of the Multics command language.
4. An explanation of the role of active functions.

The following conventions are used in command descriptions:

1. In command usage, optional arguments are shown surrounded with hyphens. For example,

   ```
   locate name1 -name2-
   ```

   would indicate that the locate command has a mandatory first argument and an optional second argument.

2. In command usage, the ellipsis form

   a1 ... an

   is used to indicate a variable number of arguments all having the same form as a1 and an.

Note that commands may be distinguished from subroutines by name; in general, subroutines have segment names which end with a trailing underscore.

Commands not found in this section may possibly be listed in the Reference Guide sections on Obsolete Procedures or Internal Interfaces.

(END)

Name: abbrev, ab

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

When it is entered, the abbrev command sets up a special command processor that is invoked for each command line input to the system. The abbrev command processor checks each input line to see if is is an abbrev request 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 (recognized by a period (.) as the first nonblank character of the line) and abbreviations are included in the line, then 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.

Usage

abbrev

Notes

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

>user_dir_dir>States>Washington>Washington.profile

The profile being used by abbrev can be changed at any time with the .u control request (see below) to any profile in the storage system hierarchy to which the user has appropriate access. The entry name of a profile segment must have the suffix ".profile". A new profile can be created by specifying a nonexisting segment to the .u control request. The segment is then created and initialized as a profile segment, assuming the user has the necessary access access permission. The user must be careful not to delete or terminate the segment that is currently being used as the profile.

The user can suppress expansion of a particular string in a command line by enclosing it within double quote characters ("). To supress 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 startup.ec segment so that he is automatically able to abbreviate whenever he is logged in. For an explanation of the startup.ec segment, see the MPM Reference Guide section, Protocol for Logging In, under Start Up.

Control Requests

Before abbrev expands a command line (to pass it on to the normal command processor), it first checks to see if the command line is an abbrev request line. An abbrev request line is recognized by its having a period (.) as the first nonblank character of the line. This means that an abbrev request is recognized only at the front of a command line. Any command line interpreted as an abbrev request line is treated specially and is neither checked for embedded abbreviations nor (even in part) passed on to the normal command processor. The one exception to this rule is a command line with a apsce character following the period; the rest of the line is passed to the normal command processor with no expansion being done.

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

.a <abbr> <rest of line>

Add the abbreviation <abbr> to the current profile. 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. The string it stands for must be no longer than 132 characters.

.ab <abbr> <rest of line>

Add an abbreviation which 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 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 which 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.

.f — Enter a mode (the default mode) which forgets each command line after executing it. See the .r and .s requests.

.l <abbr1> ... <abbrn> — List the specified abbreviations with the strings they stand for. If no abbreviations are specified, all abbreviations in the current profile are listed. If no letters are specified, all abbreviations in the current profile 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 are listed.

.q — Quit. 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 explictly invoked again.

.r	Enter a mode which remembers the last line expanded by abbrev. See the .f and the .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 profile that the user wants to use. <profile> is the pathname of the profile to be used and can contain abbreviations already defined.
.p	This request prints the name of the profile being used.
.<space> <rest of line>	If the request character is a space, the entire command line is passed on to the normal command processor (after removing the period) with no expansion being performed. The user can thus 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 which 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 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 which abbrev treats as break characters are:

tab
new line
space
double quote "
dollar sign $

apostraphe	'
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" a lot as he edits his FORTRAN source segments. 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
```

Note that 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 commmand 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
```

However, if he typed

```
w example.ft
```

he would create a segment by exactly that name (example.ft).

Command
Standard Service System
1/27/72

Name: addname, an

The addname command adds an alternate name to the existing name(s) of the segment, directory, or link specified. See also deletename and rename in the MPM.

Usage

addname path entry1 ... entryn

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

2) entryi is the additional name(s) to be added to the segment, directory, or link.

Notes

The user must have write access on the directory containing the segment, directory, or link to be modified.

The equals and star conventions may be used.

The entryi argument must be unique in the directory. If there is a duplication, the initial instance of entryi will be removed and the user will be informed of this action, unless removing the initial instance would leave the segment, directory, or link without a name. In the latter case, the user will be interrogated as to whether he wishes the segment, directory, or link deleted; if he does not, entryi will not be added to the segment, directory, or link specified.

Example

addname >sys_lib>Smith.Multics.pl1 Jones.==

would add the name Jones.Multics.pl1 to the segment Smith.Multics.pl1 in the directory sys_lib.

(END)

Command
Development System
7/29/71

Name: adjust_bit_count, abc

The adjust_bit_count command may be used to set the bit count of segments which for some reason do not have the bit count set properly (e.g., the program which was creating the segment got a fault, or the process terminated without the bit count being set, etc.).

Usage

adjust_bit_count path1 ... pathn -option-

1) pathi — are the pathnames of the segments to be adjusted.

2) option — may be -character (-ch). If the option occurs anywhere on the command line, it applies to all pathname arguments such that resetting of the bit count is done to the last nonzero character in the segment. The default is to reset the bit count to correspond to the last nonzero 36 bit word in the segment.

   If the bit count could be computed but could not be reset (e.g., improper access to the segment), the computed value will be printed such that the user may then use the set_bit_count command (see the MPM) after resetting access or other necessary corrective measures.

(END)

Name: alm

ALM is the standard Multics assembly language. It is commonly used for privileged supervisor code, compiler support operators and utility packages, and data bases. It is occasionally used for efficiency or to use hardware features not accessible in compiler languages; however, its routine use is discouraged.

The ALM language is described briefly in this section as there is no language reference manual available to users. The 6180 Processor Reference Manual has not yet been published, but the 645 Processor Reference Manual (amended for 6180 processor) can be used to understand the instruction set.

The alm command invokes the ALM assembler to translate a segment containing the text of an assembly language program into a Multics standard object segment. A listing segment can also be produced. These segments are placed in the user's current working directory.

Usage

alm segment_name -control_arguments-

1) segment_name — specifies the path name of the source program to be assembled. The suffix ".alm" is added automatically by the alm command unless it is already present.

2) control_arguments — control optional functions of the assembler. They can only appear after the segment_name argument, and none are required. Legal control arguments are:

-list, -ls — An assembly listing segment is produced if and only if this control argument is specified.

-no_symbols — By default the listing segment produced by the -list control argument contains a cross-reference table. This control argument suppresses the table.

-quiet, -qe — This control argument prevents errors from being typed out on the terminal. Errors are flagged in the listing (if any) in any case.

Notes

The entry name of an ALM source segment consists of the name of the object segment concatenated with the string ".alm". Similarly the name of the listing segment produced by the assembly has a name consisting of the name of the object segment concatenated with the string ".list". The pathname argument to the alm command identifies the source segment; if the ".alm" suffix is omitted, the command appends it. This does not affect the names of the object and listing segments.

The assembly listing is made into a multisegment file if necessary.

The assembler is serially reusable and sharable, but not reentrant. That is, it cannot be interrupted during execution, invoked again, then restarted in its previous invocation.

Error Conditions

Errors arising in the command interface, such as inability to locate the source segment, are reported in the normal Multics manner. Some conditions can arise within the assembler that are considered to be malfunctions in the assembler; these are reported by a line typed out and also in the listing. Either of the above cases is immediately fatal to the translation.

Errors detected in the source program, such as undefined symbols, are reported by placing one-letter error flags at the left margin of the offending line in the listing file. Any line so flagged is also printed on the user's terminal, unless the -quiet control argument is in effect. Flag letters and their meanings are given below.

B mnemonic used belongs to obsolete (645) processor instruction set.

C obsolete (645 compatibility check).

E malformed expression in arithmetic field.

F error in formation of pseudo-operation operand field.

M reference to a multiply-defined symbol.

N unimplemented or obsolete pseudo-operation.

O unrecognized opcode.

P phase error. Location counter at this statement has changed between passes, possibly due to misuse of org pseudo-operation.

R expression has invalid relocatability.

S error in the definition of a symbol.

T undefined modifier (tag field).

U reference to an undefined symbol.

X segdef pseudo-operation used in mastermode or executeonly procedure.

7 digit 8 or 9 appears in an octal field.

The errors B, E, M, O, P, and U are considered fatal. If any of them occurs, the standard Multics "Translation failed" error message is reported after completion of the translation.

ALM Language

An ALM source program is a sequence of statements separated by new line characters or semicolons. The last statement must be the end pseudo-operation.

Fields must be separated by white space, which is defined to include space, tab, new page, and percent characters.

A name is a sequence of upper and lower case letters, digits, underscores, and periods. It must begin with a letter or underscore, and cannot be longer than 31 characters.

Labels

Each statement can begin with any number of names, each followed immediately by a colon. Any such names are defined as labels, with the current value of the location counter. Note that a label on a pseudo-operation that changes location counters or forces even alignment (such as org or its) might not refer to the expected location. White space can appear before, after, or between labels, but not before the colon, and no white space is required.

Opcode

The first field after any labels is the opcode. It can be any instruction mnemonic described in the appropriate processor reference manual, or any one of the pseudo-operations listed below. It can be omitted, and any labels are still defined. White space can appear before the opcode, but is not required.

Operand

Following the opcode, and separated from it by mandatory white space, is the operand field. For instructions, the operand defines the address, base, and tag (modifier) of the instruction. For each pseudo-operation, the operand field is described below in the list of pseudo-operations. The operand field can be omitted in an instruction. Those pseudo-operations that use their operands generally do not permit the operand field to be omitted.

Comments

Since the assembler ignores any text following the end of the operand field, this space is commonly used for comments. In those pseudo-operations that do not use the operand field, all text following the opcode is ignored and can be used for comments. Also, a quote character (") in any field introduces a comment that extends to the end of the statement. (The only exceptions are the acc and aci pseudo-operations, for which the quote character can be used to delimit literal character strings.) Note that semicolon ends a statement and therefore ends a comment as well.

Instruction Operands

The operand field of an instruction can be of several distinct formats. Most common is the direct specification of base, address, and modifier. This consists of three subfields, any of which can be omitted. The first subfield specifies a base register by number, user-defined name, or predefined name (ap, ab, bp, bb, lp, lb, sp, sb). The subfield ends with a vertical bar. If the base register and bar are omitted, no base register is used in the instruction.

The second subfield is any arithmetic expression, relocatable or absolute. This is the address part of the instruction, and defaults to zero. Arithmetic expressions are defined below.

The last subfield is the modifier or tag. It is separated from the preceding subfields by a comma. If the tag subfield and comma are omitted, no instruction modification is used. (This is an all zero modifier.) Legal modifiers are defined below.

Other formats of instruction operands are used to imply base registers. These cannot have the base register subfield specified explicitly.

If a symbolic name defined by temp, tempd, or temp8 is used in the address subfield (it can be used in an arithmetic expression) then the base register sp is implied. This form can have a tag subfield.

Similarly, if an external expression is used in the address subfield then the base register lp is implied; this causes a reference through a link. If a modifier subfield is specified, it is taken as part of the external expression; the instruction has an implicit n* modifier to go through the link pair. External expressions are defined below.

A literal operand begins with an equals sign followed by a literal expression. The literal expression can be enclosed in parentheses. It has no base register but can have a tag subfield. A literal reference normally causes the instruction to refer to a word in a literal pool that contains the value of the literal expression. However, if the modifier du or dl is used, the value of the literal is placed directly in the instruction address field. Literal expressions are defined below.

Examples of Instruction Statements

```
xlab:   lda     ap|2,*                          " Example 1.
        eax7    xlab-1

        rccl    <sys_info>|clock_ ,*            " Example 2.

        segref  sys_info,time_delta             " Example 3.
        adl     time_delta+1

        temp    nexti                           " Example 4.
        lxl0    nexti,*

        link    goto,<unwinder_>|unwinder_      " Example 5.
        tra     lp|goto,*

        ana     =o777777,du                     " Example 6.
        ada     =v36/list_end-1
```

Example 1 shows direct specification of address, base, and tag fields. In the second instruction, no base is specified, and the symbol xlab is not external, so no base is used.

Example 2 shows an explicit link reference. Indirection is specified for the link as the item at clock_ (in sys_info) is merely a pointer to the final operand.

Example 3 uses an external expression as the operand of the adl instruction. In this particular case, the operand itself is in sys_info.

Example 4 uses a stack temporary. Since the word is directly addressable using sp, the modifier specified is used in the instruction.

Example 5 shows a directly specified operand that refers to an external entity. It is necessary in this case to specify the base and modifier fields, unlike segref.

Example 6 uses two literal operands. Only the second instruction causes the literal value to be stored in the literal pool.

Arithmetic Expression

An arithmetic expression consists of names (other than external names) and decimal numbers joined by the ordinary operators + - * /. Parentheses can be used with the normal meaning.

An asterisk in an expression, when not used as an operator, has the value of the current location counter.

All intermediate and final results of the expression must be absolute or relocatable with respect to a single location counter. A relocatable expression cannot be multiplied or divided.

Logical Expression

A logical expression is composed of octal constants and absolute symbols combined with the Boolean operators + (OR), - (XOR), * (AND), and ¬ (NOT). Parentheses can be used with the normal meaning.

External Expression

An external expression refers symbolically to some other segment. It consists of an external name or explicit link reference, an optional arithmetic expression added or subtracted, and an optional modifier subfield. An external name is one defined by the segref pseudo-operation. An explicit link reference must begin with a segment name enclosed in angle brackets and followed by a vertical bar. This can optionally be followed by an entry name in square brackets. For example:

```
<segname>| entryname
<segname>|0,5*
```

A segment name of *text or *link indicates a reference to this procedure's text or linkage sections.

A link pair is constructed for each combination of segment name, entry name, arithmetic expression, and tag that is referenced.

Literal Expression

A literal reference causes the instruction to refer to a word in a literal pool that contains the value specified. An exception occurs, in that the modifiers du and dl cause the value to be stored directly in the address field of the instruction; the effect is the same when executed. The various formats of literals are described as follows.

A decimal literal can be signed. If it contains a decimal point or exponent, it is floating point. If the exponent begins with "d" instead of "e", it is double precision. A binary scale factor beginning with "b" indicates fixed point, and forces conversion from floating point.

An octal literal begins with an "o" followed by up to twelve octal digits.

ASCII literals can occur in two forms: one begins with a decimal number between 1 and 32 followed by "a" followed by that many data characters, which can cross statement delimiters. The other form begins with "a" followed by up to four data characters, which can be delimited by the new line character.

A GBCD literal begins with "h" followed by up to six data characters, which can be delimited by the new line character. Translation is performed to the 6-bit character code.

An ITS (ITP, ITB) literal begins with "its" ("itp", "itb") followed by a parenthesized list containing the same operands accepted by the its (itp, itb) pseudo-operation. The value is the same as that created by the pseudo-operation.

A variable-field literal begins with "v" followed by any number of decimal, octal, and ASCII subfields as in the vfd pseudo-operation. It must be enclosed in parentheses if a modifier subfield is to be used.

Modifiers

These specify indirection, index register address modification, immediate operands, and miscellaneous tally word operations. They can be specified as 2-digit octal numbers (particularly useful for instructions like stba), or symbolically using the mnemonics described here.

Simple register modification is specified by using any of the register designators listed in the table. It causes the contents of the selected register to be added to the final effective address.

Register-then-indirect modification is specified by using any of the register designators followed by an asterisk. If the asterisk is used alone it is equivalent to the n* modifier. The register is added into the effective address, then the address and modifier fields of the word addressed are used in determining the final effective address. Indirection cycles continue as long as the indirect words contain an indirection modifier.

Indirect-then-register modification is specified by placing an asterisk before any one of the register designators listed below. See the processor manual.

Direct modifiers are du and dl. They cause an immediate operand word to be fabricated from the address field of the instruction. For dl, the 18 address bits are right-justified in the effective operand word; for du they are left-justified. In either case, the remaining 18 bits of the effective operand are filled with zeros.

Segment addressing modifiers its, itb, and itp can only occur in an indirect word pair on a double-word boundary. its causes the address field of the even word to replace the segment number of the effective address, then continues the indirect cycle with the odd word of the pair. Nearly all indirection in Multics uses ITS pairs. For itb and itp, see the processor manual.

Tally modifiers i, ci, sc, scr, ad, sd, id, di, idc, and dic control incrementing and decrementing of the address and tally fields in the indirect word. They are difficult to use in Multics because the indirect word and the data must be in the same segment. See the processor manual.

Fault tag modifiers f1, f2, and f3 cause distinct hardware faults whenever they are encountered. f2 is reserved for use in the Multics dynamic linking mechanism; the others result in the signalling of the conditions "fault_tag_1" and "fault_tag_3".

designators		register
x0	0	index register 0
x1	1	index register 1
x2	2	index register 2
x3	3	index register 3
x4	4	index register 4
x5	5	index register 5
x6	6	index register 6
x7	7	index register 7
n		none (no modification)
au		A bits 0-17
al		A bits 18-35 or 0-35
qu		Q bits 0-17
ql		Q bits 18-35 or 0-35
ic		instruction counter

EIS modifiers

An EIS modifier appears in the first word of an EIS multi-word instruction. It affects the interpretation of operand descriptors in subsequent words of the instruction. No check is made by ALM that the modifier specified is consistent with the operand descriptor specified elsewhere.

An EIS modifier consists of one or more subfields separated by commas. Each subfield contains either a keyword as listed below, or a register designator, or a logical expression. The values of the subfields are OR'ed together to produce the result.

keyword	meaning
pr	Descriptor contains a base register reference.
id	Descriptor is an indirect word pointing to the true descriptor.
rl	Descriptor length field names a register containing data length.

Pseudo-operations

end
terminates the source file.

include segmentname
inserts the text of the segment segmentname.incl.alm immediately after this statement. A standard include library search is done to find the include file. See the MPM Reference Guide section, The System Libraries and Search Rules.

name objectname
respecifies the object segment name as it appears in the object segment. By default the storage system name is used.

use name
assembles subsequent code into the location counter name. Default location counter is ".text.".

join /text/name1,name2,.../link/name3,name4,...
appends the location counters name1, name2, etc. to the text section, and appends the location counters name3, name4, etc. to the linkage section. The text and link parts can be used alone or together; any number of names can appear. Each name must have been previously referred to in a use statement. Any location counters not joined are appended to the text section.

org expression
sets the location counter to the value of the absolute arithmetic expression expression. The expression must not use symbols not previously defined.

even
odd
eight
sixtyfour
inserts padding (nop) to a specified word boundary.

mod expression
inserts padding (nop) to an (expression) word boundary.

mastermode
executeonly
requests special entry-checking code. Obsolete.

inhibit on
inhibit off
sets a mode affecting interrupt inhibit bit that is assembled into subsequent instructions. For system use.

null
rem
is ignored. Used for comments.

bool name,expression
defines the symbol name with the logical value expression. See the definition of logical expressions above.

equ name,expression
defines the symbol name with the arithmetic value expression.

set name,expression
assigns the arithmetic value expression to the symbol name. Its value can be reset in other set statements.

link name,extexpression
defines the symbol name with the value equal to the offset from lp to the link pair generated for the external expression extexpression. Note that an external expression can include a tag subfield. Note also that name is not an external symbol, so an instruction should refer to this link by:
lp|name,*

segref segname,name1,name2,...
defines the symbols name1, name2, etc. as external symbols referencing the entry points name1, name2, etc. in segment segname. This defines a symbol with an implicit base register reference.

temp name1(n1),name2(n2),...
defines the symbols name1, name2, etc. to reference unique stack temporaries of n1, n2, etc. words each. n1, n2, etc. are absolute arithmetic expressions, and can be omitted (the parentheses should also be omitted). The default is one word per name.

tempd name1(n1),name2(n2),...
is similar to temp, except that n1 (n2, etc.) double words are allocated, each on a double word boundary.

temp8 name1(n1),name2(n2),...
is similar to temp, except that 8-word units are allocated, each on an 8-word boundary.

acc /string/
assembles the ASCII string string into as many contiguous words as are required (up to forty-two). The delimiting character / can be any non-white-space character. The quoted string can contain new line and semicolon characters. The length of the string is placed in the first character position in acc format.

aci /string/
is similar to acc, but no length is stored. The first character position contains the first character in aci format.

bci /string/
is similar to aci, but uses GBCD six-bit character codes.

dec number1,number2,...
assembles the decimal integers number1, number2, etc. into consecutive words.

oct number1,number2,...
is like dec, with octal integer constants.

zero expression1,expression2
assembles expression1 into the left 18 bits of a word and expression2 into the right 18 bits. Both subfields default to zero.

arg operand
is assembled exactly like an instruction with a zero opcode. Any form of instruction operand may be used.

vfd T1L1/expression1,T2L2/expression2,...
is variable format data. expressioni is of type Ti and is stored in the next Li bits of storage. As many words are used as required. Individual items can cross word boundaries and exceed 36 bits in length. Type is indicated by the letters "a" (ASCII constant) or "o" (logical expression) or none (arithmetic expression). Regardless of type, the low order Li bits of data are used, padded if needed on the left. Ti can appear either before or after Li.

Restrictions: The total length cannot exceed ten words. A relocatable expression cannot be stored in a field less than 18 bits long, and it must end on either bit 17 or bit 35 of a word.

bss name,expression
defines the symbol name as the address of a block of expression words at the current location. name can be omitted, in which case the storage is still reserved.

bfs name,expression
is like bss, but name is defined as the address of the first word after the block reserved.

its segno,offset,tag
generates an ITS pointer to the segment segno, word offset offset, with optional modifier tag. If the current location is not even, a word of padding (nop) is inserted. Note that such padding causes any labels on the statement to be incorrectly defined.

itp baseno,offset,tag
itb baseno,offset,tag
generates an ITB (ITP) pointer referencing the base register baseno. Not commonly used in Multics.

firstref extexpression1(extexpression2)
the procedure extexpression1 is to be called with the argument pointer extexpression2 the first time (in a process) that this object segment is linked to by an external symbol. If extexpression2 and the parentheses are omitted, an empty argument list is supplied. The expressions are any external expressions, including tags.

segdef name1,name2,...
makes the labels name1, name2, etc. available to the linker for referencing from outside programs, using the symbolic names name1, name2, etc. Such incoming references go directly to the labels name1, name2, etc., so the segdef pseudo-operation is usually used for defining external static data. For program entry points the entry pseudo-operation is usually used.

entry name1,name2,...
generates entry sequences for labels name1, name2, etc. and makes the externally-defined symbols name1, name2, etc. refer to the entry sequence code rather than directly to the labels. The entry sequence performs such functions as initializing base register lp to point to the linkage section, which is necessary to make external symbolic references (link, segref, explicit links). The entry sequence can use (alter) base register bp, index registers 0 and 7, and the A and Q. It requires sp and sb to be properly set (as they normally are).

getlp
sets the base register lp to point to the linkage section. This can be used with segdef to simulate the effect of entry. This operator can use base register bp, index registers 0 and 7, and the A and Q, and requires sp and sb to be set properly.

save expression
push expression
creates a new stack frame for this procedure, containing expression words. If expression is omitted (the usual case), the frame is just large enough to contain all cells reserved by temp, tempd, and temp8. This operator can use base registers bp and sp, index registers 0 and 7, and the A and Q, and requires sp and sb to be set properly.

short return
is used to return from a procedure which has not performed a save. This operator requires sp and sb to be set properly.

return
is used to return from a procedure which has performed a save. This operator requires sp and sb to be set properly.

call routine(arglist)
calls out to the procedure routine using the argument list at arglist. Both routine and arglist can be any legal instruction operand, including tags. If arglist and the parentheses are omitted, an empty argument list is created. All registers are saved and restored by call. This operator requires that sp and sb be set properly.

short call routine
calls out to routine using the argument list pointed to by ap. Only lp, sp, and sb are preserved by short call, and sp and sb must be properly set.

rpt tally,delta,term1,term2,...
generates the machine rpt instruction as described in the processor manual. tally and delta are absolute arithmetic expressions. The termi specify the termination conditions as the names of corresponding conditional transfer instructions. This same format can be used with the rpt, rpd, rpda, and rpdb pseudo-operations.

rptx ,delta
generates the machine rpt instruction with a bit set to indicate that the tally and termination conditions are to be taken from index register 0. This format can be used with rplx and rpdx.

eax index,operand
assembles into an eaxn instruction, where n is the value of the absolute arithmetic expression index. This format can be used for all index register instructions.

tsp base,operand

assembles into a tspn instruction, where n is found as follows: If base is a built-in base name (ap, ab, etc.) that register is selected. Otherwise, base must be an absolute arithmetic expression whose value is n. This format can be used for all base register instructions except spri.

awd operand
awdx operand

generates the 6180 EIS instruction awd as described in the processor manual. operand must specify a base register, as this instruction selects its output register that way. The awdx pseudo-operation causes the offset to be cleared before the addition, thus effecting a load. This format can be used with abd, a4bd, sbd, sbdx, etc.

mlr (MF1),(MF2),fill(octexpression),enablefault

generates the first word of an EIS multiword instruction. MF1 and MF2 are EIS modifier fields as described above. Certain keywords (fill, bool, and mask) require logical expression operands that specify the bits to be placed in the appropriate parts of the instruction. Other keywords (round, enablefault, ascii) cause single option bits in the instruction to be set. Keywords can occur in any order, before or after any MF fields. This format can be used for all 6180 EIS multiword instructions.

desc4a address(offset),length
desc6a address(offset),length
desc9a address(offset),length

generates the operand descriptor that usually follows the first word of an EIS multiword instruction. address is any arithmetic expression, possibly preceded by a base register subfield as in an instruction operand. offset is an absolute arithmetic expression giving the offset (in characters) to the first bit of data. It can be omitted if the parentheses are also omitted. length is either a built-in index register name (al, au, ic, x0, etc.) or an absolute arithmetic expression for the data length field of the descriptor. The character size (in bits) is specified as part of the pseudo-operation name.

descb address(offset),length

generates an operand descriptor for a bit string. offset and length are in bits.

```
desc4fl   address(offset),length,scale
desc4ls   address(offset),length,scale
desc4ns   address(offset),length,scale
desc4ts   address(offset),length,scale
```

generates an operand descriptor for a decimal string. scale is an absolute arithmetic expression for a decimal scaling factor to be applied to the operand. It can be omitted, and is ignored in a floating-point operand. Data format is specified in the pseudo-operation name: desc4fl indicates floating point, desc4ls indicates leading sign fixed point, desc4ns indicates unsigned fixed point, and desc4ts indicates trailing sign fixed point. Nine-bit digits can be specified by using desc9fl, desc9ls, desc9ns, and desc9ts.

 (END)

Command
8/20/73

Name: alm_abs, aa

This command submits an absentee request to perform ALM assemblies. The absentee process for which alm_abs submits a request assembles the segments named, appends the output of print_link_info for each segment to the segment segnamei.list if it exists, and dprints and deletes segnamei.list. If the -output_file control argument is not specified, an output segment, segname.absout, is created in the user's working directory (if more than one segname is specified, the first is used). If the segment to be assembled cannot be found, no absentee request is submitted.

Usage

alm_abs segname1 ... segnamen -alm_control_args-
-alm_abs_control_args-

1) segnamei — is the path name of a segment to be assembled.

2) alm_control_args — can be one or more nonobsolete control arguments accepted by the ALM assembler and described in alm. (See the write-up in the MPM.)

3) alm_abs_control_args — can 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 <= 3). The default queue is 3. segnamei.list is also dprinted in queue n.

-copy n, -cp n — specifies the number of copies (n <= 4) of segnamei.list to be dprinted. The default is 1.

-hold — specifies that alm_abs should not dprint or delete segnamei.list.

-output_file f, -of f — specifies that absentee output is to go to segment f where f is a path name.

Notes

Control arguments and segment names can be mixed freely and can appear anywhere on the command line after the command. All control arguments apply to all segment names. An unrecognizable control argument causes the absentee request not to be submitted.

Expanded segments containing include files are not deleted.

Unpredictable results can occur if two absentee requests are submitted which could simultaneously attempt to assemble the same segment or write into the same .absout segment.

When doing several assemblies, 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 links that need to be snapped when setting up a process and when invoking the assembler need be snapped only once.

Command
Development System
5/17/72

Name: answer

This command provides a preset answer to any questions asked by some other command. It does this by establishing a handler for the condition command_question, and then executing the subject command. If the subject command calls command_query_ to ask a question, the handler will be invoked to supply the answer. The handler is reverted when "answer" exits.

Usage

answer ans -control_args- commandline

1) ans — is the desired answer to any question.

2) control_args — may be chosen from the following list of control arguments:

-brief, -bf — suppresses printing of both the question and the answer.

-times n — gives the prespecified answer n times only (where n is an integer); then it acts as if "answer" had not been called.

3) commandline — is any Multics command line.

Notes

If a question is asked which requires a yes or no answer, and the preset answer is neither yes nor no, the handler will not be invoked.

Examples

To delete a directory without being asked if you want to delete it:

```
answer yes -bf dd test_dir
```

To see the first three blocks of an info segment named fred.info, and then be interrogated:

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

To see only the first three blocks:

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

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

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

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

(END)

Name: apl

The apl command invokes the Multics APL interpreter, which is completely described in the APL User's Guide. The Multics APL language is nearly identical to the language APL/360, which has earned wide acceptance.

APL can be characterized as a line-at-a-time desk calculator with many sophisticated operators and a limited stored-program capability. One needs little or no prior acquaintance with digital computers 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 up 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 console at the 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 again on another day.

Usage

apl

Notes

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

 (END)

Command
11/8/72

Name: archive, ac

An archive segment is a single segment which is formed by combining together an arbitrary number of separate segments; these constituent parts of an archive segment are called components. The archive segment is particularly useful as a means of conserving storage space by eliminating the breakage which occurs when the contents of segments do not fill complete pages of storage. It is convenient as a means for packaging segments; it is used in that manner when interfacing with the Multics binder.

The archive command is furnished to maintain archive segments for the user. There are four general classes of operations performed by the archive command. These are:

1) printing a table of contents;
2) extracting components;
3) deleting components;
4) replacing, updating, or appending components.

The first two classes of operations use the contents of archive segments; the last two classes of operations change the contents of archive segments. Various features are combined with these operations to ease the use of archive segments.

The copy feature may be combined with the replacement and deletion operations. It causes the updated archive to be placed in the working directory when the original archive segment is found elsewhere in the storage system. The copy feature behaves as if the archive segment were first copied into the user's working directory and then updated as requested.

Deletion can be combined with the replacement operations to cause segments to be deleted from the storage system after they have been replaced or added to an archive segment. The force feature can be used in conjunction with deletion to cause the safety function to be bypassed. (This is analogous to the operation of the deleteforce command.) This form of deletion should not be confused with the operation which deletes components from archive segments.

The update feature causes components of archives to be replaced only if the date-time modified of the segment in the storage system is later than that associated with the component in the archive. If a component requested for updating is not found in the archive, it is not added to the archive.

The append feature can be used during replacement when additions only are to be made. If the archive command finds a component already present in the archive segment, a diagnostic is printed and replacement is not performed.

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

The star convention can be applied to the archive segment path name during the extraction and table of contents operations; it cannot be used during replacement and deletion operations. Component names may not be specified using the star convention.

No commands other than archive, archive_sort, and reorder_archive should be used to manipulate the contents of archive segments; using a text editor or other similar commands will result in unspecified behavior.

Usage

archive key archivepath path1 ... pathn

1) key is one of the following:

Key	Function	Comments
Table of Contents Operations		
t	print table of contents	prints the entire table of contents if no components are named by the pathi arguments; otherwise it prints information about the named components only; a title and column headings are printed before the first listed component.
tl	print table of contents in long form	operates like t, printing all information for each component.
tb	print table of contents, briefly	Operates like t, except that the title and column headers are suppressed.
tlb	print tables of contents in long form, briefly	Operates likes tl, except that the title and column headers are suppressed.

Replacement Operations

r	replace	replaces or adds components in the archive. If no component names are given, all components of the archive for which segments by the same name are found in the user's working directory are replaced. When components are named, if they are found in the existing archive segment, they are replaced by segments in the storage system; otherwise they are added.
rd	replace and delete	operates like r and deletes all segments which have been placed in the archive after the archive has been updated.
rdf	replace and deleteforce	operates like r and forces deletion of all replaced segments after the archive has been updated.
cr	copy and replace	operates like r, placing the updated archive in the user's working directory instead of changing the original archive segment.
crd	copy, replace, and delete	operates like rd, placing a copy in the user's working directory.
crdf	copy, replace, and deleteforce	operates like rdf, placing a copy in the user's working directory.

Append Operations

a	append	appends named components to the archive segment. If any named component is found within the archive, a diagnostic is issued and the component is not replaced. At least one component must be named by the pathi arguments.
ad	append and delete	operates like a and deletes all appended segments after the archive has been updated.
adf	append and deleteforce	operates like a and forces deletion of all appended segments after the archive has been updated.

ca	copy, append	operates likes a, placing the new archive segment in the user's working directory.
cad	copy, append, and delete	operates like ad, placing the new archive segment in the user's working directory.
cadf	copy, append, and deleteforce	operates like adf, placing the new archive segment in the user's working directory.

Update Operations

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 found in the archive. If the component is not found in the archive, it is not added to the archive segment.
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 the new archive in the user's working directory.
cud	copy, update, and delete	operates like ud, placing the new archive in the user's working directory.
cudf	copy, update, and deleteforce	operates like udf, placing the new archive in the user's working directory.

Deletion Operations

d	delete	deletes from the archive those components specified by arguments.
cd	copy, delete	operates like d, placing the updated archive in the working directory.

Extraction Operations

x	extract	extracts from the archive those components specified by arguments, placing them in segments in the storage system, as specified by the path name arguments. The mode stored in the archive is given to the segment for the user performing the extraction. Deletes segments if already present, observing the duplicated name convention in a manner similar to the copy command. If no component names are given, it extracts all components, placing them in the working directory. The archive segment is not modified.
xf	extract and deleteforce	operates like x, deleting or removing names from any segments found where the new segment is to be created.
2)	archivepath	is the path name of the archive segment to be created or manipulated. The suffix ".archive" will be added if the user does not supply it. If the segment does not exist, it will be created for replace or append operations. The star convention may be used for extraction and table of contents operations.
3)	pathi	specifies the components to be operated on for table of contents and deletion operations. For replacement and extraction operations, it specifies the path name of a segment corresponding to a component whose name is the entry name portion of the path name. The star and equal conventions may not be used.

Notes

Each component of an archive segment retains certain attributes of the corresponding segment in the storage system. 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 are maintained. 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 associated with the component.

The archive command maintains the order of components contained within an archive segment. When new components are added, they are placed at the end.

The archive command automatically creates an archive segment during replacement operations when no original archive exists.

The archive command cannot be used recursively. Internal consistency 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.

During the operation of the archive command for replacement or deletion, because the replacement operation is not indivisible, it is possible for the updating operation to be stopped before it has been completed and after the original segment has been truncated. This may happen, for example, if a record quota overflow is received. When this situation occurs, a message is printed informing the user what has happened. In this case, the only good copy of the updated archive segment will be contained in the process directory.

Archive segments may be placed as components inside of archive segments, preserving their identity as archives, and then later may be extracted intact.

When the archive command detects an internal consistency error, it prints a status message and stops performing the requested operation. For table of contents and extraction operations, it will have completed requests for 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.

No more than thirty-two components may be named for any one request. For replacement operations where no components are named, more than thirty-two components may be replaced or appended, but if deletion is requested, only the first thirty-two segments will be deleted. A message is printed when this situation occurs.

The archive command observes the protected segment convention by interrogating the user when appropriate.

 (END)

Name: archive_sort, as

This command is used to sort the components of an archive segment. The components are sorted into ascending order by entry name using the standard ASCII collating sequence. The original archive segment is replaced by the sorted archive.

Usage

archive_sort path1 .... pathn

1) pathi is the path name of an archive segment to be sorted. (The user need not supply the ".archive" suffix.)

Notes

There may be no more than 1000 components in an archive segment which is to be sorted.

Storage system errors encountered while attempting to move the temporary sorted copy of the archive segment back into the user's original segment will result in diagnostics, and the preservation of the sorted copy in the user's process directory (telling the user its name). If the original is protected, the user will be interrogated to determine whether or not it should be overwritten.

 (END)

Name: basic

The basic command invokes the BASIC compiler to translate a segment containing BASIC source code. If the compile option is not specified, the compiled code is then executed.

Usage

basic source_name -option1- ... -optionn-

1) source_name — is the path name of the segment to be translated. The characters .basic may or may not appear as part of the path name. They must appear, however, on the segment itself.

2) optioni — is selected from the following list of options. The options may appear in any order.

-time n, -tm n — specifies a time limit of n CPU seconds where n is an integer. When the time limit is exceeded, execution stops, and the user is asked if he would like to continue execution. If he answers yes, a new timer is set giving the user the same amount of time.

-compile — indicates to compile the program and produce an object segment rather than immediately executing the code. The compiled object segment is saved in the user's working directory with the characters .dobj appended in place of .basic. The object segment is not a Multics standard object segment and can only be executed using the basic_run command.

-library, -lb — indicates that the Dartmouth library is to be searched for the source segment. No other directory is searched.

Notes

This implementation of BASIC is described in BASIC, Sixth Edition, published in 1971 by the Kiewit Computation Center, Dartmouth College, in Hanover, New Hampshire.

The following is a list of differences between the Dartmouth and Multics implementations of BASIC:

1) The Multics storage system conventions differ from those at Dartmouth. Therefore, if a user refers to a segment as

   20 file #1:"alpha"

   Multics will search for a segment named alpha in the user's working directory. If alpha is not found, the directory is searched for alpha.basic. If this is not found, the segment alpha is created.

2) The number sign (#) must be entered with an escape character preceding it to avoid the Multics interpretation as an erase character. The upward arrow character is entered as a circumflex on Multics.

The current version of the BASIC compiler is a proprietary program of Dartmouth College. It has been made available to users of the M.I.T. Information Processing Center with the permission of Dartmouth College. The BASIC compiler may not be used at other computer installations without permission of Dartmouth College.

 (END)

Command
Development System
2/16/72

Name: basic_run, br

This command will execute an object segment created by the BASIC compiler. For information on the BASIC language, see the write-up on the "basic" command in the MPM.

Usage

basic_run pathname -option1- ... -optionn-

1) pathname — is the path name of the object segment to be executed. The characters ".dobj" may or may not appear as part of the path name but must be the last component of the BASIC object segment.

2) optioni — is selected from the following list of options which may appear in any order:

-time n, -tm n — specifies a time limit of n CPU seconds where n is an integer. When the time limit is exceeded, execution stops, and the user is asked if he would like to continue execution. If he answers yes, a new timer is set giving the user the same amount of time.

-library, -lb — indicates that the Dartmouth library is to be searched for the object segment. No other directory is searched.

The current version of the BASIC compiler is a proprietary program of Dartmouth College. It has been made available to users of the M.I.T. Information Processing Center with the permission of Dartmouth College. The BASIC compiler may not be used at other computer installations without permission of Dartmouth College.

(END)

Command
Development System
1/3/72

Name: basic_system, bs

basic_system is the standard BASIC source editor and run dispatcher. A BASIC source segment path name 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 Multics "basic" command, which only compiles a program.

Usage

basic_system pathname

1) pathname is the path name of an existing or a to be created segment. The .basic suffix is assumed.

Requests

The basic_system editing requests are:

line number basic source line

adds or replaces a basic source line in proper sequence. The line number must be less than 10,000.

line number deletes that source line if such a line number exists.

save stores the current internal source segment in the segment specified in the command line.

quit returns from basic_system. The current internal segment is lost.

list prints the entire current internal segment.

run calls BASIC with the current internal source segment.

Any other type of request is ignored.

Notes

If you quit out of a BASIC compilation or execution run, immediately issue the program_interrupt (pi) command in order to get back to basic_system. Otherwise, any unsaved BASIC program within basic_system will be lost.

Refer to BASIC, Fifth Edition, published by the Kiewit Computation Center, Dartmouth College, Hanover, New Hampshire, September 1970, for detailed information on the BASIC language syntax.

The current version of the BASIC compiler is a proprietary program of Dartmouth College. It has been made available to users of the M.I.T. Information Processing Center with the permission of Dartmouth College. The BASIC compiler may not be used at other computer installations without permission of Dartmouth College.

(END)

Name: bind, bd

This is the command interface to the Multics binder which, given one or more separately translated procedure object segments, produces a single inclusive and compact bound procedure object segment. The bound segment will be in standard object segment format if all input segments are standard object segments. (See the MPM Subsystem Writers' Guide section, The Multics Standard Object Segment.) This write-up describes version 8.1 of the binder:

Usage

bind arc1 ... arcn --update- -upd1- ... -updn- -control_arg-

1) arci — is the pathname of an archive segment containing one or more component object segments to be bound. Up to 16 (current arbitrary implementation limit) input archive segments may 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 may or may 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 (updi) specifies update rather than input object segments. (See below.)

3) updi — is the pathname of an optional archive segment containing update object segments. Up to a combined total of 16 input and update segments may 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 will be bound.

4) control_arg — the bind command accepts either 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 ".list" suffix. The listing segment is generated for the purpose of dprinting, and contains the bound segment's .bind control segment, its bind map, and that information from the bound object segment which would be printed by the print_link_info command. See the MPM write-ups for dprint and print_link_info.

-map produces a .list listing segment which contains only the bind map information.

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

The Bindfile

As is discussed in more detail in Syntax of the Bindfile below, special binding instructions may be provided in symbolic form in a special ASCII segment known as the bindfile whose entry name must contain the suffix ".bind". The bindfile must be archived into any one of the input archive segments (at any location within that archive segment) where it will be automatically located and recognized by the binder.

In the case in which 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.

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 entry name 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 to it the .list suffix. 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 (as explained below) will cause additional segment names to be added to the bound segment. Note that the primary name of the bound object segment must not be the same as the name of any component, since both the bound object name and the component names must be stored in the definition section.

Background Information

In order to understand the purpose of the bindfile, some knowledge of the binder's functions is required.

A Multics procedure object segment consists of an internal part (pure text, internal static, and symbol table) which is the machine code representation of the source program, and an external part (definition section and linkage information) which defines certain external variables by a symbolic name for the purpose of dynamic interprocedure linking.

The binding process performs two distinct operations: a) a bound object segment is produced whose internal part (i.e., text, internal static, and symbol table) is a concatenation of the respective (relocated) portions of the component object segments; and b) all interprocedure references among the bound component object segments are prelinked at bind time.

The external part of the bound segment is newly generated to reflect the bound object segment's interface with the external world. Many external symbols previously defined within the component objects may now be internal to the bound object segment and need, therefore, no longer be defined as external to it.

The binder performs its internal prelinking by establishing direct text-to-text or text-to-internal static references among the bound component objects. For nonstandard object segments, dynamic links to procedure entry points are established through an indirect entry sequence located in the entered procedure's linkage section. The purpose of this indirection is to properly reload the linkage pointer (LP) register before the procedure itself is entered. The binder dispenses with this indirection in its internal prelinking because the entire bound object segment requires a single value of LP which has been set properly when the bound object was first entered. Under certain circumstances, however, it is customary in Multics to pass entry values to external procedures (e.g., condition_, ipc_$decl_ev_call_chn) which later may invoke the entry specified by such an entry value. If such an entry value refers to an entry point which is internal to the bound segment, but which is not internal to a component originally in standard object segment format, that entry point's indirect entry sequence in the linkage section must be regenerated to assure that a transfer of control to that entry point will cause the LP to be reloaded properly.

The main purpose of the bindfile is to specify which external symbols within the component objects are to be retained in the bound objects, which are to be deleted, and which are to

be prelinked indirectly through an entry sequence in the linkage section because they are used as parameters in calls to procedures such as condition_ or ipc_$decl_ev_call_chn.

Syntax of the Bindfile

The binder's symbolic instructions have their own syntax which 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, regular statements pertain to a single component object within the bound object segment. Master statements are identified by master key words which are distinct from regular key words in that they begin with a capital letter; regular key words begin with a lower case letter. A key word designates its parameters and a certain action to be undertaken by the binder pertaining to those parameters.

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 which 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. The synonym statement has two uses. First, it facilitates the Multics linker's lookup of entries in bound

segment components that have several external segment names associated with them. Second, it allows the binder to locate the component object, for the purpose of prelinking, even if it is referenced by names other than its objectname. 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, listnames and listtotals 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, listnames, ln, listtotals,
               lt;
```

Failure to state segment names results in most inefficient linker performance.

retain — the parameters are the symbolic names of external symbols (i.e., entry names and segdefs) declared within the component object segment which the user wishes to retain (i.e., have regenerated) as external symbols of the bound object segment.

delete — the parameters are the symbolic names of external symbols (i.e., entry names and segdefs) declared within the component object segment which the user does not wish to be regenerated as external symbols of the new bound segment.

indirect — the parameters are the symbolic names of entry names (no segdefs) of the component object segment which are to be prelinked indirectly through a regenerated entry sequence in the bound segment's linkage section. For components not in standard object segment format, an indirect statement must be specified for all entry points used as parameters in calls to procedures such as condition_ and ipc_$decl_ev_call_chn.

The retain, delete and indirect statements are considered to be 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 which are not to be prelinked during binding; i.e., all references to such symbolic names will be regenerated in the form of links to external symbols. The no_link statement implies a retain statement for the specified symbols.

global — the global statement may have as its parameter either retain, delete, indirect or no_link which becomes effective for all external symbols of the component object. An explicit retain, delete, indirect 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, to allow 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 master key word No_Table, which is described below.

Master Key Words

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 will be 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 may be a subset of the input sequence, allowing the archive segments to

contain additional segments which 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 them to be omitted from the bound segment except when they are needed by (version II) PL/I I/O runtime routines. If this key word is not given, all symbol tables will be 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*/
```

Error Messages

The binder produces three types of error messages. Messages beginning with the word "Warning:" do not necessarily represent errors. 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 #n ....
```

where n is the line number of the offending statement, to allow easy retrieval through use of a context editor. If one of the latter occurs the binder aborts, as it would not be able to bind according to the user's specifications.

The message

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

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

Notes

The binder may be considered, in some sense, to be a language processor whose source language is the collection of component object segments. For components not in standard object segment format, it has to be cognizant of the code generation peculiarities of the diverse translators supported by Multics. Therefore, before processing a component object segment, it looks up, in the component's symbol table, the name of the processor which produced that component object segment. If that component is not in standard format and if that language processor is unknown to the current version of the binder, it will display an error message and refuse to handle that component object. Binding is terminate at that point.

Also, because the binder does not preserve the original linkage section of the component object segment, code which makes assumptions about the existence of a given link in the linkage section (i.e., which would cease to work if that link were removed by the binder), or which assumes a certain structure of links (i.e., an array of links used as a transfer vector and accessed through indexing) is not bindable by the Multics binder and will normally result in a display of an appropriate error message or, occasionally, in unpredictable results during the execution of the bound segment.

Examples

Following are examples of bindfiles.

1) Bindfile for alm -- bound_alm_.bind

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

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

2) Bindfile for debug -- bound_debug_.bind

```
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 entry names debug$debug and
                            debug$db*/
indirect:     fault;      /*indirect entry sequence for condition
                            handler debug$fault*/

objectname:   list_arg_;
retain:       list_arg_;  /*retain entry name list_arg_$list_arg_*/
objectname:   gr_print;
retain:       gr_print;   /*retain entry name gr_print$gr_print*/
```

 (END)

Name: calc

The calc command provides the user with a calculator capable of evaluting arithmetic expressions with operator precedence, a set of often-used functions, and an addressable-by-identifier memory.

Usage

calc

initiates the command. The user may then type in any number of expressions, assignment statements, list commands, or a quit command, separated from each other by one or more carriage returns.

Expressions

Arithmetic expressions involving real values and the operands +, -, *, /, and ** (addition, subtraction, multiplication, division, and exponentiation) may be typed in. Prefix plus and minus are allowed. Parentheses may be used, and blanks between operators and values are ignored. Calc will evaluate the expression and print out the results. For example, if the user typed:

```
2 + 3 * 4
```

calc would respond:

```
=   14
```

The order of evaluation is as follows:

1) expressions within parentheses

2) function references

3) prefix +, prefix -

4) **

5) *, /

6) +, -

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. Note that this means that 2**3**4 is evaluated as (2**3)**4.

Numbers may 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) may 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 may be nested to any level:

```
sin(ln(var)*45*pi/180)
```

Assignment Statement

The value of an expression may be assigned to a variable. The name of the variable must be 1 to 8 characters in length, and must be made up of letters (upper and/or lower case) 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. "pi" and "e" have pre-assigned values of 3.14159265 and 2.7182818, respectively.

List Command

If "list" is typed calc will print out the names and values of all the variables that have been declared so far.

Quit Command

Typing "q" will cause calc to return to the calling program; i.e. to command level.

Examples of Use

```
user:     calc
          2+2
calc:     =   4
u:        r = 1.5
          pi*r**2
c:        =   7.068583
u:        sin(0.01)
c:        =   9.999832E-3
u:        143e11+(12e13
c:        too many (
u:        143e11+(12e13)
c:        =   1.343E+14
u:        list
c:
          r     =   1.5
          e     =   2.718282
          pi    =   3.141592
u:        q
```

 (END)

Command
Development System
9/22/71

Name: cancel_abs_request, car

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

Usage

cancel_abs_request pname -option1- ... -optionn-

1) pname — is the pathname of the absentee control segment associated with this request. The pathname must be typed as it was given in the original request.

2) optioni — is selected from the following list of options and may appear anywhere on the command line:

-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 option is omitted, the third priority queue is searched unless the -all option 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 pname 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.

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

Example

```
car >udd>Multics>Jones>dump>translate
```

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

(END)

Name: cancel_daemon_request, cdr

The cancel_daemon_request command allows a user to delete a dprint or dpunch request which is no longer required. Normally the deletion can be made only by the user who originated the request. See the MPM command write-ups for dprint and dpunch.

Usage

cancel_daemon_request pname -ctl_arg1- ... -ctl_argn-

1) pname — is the path name of the segment for which the dprint or dpunch request is to be cancelled. The path name must be typed exactly as it was given in the original request.

2) ctl_argi — is selected from the following list of control arguments and can appear anywhere in the command line:

-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 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 "Dprint (dpunch) of pname cancelled" is to be omitted.

Notes

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

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

Example

```
cancel_daemon_request >Jones_dir>dump>translate.list
```

would delete the last request which the user had made in queue 3 to print or punch the segment >Jones_dir>dump>translate.list.

 (END)

Command
Standard Service System
02/16/71

Name: change_default_wdir, cdwd

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

Usage

change_default_wdir -path-

1) path is the directory which 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 change_wdir. When the change_wdir command is issued with no arguments, the default working directory becomes the current working directory; if no default working directory has been established, the change_wdir command prints an error message.

See also change_wdir and print_default_wdir in the MPM.

(END)

Command
Development System
9/28/71

Name: change_error_mode, cem

This command is used to control the amount of information printed by the default error handler. It affects all the messages for the life of a process or until it is invoked again.

Usage

change_error_mode -option-

1) option may be chosen from the following list of options:

-brief, -bf prints the condition name.

-long, -lg prints more complete messages. In particular, if a segment is bound, both the offset relative to the procedure and the offset relative to the segment are printed. When there is a crawl out, both sets of machine conditions are printed.

Note

The normal mode, with the message intermediate in length between the brief and long messages, is the default, and can be reset by giving this command with no arguments.

(END)

Command
Standard Service System
02/16/71

Name: change_wdir, cwd

The change_wdir command changes the user's current working directory to the directory specified as an argument.

Usage

change_wdir -path-

1) path is the pathname of the directory to change to. If path is not given, the default working directory is assumed.

Notes

If path specifies a nonexistent directory, an error message will be printed.

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 a mode of "rewa" for path. Restricted uses are possible in accordance with the mode attributes on the directory. For example, the mode must be at least "r" to list the directory.

See also change_default_wdir and print_default_wdir in the MPM.

(END)

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-

1) control_args can be selected from the following:

-date string, -dt string
If this argument is given, check_info_segs uses the date specified by string instead of the date in the user profile. string must be acceptable to the convert_date_to_binary_ subroutine. (See the convert_date_to_binary_ write-up in the MPM.) 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 and time modified as well as the name of any segment selected.

-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. This 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 current 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. Note that cmdline must be enclosed in quotes if it contains blanks.

-pathname path, -pn path — If this control argument is specified, check_info_segs assumes that path is a pathname with one or more asterisks in the entryname portion. All new or modified segments that match path are selected.

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 control argument is not specified, the defaults are >documentation>info>*.info and >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, it is created.

check_info_segs checks the time modified for any segment pointed to by a link, not the time the link was modified. The command types a warning message if any segment pointed to by a link does not exist.

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

Examples

1) Check for info segments modified since the specified date:

```
check_info_segs -date "07/01/73 0900."
```

2) Print all modified info segments:

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

Note that 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.

3) Print just the first block of any modified info segment:

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

Note that the -pn argument must be given to the help command, since check_info_segs supplies an absolute path name as the other argument in the command line.

4) Check for all modified info and peruse_text segments:

```
cis -pn >doc>pt>*.pt -nud
cis
```

Note the use of the -nud argument to prevent the time of last invocation from being updated in the first command line.

 (END)

Name: close_file, cf

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

Usage

close_file -ctl_arg- -filename1- ... -filenamen-

1) ctl_arg can have the value -all to close all open files. In this case, no filenamei appears.

2) filenamei is the name of an open FORTRAN or PL/I file.

Notes

The format of a FORTRAN file name is filenn where nn is a 2-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 filenamei, all PL/I files of that name and, if applicable, the FORTRAN file of that name are closed.

 (END)

Name: code, decode

In order to provide maximum security for data stored in a Multics segment, the code command is provided to encipher a segment's contents according to a key which need not be stored in the system. The decode command, given an enciphered segment and the same key, reconstructs the original segment. The two segments, original and enciphered, have the same length.

Usage

To encipher:

code name1 -name2-

1) name1 is the path name of the segment to be enciphered.

2) name2 is the path name of the enciphered segment to be produced. If name2 is not provided, it is taken to be the same as name1. This command always appends the suffix ".code" to name2 to produce the name of the enciphered segment.

To decipher:

decode name1 -name2-

1) name1 is the path name of the enciphered segment. The ".code" suffix should not be specified.

2) name2 is the path name of the deciphered segment to be produced.

Notes

The code command requests an encipherment key (1-11 characters not including space, semicolon, or tab) from the terminal. 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.

The decode command requests the key from the terminal only once, and produces name2 from the enciphered segment name1.code.

 (END)

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 -ctl_arg1- -ctl_arg2-

1) path1, path2 — are the path names 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 with the first word of the segment if no offset is specified.

3) ctl_argi — specifies one of the following control arguments:

-mask n — specifies that 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, -lg n — specifies that the comparison should continue for no more than n (octal) words.

Notes

The actual number of words to be compared is the minimum of the control argument -length, the word count of the first segment minus its offset, and the word count of the second segment minus its offset. 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.

The command lists any differences found 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 described in the MPM.

 (END)

Name: compare_ascii, cpa

This command compares two ASCII files and prints out the changes made to the file specified by oldpath to yield the file newpath. The output is organized with the assumption that the first file was edited resulting in the second file.

Usage

compare_ascii oldpath newpath -n1- -n2-

1) oldpath — is the pathname of the first file.

2) newpath — is the pathname of the second file (the result of editing the first file).

3) n1 — is a decimal number (optional) specifying the minimum number of characters which must be the same before compare_ascii will assume the segments are again "in sync".

4) n2 — is a decimal number (optional) analogous to n1 specifying the minimum number of lines that must be the same. It is required that n1 be specified to exercise this option.

Notes

The equals convention may be used. If n1 and n2 are both given, they must both be exceeded for the segments to be considered "in sync" again.

(END)

Command
Standard Service System
3/17/72

Name: copy, cp

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

Usage

copy path1_1_ path2_1_ ... path1_n_ path2_n_ -ca_1_- ... -ca_n_-

1) path1_i_ are the segments to be copied.

2) path2_i_ are the copies to be created. If the last path2_i_ is not given, the copy is placed in the working directory with the entry name of the segment from which it was copied.

3) ca_i_ may be chosen from the following list of control arguments:

-name, -nm causes multiple names to be copied.

-acl causes the ACL to be copied.

-all, -a causes multiple names and ACLs to be copied.

-brief, -bf causes the messages "Bit count inconsistent with current length..." and "Current length is not the same as records used..." to be suppressed.

The control arguments may appear once anywhere in the command line and apply to the entire command line.

Notes

Read access is required for path1_i_. Execute access is required for the directory containing path1_i_. Execute, write, and append access are required for the directory containing path2_i_.

The star and equal conventions may be used.

Example

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

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

(END)

Command
Standard Service System
6/29/72

Name: create, cr

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

Usage

create path1 ... pathn

1) pathi is the name of the segment to be created.

Notes

The user must have execute and append access 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 will be 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 will be removed and a message to that effect issued to the user.)

The user is given rewa access to the segment created.

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

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. A noted above, the directories new_dir and alpha must already exist.

(END)

Command
9/26/73

Name: createdir, cd

The createdir command causes a storage system directory branch of specified name 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 write-up of the create command for creation of segments.

Usage

createdir path1 ... pathn

1) pathi specifies the name of the subdirectory to be created.

Notes

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

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 user is interrogated as to whether he wishes the subdirectory bearing the old instance of the name to be deleted. (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 for the subdirectory created.

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

Example

createdir sub >my_dir>alpha>new

would create 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.

 (END)

Command
Development System
3/26/73

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 concept of breakpoints (commonly called breaks) is implemented and thus makes it possible for the user to gain control during program execution for whatever reason he may have. 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 in common use and to refer to variables of interest directly by name.

debug uses a segment in the initial working 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 username.breaks where username is the login name of the user.

Usage

debug

Notes

The debug command provides the user with the following functions:

1) it can look at data or code;

2) it can modify data or code;

3) it can set a break;

4) it can perform (possibly nonlocal) transfers;

5) it can call procedures;

6) it can trace the stack being used;

7) it can look at procedure arguments;

8) it can control and coordinate breaks;

9) it can continue execution after a break fault;

10) it can change the stack reference frame;

11) it can print machine registers; and

12) it can 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) may be placed on a line separated by semicolons (;).

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 rather than 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 metalinguistic and are not in the debug syntax.) The segment name is either a path name, a reference name, or a segment number, and defines what is called the working segment. If the working segment is a procedure segment with an active stack frame, the stack frame may be referenced by specifying &s as the segment ID. If the working segment has an active linkage section (i.e., one with an entry in the Linkage Offset Table (LOT) for the working ring), this may be referenced by specifying &l as the segment ID. A segment ID of &t refers to the working segment itself.

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 may be either a number or a symbolic name. (Note that all numbers are treated as octal except in a few cases specified later.) If a symbolic name is specified, a symbol table must exist for the working segment. See the pl1 command in the MPM 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 he could use

```
/test/i&s+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/i&s$a
```

would result in exactly the same output as above. Note the lack of a + sign 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 may 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.

An example:

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

would generate a working pointer which points at 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 which is to be addressed. The offset field is ignored, but a relative offset may be specified.

An example:

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

will cause 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 will be 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 which 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 which would cause some field to change. If this is the case, debug may realize 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&1
```

if regp is found at a relative offset of 140 (octal) in the linkage section. Note that the segment ID was changed to &1 where it will remain 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. looking up 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 path name, the specified segment is initiated (it may already have been known) and the returned segment number is used as above.

The entire set of defaults which 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.

Operator Field of Data Requests

After decoding the generalized address and coming up with 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 "new line" 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.

Data Requests to Look at Data

For the print request, there are two operands (both optional). The first operand is a single character specifying the output mode desired. (See Appendix 1 for a list of acceptable output modes. The default is half carriage octal.) The second operand is a number indicating how much output is being requested. For example,

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

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

The following output modes are available for data requests (see Appendix 1 for a full description):

1) o octal

2) h half carriage octal

3) d decimal

4) a ASCII

5) i instruction

6) p pointer

7) s source statement

8) l code for line number

9) n no output (just change defaults)

10) e floating point with exponent

11) f floating point

12) b bit string

13) g graphic

The request

```
+36,a16
```

requests that 16 (octal) 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. Note that 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, i.e., one word for octal output, one line for source output, one character for ASCII output, etc.

Data Requests to Modify Data

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

```
i = 7;  p(1) = 206|10, 206|32
```

would assign the octal value 7 to i and the values 206|10 and 206|32 to p(1) and p(2), respectively. (It is assumed that both are variables which 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) an octal number;

2) a decimal number (a number preceded by &d);

3) character strings;

4) register values (see Registers below);

5) instruction format input;

6) floating point;

7) pointers;

8) bit strings;

9) variables.

Note that all numbers typed to debug are assumed to be octal unless immediately preceded by the characters &d which cause such a number to be interpreted as decimal. There are three exceptions. Subscripts, the bit offset of a pointer, and the number immediately following a segment ID of &a are all assumed to be decimal.

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

The word offset portion of a pointer value being input may 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 as follows:

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

The address may specify a base register or a number. For example,

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

Note that some value must be given for the address field. The zero-op-code 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 may 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) will fill in 145|13001 with the decimal 16 and hence overwrite the "s" of the previous argument.

In order to better specify more complicated assignments, a repetition factor is provided. If a single number (octal, or if &d escaping is used, 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 = (40)" " "alpha"
```

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

Data Requests to Set Breaks

A breakpoint is a special modification to the code of a program which, 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 will be 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.

Data Requests to Alter Program Control

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

```
/216/2176>
```

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

If no active stack frame is found, debug comments on the fact and awaits further requests.

Data Requests to Call a Procedure

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 which 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 n+1'st argument 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 will return a value into wherever the working pointer is pointing. For example,

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

This will cause the sqrt_ function to be called with the first argument 2.0 and the return argument of fi. Note that 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
```

will set up a call to who$who with no arguments. Note that

```
:= 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,

```
:= hcs_$usage_values(0,0)

%1,d

%2,d
```

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

```
%2 := hcs_$usage_values(0)
```

If this were done, the second 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.

Note that 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
```

Note that 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 may 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 register considered as a single register

21) exp exponent register

22) tr timer register

23) ralr ring alarm register

24) regs all of 10) through 23)

25) ppr procedure pointer register

26) tpr temporary pointer register

27) even even instruction of Store Control Unit (SCU) data

28) odd odd instruction of SCU data

29) scu all SCU data

30) all all machine conditions

The user can change these registers at will 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 will be set to those of these registers.

The values in the registers are automatically filled in by debug (when it is called or faulted into) with those of the last fault found in the stack. The user can override these values with the fill registers and crawl out 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 may type

```
$user
```

The setting and displaying of registers follows the syntax of data requests. However, only the register name and a possible new value may appear in a register request. Registers may 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 may 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 Appendix 2 for a complete review of all requests.

1) Trace Stack

.ti,j

The stack is traced from frame i (counting from 0 at the base of the stack) for j frames. If i is less than 0, tracing begins at 0; if i is greater than the last valid frame, then only the last frame is traced. If i is not specified, it is assumed to be 0; if j is not specified, all valid stack frames after i will be 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 entry name invoked for the stack frame (i.e., the label on the entry or procedure statement).

Examples:

.t2,3

.t100

2) Pop or Push Stack

The general form is:

.+i or .-i

The working segment is changed by moving up or down the stack i frames. For example, if the working segment's active stack frame is at depth 4 in the stack, then

.+3

will change 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 sp|0, &s, and 0, respectively.

3) Set Stack

The general form is

.i

The working segment is set to that of the i'th stack frame (starting at 0). The defaults are set as in pushing or popping the stack.

4) Execute Multics Command

The general form is

..<Multics command line>

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

5) 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 depth in octal, unless there is no stack frame for the working segment, in which case the number is -1. The working segment appears between the slashes (test_seg above); if .D is used, the full path name 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
```

6) 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 will be skipped. Note that the first instruction executed upon continuation is the instruction on which the break occurred. If a t follows the c, debug will continue in temporary break mode (see below). If an r follows the c, debug will reset 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
```

7) 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 will return to the last procedure which explicitly called debug.

8) Change Output Mode

Requests pertaining to debug's console output begin with ".m".

a) 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 console; the column headings are not printed for a stack trace; the printing of register contents is somewhat more compact; some error messages are abbreviated.

b) Enter long output mode

.ml

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

9) 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 which 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 may have several breaks with the same number defined in different object segments .

There are two types of global requests which 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 initial working 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*n* args

where the g, the number *n*, 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:

a) 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.

b) 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.

c) Execute a debug command line 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 will cause <rest of line> to be interpreted as a debug input line whenever the appropriate break(s) is (are) encountered. If <rest of line> is null, the specified breaks will have this execute feature reset to normal.

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

.boi disable (turn off) 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 re-enabled by the .bn request. If the break was already disabled, the request has no effect.

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

.bni enable (turn on) 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.

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

.bgt <rest of line>

This will cause <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 command lines associated with individual breaks. If <rest of line> is null, a previously-established temporary command line is disestablished.

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

.bc*i* arg1 -rel- arg2

.bc arg1 -rel- arg2

arg1 and arg2 may be constants or variables. -rel- may 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

will cause break 3 to fault whenever it is encountered and the value of i is nonzero. Also,

.bc3 i = j

will cause break 3 to fault whenever it is encountered and the value of i is the same as the value of j. Note that 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.

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

.bs*i* *n*

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

i) Print or change the default object segment. The form for this request is

.bd name

where name is the (relative) path name or segment number of the segment to become the default object segment. If name is not specified, the path name of the default object segment is printed.

j) List the current segments which have breaks. The form for this request is:

.bp

This request merely interprets the break segment in the initial working directory.

10) Print Arguments

The general form is

.ai,m

Argument i for the current stack frame will be printed in the mode specified by m. If i is not specified, all arguments are printed. If m is not specified, debug will decide the output mode. Valid values for m are:

a) o full word octal;

b) p pointer;

c) d decimal;

d) a ASCII;

e) b bit string;

f) l location of argument;

g) e,f floating point;

h) ? debug will decide (the default value for m).

Examples:

.a3

ARG 3: ">user_dir_dir"

.a3,o

ARG 3: 076165163145

11) 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.)

12) Crawl-Out Registers

The general form is:

.C

For register requests debug uses the fault data associated with the last crawl-out (abnormal exit from an inner ring).

Program Interrupt Feature

The user may 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:

pi

which is the standard program interrupt manager. (See the program_interrupt write-up in the MPM.)

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, and only then, i.e, a temporary break, if not continued, must be explicitly reset by the user.

Since temporary breaks are set sequentially in a program (i.e., at the next statement in the source program), any transfers within a program may either skip a temporary break or cause code to be executed which was stopped earlier with a temporary break. Temporary break mode is designed to be used in programs which are fairly uniform and sequential in their flow of control. Note that a user should generally 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 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, hence, not necessarily into the same segment as before the request.

Implementation of Breakpoints

Breakpoints are implemented by using a special instruction (mme2) which causes a hardware fault whenever it is executed. 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 will execute the instruction which used to exist where the break word was placed.

debug 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 Programs

If a symbol table were created for a PL/I program using the table option, then names of labels, scalars, structures, and arrays may be used. The only restrictions are that 1) the entire structure name must be specified; 2) the only expressions which are allowed for subscripts are of the form

variable ± constant

where variable may 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 which is based or part of a substructure, a bit offset may 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 may still contain a bit offset. For example, if the working pointer has the value

```
151|3706(13)
```

then the request

```
+16,b3
```

will set the working pointer to

```
151|3724(13)
```

and print the three bits at this location.

Appendix 1

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 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 will be printed in response to a single request.

5) i instruction

The data is printed in instruction format much as an assembler might do.

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 which generated the code pointed to by the working pointer (assumed to be pointing into the text). For example,

```
/test/loop_here+32,s2
```

will print two lines of source code starting with the line which generated the code 32 (octal) words after the label loop_here.

Another example:

```
/test/&a219,s
```

will print 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 comments on the fact, increments the line number, and tries again (forever).

8) l code for line number

The code associated with the specified line number is printed. The line number is determined as in s type output. For example,

```
/test/&a27,l
```

will print the code generated for line 27 (decimal) of test.lang. Note that any number following the l will be 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 will be 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).

Appendix 2

Data and Control Requests

1) Data Requests

/seg name/	offset	seg ID	rel offset	operator	operands
path name	number	&t	number	,	operands
ref name	symbol	&s	register	=	input list
seg number		&l		<	function list
&n seg name		&an		>	
		&pn		:=	

Segment ID	Operators	Registers	Output Modes
&t text	, print	$a	o octal
		$q	h half carriage octal
&s stack	= assign	$aq	d decimal
		$x0	a ASCII
&l linkage	< set break	.	i instruction
		.	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	
		$prs	
		$regs	
		$scu	
		$all	

2) Control Requests

.t*i*,*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 (ignore next *i* break faults)
.ct,*i*	continue, in temporary break mode
.cr,*i*	continue, in normal mode
.q	return from debug to caller
.br*i*	reset break *i*
.br	reset the breaks of the default object segment
.bgr	reset all breaks
.bl*i*	list break *i*
.bl	list the breaks of the default object segment
.bgl	list all breaks
.be*i* <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
.bo*i*	disable break *i*
.bo*i*	disable break *i*
.bo	disable the break of the default object segment
.bgo	disable all breaks
.bn*i*	enable break *i*
.bn	enable the breaks of the default object segment
.bgn	enable all breaks
.bgt <line>	establish a temporary global command
.bc*i* a1 -rel- a2	make conditional break *i*
.bc a1 -rel- a2	make conditional all breaks of default object segment
.bs*i* *n*	set skips of break *i* to *n*
.bd name/no.	set (or print) default object segment
.bp	print all segments with breaks
.a*i*,*m*	print argument *i* in mode *m* (modes: o, p, d, a, b, l, e, f, ?)
.f	use registers from last fault
.C	use crawl-out registers
.mb	change to brief output mode
.ml	change to long output mode

 (END)*

Name: decam, dcm

The decam (desk calculator with memory) command performs the functions of a ten-key desk calculator. In addition, it has a small memory and variable radix input/output.

Usage

decam

initiates the calculator; when ready to accept the first request decam types "Go". No further responses will be typed by decam unless it is asked to print a result or an illegal request is given. Successive requests are separated by new-line characters. All blanks are ignored. One result register is maintained, called A here. The following requests modify the contents of A as shown (n is any integer):

typed request	computation performed	
=n	n -> A	initialize A with n
+n	A + n -> A	add n to A
-n	A - n -> A	subtract n from A
*n	A X n -> A	multiply A by n
/n	A / n -> A	divide A by n
%n	n / A -> A	divide n by A
p		print contents of A

One additional request

q

will return the user to command level.

Storage Cells

Eight storage cells, named s, t, u, v, w, x, y, and z, may also be used as operands in the above requests by replacing the integer n with the name of a storage cell. A value may be stored in a storage cell by

x = n

where storage cell x receives the value of n. Of course any of the eight storage cell names may be used. If n is omitted the value of A is used. Storage cell s contains the radix which is used for input/output conversion. It initially contains ten (decimal). The contents of s must be in the range from 2 to 20 inclusive.

To print storage cell x, type

p x

Note

All computations are done with 35-bit integers, giving about 10 digit precision when the radix is ten.

Example

The following example sums two set of numbers, then divides the first sum by the second. The right hand column describes the activity in the left hand column.

```
decam           user invokes decam
Go              response from decam
=0              user initalizes A
+214            user adds first set of numbers
+27             ..
+818            ..
p               user requests result to be printed
1059            decam prints the result
                decam prints a blank line
x=              user saves result in cell x
=0              user resets A to zero for second addition
+14             user adds second set of numbers
+23             ..
+79             ..
p               user requests result to be printed
116             decam prints the result
                decam prints a blank line
%x              user divides A into x
p               user requests result to be printed
9               decam prints the result
                decam prints a blank line
q               user returns to command level
```

Note that if the second set of numbers had been summed first, the divide request would have been /x (divide A by x) instead of %x (divide A into x).

Syntax

Only a subset of the allowed syntax of the desk calculator has been described above. The full syntax description is given as follows:

<name> ::=	s|t|u|v|w|x|y|z| <null field>
<integer> ::=	(usual definition--the letters a through j are used as digits as necessary, for radix > 10.)
<operand> ::=	<name> | <integer>
<operator> ::=	+|-|*|/|%|p|q|=
<request> ::=	<name> <operator> <operand>

 (END)

Command
Standard Service System
7/24/72

Name: delete, dl

The delete command causes the specified segments and multi-segment files to be deleted. See also delete_dir (for deleting directories) and deleteforce (for deleting protected segments without being interrogated).

Usage

delete path1 ... pathn

1) pathi is the name of the segment to be deleted.

Notes

The user must have write access for both the segment and its directory. If only the segment's write access is lacking, he will be interrogated as to whether he wishes to delete the segment. See also the MPM section for deleteforce.

If pathi is a link, delete will print a message; it will not delete either the segment in question or the link. See the MPM section for unlink. If pathi is a directory, delete will print a message; it will not delete the directory. See the MPM section for deletedir.

The star convention may be used.

(END)

Command
Standard Service System
6/29/72

Name: delete_dir, dd

The delete_dir command causes the specified directories (and any segments they contain) to be deleted. Subdirectories and their segments and subdirectories will also be deleted. See the MPM sections for delete (for deleting segments) and deleteforce (for deleting protected segments).

Usage

delete_dir path1 ... pathn

1) pathi is the name of the directory to be deleted.

Notes

The user must have write access for both the directory and its superior directory. The star convention may be used. Before deleting each specified directory, delete_dir will ask the user if he wants to delete that directory. It will be deleted only if the user types "yes".

Warning: protected segments in pathi or any of its subdirectories will be deleted.

(END)

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 added to the directory. For a discussion of Initial ACLs, see the MPM Reference Guide section, Access Control.

Usage

delete_iacl_dir pathname acname1 ... acnamen -control_arg-

1) pathname specifies the directory in which the directory Initial ACL should be changed. If it is "-wd" or "-working_directory" or omitted then the working directory is assumed. If it is omitted then only the "-a" option for acnamei is allowed. If no arguments are given then the entry for the user's name and project is deleted from the Initial ACL of the working directory. The star convention may be used.

2) acnamei is an access control name. If no acnamei is specified then the user's name and project are assumed. acnamei must be of the form person.project.tag. If one or more of the components is missing, then all entries in the Initial ACL that match the given components are deleted. Components missing on the left must be delimited by periods; however, the periods may be omitted on the right.

3) control_arg may be -ring (-rg). If may appear anywhere on the line and affects the whole line. If it is present it must be followed by a digit, where user's ring $\leq$ digit $\leq$ 7, which specifies which ring's Initial ACL for directories should be affected. If this option is not given then the user's ring is assumed.

Examples

```
delete_iacl_dir news .Faculty ..a
```

will delete from the directory Initial ACL of the news directory all entries with project name Faculty and all entries with instance tag a.

```
did -a
```

will delete all entries from the directory Initial ACL of the working directory.

```
did store Jones -rg 5
```

will delete from the directory Initial ACL in ring 5 in the directory store, all entries beginning with person name Jones.

 (END)

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 added to the directory. For a discussion of Initial ACLs, see the MPM Reference Guide section, Access Control.

Usage

delete_iacl_seg pathname acname1 ... acnamen -control_arg-

1) pathname specifies the directory in which the segment Initial ACL should be changed. If it is "-wd" or "-working_directory" or omitted then the working directory is assumed. If it is omitted then only the "-a" option for acnamei is allowed. If no arguments are given then the ACL entry for the user's name and project is deleted from the Initial ACL of the working directory. The star convention may be used.

2) acnamei is an access control name. If no acnamei is specified then the user' name and project are assumed. acnamei must be of the form person.project.tag. If one or more of the components is missing, then all entries in the Initial ACL that match the given components will be deleted. Components missing on the left must be delimited by periods; however, the periods may be omitted on the right.

3) control_arg may be -ring (-rg). It may appear anywhere on the line and affects the whole line. If it is present it must be followed by a digit, where user's ring $\leq$ digit $\leq$ 7, which specifies which ring's Initial ACL for segments should be affected. If this control argument is not given then the user's ring is assumed.

Examples

```
delete_iacl_seg news .Multics ..a
```

will delete from the segment Initial ACL of the news directory all entries with project name Multics and all entries with tag a.

```
dis -a
```

will delete all entries from the segment Initial ACL in the working directory.

```
dis store Jones -rg 5
```

will delete from the segment Initial ACL for ring 5 in the directory store all entries beginning with a person name Jones.

 (END)

Name: deleteacl, da

This command deletes entries on an Access Control List (ACL) of either a segment or a directory.

Usage

deleteacl pathname acname1 ... acname2

1) pathname specifies the ACL of the segment or directory which should be changed. If it is "-wd" or "-working_directory" or omitted, then the working directory is assumed. If it is omitted, then only the "-a" option for acnamei is allowed. If no arguments are given, then the ACL entry for the user's name and project is deleted from the ACL of the working directory. The star convention may be used.

2) acnamei is an access control name. If no acnamei is specified, then the user's name and project are assumed. acnamei must be of the form person.project.tag. If one or more of the components is missing, then all entries in the ACL that match the given components will be deleted. Any components missing on the left must be delimited by periods; however, the periods may be omitted on the right. If an acnamei would include *.SysDaemon.*, but does not have all three components specified, the ACL entry for *.SysDaemon.* will not be deleted if it exists. If acnamei is "-a" or "-all" then the whole ACL will be deleted, and the *.SysDaemon.* rw entry (or sma for directories) will then be added to the empty ACL.

Note

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 will prevent Backup.SysDaemon.a from saving the segment or directory (including the hierarchy inferior to the directory) on tape, Dumper.SysDaemon.a from reloading it, and Retriever.SysDaemon.a from retrieving it.

Examples

```
deleteacl  news  .Faculty  ..a
```

will delete from the ACL of news all entries with the project name Faculty and all entries with the instance tag "a".

```
da  -a
```

will delete all entries from the ACL of the working directory and then add an entry "*.SysDaemon.* sma".

```
da  test.pl1  *.*.*  Doe
```

will delete from the ACL of test.pl1 an entry for *.*.* and all entries with the person name Doe.

 (END)

Command
Standard Service System
7/6/72

Name: deleteforce, df

The deleteforce command causes the specified segments to be deleted, regardless of whether or not they are protected.

Usage

deleteforce path1 ... pathn

1) pathi is the name of the segment to be deleted.

Notes

The user must have write access for both the segment and its directory.

If pathi is a link, deleteforce will print a message; it will not delete either the segment in question or the link. See the MPM section for unlink. If pathi is a directory, deleteforce will print a message; it will not delete the directory. See the MPM section for delete_dir.

The star convention may be used.

(END)

Command
Standard Service System
6/29/72

Name: deletename, dn

The deletename command deletes specified names from segments or directories which have multiple names. See the MPM sections for addname (for adding names to storage system entries) and rename (for changing names of storage system entries).

Usage

deletename path1 ... pathn

1) pathi is the name which is to be deleted.

Notes

In keeping with standard practice, pathi may be a relative path name or an absolute path name; its final portion (the storage system entry in question) will be deleted from the segment or directory it specifies, provided that doing so does not leave the segment or directory without a name. In the latter case, the user will be interrogated as to whether he wishes the segment or directory in question to be deleted.

The star convention may be used.

Example

deletename alpha >my_dir>beta

would delete the name alpha from the list of names for the appropriate segment in the current working directory, and would delete the name beta from the list of names for the appropriate segment in the directory >my_dir. Neither alpha nor beta may be the only name for their respective segments.

(END)

Command
Development System
8/18/71

Name: display_component_name, dcn

The display_component_name command converts an offset within a bound segment (e.g., bound_zilch_|23017) into an offset within the referenced component object (e.g., comp|1527). It works on segments bound with version number 4 (and subsequent versions) of the binder. It is especially useful when it is necessary to convert an offset within a bound segment (as displayed by the default error handler or by a stack trace) into an offset corresponding to a compilation listing.

It is intended to be a temporary command and will probably be removed when the debugging and diagnostic tools have been modified to perform the appropriate offset conversion themselves.

Usage

display_component_name path offset1 ... offsetn

1) path is the pathname of a bound object segment.

2) offseti is an octal offset within the text of the bound object segment specified by "path".

Example

The command

```
display_component_name bound_zilch_ 17523 64251
```

might respond with the following display:

```
17523     component5|1057
64251     component7|63
```


(END)

Name: do

The purpose of the do command is to expand a command string given as its first argument by replacing the parameter designators &1 ... &9 found in it with the actual arguments supplied following the command string. The resultant expanded string is then passed to the Multics command processor for execution. If abbreviations are being expanded in the process, any abbreviations in the expanded string are first expanded. (See the writeup for the abbrev command.) Control arguments exist to print the expanded command line, to suppress its execution, or to pass it back as the value of an active function.

Usage

do "command_string" -arg1- ... -argn-

1) command_string — is a command line (in quotation marks). Each instance of the paramter designator &i (where i is a number from 1 to 9) found in command_string is replaced by the corresponding actual argument argi. If any argi is not supplied, then each instance of &i in command_string is replaced by the null string. 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 pair && is replaced by a ampersand. Any other ampersand discovered in command_string causes an error message to be printed and the expansion to be terminated. Any argument supplied but not mentioned in a paramter designator is ignored.

2) argi — is a character string argument to replace a parameter designator &i in command_string.

Usage as an Active Function

If the do command is called as an active function

|do "command_string" arg1 ... argn|

then, instead of executing the resultant expanded string, the do command passes it back as the value of the active function.

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 as follows:

do mode

where mode is one of -long, -lg, -brief, -bf, -nogo, -go, -absentee, or -interactive.

If the long/brief mode is long, then the expanded command string is printed on the string error_output before it is executed or passed back. If the long/brief mode is brief, then the string is not printed. The default for this mode is brief.

If the nogo/go mode is nogo, then the expanded command string is not passed to the command processor for execution. If the nogo/go mode is go, then the expanded string is passed to the command processor (if the do command was invoked as a command). If do is invoked as an active function, then the nogo/go mode is ignored. The default for this mode is go.

If the absentee/interactive mode is absentee, then the do command establishes itself as a default on unit during the execution of the expanded command string. This is mainly of use in an absentee environment, in which any invocation of the standard default on unit terminates the process. When do is the default on unit, any signal caught by do merely terminates execution of the command string, not the process. A number of conditions, however, are not handled by do but are passed on for their standard Multics treatment; they are cput, alrm, quit, program_interrupt, command_error, command_query_error, command_question, and record_quota_overflow. (For a description of these conditions see the MPM Reference Guide section, List of System Conditions and Default On Unit Actions.) If the absentee/interactive mode is interactive, then do does not catch any signals. The default for this mode is interactive.

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 string, and &r1 ... &r9, to request that the actual argument be requoted as well as have its quotes doubled during substitutution.

Quote-doubling can be described as follows. Each parameter designator in the input string to be expanded is found nested a certain level deep in quotes. If a designator is found to be not 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 &qi is found nested to quote-level L, then, as argi is substituted into the expanded string each quote character found in argi is multiplied by 2**L during insertion. This permits the quote character to survive the quote-stripping action to which the command processor subsequently subjects the expanded string. If &qi is not located between quotes, or if argi contains no quotes, then the substitutions performed for &qi and for &i are identical.

If the parameter designator &ri is specified, then the substituted argument argi is placed between an additional level of quotes before having its quotes doubled. More precisely, if the parameter designator &ri is found nested to quote-level L, then 2**L quotes are inserted into the expanded string, argi is substituted into the expanded string with each of its quotes multiplied by 2**(L+1), and then 2**L more quotes are placed following it. If argument argi is not supplied, then nothing is placed in the expanded string; this providis a way to distinguish between arguments that are not supplied and arguments that are supplied but null. If argument argi is present, then the expansions of &ri and of &qi written between an additional level of quotes are identical.

Accessing More than Nine Arguments

In addition to the normal parameter designators in which the argument to be substituted is specified by a single digit, do also allows the designators &(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, the abbrev command. Consider the following abbreviations:

```
ADDPLI   do "fo &1.list;ioa_ ¬|;pli &1;co"
AUTHOR   do "ioa_$nnl &1;status -author $1"
AX       do "if is &1 -then ""df &1"""""
FO       do "AX &1;fo &1"
P        do "pl1 &1 -list &2 &3"
```

The command

```
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

```
FO gamma
```

expands to

```
AX gamma;fo gamma
```

which is expanded to

```
if is gamma -then df gamma;fo gamma
```

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.

Name: dprint, dp, dpl, dp2

The dprint (daemon print) command allows specified segments to be queued for printing on the Multics line printer. It is similar to the dpunch command and the actual printing is managed by the same system process which manages segment punching. The output, identified by the personal name contained in the requestor's user ID, is available from Operations. Because queue traffic is highly variable, no guarantee can be made as to how long the printing will take.

The entry dpl places requests in the top priority queue, dp2 places them in the second priority queue, and dp and dprint place them in the lowest priority queue. All requests in the first queue are processed before any requests in the other queues, etc. Higher priority queues have a higher cost associated with them.

Usage

dprint -ctl_arg1- ... -ctl_argn- -path1- ... -pathn-

1) ctl_argi — can be chosen from the following list of control arguments and can appear anywhere in the command line:

-brief, -bf — the message "j requests signalled, k already queued." is suppressed. This control argument cannot be overruled later in the command line.

-copy n, -cp n — causes n copies (n <= 4) of subsequent segments to be printed. This control argument can be overruled by a subsequent -copy control argument. If the -delete control argument is specified for the segment, it does not take place until after the last copy has been printed. The default value for n is 1.

-queue n, -q n — all subsequently specified segments are printed in priority queue n (n <= 3). This control argument can be overruled by a subsequent -queue control argument. It overrides any specification made through the use of

	the command names dp1 or dp2. The default value for n is 3.
-delete -dl	all subsequently specified segments in the command line are deleted after printing.
-header heading -he heading	subsequent output is identified by the string "heading" as well as by the user ID. This control argument can be overruled by a subsequent "-header" control argument.
-destination dest -ds dest	subsequent output is labeled with the the string "dest", which is used to determine where to deliver the output. (If this control argument is omitted, the requestor's project ID is used.) This control argument can be overruled by a subsequent -destination control argument.
2) pathi	is the path name of a segment to be queued for printing.

Notes

All control arguments can appear anywhere in the command line. If present, they affect only segments specified after their appearance.

The -brief control argument affects only the message printed after the command is finished and not the processing of segments.

The -copy control argument limits the maximum number of copies to 4.

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 printing.

The command dp (or dp1 or dp2), with no arguments specified, results in a message giving the status of the queue.

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.

 (END)

Name: dpunch, dpn, dpn1, dpn2

The dpunch (daemon punch) command allows specified segments to be queued for punching by the Multics on-line card punch. It is similar to the dprint command and the actual punching is managed by the same system process that manages segment printing. Output is available from Operations. Because queue traffic is highly variable, no guarantee can be made as to how soon the punching will be performed.

See also the MPM Reference Guide Section, Input and Output Facilities.

The entry dpn1 places requests in the top priority queue, dpn2 places them in the second priority queue, and dpn and dpunch place them in the lowest priority queue. All requests in the first queue are processed before any requests in the other queues, etc. Higher priority queues have a higher cost associated with them.

Usage

dpunch -ctl_arg1- ... -ctl_argn- -path1- ... -pathn-

1) ctl_argi — can be chosen from the following list of control arguments and can appear anywhere in the command line:

-brief, -bf — the message "j requests signalled, k already queued." is suppressed. This control argument cannot be overruled later in the command line.

-copy n, -cp n — causes n copies (n <= 4) of all subsequent segments to be punched. This control argument can be overruled by a subsequent -copy control argument. If the -delete control argument is specified for the segment, it does not take place until after the last copy has been punched. The default value of n is 1.

-queue n, -q n — all subsequently specified segments are punched in priority queue n (n <= 3). This control argument can be overruled by a subsequent -queue control argument.

It overrides any specification made through the use of the command names dpn1 and dpn2. The default value for n is 3.

-delete
-dl
: all subsequently specified segments in the command line are deleted after punching.

-header heading
-he heading
: the string "heading" is added to the deck's identifying information for all subsequently specified segments in the command line unless overruled by a subsequent -header control argument.

-destination dest
-ds dest
: the string "dest" is printed on the accompanying sheet identifying the output, and is used to determine where to deliver the deck. (If this control argument is omitted, the requestor's project ID is used.) This control argument can be overruled by a subsequent -destination control argument.

-mcc
: the following segments in the command line are to be punched under character conversion. This control argument can be overruled by either the -raw or -7punch control arguments.

-raw
: the following segments in the command line are to be punched with no conversion. This control argument can be overruled by either the -mcc or -7punch control arguments.

-7punch
-7p
: the following segments in the command line are to be punched under -7punch 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.

2) path*i* is the path name of a segment to be queued for punching.

Notes

All control arguments can appear anywhere in the command line. If present, they affect only segments specified after their appearance.

The -brief control argument does not affect the processing of segments but only the message printed after the command is finished.

The -copy control argument limits the maximum number of copies to 4.

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 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.

The dpunch command (or dpn or dpn1 or dpn2), with no arguments specified, results in a message giving the status of the specified queue.

It is suggested that, before deleting the segment that was punched, the user read the deck back in and compare it with the original to ensure the absence of errors.

Example

```
dpunch a b -mcc -he Doe c.pl1 -dl -7p -he "J. Roe" a
```

would cause segments a and b in the current working directory to be 7punched (the default conversion mode) and c.pl1 to be punched under character conversion with "for Doe" added to the heading. Segment a is 7punched with "for J. Roe" added to the heading and the segment is then deleted after punching.

 (END)

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 representation.

Usage

dump_segment seg offset num -control_arg-

1) seg — is the pathname or segment number of the segment to be dumped. If it is a pathname, but looks like a number, the preceding argument should be -name or -nm.

2) offset — is the offset of the first word to be dumped. If omitted, the entire segment is dumped.

3) num — is the number of words to be dumped. If omitted, 1 is assumed.

4) control_arg — can be chosen from the following list of control arguments:

-long, -lg — causes eight words to be printed on a line. Four is the default. This control argument cannot be used together with any of the other control arguments. (Its use with -bcd or -character would result in a line longer than 132 characers.)

-character,-ch — causes the ASCII representation of the words to also appear on each line. Characters that cannot be printed are represented as periods.

-bcd — causes the BCD representation of the words to also appear on each line. There are no non-printable BCD characters, so periods can be taken literally.

-short, -sh causes lines to be compacted, 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.

 (END)

Name: edm

The edm command is the standard Multics context editor. It is used for creating and editing ASCII segments. This command cannot be called recursively. See also the MPM Introduction Chapter 3, Beginner's Guide to the Use of Multics.

Usage

edm -pathname-

1) pathname specifies the segment to be edited. The argument is optional. If pathname is not given, edm will begin in input mode (see Notes below), ready to accept whatever is typed subsequently as input; when the created segment is written out, its name may be specified as part of the write request. If pathname is given, but the segment does not yet exist, edm will also begin in input mode; otherwise, edm will begin in edit mode, ready to edit the segment specified by pathname.

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 characters must be issued. The mode change characters are the period (.) followed by a "new line" (carriage return-new line), issued as the only characters 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 characters.

The edm requests are predicated on the assumption that the segment consists of a series of lines to which there is a conceptual pointer which 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; for these, the character is generally the first letter of the name of the request. Only the single character is accepted by the command. Three requests have been considered sufficiently dangerous, or likely to confuse the unwary user, that their names must be specified in full.

If the user presses the quit button 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 MPM write-up for program_interrupt.) 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 will be lost. If the user wishes to keep the input he must type start following the quit.

Requests

The requests are as follows (detailed descriptions follow the list, in the order of the list):

-	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)
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

- Request

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;
```

= Request

Format: =

Purpose: Print current line number.

Pointer: Unchanged.

, Request

Format: ,

Purpose: The editor will print lines, starting with the current one, leaving off the carriage return. It then switches to input mode, letting you type the rest of the line (comment, "new line", etc.). The process then repeats with the next line. The mode change characters will return you to edit mode.

Pointer: Left pointing to the last line printed.

b Request

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.

c Request

Format: c n /string1/string2/

Purpose: Replace every instance of string1 by string2 in the number of lines indicated to be searched by the integer n. edm responds to each change by printing the line with the changed text in red if the user is in verbose mode (see the v request), or with "edm: Substitution failed." if string1 is not found.

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 shown in the format). A delimiter following string2 is optional.

Default: If an integer is absent, only string1 of the current line is changed. 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 may also be given as s (for substitute).

d Request

Format: d n

Purpose: Causes 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 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.

E Request

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.

f Request

Format: f string

Purpose: Search segment for a line beginning with the string. Search starts at the line following the current line and continues around the entire segment until the string is found or until return to the current line. The current line is not searched. If the string is not found, the error message "edm: Seach 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, edm searches for the string requested by the last f or l request.

i Request

Format: i newline

Purpose: Insert newline 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 new line.

Pointer: Set to the inserted line.

Default: If newline is null, a blank line is inserted.

Note: Immediately after a t (top) request, an i request causes the newline to be inserted at the beginning of the segment.

k Request

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.

l Request

Format: l string

Purpose: Search segment for a line containing the string. Search starts at the line following the current line and continues around the entire segment until the string is found or until return 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, edm searches for the string requested by the last l or f request.

Example:

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

Request:    l x =
```

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

merge Request

Format: merge path

Purpose: The segment specified by path is inserted after the current line.

Spacing: A single space following merge is not significant.

Pointer: Set to the last line of the inserted segment.

Default: If path is not given, the name given in the invocation of edm is used.

n Request

Format: n n

Purpose: Move pointer down the segment n lines.

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 request.

p Request

Format: p n

Purpose: n lines will be printed, 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 may be aborted by pressing the quit button and typing pi or program_interrupt. This will put edm in a state where it is ready to accept another request. (See the MPM write-up for program_interrupt.)

q Request

Format: q

Purpose: To 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 will warn the user that the changes will be lost and ask if he still wishes to quit.

Pointer: If the user is queried and answers no, then the pointer is unchanged.

qf Request

Format: qf

Purpose: To exit from edm directly without being warned or queried.

r Request

Format: r newline

Purpose: Replace current line with newline.

Spacing: One space between r and newline is not significant. All other leading and embedded spaces become part of the text of the new line.

Pointer: Unchanged.

Default: If newline is null, a blank line replaces the current line.

s Request

Note: Used identically to the c request.

t Request

Format: t

Purpose: Moves pointer to the first line 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.

updelete Request

Format: updelete

Purpose: Delete all lines above (but not including) the current line.

Pointer: Unchanged.

upwrite Request

Format: upwrite path

Purpose: All the lines above the current line (but not including the current line) are saved in the hierarchy in the segment specified by path.

Spacing: A single space following upwrite is not significant.

Pointer: Unchanged.

Default: If path is not given, the name given in the invocation of edm is used.

Note: The lines written out are deleted from the edit buffers and thus are no longer available for editing.

v Request

Format: v

Purpose: Causes 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.

w Request

Format: w path

Purpose: To write out (save) the edited copy. path can stipulate the directory and the entry name within the directory in which the segment is to be saved. If only the entry name for the saved copy is given, the working directory is assumed.

Spacing: A space between w and path is not significant.

Pointer: Set to "no line" at the end of the segment.

Default: If path is null, and if the original name of the segment is not null, the edited segment is saved under the original name; the original segment is deleted. If path is null and no previous segment exists, 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.

 (END)

Name: endfile

The endfile command causes the FORTRAN I/O system to close one or all of the FORTRAN I/O files which are still open after the end of the execution activities during which the I/O files were referenced. If is useful when a FORTRAN program did not proceed to completion, such as when it was interrupted by the user pressing his quit button.

Usage

```
endfile file_id
```

1) file_id is a one- or two-digit number which identifies the file to be closed. If file_id is "-all", then all of the FORTRAN I/O files still open in the process are closed.

 (END)

Command
Standard Service System
5/24/72

Name: enter, e
enterp, ep

These commands are used by anonymous users to gain access to Multics. enter is actually a request to the answering service to create a process for the anonymous user. Therefore, these commands can only be used from a terminal connected to the answering service; that is, one which has just dialed up, or one which has been returned to the answering service after a terminated session with a "logout -hold" command.

Anonymous users who are not to supply a password use the enter (e) command. Anonymous users who are to supply a password use the enterp (ep) command.

Usage

enter -anonymous_name- project -control_args-

1) anonymous_name — is an optional identifier which is not checked by the system, but is passed to the user's process overseer as if it were a person ID. If anonymous_name is not specified, it will be assumed to be the same as the project ID.

2) project — is the identification of the user's project.

3) control_args — may be chosen from the following list of control arguments:

-brief, -bf — Messages associated with a successful login will be suppressed. If the user is using the standard process overseer, the message of the day will not be printed.

-home_dir path
-hd path — The user's home directory will be set to the path specified, if the user's project administrator allows him to specify his home diretory.

-process_overseer path
-po path — The user's process overseer will be the procedure given by the path specified, if the user's project administrator allows him to specify his process overseer.

-no_print_off, -npf	The system will overtype several lines to provide a black area for the user to type his password.
-print_off, -pf	The system will not overtype an area for the password, since the user's terminal responds to the printer-off control sequence.
-no_preempt, -np	If the user can only be logged in by preempting some other user in his load control group, refuse his login instead.
-no_start_up, -ns	If the user has a start_up.ec segment, and the project administrator allows the user to avoid it, instruct the standard process overseer not to execute it.
-account id, -ac id	Replace the normal account identifier for the user with id. (This control argument currently has no effect.)
-force	If the user has the guaranteed login attribute, log the user in if at all possible.

Note

See the MPM Reference Guide section on the Protocol for Logging In for an explanation of the responses to the enter and enterp commands.

(END)

Command
Development System
3/20/72

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 its output in another segment. The time before which this process is not to be created may be specified.

The principal difference between an absentee process and an interactive one is that "user_input" is attached to an absentee control segment containing commands and control lines; "user_output" is attached to an absentee output segment as well. The absentee control segment has the same syntax as an exec_com segment. (See exec_com in the MPM.)

Usage

enter_abs_request pname -ca*1*- ... -ca*n*- -ag arg*1* ... arg*n*

1) pname — specifies the path name of the absentee control segment associated with this request. The entry name must have the suffix .absin although it may be omitted in the command. Pname must be the first argument to the command.

2) ca*i* — is selected from the following list of control arguments and may appear in any position:

-output_file pname
-of pname — indicates that the user wishes to specify the name of the output segment. It must be followed by the path name of the absentee output segment.

-restart, -rt — indicates that the computation specified by this request may 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 will use. It must be followed by a positive integer

specifying the limit, in seconds. The default is no user supplied limit. There is a system enforced limit which an absentee process may use. Currently this absolute limit is twenty minutes.

-queue n, -q n — indicates in which priority queue the request is to be placed. It must be followed by an integer specifying the number of the queue. If this option is omitted, the request is placed in the third queue.

-time "deferred_time"
-tm "deferred_time" — 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. (See convert_date_to_binary_ in the MPM.) If the time consists of more than one component, it should 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 which 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 will be 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) argi — is an argument to the absentee control segment.

Notes

If the path name of the output segment is not specified, the output of the absentee process will be directed to a segment whose path name is the same as the absentee control segment, except that it has the suffix .absout.

The command checks for the existence of the absentee input segment and will reject 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 was issued at the deferred time.

Examples

Suppose that a user wants to request an off-line compilation. A control segment would be constructed called absentee_pl1.absin containing:

```
cwd current

pl1 x -table -symbols

dp -dl x.list

logout
```

The command line

```
enter_abs_request absentee_pl1.absin
```

would cause an absentee process to be created (some time in the future) which would:

1) set the working directory to a directory "current" inferior to the user's normal initial working directory;

2) compile a PL/I program named x.pl1 with two options;

3) dprint one copy of the list segment;

4) log out.

The output of these tasks would appear in the same directory as absentee_pl1.absin in a segment called absentee_pl1.absout.

Suppose that an absentee control segment, trans.absin, contained the following:

```
cwd &1
&2 &3 -list &4
&goto &2.a
&label pl1.a
fo &3.list; ioa_ "|; pli &3; co
&label alm.a
dp -dl &3.list
&goto &2.b
&label pl1.b
&3
&label alm.b
logout
```

The command

```
ear trans -li 300 -rt -ag work pl1 x -map
```

would cause a request for an absentee process to be made in queue 3 which will set the working directory to the directory "work" inferior to the normal initial working directory, then compile a PL/I program x.pl1 in that directory, produce a listing segment containing a map, append to the listing segment linkage information, issue a dprint request for the listing segment, and execute the program x, just compiled in the absentee process. There would be a CPU limit of five minutes placed on this process.

The command

```
ear trans -rt -tm "Monday midnight" -q 2 -ag comp alm yz
```

would cause a request for an absentee process to be placed in queue 2 which will set the working directory to the directory "comp" inferior to the initial working directory, assemble an ALM program named yz.alm, produce a listing segment, and issue a dprint request for the listing segment constructed. This process will not be created until after midnight of the next Monday.

Both absentee processes would issue a logout command as the last command in the process.

Both absentee computations could be restarted from the beginning if interrupted for any reason.

(END)

Name: exec_com, ec

The exec_com command is used to execute a series of command lines contained in a segment. It 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:

1. substitution of arguments to the command for special strings in the exec_com segment;

2. control of I/O streams;

3. generating command lines, control statements and input lines conditionally;

4. combining several exec_com sequences into one segment; and

5. altering the flow of control.

Usage

exec_com pathname -arg1- -arg2- ... -argn-

1) pathname — is the pathname of the segment containing the commands to be executed and control statements to be interpreted. The entry name of the segment must have the suffix ".ec", although the suffix can be omitted in the command invocation.

2) argi — is the string to be substituted for special strings in the exec_com segment.

The Input Segment

The exec_com segment should contain only command lines, input lines and control statements. It is normally 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 often used.

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 substution 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, arg1 is substituted for the string "&1" and arg10 is substituted for "&10".

The character "&" should be followed by a number, i, or by the string "ec_name". If argi is not provided, "&i" is replaced by the null string. The string "&ec_name" is replaced by the entry name of the exec_com segment without the ".ec" suffix. the string "&0" is replaced by the pathname argument to exec_com, just as it was given to the command.

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 there are twelve control statements: &label, &goto, &attach, &detach, &input_line, &command_line, &ready, &print, &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, which 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 I/O stream "user_input".

&attach	causes the I/O stream "user_input" 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 from the stream "user_i/o".
&detach	causes the I/O stream "user_input" to be reverted to its original value. The default is detach rather than attach.
&input_line on	causes input lines returned when using the &attach feature to be written on the stream "user_output".
&input_line off	causes such input lines not to be written out. The default is on.

3. &command_line, &ready and &print

These statements allow the control of the I/O stream "user_output". 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 stream "user_output" before they are executed.
&command_line off	causes subsequent command lines not to be written out. The default is on.
&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. The default is off.
&print char_string	causes the character string following &print to be written out on the I/O stream "user_output". The character "¬" is treated as a special character in an &print statement. The following is a list of strings that can appear and the characters

that replace them:

string	replacement
¬/	new line character
¬|	form feed
¬-	horizontal tab
¬¬	¬

No other characters should appear following "¬" in the &print statement.

4. &quit

&quit causes 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 generate command lines, input lines and control statements conditionally.

The form of these control statements is:

```
&if [ACTIVE_FUNCTION -arg1- ... -argn-]
&then THEN_CLAUSE
&else ELSE_CLAUSE
```

An active function 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 statement.

&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 statement.

&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 statement.

The active functions described in the MPM Reference Guide section, Logical Active Functions, 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 insure compatibility with control requests to be added to exec_com in the future.

The segment executed by exec_com can contains calls to exec_com. The user is cautioned against frivolous use of this feature when using 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 on its storage system entry that can replace "&ec_name" (concatenated with a ".ec" suffix).

Examples

1. Assume that the segment a.ec in the user's working directory contains:

```
pl1 &1 -table -list
dprint -dl &1.list
&quit
```

The command

```
exec_com a foo
```

would cause the following to be executed:

```
pl1 foo -table -list
dprint -dl foo.list
```

2. 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 -dl &1.list
&quit
```

The command

```
exec_com b my_file
```

would cause the following to be executed:

```
print my_file 1 99
```

The command

```
exec_com a foo
```

would cause the following to be executed:

```
pl1 foo -table -list
dprint -dl foo.list
```

3. 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 -dl &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 to be executed:

```
pl1 foo -table -list
dprint -dl foo.list
```

If the segment foo.pl1 did not exist, the command

```
exec_com d foo
```

would output the following:

```
foo.pl1 not found
```

4. 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/1
end text x compile exec_com
```

Names: file_output, fo
console_output, co

The file_output command allows the user to direct the I/O output stream "user_output" to a specified segment. The console_output command allows the user to direct it back to the terminal.

Usage

file_output -pathname-
console_output

1) pathname is an optional segment path name. If is not present, the file_output command will direct output to the segment, output_file, in the user's working directory. If the specified segment does not exist (pathname or output_file), it will be created. If it does already exist, subsequent output will be appended to the end of the segment.

Note

To avoid getting ready messages in the output file the file_output and console_output commands should appear on the same line of console input. (See Examples below.)

Examples

The sequence of commands

```
file_output my_info

list -a

list -p >sample_dir -d

console_output
```

will place in the segment 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. Note that the ready messages from the file_output command and the two invocations of the list command will also appear in my_info.

The command line

```
fo my_info; list -a; list -p >sample_dir -d; co
```

has the same affect as the first example except that no ready messages will appear in my_info.

 (END)

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.

Usage

fortran pathname -control_arg1- ... -control_argn-

1) pathname — is the path name of a FORTRAN source segment that is to be translated by the FORTRAN compiler. A directory path name and an entry name, segname, are derived from path name by calling expand_path_. The compiler takes its input from segname.fortran.

2) control_argi — can be chosen from the following list of control arguments:

-source, -sc — produces a line-numbered printable ASCII listing of the program. The default is no listing.

-symbols, -sb — lists the source program as above and all the names declared in the program with their attributes. The default is no symbols.

-map — lists the source program and symbols as above 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. The default is no map.

-assembly — lists the source program as for the -source control argument followed by an assembly-like listing of the compiled program. Note that producing an assembly-like listing significantly increases compilation time and should be avoided whenever possible by using the -map control argument. The default is no list.

-list, -ls — lists the source program and symbols as for the -symbols control argument followed by Note that use of the -list control argument

significantly increases compilation time and should be avoided whenever possible by using the -map control argument. The default is no list.

-brief, -bf causes error messages written into the stream "error_output" to contain only an error number, statement identification, and, when appropriate, the identifier or constant in error. In the normal, non-brief mode, an explanatory message of one or more sentences is also written.

-severity*i*, -sv*i* causes error messages whose severity is less than *i* (where *i* is 1, 2, 3, or 4; e.g., severity3) to not be written into the "error_output" stream although all errors are written into the listing. The default value for *i* is 1.

-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. Use of this control argument adds 5-10% 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 progran 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 used 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 satement labels for use by symbolic debuggers. The table appears as 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, to check for subscript values exceeding the declared bound dimension. 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. See the MPM command write-up of the profile command.

The following two control arguments are available for users who wish to maintain their FORTRAN source segments in ANSI card format.

-card
specifies that the source segment is in card image format.

-convert
specifies that the source segment is in card image format. The compiler generates a segment, segname.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 lower case.

2) Column 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) Columun one contains a "C", "c" or "*". The card image is a comment and columns 1-6 are preserved as is.

b) Column 6 contains a character other than zero or blank. Columns 1 thru 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. Columns 1-5 can contain blanks or numerals. The numerals present are concatenated to form as a single string and are followed by a tab.

The following control arguments are available, but are probably not of interest to the normal 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 typed 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.

-link -lk — generates a link to the operator segment instead of loading its address from the stack. This control argument is provided for users who must be able to switch operator segments easily, and is not suggested for the general user because of increased execution overhead.

Further information on the above control arguments is contained under Error Diagnostics and Listing.

Notes

A normal compilation produces an object segment, segname, and leaves it in the user's working directory. If segname existed previously in the directory, its Access Control List (ACL) is saved and given to the new copy of segname. Otherwise, the user is given "re" access to the segment with ring brackets V,5,5 where V is the validation level of the process active when the object segment is created.

The user's control arguments control the absence or presence of the listing segment for segname.fortran and the contents of that listing. If created, the listing segment is named segname.list. The ACL is as described for the object segment except that it is given "rwa" access when newly created. Previous copies of segname and (if the list option is on) segname.list are replaced by the new segments created by the compilation.

Note that because of the Multics standard which 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.

Severity Level	Meaning
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 right 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.

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" stream and returns to its caller.

Error messages are written into the stream "error_output" as they occur. Thus, a user at his console 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 set the brief option so that the message is shorter. An example of an error message in its long form is:

```
WARNING 156 IN STATEMENT 1 ENDING 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 option had been set the user would see instead:

```
WARNING 156 IN STATEMENT 1 ENDING ON LINE 5
j
SOURCE:  5     continue
```

Once a given error message has been typed on the user's console in the long form, all further instances of that error use the brief mode.

If the listing option is on, the error messages are also written into the listing segment. They appear, sorted by line number, after the listing of the source program. Because of an implementation restriction, no more than 100 messages are printed in the listing.

Listing

The listing created by fortran 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 which are:

1) type, dimension, common statements, etc.;

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, attributes, and references. Then comes a listing of external operators used followed by a listing of the error messages.

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 ending 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 which contains the source statement which produced the instruction. Op code, base-register, and modifier mnemonics are printed alongside 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.

 (END)

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 segnamei.list if it exists, and dprints and deletes segnamei.list. If the -output_file control argument is not specified, and output segment, segname.absout, is created in the user's working directory (if more than one segname is specified, the first is used). If the none of the segments to be compiled can be found, no absentee request is submitted.

Usage

fortran_abs segname1 ... segnamen -fortran_control_args-
-fortran_abs_control_args-

1) segnamei — is the path name of a segment to be compiled.

2) fortran_control_args — can be one or more nonobsolete control arguments accepted by the FORTRAN compiler and described in fortran. (See the write-up in the MPM.)

3) fortran_abs_control_args — can 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 <= 3). The default queue is 3. segnamei.list is also dprinted in queue n.

-copy n, -cp n — specifies the number of copies (n <= 4) of segnamei.list to be dprinted. The default is 1.

-hold — specifies that fortran_abs should not dprint or delete segnamei.list.

-output_file f, -of f — specifies that absentee output is to go to segment f where f is a path name.

Notes

Control arguments and segment names can be mixed freely and can appear anywhere on the command line after the command. All control arguments apply to all segment names. An unrecognizable control argument causes the absentee request not to be submitted.

Expanded segments containing include files are not deleted.

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 links that need to be snapped when setting up a process and when invoking the compiler need be snapped only once.

 (END)

Name: fs_chname

The fs_chname command is an interface to the storage system subroutine hcs_$chname_file. It causes an entry name 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 by-pass the star convention or to manipulate strangely-named segments. For segments with ordinary names, the rename, addname and deletename commands perform the same function. See the MPM write-ups for these commands.

Usage

fs_chname dir_name entry_name oldname newname

1) dir_name is the directory name portion of the path name of the segment in question.

2) entry_name is the entry name portion of the path name of the segment in question.

3) oldname is an old entry name to be deleted. See Notes below.

4) newname is a new entry name to be added. See Notes below.

Notes

When both an old entry name and a new entry name appear in the command line, the new entry name replaces the old entry name. This is equivalent to using the rename command.

If the old entry name is a null character string (""), then the new entry name is added to the segment. This is equivalent to using the addname command.

If the new entry name is a null character string (""), then the old entry name is deleted from the segment. This is equivalent to using the deletename command.

The user must have write attribute on the directory containing the entry in order to make any name changes.

Examples

```
fs_chname >my_dir alpha>beta alpha>beta alpha_beta
```

This example would replace the incorrect entry name alpha>beta (which 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.=
```

This example would add the entry name story.= to the specified segment. The addname 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 entry name story.equal to the segment. See the MPM Reference Guide section, Constructing and Interpreting Names, for a discussion of the equals convention.

 (END)

Name: get_com_line, gcl

The get_com_line command prints on the user's terminal the current value of the maximum expanded command line size. An expanded command line is one obtained after all active strings have been processed.

Usage

```
get_com_line
```

Note

The default maximum length of an expanded command line is 128 characters. It may be changed using the set_com_line command. For a discussion of the command language (including the treatment of active strings), see the MPM Reference Guide section, The Command Language.

 (END)

Name: getquota, gq

The getquota command returns information about the secondary storage quota and pages used for a specified directory.

Usage

getquota -ctl_arg- pathnamei ... pathnamen

1) ctl_arg — if -long or -lg, it specifies that the long form of output is to be used.

2) pathnamei — is the name of the directory about which quota information is desired. If pathnamei is -wd or -wdir, the working directory will be used. If no arguments are given, the working directory will be used. The star convention may by used to obtain quota information about several directories.

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 path name is specified. When only one path name 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, the number of immediately inferior directories with nonzero quotas is printed. The time-page product in units of page-days is also returned 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 interior directories with nonzero quotas he will have to print this product for all things directories in order to obtain the charge.

 (END)

Name: help

The help command assists users in obtaining information about commands and subsystems.

Asking for help with a command causes information about that command to be printed on the the user's terminal. After a small but useful amount of information has been typed the user is asked if he wants more help. If the user replies "yes" another block of information is typed and the user again questioned. Otherwise the command exits. Typing the command "help" (or "help help") causes information about the help command to be typed. A count of the lines to follow is printed before each time the user is asked if the wants more help.

Usage

help -name- -control_arg-

1) name — is the name of an information segment that the user wishes to read. It is the first component of a segment, in the installation information directory >documentation >iml_info or in the system information directory >documentation>info, that has .info as its second component. If the name argument is present, control_arg cannot be present.

2) control_arg — can be present only if the name argument is not present, and can have as its value:

-pathname xxx, -pn xxx — if this control argument is specified, help types the contents of the info segment whose path name is xxx instead of looking in the system or installation information directories.

Example

In the following example, messages typed by the user are underlined for clarity but are not underlined in the actual script.

help
(5 lines follow)
7/28/72 - help now gives following line count in its question.

The help command types out system information segments located in the directory >documentation>info and installation information segments located in the directory >documentation>iml_info.
Type "help name" to see the segment name.info.

17 lines follow. More help? yes

To see what system info segments are available, type
list -pn >udd>documentation>info *.info
(There are currently over 200 info segments.)

Some useful info segments are:

motd	message of the day
news	general information, on-line installations
sys	supervisor change information
pl1	status of PL/I compiler
fortran	status of FORTRAN compiler
basic	status of BASIC compiler
bugs	current list of system bugs
doc	documentation and assistance
pt	introduction to "peruse_text", which gives additional information on many commands

Rest of segment has 10 lines. More help? yes

help accepts one control argument:

-pathname xxx, -pn xxx — if this control argument is specified, help types out the info segment whose path name is xxx, instead of looking in the system or information directory. If the suffix .info is missing from xxx, help appends it.

Note

The data segments are composed of blocks of ASCII character information, with the blocks arranged in descending order of importance of their contents. Lines should be less than 60 characters long. Each block except the last is terminated by the ASCII character \006, which causes the help command to ask if more help is wanted. The first line of an info segment should contain a creation (or updating) date.

Command
Standard Service System
02/12/71

Name: hold, hd

The hold command may be issued after a quit signal or an unclaimed signal has interrupted a process. This will cause the history of the process up to the point of interruption to be preserved. That is, the current state of the call-save-return stack is saved. This history is preserved until a release command is issued.

Usage

hold

(END)

Command
Standard Service System
8/22/72

Name: how_many_users, hmu

This command tells how many users are 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 up, the number of absentee users and the maximum number of absentee users is printed also.

Usage

how_many_users -ctl_arg1- ... -ctl_argn- -arg1- ... -argn-

1) ctl_argi may 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, 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 argi.

2) argi specifies that only selected users are to be listed, and may be one of the following:

Name lists a count of logged in users with user name "Name".

.Project lists a count of logged in users with a project ID of ".Project".

Name.Project lists a count of logged in users with the name and project of "Name.Project".

Notes

Absentee counts in a selective use of how_many_users (i.e., when an argi is specified) are denoted by an asterisk (*).

Up to twenty classes of selected users are permitted.

Examples

1) Print summary information.

```
hmu

Multics 15.16, load 5.0/50.0; 6 users
```

2) Print summary information on absentee users.

```
hmu -absentee

Absentee users 0/2
```

3) Pring long information.

```
hmu -long

Multics 15.16; MIT, Cambridge, Mass.
Load = 24.5 out of 50.0 Units; users = 23
Absentee users = 0; Max absentee users = 2
System up since 08/01/72  0644.5
Last shutdown was at 08/01/72  0517.9
```

4) Print brief information about the project SysDaemon.

```
hmu -bf .SysDaemon

SysDaemon = 3 + 0*
```

5) Print brief information about the person Smith.

```
hmu -bf Smith

Smith = 1 + 1*
```

 (END)

Name: indent, ind

The indent command improves the readability of a PL/I source segment by by indenting it according to a set of standard conventions described below.

Usage

indent oldpath -newpath- -control_args-

1) oldpath — is the path name of the input PL/I source segment. If the source segment name does not have a suffix of .pl1, the suffix will be assumed.

2) newpath — is the path name of the output PL/I source segment. If the output segment name does not have suffix of .pl1, the suffix will be assumed. If this argument is omitted, newpath will be assumed to be the same as oldpath, and the indented copy of the program will replace the original copy.

3) control_args — may 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.)

-lmargin XX, -lm XX — set the left margin (indentation for normal program statements) to XX. If this argument is omitted, the default for XX is 11.

-comment YY, -cm YY — set the comment column to YY. Comments are lined up in this column unless they begin a line and are preceded by a blank line (or are at the beginning of the program or are a comment beginning in column 1). If this argument is omitted, the default for YY is 61.

-indent ZZ, -in ZZ — set indentation for each level to ZZ. Each do, begin, proc, and procedure statement will cause an additional ZZ spaces of indentation until the matching end statement is encountered. If this

argument is omitted, the default for ZZ is 5.

Conventions

Declaration statements are indented five spaces (with any identifiers that appear on extra lines, but which are still part of the same declaration, being lined up under the first identifier on the first line of the statement). Structure declarations are indented according to level number; after level two, additional levels are indented two more spaces each.

Multiple spaces are replaced by a single space, except inside of strings or for non-leading spaces and tabs in comments. 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 will print a warning message. Language keywords (do, begin, end, etc.) are recognized only at parenthesis level zero.

Restrictions

Lines longer than 350 characters will be split, since they overflow indent's buffer size. This is the only case in which indent will split a line.

Labeled end statements will not close multiple open do statements.

The indent command assumes that the identifiers begin, end, procedure, proc, declare, and dcl are reserved words, and are always language keywords. Thus, indent will become confused if the input contains a statement like

```
go to begin;
```

since it will think that the statement delimits a begin block.

Structure level numbers greater than 99 will not indent correctly.

 (END)

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 MPM Reference Guide section, The System Libraries and Search Rules.

Usage

initiate pathname -ref1- -ref2- ... -refn- -control_arg-

1) pathname — is the path name of the segment to be initiated.

2) refi — are optional reference names by which the segment may be known without further initiating. See Notes below.

3) control_arg — may be the string "-s" and may 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, refi, are given in the command line, then the segment will be initiated by the entry name part of the path name. If any reference names, refi, are present in the command line, the segment will not be initiated by its entry name, but by the reference names so given. If the path name is a single element name then the directory assumed is the working directory. The < and > symbols are recognized in the path name; the star convention may not 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 non-null access to that segment.

Examples

```
initiate >udd>m>mmm>gamma x y
```

would initiate the segment >udd>m>mmm>gamma with the names x and y.

```
initiate pop
```

would initiate the segment pop in the working directory and give it the reference name pop.

```
initiate xx u v -s
```

would initiate the segment xx in the working directory with the reference names u and v and would print out the assigned segment number.

 (END)

Command
Standard Service System
8/18/71

Name: iocall

Often it is useful to issue I/O system function calls from command level. The iocall command is provided for this purpose. It will perform the following functions: 1) it will accept a variety of argument formats, supplying useful default arguments where required; 2) it will print the values of return arguments; 3) it will decode and print status returned by the I/O system.

More information on I/O can be found in chapters 1 and 2 of the Introduction to Multics, the Reference Guide Section, and in the description of ios_.

Usage

iocall function_name stream_name argument1 ... argumentn

1) function_name — is one of the I/O system calls: abort, attach, changemode, detach, getsize, order, read, readsync, resetread, resetwrite, seek, setsize, tell, worksync, write, writesync.

2) stream_name — is the stream_name argument present on all function calls.

Notes

Below is a list of the function calls accepted by iocall. The list starts with the complete I/O system call. Following the call is a list of the variations of the call acceptable by iocall. Following this are notes on special cases associated with the call.

1) attach

call ios_$attach (stream_name, type, device/stream_name, mode, status);

iocall attach stream_name type device/stream_name -mode1- ... -moden-

The various modei are concatenated and separated by commas to form the mode argument. If there are no modei, mode is set to the null string.

2) detach

```
call ios_$detach (stream_name, device/stream_name,
     disposal, status);

iocall detach stream_name
iocall detach stream_name device/stream_name
iocall detach stream_name device/stream_name disposal
```

If the arguments device/stream_name or disposal are missing, the null character string is supplied. The status argument is supplied and decoded.

3) read

```
call ios_$read (stream_name, workspace, offset, nelem,
     nelemt, status);

iocall read stream_name segment
iocall read stream_name segment nelem
iocall read stream_name segment offset nelem
```

The offset and nelem arguments, if present, are in decimal. If the offset argument is missing, 0 is supplied. If the nelem argument is missing, the maximum size of the segment argument is provided as a multiple of the current element size. A pointer to the base of the segment argument is supplied as the workspace argument. If the segment argument does not exist, it is created in the working directory. The status argument is supplied and decoded.

4) write

```
call ios_$write (stream_name, workspace, offset, nelem,
     nelemt, status);

iocall write stream_name
iocall write stream_name segment nelem
iocall write stream_name segment offset nelem
```

The offset and nelem arguments, if present, are in decimal. If the offset argument is missing, 0 is supplied. If the nelem argument is missing, the bit count of the segment argument is provided as a multiple of the current element size. A pointer to the base of the segment argument is supplied as the workspace argument. The status argument is supplied and decoded.

5) seek

```
call ios_$seek (stream_name, ptrname1, ptrname2, offset,
     status);

iocall seek stream_name ptrname1
iocall seek stream_name ptrname1 ptrname2
iocall seek stream_name ptrname1 ptrname2 offset
```

The offset argument, if present, is decimal. If the offset argument is missing, 0 is supplied. If ptrname2 is missing, "first" is supplied. The status argument is supplied and decoded.

6) tell

```
call ios_$tell (stream_name, ptrname1, ptrname2, offset,
     status);

iocall tell stream_name ptrname1
iocall tell stream_name ptrname1 ptrname2
```

If ptrname2 is missing, "first" is supplied. The offset argument is supplied and its value is printed in decimal on return. The status argument is supplied and decoded.

7) setsize

```
call ios_$setsize (stream_name, elementsize, status);

iocall setsize stream_name elementsize
```

The elementsize argument is decimal. The status argument is supplied and decoded.

8) getsize

```
call ios_$getsize (stream_name, elementsize, status);

iocall getsize stream_name
```

The elementsize argument is provided and its value is printed in decimal on return. The status argument is supplied and decoded.

9) order

```
call ios_$order (stream_name, request, argptr, status);

iocall order stream_name request
```

The argptr argument is supplied as a null pointer. The status argument is supplied and decoded.

10) changemode

```
call ios_$changemode (stream_name, mode, old_mode, status);

iocall changemode stream_name -mode1- ... -moden-
```

The various modes are concatenated and separated by commas to form the mode argument. If there are no mode*i*, mode is set to the null character string. The old_mode argument is supplied and its value printed on return. The status argument is supplied and decoded.

11) resetread

```
call ios_$resetread (stream_name, status);

iocall resetread stream_name
```

The status argument is supplied and decoded.

12) resetwrite

```
call ios_$resetwrite (stream_name, status);

iocall resetwrite stream_name
```

The status argument is supplied and decoded.

13) abort

```
call ios_$abort (stream_name, old_status, status);

iocall abort stream_name
```

A zero bit string is supplied as the old_status argument. The status argument is supplied and decoded.

14) readsync

```
call ios_$readsync (stream_name, smode, limit, status);
```

```
iocall readsync stream_name smode
iocall readsync stream_name smode limit
```

The limit argument is in decimal, if present, and is set to a large value if absent. The status argument is supplied and decoded.

15) writesync

```
call ios_$writesync (stream_name, smode, limit, status);

iocall writesync stream_name smode
iocall writesync stream_name smode limit
```

The limit argument is decimal, if present, and is set to a large value if absent. The status argument is supplied and decoded.

(END)

Name: iomode

The iomode command changes the type of code conversion performed by the I/O system for a specified device.

Usage

iomode mode -ioname-

1) mode — specifies the type of code conversion. It may be "edited" or "normal". The edited mode suppresses all escapes; i.e., non-available graphics. The normal mode includes escapes.

2) ioname — specifies the stream name associated with the device whose mode is to be set. If ioname is not specified the stream name "user_output" is assumed. For the normal user this will have the effect of setting the mode of his terminal.

Note

Other modes are available in the I/O system and may be set by using the iocall command (with the changemode function) or the ios_$changemode subroutine. See the MPM write-ups for iocall and ios_. The iomode command merely calls ios_$changemode to make the change.

 (END)

Command
2/12/73

Name: line_length, ll

The line_length command allows the user to control the maximum length of a line written on the device which his process is using for output on the "user_output" I/O stream. This device will usually be his terminal.

Usage

line_length maxlength

1) maxlength is the maximum number of characters which may henceforth be printed on a single line using the I/O stream "user_output". In most cases, this is the maximum length of a line of output printed at the user's terminal.

 (END)

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 the MPM Reference Guide section, Segment, Directory, and Link Attributes.

Usage

link path1*1* path2*1* ... path1*n* -path2*n*-

1) path1*i* specifies the segment to which path2*i* is to point.

2) path2*i* specifies the link to be created. If not given (in the final argument position of a command line only) a link to path1*i* will be created in the working directory with the entry name portion of path1*i* as its entry name.

Notes

Entry names must be unique within directories. Therefore, if the creation of a link would lead to a duplicate name, the user is interrogated as to whether he wishes the entry bearing the old instance of the name to be deleted. If not, the link will not be created.

The star and equals conventions may be used.

The user must have append access in the directory in which the link is to be created.

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.

 (END)

Name: lisp

The lisp command invokes a LISP subsystem that provides an interpreter of the MACLISP dialect of LISP for interactive use on Multics.

LISP is a recursive language that is suited for many applications. The Multics implementation is designed to be very fast yet not limited by storage capacity as many other LISP systems are. Over two hundred fifty system functions are provided for diverse user needs while a compiler is provided so that a user can compile his own functions from functions written for interpretive use.

Usage

lisp -pathname- -argument1- ... -argumentn-

1) pathname is a path name of a saved environment from which the subsystem constructs an initial active environment. If not specified, the standard LISP saved environment is used. The user can save an environment from inside the LISP subsystem and then use this saved environment at some later time to initialize a new lisp command to the same state as when he saved it.

2) argumenti is an arbitrary argument which can be referred to from within the LISP subsystem with the appropriate system function.

Notes

For a complete description of the MACLISP dialect of LISP, consult the document LISP Reference Manual (MACLISP Dialect). The file lisp.info describes methods for obtaining this document in addition to other useful information. Also, refer to the MPM write-up for the Multics command, lisp_compiler.

This MPM write-up is divided into two parts with the first part describing the basic structure of the interpreter and the methods by which the user can control it while the second part summarizes some of the characteristics that distinguish the MACLISP dialect from other dialects of LISP. Both parts assume that the user has some prior knowledge of LISP, such as having read Weissman's LISP 1.5 PRIMER or part of the LISP 1.5 Manual.

Part I: Essential Facts for Using the Interpreter

Accomplishing Evaluation

Explanations in the following text are illustrated by an example of a terminal session. Line numbers have been added for reference and a star is placed after the numbers for lines that have items entered by the user.

```
 1*      lisp
 2       *
 3*      (cons (quote a) (quote b))
 4       (a . b)
 5*      (car (quote (a b c d)))
 6       a
 7*      3245
 8       3245
 9*      (setq foo (quote bar))
10       bar
11*      foo
12       bar
13*      (quit)
14       r 410  1.474  9.264  204
```

When the lisp command is issued, an initial environment of atoms, functions, and list structure is constructed from a saved environment. After this environment has been constructed, the interpreter reaches its basic state, known as top level. Whenever the interpreter reaches this state from some other state it outputs a star to the user's terminal as is seen in line 2. MACLISP has an eval type top level as opposed to some other dialects of LISP that have an evalquote type top level. Thus the user must type one form to be evaled followed by a new line character, instead of two S-expressions, one to be applied as a function to the second. Note that one has to explictly quote an S-expression if one does not want to have it evaluated. On line 3 the user types a form that is a simple function call. The evaluation is printed on line 4. Lines 5 and 6 illustrate the same thing. On line 7 the user simply types a number followed by a new line character. Numbers evaluate to themselves which is shown on line 8. Note that it is very easy to forget that the interpreter starts out operating in base eight, octal.* In

*The input radix can be varied by resetting the system variable, ibase, and the output radix can be varied by resetting the variable, base.

MACLISP, atoms that can have values are called symbols. On line 9 the user sets the symbol, foo, to have the value, bar. Then on line 11 the user evaluates this symbol. Finally we give a simple example of control of the interpreter. In order to leave the subsystem permanently one uses the function quit which takes no arguments. This is illustrated above by line 13. All atoms and list structure that have been created by the user are then destroyed and the subsystem returns control to Multics.

Control of the Interpreter

A MACLISP interpreter gives a user a great amount of control over its behavior. It has many switches that can be set by the user and functions that he can replace. Some of these features are discussed in Part II. Many of the switches can be set in two different ways, either by using some function or by a general method that has real time effect. This latter method is described immediately below.

If at any time the user depresses the interrupt (break or attention) key on his terminal, the LISP interpreter responds by prompting the user and then waits for input. The user can then type a single letter command followed by a new line character. It must be stressed that these commands have effect in real time, they happen when they are given and not after the interpreter is finished doing whatever it was doing at the time the attention key was depressed. Three important commands are:

z gives the user a Multics Command Processor at a higher stack frame. This is identical to what happens in most Multics programs when the interrupt key is depressed. The Multics command, start, starts lisp running again in the standard manner. If the command, program_interrupt, is issued, then lisp again prompts the user, but only accepts the three characters mentioned on this page.

g interrupts the current LISP program and returns control to the top-level function of the interpreter. The internal LISP stacks are unwound and temporary bindings of variables are restored.

b enters a "break loop", a read-eval-print loop at a higher level in the LISP stacks. The user can examine variables, do whatever else he wishes to do and then return to the previously running program by typing the atom, <dollar-sign>p ($p).

Error Conditions

If the user tries to evaluate some item that causes a LISP error, the interpreter usually creates a break loop and informs the user of this by typing something such as ";bkpt unbnd-vrbl". In many cases the user can then modify the incorrect item and then restart the program. However, for now it is sufficient to know that if the user types the atom, $p, the interpreter returns to top level and the user can try again.

Part II: Features of the MACLISP Dialect

Features Common to All MACLISP Implementations

1) Debugging System -- As the name, lisp subsystem, implies, a MACLISP interpreter provides a complete system in which to work. Thus one of its important features is a sophisticated debugging system. Functions are provided to set break-points, do traces, do back traces, examine variables at various levels in the LISP stack, reset variables, examine arguments in function calls, return values for function calls that are still stacked up, and others. In addition, as mentioned in Part I, most errors signalled by the interpreter are correctable from the break loop created by the error. These break loops are created by the user interrupt system that is mentioned below.

2) Real Time Control Characters -- A method is provided for giving commands from the console to the interpreter while it is also doing evaluation.

3) Programmable Portions of the Interpreter -- Many parts of the interpreter can be modified by the user. Besides offering versatility for regular programming, this facilitates the use of LISP as a language for implementing other languages and subsystems. For example, the languages Micro-Planner and CONNIVER and the mathematical laboratory, MACSYMA, are all written in LISP.

Reader Syntax Table -- The syntax categories of characters can be set by the user.

Macro Characters -- One of the things that can be entered in the syntax table is whether or not a character is a macro character. When the reader encounters a macro character, it invokes a function associated with that character. For example the system comes supplied with two macro characters, <accent-acute> (') which is the quote macro and <semi-colon> (;) which is the comment macro. When the reader sees

'<S-expression>

the associated function for the macro character ' transforms it into

(quote <S-expression>)

when a <semi-colon> is encountered the associated function discards the rest of that line.

Multiple Obarrays -- Instead of a single obarray (or oblist as it is known in other LISP's) several may be used. The user can create new obarrays, either totally or incrementally different, and then instruct the LISP reader to use a particular one. This allows use of modular systems by preventing name conflicts.

Variable Top-level Function -- The user can set the top-level function to be anything. For example the user could make an evalquote top level.

User Interrupt Functions -- On the occurrence of certain conditions such as various errors, an alarmclock ringing, or certain real time control characters being entered, the interpreter executes various user interrupt functions. For example many of the error interrupts are preset to a break-loop producing function. The user has the ability to set any of the user interrupt functions.

4) Arbitrary Precision Arithmetic -- In addition to a large number of arithmetic functions including exponential and trigonometric functions, MACLISP's arithmetic capabilities are further enhanced by the ability to do integer arithmetic to arbitrarily large precision.

5) Compiled Code -- Due to the structure of the interpreter, the code produced by the compiler is very efficient as compared to other dialects of LISP. Refer to the MPM write-up for the Multics command, lisp_compiler.

Features Specific to the Multics Implementation

1) The entire address space of the Multics virtual memory can be used for LISP storage.

2) Functions can be written in the other languages that exist on Multics.

3) A new data type called character strings exists, and functions are provided to manipulate them.

References

1) PDP-6 LISP (LISP 1.6), A.I. Memo No. 116A, revised 1967, Project MAC, Massachusetts Institute of Technology.

2) Crisman, P.A., Editor, Compatible Time-Sharing System: A Programmer's Guide, 2nd edition, M.I.T. Press, 1965.

3) McCarthy, John, et al, LISP 1.5 Programmer's Manual, 2nd edition, M.I.T. Press, 1966.

4) Moon, D.A., et al, LISP Reference Manual (MACLISP Dialect), Project MAC, Massachusetts Institute of Technology, 1973.

5) Weissman, Clark, LISP 1.5 Primer, Dickerson Publishing Company, 1967.

6) White, John L., An Interim LISP User's Guide, A.I. Memo No. 190, Project MAC, Massachusetts Institute of Technology, 1970.

Name: lisp_compiler, lcp

The lisp_compiler command invokes a compiler that translates a file of LISP functions written for interpretive use into a standard Multics object segment. These compiled functions can then be used from within an actively running LISP.

Usage

lisp_compiler -pathname- -control_arg1- ... -control_argn-

1) pathname — is the path name of a segment or multi-segment file to be compiled. If the name does not end in .lisp, the suffix is supplied. This file can contain function definitions, declare's, and other LISP code which will be executed when the object segment is made known to the LISP environment by the fasload function.

2) control_argi — can be chosen from the following list of control arguments:

-time, -tm — prints out the time taken to compile each function.

-total_time, -total, -tt — at the end of the compilation, prints out the CPU time, paging, etc.

-nowarn, -nw — do not print warning messages.

-pathname xxx, -pn xxx, -p xxx — the following argument (xxx) is the path name of the segment to be compiled. This control argument specifies that the name should be used exactly as given. The .lisp suffix is not appended to it, and it can begin with a minus sign without adverse effect.

-all_special — make all variables "special".

-macros, -mc — copy macro definitions into the output so that they will be defined at run time.

-genprefix xxx, -gnp xxx — Sets the prefix for generated function names to xxx.

-check, -ck — do not generate an object segment, just check for errors.

-eval xxx — evaluate the S-expression xxx before beginning the compilation.

Notes

The object segment created by this command is placed in the working directory with a name which is the first component of the name of the source file; i.e., the name of the source file up to but not including the first period.

Include files can be used by inserting the statement:

(%include name) or (%include "name")

The include file name.incl.lisp is inserted into the input at that point. The standard include file search rules are used (see the MPM Reference Guide section, The System Libraries and Search Rules).

For a complete description of the MACLISP dialect of LISP, consult the document LISP Reference Manual (MACLISP Dialect). The help segment lisp.info describes methods for obtaining this document in addition to other useful information. Also, refer to the MPM write-up for the Multics command, lisp.

 (END)

Name: list, ls
listotals, lt
listnames, ln

The list command enables the user to determine the names (including multiple names), modes, times last used and modified, and lengths of either all the entries in a specified directory or selected entries. The related listotals and listnames commands give information only about counts and lengths, and names, respectively. All three commands present information about segments, directories, multi-segment files, and links (in that order, where appropriate), and respect various options as to searching and sorting, as explained below.

Usage: list, ls

list entry1 ... entryn -opt1- ... -optn-

1) entryi	is an entry name (see the -pn option for a directory). If no entries are specified, all the entries of the relevant directory are dealt with.
2) opti	is chosen from the following list of options (see also Notes below):
-pathname, -pn	uses the directory specified in the immediately following path name. The desired entries follow the path name. The default is the current working directory. -pn may occur more than once in a single invocation of the command. Entries preceding the first -pn refer to the working directory. (See Examples below.)
-segment, -sm	lists segments only. The default is on.
-directory, -dr	lists directories only. The default is off.
-multisegment_file -msf	lists multi-segment files only. The default is off.

-link, -lk	lists links only. The default is off.
-branch, -br	lists branches, i.e., segments, directories, and multi-segment files only. The default is off.
-all, -a	lists all segments, directories, multi-segment files, and links. The default is off.
-date_time_used -dtu	sorts on and prints the date and time last used. The order is reverse chronological, i.e., the most recent first. The default is off.
-date_time_modified -dtm	sorts on and prints the date and time last modified. The order is reverse chronological, i.e., the most recent first. The default is off.
-reverse, -rv	reverses the order (to chronological or least recent first) for the -dtu or -dtm options. If -dtu is wanted, only -rv need be given. The default is off.

Notes

In the discussion of options, default means "what is assumed if this option is not given for a particular invocation of the command". on means the specified action is taken. off means the specified action is not taken.

The last three options (-dtu, -dtm, -rv) do not apply to links.

The star convention may be used in the entryi argument.

Conflicting options (e.g., -dtu and -dtm) should not be given in a single invocation of the command. Note that -sm, -dr, and -lk do not conflict.

Examples

```
list -a -rv
```

would list all the segments, directories, multi-segment files, and links in the current working directory, with the segments, directories, and multi-segment files sorted on the date and time last used, in chronological order.

```
list a b -pn >udd>Multics>Doe *.pl1 -pn >udd>MAC>Roe **
```

would list segments a and b (if present) in the working directory, all two component segments with a second component of pl1 in the directory >udd>Multics>Doe, and all of the segments in the directory >udd>MAC>Roe.

Usage: listotals, lt

```
listotals entry1 ... entryn -opt1- ... -optn-
```

1) entry*i* as above.

2) opt*i* as above except that the last three options are not valid (-dtu, -dtm, and -rv).

The response is a count of, and the total number of records occupied by, segments, directories, and multi-segment files, and a count of links, as appropriate to the particular options chosen.

Usage: listnames, ln

```
listnames entry1 ... entryn -opt1- ... -optn-
```

1) entry*i* as above.

2) opt*i* as above except that the last three options are not valid (-dtu, -dtm, and -rv).

The response is a list of the names of the segments, directories, multi-segment files, and links, as appropriate to the particular options chosen.

(END)

Command
Development System
3/14/72

Name: list_abs_requests, lar

The list_abs_requests command allows the user to obtain information about absentee requests. Normally the user will be allowed information only concerning requests which he has made.

Usage

list_abs_requests option1 ... optionn

1) optioni — is selected from the following list of options and may appear anywhere on the command line:

-total, -tt — indicates that the user wants only the total of the requests in the queue.

-long, -lg — indicates that all of the information pertaining to an absentee request will be printed. If this option is omitted, only the full path name of the absentee control segment will be printed.

-queue n, -q n — indicates which queue is to be searched. It must be followed by an integer specifying the number of the queue. If this option is omitted, the third priority queue is searched unless the -all option 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.

Note

The -total and -long options are incompatible.

Examples

1) 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

2) list_abs_requests -long -queue 1

```
     Queue 1: 2 requests. 27 total requests.

     Absentee input segment:     >udd>M>Day>dump>translate.absin
     Restartable:                yes
     Deferred time:              09/16/71  2300.0 edt Thu
     Argument string:            "pl1"
                                 "abcd"
                                 "-table"
                                 "-map"

     Absentee input segment:     >udd>M>Day>bind>auto_bind.absin
     Restartable:                no
     Cpu limit:                  600 seconds
     Absentee output file:       >udd>M>Day>bind>bd.out
```

3) 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.
```

 (END)

Command
5/18/73

Name: list_daemon_requests, ldr

The list_daemon_requests command prints on the user's terminal information about dprint and dpunch requests. Normally the user is allowed to obtain information concerning only requests which he has previously made. See the MPM command write-ups for dprint and dpunch.

Usage

list_daemon_requests control_arg1 ... -control_argn-

1) control_argi — is 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 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 omitted, only the full path name of the segment to be printed or punched is printed.

-queue n, -q n — indicates which 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 are to be searched starting with the highest priority queue and ending with the lowest priority queue.

Note

The -total and -long control arguments are incompatible, and cannot be used in the same list_daemon_requests command line.

Examples

1) list_daemon_requests

```
     Queue 3: 3 requests. 6 total requests.

          >Jones_dir>dump>translate.list
          >Jones_dir>doc>ldr.runoff
          >Jones_dir>Jones.profile
```

2) list_daemon_requests -long -queue 1

```
     Queue 1: 2 requests. 27 total requests.

     Pathname:       >Smith_dir>foo.list
     Type:           print
     Copies:         1
     Delete:         yes
     For:            Jones

     Pathname:       >doc>info>motd.info
     Type:           print
     Copies:         3
     Delete:         no
     To:             575 Tech Sq.
```

3) list_daemon_requests -total -all

```
     Queue 1: 2 requests. 15 total requests.

     Queue 2: 0 requests. 0 total requests.

     Queue 3: 0 requests. 39 total requests.
```

 (END)

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 added to the directory. For a discussion of Initial ACLs, see the MPM Reference Guide section, Access Control.

Usage

listacl pathname acname1 ... acnamen -control_arg-

1) pathname — specifies the directory in which the directory Initial ACL should be listed. If it is "-wd", "-working_directory" or omitted then the working directory is assumed. If it omitted then no acnamei may be specified. The star convention may be used.

2) acnamei — is an access control name. If no acnamei is specified then the whole Initial ACL will be listed. acnamei must be of the form person.project.tag. Any components missing on the left must be delimited by periods; however, the periods may be omitted on the right. If one or more of the components is missing then all access names that match the given components will be listed. If acnamei is "-a" then the whole Initial ACL will be listed.

3) control_arg — may be -ring (-rg). It may appear anywhere on the line and affects the whole line. If present it must be followed by a digit, where $0 \leq \text{digit} \leq 7$, which specifies which ring's Initial ACL should be listed. If the control argument is not given then the user's ring is assumed.

Examples

```
list_iacl_dir all_runoff .Faculty Fred
```

will list, from the directory Initial ACL in the directory all_runoff, all entries with the project name Faculty and all entries with the person name Fred.

```
lid -wd -a -rg 5
```

will list all entries in the directory Initial ACL for ring 5 in the working directory.

 (END)

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 added to the directory. For a discussion of Initial ACLs, see the MPM Reference Guide section, Access Control.

Usage

listacl pathname acname1 ... acnamen -control_arg-

1) pathname — specifies the directory in which the segment Initial ACL should be listed. If it is "-wd", "-working_directory" or omitted then the working directory is assumed. If it is omitted then no acnamei may be specified. The star convention may be used.

2) acnamei — is an access control name. If no acnamei is specified then the whole Initial ACL will be listed. acnamei must be of the form person.project.tag. Any components missing on the left must be delimited by periods; however, the periods may be omitted on the right. If one or more of the components is missing then all access names that match the given components will be listed. If acnamei is "-a" then the whole Initial ACL will be listed.

3) control_arg — may be -ring (-rg). It may appear anywhere on the line and affects the whole line. If present it must be followed by a digit, where $0 \leq \text{digit} \leq 7$, which specifies which ring's Initial ACL should be listed. If the option is not given then the user's ring is assumed.

Examples

```
list_iacl_seg all_runoff .Faculty Fred
```

will list, from the segment Initial ACL in all_runoff, all entries with the project name Faculty and all entries with the person name Fred.

```
lis -wd -a -rg 5
```

will list all entries in the segment Initial ACL for ring 5 in the working directory.

 (END)

Command
Development System
10/14/71

Name: list_ref_names, lrn

This command accepts both segment numbers and pathnames and prints the reference names and segment numbers by which segments are known. When segment numbers are specified, it also prints pathnames.

Usage

list_ref_names a1 ... an

1) ai can be segment numbers, pathnames, or options.

If ai is a segment number, the pathname and reference names of segment ai will be printed.

If ai is a pathname, the segment number (in octal) and the reference names of segment ai will be printed. If ai looks like an option (i.e., if it is preceded by a minus sign) or a number, then ai should be preceded by -name or -nm.

The following options are available for use with this command:

-from i
-to k — These two options allow one 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 option is omitted, the segment number of the first non-ring 0 segment will be assumed, unless -all is used (see below). If the -to option is omitted, the highest used segment number will be assumed.

-brief, -bf — This option suppresses printing of the reference names for the entire execution of the command. This option may appear anywhere in the line.

-all, -a — This option causes the pathnames and reference names of all known segments to be printed, as well as the reference names of ring 0 segments. This option may appear anywhere in the line. The -a option is equivalent to -from 0.

Notes

All of the above forms (segment numbers, pathnames, and options) may be mixed. For example:

```
list_ref_names pathone 156 -from 230
```

In the above command line, the segment number (in octal) and the reference names of pathone are returned. The pathname and reference names of segment 156 and of all segments from 230 on are also returned.

If called with no arguments, list_ref_names prints information on non-ring 0 segments only.

(END)

Name: listacl, la

This command lists some or all of the entries on an Access Control List (ACL) of either a segment or directory.

Usage

listacl -pathname- -acname1- ... -acnamen- -control_arg-

1) pathname specifies the segment or directory for which the ACL should be listed. If it is "-wd", "-working_directory" or omitted, then the working directory is assumed. If it is omitted then no acnamei may be specified. The star convention may be used.

2) acnamei is an access control name. If no acnamei is specified then the whole ACL will be listed. acnamei must be of the form person.project.tag. Any components missing on the left must be delimited by periods; however, the periods may be omitted on the right. If one or more of the components is missing then all access names that match the given components will be listed. If acnamei is "-a" then the whole ACL will be listed.

3) control_arg may be -ring_brackets (-rb). It may appear anywhere on the line and affects the whole line. If it is present, the ring brackets will be listed. Ring brackets are discussed in the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

Examples

listacl notice.runoff .Faculty Doe

will list, from the ACL of notice.runoff, all entries with the project name Faculty and all entries with the person name Doe.

la *.pl1 -rb

will list the whole ACL and the ring brackets of every segment in the working directory that has a two-component name with second component pl1.

 (END)

Name: login, l

The login command is used to gain access to the system. login is actually a request to the answering service to start the user identification and process creation procedures. Therefore, this command can only be issued from a terminal connected to the answering service; that is, one which has just dialed up, or one which has been returned to the answering service after a session terminated with a "logout -hold" command.

The login command will request a password from the user (and will attempt to insure either that the password does not appear at all on the user's terminal or that it is throughly hidden in a line of cover-up characters). The password is a string of one to eight letters and/or integers associated with the person ID. It is maintained by the system administration.

After the user responds with his password, the system will look up the person ID, the project ID, and the password in its tables, and verify that the person ID is valid, that the user is a legal user of the project, that the project ID is valid 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 the new process.

Usage

login person -project- -control_args-

1) person — is the user's registered personal identifier. This argument must be supplied.

2) project — is the identification of the user's project. If this argument is not supplied, the default project ID associated with the person ID will be assumed. See the -change_default_project control argument below for changing the default project ID to the project ID specified by this argument.

3) control_args — may be selected from the following:

-brief, -bf	Messages associated with a successful login will be suppressed. If the standard process overseer is being used, then the message of the day is not printed.
-home_dir path, -hd path	The user's home directory will be set to the path specified, if the user's project administrator allows him to specify his home directory.
-process_overseer path, -po path	The user's process overseer will be the procedure given by the path specified, if the user's project administrator allows him to specify his process overseer.
-no_print_off, -npf	The system will overtype several lines to provide a black area for the user to type his password.
-print_off, -pf	The system will not overtype several lines for the password, since the user's terminal responds to the printer off control sequence.
-no_preempt, -np	If the user can only be logged in by preempting some other user in his load control group then the login does not take place.
-no_start_up, -ns	If the user has a start_up.ec segment, and the project administrator allows the user to avoid it, instruct the standard process overseer not to execute it.
-account id, -ac id	Replace the normal account identifier for the user with id. (This option currently has no effect.)
-force	If the user has the guaranteed login attribute, log the user in if at all possible. Only system users who perform emergency repair functions will have the necessary attribute.

-change_password, -cpw	Request to change the user's password to a newly given password. The login command will request the old password, and then a new password. If the old password is correct, the new password will replace the old for subsequent logins, and the message "password changed" will be printed at the user's terminal. Note that the user should not type the new password as part of the control argument.
-change_default_project, -cdp	Request to change the user's default project ID to be the project ID specified in this login command line (see the description of the second argument above). If the password given by the user is correct, the default project ID will be changed for subsequent logins, and the message "default project changed" will be printed at the user's terminal.

Notes

Several parameters of the user's process, as noted above, can be controlled by the user's project administrator. The project administrator may allow the user to override some of these attributes by specifying control arguments in his login line. See the MPM Reference Guide section, Protocol for Logging In, for more information about these variable parameters and their usual values.

 (END)

Name: logout

The logout command terminates a user session and ends communication with the Multics system.

Usage

logout -control_args-

1) control_args is an optional control argument that can be chosen from the following:

-hold — 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 also the MPM Reference Guide section, The Multics Command Language Environment.

 (END)

Name: mail, ml

The mail command allows the user to send a segment to another user or to print messages sent to him. Mail sent to a user is placed in the segment "mailbox" in his home directory. The mailbox is provided with a lock.

Usage

mail -path- -person1- -project1- ... -personn- -projectn-

1) path is the pathname of a mailbox segment to be printed (if no personi projecti pairs are supplied) or of a segment to be sent to other user or users (when one or more personi projecti pairs are specified).

2) personi is the name of a person to whom mail is to be sent.

3) projecti is the name of a project on which personi works.

Printing Mail

The mailbox segment named by path will be locked and its contents printed out preceded by a line of the form

x messages, y lines

If path is omitted, the segment "mailbox" in the user's home directory is printed. After the mail is printed, the mail command will ask whether to delete the mail. If the answer is no, the command mail unlocks the mailbox and terminates it; if the answer is yes, mail truncates the mailbox to zero length.

Sending Mail

The contents of the segment path will be copied into the segment

>user_dir_dir>projecti>personi>mailbox

for each person-project pair specified. When mail is sent, the command first locks the target mailbox and then copies the contents of the mail into the target mailbox, preceded by a header identifying the sender:

From Person.Project date time

The segment to be mailed must be less than one record long. Illegal characters will be removed from the mail as it is sent.

If path is *, mail will respond "Input" and accept lines from the keyboard until a line with only a period (.) is typed; the typed lines then will be sent to the specified addresses.

Note

The user of the mail command must have rwa access to any mailbox he accesses, and e access to the directory superior to the mailbox in order to read or send mail.

Entry: mail$unlock

The user will receive a message of the form

```
mail:  The segment is already locked.
       mailbox busy, try again later.
```

if the mailbox he attempts to access is locked. Because quitting out of mail may leave a mailbox locked, this entry is provided to force the lock to zero. The mailbox specified by path will have its lock cleared. If path is not specified, the mailbox in the user's home directory will be unlocked.

Usage

```
mail$unlock -path-
```

1) path as above.

Creating a Mailbox Segment:

In order to receive mail, there must be a segment in the user's home directory called mailbox whose ACL is set to rwa for all users. To create such a segment perform the following sequence of commands in your home directory.

```
create mailbox
setacl mailbox rwa *.*.*
```


(END)

Name: make_peruse_text, mpt

In order that the peruse_text command may perform efficiently, it is necessary to translate a source segment (document) which is written for use with peruse_text. This translation produces a segment comprised of the original text followed by some information used by peruse_text to access a labeled title without performing a line-by-line search.

Usage

make_peruse_text pathname

1) pathname specifies that the segment pathname.pts is the source segment. The resulting new segment will be placed in the user's working directory under the name entryname.pt, where entryname is the entry name portion of pathname.

Notes

When writing an on-line document for use with peruse_text, the writer should pay particular attention to the formatting of the document title and the topic headings. A sample outline of a typical document is given at the end of this description.

1) Title of Document

The first line must be the title of the document. When peruse_text is first invoked, or when the call or return requests are issued, peruse_text searches for the first "new line" character, then prints the line ended by this "new line" character.

2) Topic Headings

Each topic in a document is introduced by a labeled title similar to a topic heading in an outline. These topic headings consist of the ASCII character \006, immediately followed by a label (see Label Syntax below), followed by a space or spaces and the title of the topic. The complete topic heading must be written on one line.

3) Label Syntax

Each topic heading in a peruse_text document begins with the ASCII character \006, and is immediately followed by a label. A label is composed of a sequence of from one to

eight integer elements separated by periods (e.g., 1., 3.1, 4.1.12). A label must contain at least one period. Each integer element can take on values between 0 and 511. The integer elements of topic headings must be in ascending sequence from the start of the document to the end of the document. However, they do not have to be consecutive integers; numbers can be skipped. If make_peruse_text finds errors in label format or sequence, it prints diagnostic messages and aborts the translation of the segment specified by pathname.

4) Recommended Style

It is recommended that topic headings should be followed by some associated text which describes the topic, and/or refer the user to other topics. If the inclusion of associated text does not seem appropriate, then the topic heading is probably unnecessary. The format of this text is at the discretion of the user, but should be terse.

First level topics should have single-level labels, i.e., 1., 2., not 1.0, 2.0, etc. Topic headings (excluding the label) and the document title should usually be all capital letters. Numbers are permitted in both.

Each topic heading that is subordinate to another topic should be indented one more space than its immediately superior topic heading. This can be achieved by inserting one or more blank characters before the ASCII \006 character.

To prevent excessive printing at the terminal, no single topic should be divided into more than five subtopics at the next subordinate level. In addition, the number of lines of text following a topic heading should be on the order of six.

Lines should be no more than 72 characters long, so that they may fit on one line of all commonly used terminals.

Example

```
                FILE OUTPUT, CONSOLE OUTPUT (6-2-71)

1.              FILE OUTPUT COMMAND
 1.1            USAGE
 1.3            DEFAULT
 1.4            FEATURES AND CONSTRAINTS
 1.5            EXAMPLE
2.              CONSOLE OUTPUT COMMAND
 2.1            USAGE
 2.4            FEATURES AND CONSTRAINTS
 2.5            EXAMPLE
```


(END)*

Name: memo

The memo command makes possible the use of Multics as an interactive notebook and reminder list containing memos. It 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 will be printed on the console. Memos can also be set which 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 which, rather than reminding him to do something at a certain time, actually performs the action itself. Finally the user can specify that the memo is to be repeated at regular intervals.

In the default case memo maintains its information in a segment Username.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, and additional information telling whether the memo is to be repeated or not and whether 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 will pass it to memo as a single argument.

Usage

memo -control_arg- -opt1- ... -optn- -memo_text-

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 then if nothing else appears on the command line mature memos are printed or executed, otherwise the rest of the line is used to set a memo.

-pathname memopath -pn memopath	where memopath is the path name of a memo segment to be used. The entry name of the segment must end in the suffix .memo, and if the segment does not exist, memo attempts to create it. If the user does not specify the suffix .memo, it is assumed.
-list, -ls	memos selected by the optional arguments are printed in full detail, including their maturity times, text, and information about the optional arguments, opti, used when the memos were set. No memos are executed.
-print, -pr	the text of all memos selected by the optional arguments, opti, are printed. No memos are executed.
-delete, -dl	all memos selected by the optional arguments, opti, are deleted.
-off	suppresses all memo alarms. This control argument must not be followed by any optional arguments, opti.
-on	enables the setting of memo alarms. This control argument must not be followed by any optional arguments, opti.
-brief	suppresses the message "No memos" if none are found. This control argument must not be followed by any optional arguments, opti.
2) opti	is one of the following optional arguments. Note that 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	is used when selecting memos. If any numbers are used to specify which memos are to be selected then only those memos which match one of the numbers are selected.

-date memo_time
-dt memo_time

where memo_time is a time in a form suitable for input to the subroutine convert_date_to_binary_ (see the MPM subroutine write-up for convert_date_to_binary.) memo_time is truncated to midnight preceding the date in which memo_time falls. If used while setting a memo then the trucated memo_time 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 memo_time are selected.

-time memo_time
-tm memo_time

where memo_time is a time 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 memo_time is not truncated.

-alarm, -al

if a memo is being set, this specifies that the memo is to be an alarm. When mature it will be printed or executed immediately (or as soon as alarms are enabled) and then deleted. If memos are being selected, this argument selects only those memos which are alarms.

-repeat interval
-rp interval

where interval is the interval (>=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_.

-invisible, iv

used only when setting a memo, this optional argument specifies that the memo will never be mature and will never be printed during a normal memo print.

-call

used only when setting a memo, this argument specifies that the memo is to be passed to the command processor as a command.

-match string1 .. stringn
where stringi is a character string. Memos containing substrings matching all of the stringi's are selected. The remainder of the line is interpreted as the set of the strings to be matched. The maximum number of strings which may 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 following argument must be the path name of a memo segment which is to be used. If a memo segment is specified by this means it will continue to be used for the duration of the user's process, unless changed again by the -pathname control argument. If the segment with path name memopath 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 may be used to specify the type of memo being set, and the time it will 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 which mature before the specified date or time are printed. Only those memos are printed which meet all of the specifications set by the optional arguments.

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" may be given to re-enable alarms after they have been turned off, or it may be used at login time to enable alarms without printing or executing other mature memos. Memo alarms are enabled by any use of memo except "memo -off".

Examples

In the following sequence of memo examples, input typed by the user is marked by an arrow (->). 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 only mature memos, then all memos maturing before a specified date. Finally, the only mature memo is deleted, and its successful deletion is demonstrated.

```
->memo
  memo:  Creating >udd>xproj>Jones>Jones.memo.

->memo
  No memos.

->memo get bookshelves
->memo -date 5/23/73 -repeat 2weeks -alarm Staff meeting at two.
->memo -call -date 6/1/73 -repeat 1month list -dtm -rev
->memo -time thu9am -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 -dtm -rev (call, "1month")
   4) Thu 05/17/73 0900 Weekly report due Friday. ("1week")

->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.
```

 (END)

Command
Standard Service System
4/3/72

Name: move, mv

The move command causes a designated segment (and its access control list (ACL), and multiple names, if any) to be moved to a new position in the storage system hierarchy.

Usage

move path1_1_ path2_1_ ... path1_n_ path2_n_ -ca-

1) path1_i_ is the path name of the nondirectory storage system segment to be moved.

2) path2_i_ is the path name to which path1_i_ is to be moved.

3) ca may be the following control argument:

-brief, -bf 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 may be used.

If path2_i_ already exists, the user will be interrogated as to whether he wishes it to be deleted.

The user's mode with respect to the directory portion of path2_i_ must include execute, write, and append access.

Example

move alpha >udd>Multics>Doe>= >udd>Multics>Doe>beta b

causes segment alpha to be moved from the current working directory to the directory >udd>Multics>Doe, with the name alpha, and segment beta to be moved from the directory >udd>Multics>Doe to the current working directory, with the name b.

(END)

Command
Standard Service System
6/23/71

Name: movequota, mq

The movequota command allows a user to move all or part of a quota between two directories (one immediately inferior to the other).

Usage

movequota pathname1 quota_change1 ... pathnamen quota_changen

1) pathnamei — is the pathname of a directory branch. The quota change will take place between this branch and its parent directory. "-wd" may be given to specify the working directory. The star convention may not be used.

2) quota_changei — is the number of pages to be subtracted from the parent directory quota and added to the quota on pathnamei. If this number is negative, the number of pages will be added to the parent directory quota and subtracted from the quota on pathnamei

Notes

The user must have "write" permission on both the directory specified by pathnamei and its parent directory.

After the change, the quota must be greater than or equal to the number of pages used in pathnamei unless the change would make the quota zero.

If the change would make the quota on pathnamei zero, there must be no immediately inferior directories with nonzero quota. When the quota is changed to zero, the pages used and the page-time product for pathnamei is then reflected up to the superior directory.

Example

```
movequota >udd>Multics 1000 >udd>Multics>Doe -50
```

will add 1000 pages to the quota on >udd>Multics and subtract 1000 pages from the quota on >udd. It will then subtract 50 pages from the quota on >udd>Multics>Doe and add 50 pages to the quota on >udd>Multics.

(END)

Command
Development System
09/23/70

Name: names

Entry: names$move

This command moves all the "extra" names from one multiply-named directory entry to another for any number of entry pairs. The name used to designate the segment is not moved; all others are moved.

Usage

names$move from_path1 to_path1 from_path2 to_path2 ... from_pathN to_pathN

1) from_pathi is the pathname of the segment whose names are to be moved.

2) to_pathi is the pathname of the segment to which names are to be moved.

Entry: names$copy

This entry is similar to names$move but leaves the names on the segment which originally has them. Note that in this case the names cannot be copied to a segment in the same directory because that would attempt to duplicate names in the directory. All of the names on the original segment are copied.

Usage

names$copy from_path1 to_path1 ... from_pathN to_pathN

arguments as described above.

(END)

Command
Standard Service System
02/12/71

Name: new_proc

The new_proc command creates a new process for the user, leaving him at command level in the same working directory he logged into. Although not implemented in the initial release of Multics, the process from which the command was invoked will in later versions be available for debugging.

The new_proc command is typically used to refresh static storage, possibly because a runaway program has overwritten the static storage area. Since dynamically snapped links are placed in the static storage area, they must be resnapped in the new process.

Note

If the login working directory contains a segment named start_up.ec, new_proc will cause the command

```
exec_com start_up  new_proc
```

to be automatically issued in the new process. This feature may be used to initialize per-process static variables.

Usage

new_proc

(END)

Command
Standard Service System
5/10/72

Name: page_trace, pgt

This command prints a recent history of page faults and other system events within the calling process.

Usage

page_trace -count- --long-

1) count causes the last count of system events (mostly page faults) recorded for the calling process to be printed. If count is not specified, then all the entries in the system trace list for the calling process will be printed. Currently, there is room for approximately 350 entries in the system trace list.

2) -long, -lg causes full path names to printed, where appropriate. The default is to print only entry names.

Output

The first column of output describes the type of the trace entry. If nothing is printed in this column for an entry, 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 is the page number for entries where this is appropriate. The fourth column gives the segment number for entries where this is appropriate. The last column is the entry (or path) name of the segment for entries where this is 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

To perform useful tests to determine page fault activity and frequency, the prepaging mechanism must first be crippled for the calling process. This is because the prepage driving list is the same list as the page trace list. To turn prepaging off, the command

pre_page_off

may be issued. To turn prepaging on again, the command

pre_page_on

may be issued.

Note that since it is possible for segment numbers to be reused within a process, and since only segment numbers (not names or path names) are kept in the trace array, the entry names and path names associated with a trace entry may not be correct. In fact, the entry and path names 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 will 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.

(END)

Command
Development System
8/18/72

Name: peruse_text

The peruse_text command allows a user to extract information from a formatted segment with a minimum of printing at the terminal. Formatted segments consist of a labeled topic heading for each block of text. To write and prepare such a segment, refer to the MPM description of the make_peruse_text command.

The peruse_text command responds by printing the title line of the document.

Usage

peruse_text -control_arg- entryname

1) control_arg — is an optional argument which specifies, if present, that entryname is actually a path name. It can have as its value either -pathname or -pn.

2) entryname — refers to the segment entryname.pt in the specified directory. The suffix .pt need not be specified. If the -pathname control argument is used, the complete path name is specified by this argument. Otherwise, it is assumed that entryname.pt resides in the directory where all pt documents are kept, >documentation>pt (abbreviated as >doc>pt).

Label Syntax and Label Depth

Each topic in a document is introduced by a topic heading which begins with a label. A label is composed of a sequence of from one to eight integer elements separated by periods (e.g., 1., 3.1, 4.1.12). A label must contain at least one period. Each integer element can take on values between 0 and 511. The number of integer elements in a label is the depth of the label, ranging from one to eight. In the above examples of a label, the depths are 1, 2, and 3 respectively.

Subsystem Request List

The peruse_text command responds to requests to peruse topic headings and associated text. These requests are listed in the following table.

Request	Meaning	Function
t	table	print a table of the topic headings at one depth
p	print	print the specified range of topic headings with their associated text
i	index	print the specified range of topic headings
c	call	call from the segment currently being perused to peruse a specified segment
r	return	return to perusing the segment that was perused before the current segment
"string"	find	print all topic headings where the specified character string appears as part of the topic heading or the associated text
q	quit	exit peruse_text to command level

Table Request

The table request prints a table of the topic headings at one depth. The format is:

```
t -label-
```

If no argument is given, all topic headings with a depth of one are printed. If a label is supplied, all topic headings one level deeper than and subordinate to the specified label are printed. The space between t and the argument is optional.

Whenever the table request causes the last topic heading at the level specified to be printed, peruse_text appends a line containing only the word END after the last topic heading. When a topic heading has no subordinate topic headings, an asterisk appears on the same line, following the topic heading.

See Examples below for use of the table request.

Index and Print Requests

The index request prints topic headings within the range specified. The print request prints topic headings and the associated text within the range specified. The format of these requests is:

```
request_name -begin_label- -end_label- -depth-
```

1) request_name is either i or p

2) begin_label specifies the first label to be indexed or printed. This argument is optional; its default value is the current label (that is,

the value of the label last tabled, indexed, or printed).

3) -end_label — specifies the last label to be indexed or printed. It must be preceded by a hyphen. It is optional and has as a default value the value of begin_label.

4) depth — specifies the depth of topic headings and text to be indexed or printed. That is, it specifies the maximum number of integer elements in labels to be indexed or printed. It is optional and takes the maximum of the depths of begin_label and -end_label as its default value.

Spacing in the index and print request is not significant before the begin_ label and -end_label arguments. However, one or more spaces must appear before the depth argument when it is present.

As in the table request, a line consisting of the word END is printed following the last topic heading at a given level. Requests for indexing or printing which do not involve the last topic heading produce

no such line. However, unlike the table request, no asterisk appears on the line with topic headings having no inferiors.

Note that the depth specified (or assumed by default) may be larger than the depth of -end_label. In this case, printing of output will continue past the topic heading specified by -end_label until all topic headings inferior to that one and having a depth within the specified range are printed.

See Examples below for various ways of using the index and print requests.

Use of Asterisk in an Argument

An asterisk (*) has special meaning in arguments to the table, index, and print requests. It is taken as a shorthand notation for the maximum possible value of the place it occupies. Thus, an asterisk is equivalent to 511 when used as an integer element in a label argument, and is equivalent to 8 when used as a depth argument.

Note that the use of asterisk in the table request is nearly always meaningless since very few documents will have an integer

element of a label equal to 511. (Recall that the table request says to print all topic heading immediately inferior to a specifed label.) Similarly, the use of asterisk in the begin_label argument of the index and print requests is usually meaningless. Its usefulness is in the -end_label and depth arguments of the index and print requests where it effectively specifies that the request should be performed until the end of the document is reached and should go to the maximum label depth present in the document.

Default Request Names

The table, print, and index requests are the only ones concerned with the structure of the document being perused. The peruse_text command remembers which of the three was last used, and supplies it whenever the request name is omitted from a request. The default request name is thus the last request (of those three) used for the document being perused. Upon initial entry to each document, the default request is index.

Find Request

The peruse_text command accepts any quoted string (including spaces) as a request. The find request causes peruse_text to search the segment being perused for all occurrences of the quoted string and to print the labeled topic headings wherever the quoted string occurs in the title or associated text. The format of the find request is:

```
"string"
```

The value of the quoted string may be any ASCII string that might be part of the contents of the segment currently being perused. The ASCII string must be enclosed in quotation marks. The PL/I quoting convention is used if quotation marks are to be specified within the quoted string.

Call and Return Requests

The call request is used to change from perusing the current segment to perusing another segment. The return request is used to resume perusing the previous segment.

The format of the call request is.

c -control_arg- entryname

Arguments are the same as those for peruse_text. The title line of the called segment is printed immediately following the call.

The format of the return request is.

r

The title line of the segment returned to and the topic heading last referenced before the call request are printed upon return. The default request name is set to t, p, or i, whichever was the default at the time of the call request.

Quit Request

The quit request may be used to exit peruse_text to Multics command level. The format of the quit request is:

q

Program Interrupt

The user may interrupt peruse_text by pressing the QUIT button, especially during the printing of unwanted output. To immediately return to peruse_text, where peruse_text waits for a subsequent request, type the command program_interrupt (pi).

Examples

The following series of requests was performed (in the order shown) on the pt segment for the peruse_text command. A short explanation of the feature being demonstrated by each request precedes the request. The line typed by the user is marked by a small arrow (->) at the left. The rest was printed by peruse_text. The print request was slighted in the examples due to the excessive space required. However, examples of the index request cover the print request format adequately.

1) Obtain a full table of contents of the document for reference in later examples. Incidentally this shows the index request with three arguments, and the asterisk in arguments.

```
->i 1. -*. *
  1.          PREFACE
   1.1        FUNDAMENTALS OF PERUSE_TEXT
   1.2        TO READ THE REST OF THIS DOCUMENT
  2.          PERUSE_TEXT (pt) COMMAND
   2.1        USAGE
   2.2        OPTION
   2.3        DEFAULTS
   2.4        FEATURES AND CONSTRAINTS
    2.4.1     LABELS
    2.4.2     TABLE (OF CONTENTS), INDEX AND PRINT REQUESTS
    2.4.3     THE CALL AND RETURN REQUESTS
    2.4.4     THE FIND REQUEST
    2.4.5     THE QUIT REQUEST
    2.4.6     USE OF ASTERISK
    2.4.7     DEFAULTS FOR ARGUMENTS
    2.4.8     DEFAULT FOR REQUEST NAMES
    2.4.9     PROGRAM INTERRUPT
    2.4.10    DOCUMENT FORMAT
    2.4.11    "COMPILATION" OF DOCUMENTS
   2.5        EXAMPLES
  END.
```

2) Table request with no arguments.

```
->t
  1.          PREFACE
  2.          PERUSE_TEXT (pt) COMMAND
  END.
```

3) Default request name (i.e., the table request is assumed).

```
->2.
   2.1        USAGE *
   2.2        OPTION *
   2.3        DEFAULTS *
   2.4        FEATURES AND CONSTRAINTS
   2.5        EXAMPLES *
  END.
```

4) Index request with no arguments.

```
->i
   2.5        EXAMPLES
  END.
```

5) Index request with begin_label argument only.

```
->i 2.4.2
    2.4.2       TABLE (OF CONTENTS), INDEX AND PRINT REQUESTS
```

6) Index request with begin_label and -end_label arguments. Note that no space appears before -end_label.

```
->i 1.-2.
  1.            PREFACE
  2.            PERUSE_TEXT (pt) COMMAND
  END.
```

7) Index request with all arguments. All optional spaces are omitted.

```
->i1.-2. 2
  1.            PREFACE
   1.1          FUNDAMENTALS OF PERUSE_TEXT
   1.2          TO READ THE REST OF THIS DOCUMENT
  2.            PERUSE_TEXT (pt) COMMAND
   2.1          USAGE
   2.2          OPTION
   2.3          DEFAULTS
   2.4          FEATURES AND CONSTRAINTS
   2.5          EXAMPLES
  END.
```

8) Index request (assumed by default) with the -end_label argument missing and an asterisk for the depth argument.

```
->1. *
  1.            PREFACE
   1.1          FUNDAMENTALS OF PERUSE_TEXT
   1.2          TO READ THE REST OF THIS DOCUMENT
```

9) Print request with no arguments.

```
->p
   1.2          TO READ THE REST OF THIS DOCUMENT
```

Use the requests described in Section 1.1 to read the rest of this document.

10) Find requests.

```
  ->"request"
     1.1        FUNDAMENTALS OF PERUSE_TEXT
     1.2        TO READ THE REST OF THIS DOCUMENT
    2.          PERUSE_TEXT (pt) COMMAND
     2.4        FEATURES AND CONSTRAINTS
      2.4.2     TABLE (OF CONTENTS), INDEX, AND PRINT REQUESTS
      2.4.3     THE CALL AND RETURN REQUESTS
      2.4.4     THE FIND REQUEST
      2.4.6     USE OF ASTERISK
      2.4.7     DEFAULTS FOR ARGUMENTS
      2.4.8     DEFAULT FOR REQUEST NAMES
      2.4.9     PROGRAM INTERRUPT
     2.5        EXAMPLES

  ->"REQUEST"
      2.4.2     TABLE (OF CONTENTS), INDEX, AND PRINT REQUESTS
      2.4.3     THE CALL AND RETURN REQUESTS
      2.4.4     THE FIND REQUEST
      2.4.5     THE QUIT REQUEST
      2.4.8     DEFAULT FOR REQUEST NAMES
```

11) Call request, and index request showing initial default request name.

```
  ->c ec
    EXEC_COM (5-17-72)  SSS

  ->1.1 -1.5
     1.1        USAGE
     1.3        DEFAULTS
     1.4        FEATURES AND CONSTRAINTS
     1.5        EXAMPLES
    END.
```

12) Return request showing that the current label is restored.

```
  ->r
                PERUSE_TEXT ( 5-17-72)
     1.2        TO READ THE REST OF THIS DOCUMENT
```

Summary of Requests

1) Data Requests

t	Print first-level table of contents.
t label	Print topic headings at next level subordinate to label.

i	Print current topic heading.
i begin_label	Print topic heading associated with begin_label.
i begin_label -end_label	Print topic headings from begin_label to end_label to the maximum depth of the two labels.
i begin_label -end_label depth	Print topic headings from begin_label to end_label to the depth specified.
p	Print current topic heading and text.
p begin_label	Print topic heading and text associated with begin_label.
p begin_label -end_label	Print topic headings and text from begin_label to end_label to the maximum depth of the two labels.
p begin_label -end_label depth	Print topic headings and text from begin_label to end_label to the depth specified.
"string"	Print topic headings where the ASCII string appears in the topic heading or text.

2) Control Requests

c name	Call the peruse_text segment specified (in a special system directory).
c -pathname name	Call the peruse_text segment specified by complete pathname.
r	Return to the peruse_text segment from which the current one was called.
q	Return to Multics command level.

(END)

Name: p11, v2p11

The p11 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.

Usage

p11 pathname -control_arg1- ... -control_argn-

1) pathname — is the path name of a PL/I source segment that is to be translated by the PL/I compiler. If the source segment name does not have a suffix of .p11, then one is assumed.

2) control_argi — can be chosen from the following list of control arguments:

-source, -sc — produces a line-numbered printable ASCII listing of the program. The default is no listing.

-symbols, -sb — lists the source program as above and all the names declared in the program with their attributes. The default is no symbols.

-map — lists the source program and symbols as above, 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 on-line. The default is no map.

-list, -ls — lists the source programs and symbols as for the -symbols control argument, followed by an assembly-like listing of the compiled object program. Note that use of the -list control argument significantly increases compilation time and should be avoided whenever possible by using the -map control argument. The default is no list.

-brief, -bf — causes error messages written into the stream "error_output" contain only an error number, statement identification, and, when appropriate, the identifier or constant in error. In the normal, non-brief mode, an explanatory message of one or more sentences is also written.

-severity*i*, -sv*i* — causes error messages whose severity is less than *i* (where *i* is 1, 2, 3, or 4; e.g., severity3) to not be written into the "error_output" stream although all errors are written into the listing. The default value for *i* is 1.

-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. Use of this control argument adds 5-10% 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 used 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 lables for use by symbolic debuggers. The table appears as 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 print_profile command can be used to print the execution counts. See the MPM command write-up of the print_profile command.

The following control arguments are available, but are probably not of interest to the normal 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 — After compilation, this control argument prints 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 typed after a compilation.

Further information on the above control arguments is contained under the headings Error Diagnostics and Listing.

Notes

A normal compilation produces an object segment, segname, and leaves it in the user's working directory. If segname existed previously in the directory, its access control list (ACL) is saved and given to the new copy of segname. 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 active when the object segment is created.

The user's control arguments control the absence or presence of the listing segment for segname.pl1 and the contents of that

listing. If created, the listing segment is named segname.list. The ACL is as described for the object segment except that the user is given "rwa" access to it when newly created. Previous copies of segname and (if the list option is on) segname.list are replaced by the new segments created by the compilation.

Note that because of the Multics standard that restricts the length of segment names, a PL/I source segment name cannot be longer than 24 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:

Severity Level	Meaning
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 right 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.
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" stream and returns to its caller.

Error messages are written into the stream "error_output" 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 set 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 had been specified, the user would see 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 no explanatory text appears with the first instance of a info, the user should use the help command to consult >documentation>info>pl1.status.info. This situation arises when a new error is defined before the segment containing the message text is updated.

If the -list control argument has been specified, the error messages are also written into the listing segment. They appear, sorted by line number, after the listing of the source program. Because of an implementation restriction, no more than 100 messages are printed in the listing.

The pl1 command issues a warning if any variables have been declared by context or implication. This warning does not appear in the listing segment.

Listing

The listing created by PL/I 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 which are:

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, precision, or level; attributes such as "initial", "array", "internal", "external", "aligned" and, "unaligned"; and a cross-reference list. The symbol listing is followed by the error messages.

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. Opcode, base-register, and modifier mnemonics are printed alongside 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.

Reference

1) The MULTICS PL/1 Language. A semiformal definition of the Multics PL/1 language.

 (END)

Name: pl1_abs, pa, v2pl1_abs, v2pa

This command submits an absentee request to perform PL/I compilations using the Version 2 PL/I compiler. 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 segnamei.list if it exists, and dprints and deletes segnamei.list. If the -output_file control argument is not specified, an output segment, segname.absout, is created in the user's working directory (if more than one segname is specified, the first is used). If the segment to be compiled cannot be found, no absentee request is submitted.

Usage

pl1_abs segname1 ... segnamen -pl1_args- -pl1_abs_args-

1) segnamei — is the path name of a segment to be compiled.

2) pl1_args — can be one or more nonobsolete control arguments accepted by the PL/I compiler and described in pl1. (See the write-up in the MPM.) Control arguments must begin with a minus sign (-).

3) pl1_abs_args — can 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 <= 3). The default queue is 3. segnamei.list is also dprinted in queue n.

-copy n, -cp n — spcecifies the number of copies (n <= 4) of segnamei.list to be dprinted. The default is 1.

-hold — specifies that pl1_abs should not dprint or delete segnamei.list.

-output_file f, -of f — specifies that absentee output is to go to segment f where f is a path name.

Notes

Control arguments and segment names can be mixed freely and can appear anywhere on the command line after the command. All control arguments apply to all segment names. An unrecognizable control argument causes the absentee request not to be submitted.

Unpredictable results can occur if two absentee requests are submitted which 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 links that need to be snapped when setting up a process and when invoking the compiler need be snapped only once.

 (END)

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.

Usage

print path -begin- -end-

1) path is the path name of the segment to be printed.

2) begin is the line number from which to begin printing and is optional. It it is not specified, printing starts on the first line of this segment. If begin is not specified, then end may not be specified either.

3) end is the line number to end printing; it is optional and if not specified, printing ends with the last line of the segment.

Notes

If neither begin nor end is supplied, a short identifying header will precede the printing of the segment. This header is suppressed whenever begin is nonnull. See Examples below.

The command assumes that "new line" characters are appropriately embedded in the text. Output is written on the I/O output stream "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.

 (END)

Command
Standard Service System
7/28/71

Name: print_attach_table, pat

This command prints on the user's console the I/O stream name associations created by attach calls in the user's current ring.

Usage

print_attach_table -stream_name1- ... -stream_namen-

1) stream_namei is the name of a stream about which information is to be printed.

Notes

The type and identifying names of the devices associated with the indicated stream names will be printed. If no arguments are specified, the information for all streams currently attached will be printed.

(END)

Name: print_bind_map, pbm

The print_bind_map command displays all or parts of the bind map of an object segment generated by version number 4 of the binder or subsequent versions. Note: if the object segment was bound by the old bindarchive command or by earlier versions of the binder, it will ask the user to print the appropriate map segment.

Usage

print_bind_map path -comp1- ... -compn- -control_args-

1) path — is the path name of a bound object segment.

2) compi — are the optional names of one or more components of this bound object. Only the lines corresponding to these components will be displayed. A component name must contain one or more non-numeric characters. If it is purely numerical, it is assumed to be an octal offset within the bound segment and the line corresponding to the component residing at that offset will be displayed. A numerical component name may be specified by preceding it with the control argument "-name" ("-nm").

3) control_args — may be any of the following:

-long, -lg — in addition to the components' relocation values (displayed in the default brief mode), the command also displays their compilation time and their source language.

-name, -nm — the following compi argument is the purely numerical name of a component segment.

-no_header, -nhe — the command will omit all headers, printing only lines concerning the components themselves.

Note

If no component names are specified, the entire bind map is displayed.

 (END)

Command
Standard Service System
11/11/70

Name: print_dartmouth_library, pdl

This command prints the pathname of the Multics directory being used as the current user library for the Dartmouth subsystem. The user library is searched whenever a Dartmouth Basic program produces a program name ending in "***".

Usage

print_dartmouth_library

(END)

Command
Standard Service System
02/16/71

Name: print_default_wdir, pdwd

The print_default_wdir command causes the current default working directory's name to be printed at the console.

Usage

print_default_wdir

Note

See also change_wdir and change_default_wdir in the MPM.

(END)

Command
Standard Service System
7/28/71

Name: print_link_info, pli

The print_link_info command prints selected items of information for the specified object segments.

Usage

print_link_info path1 ... pathn option1 ... optionn

1) pathi — is the pathname of the object segment to be displayed.

2) optioni — is chosen from the following list of options. It may appear anywhere on the command line and applies to all pathnames. If no option is specified, all of them are assumed by default. In this case, information describing the segment's actual file system location (tree name) and the circumstances of its creation (creation time, generator name, etc.) is also displayed.

-length, -ln — displays the object segment sections' lengths.

-entry, -et — is a listing of the object segment's external definitions, giving their symbolic names and their relative addresses within the segment.

-link, -lk — is an alphabetically sorted listing of all the external symbols referenced by this object segment.

(END)

Command
Development System
02/12/71

Name: print_linkage_usage, plu

This command prints out all linkage segment usage for a process. This information is useful for debugging purposes or for analysis of a process' use of its linkage segments.

Every procedure segment and each data segment that has definitions has a linkage section associated with it.

Usage

print_linkage_usage

Notes

The information printed for each linkage block is the name of the corresponding segment, the segment number, the offset, and the length of the linkage block. The printout is in order by the segment number and the offset of the linkage block. The command prints the position and size of each block within a combined linkage segment that is not used for linkage.

(END)

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 startup.ec segment see the MPM Reference Guide section, Protocol for Logging In.

Usage

print_motd

Notes

The current segment >system_control_1>message_of_the_day is compared with the segment User.motd (where User is the user's name) found in the user's home directory. All newly inserted or modified lines are printed on the console, and the user's copy is updated for use the next time print_motd is invoked.

If the segment User.motd does not exist, print_motd will attempt to create it, print the current message of the day, and initialize User.motd.

 (END)

Command
Development System
6/23/71

Name: print_search_rules, psr

The print_search_rules command causes the printing of the currently operating search rules in the user output stream.

Usage

print_search_rules

Notes

See also set_search_rules and set_search_directories in the MPM.

(END)

Command
Standard Service System
02/11/71

Name: print_wdir, pwd

The print_wdir command prints out the name of the current working directory on the user's console.

Usage

print_wdir

(END)

Command
11/5/73

Name: profile

The profile control argument of the pl1 and fortran commands 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 source line as well as a counter that starts out as zero. Each statement in the program is modified to start with an instruction to add one to the counter associated with the statement. The profile command allows the user to print and reset these counters.

Usage

profile name1 ... namen -ctl_arg1- ... -ctl_argn-

1) namei — is the pathname or reference name of a program whose counters are to be printed or reset.

2) ctl_argi — is selected from the following list. Control arguments apply to all programs whose names appear in the command line.

-print, -pr — causes profile to 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 jumps into pl1_operators_. Note that each instruction and each jump into pl1_operators_ count as only one unit.
5. the names of all operators in pl1_operators_ used by this statement.

The total cost for all statements is printed at the end.

-brief, -bf — causes profile to omit from the statement list statements that have never been

executed.

-long, -lg — causes profile to include the statement list statements that have never been executed.

-reset, -rs — causes profile to reset to zero all counters in the specified program.

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, the output from the profile command is:

LINE	STM	COUNT	COST	PROGRAM
				example
7	1	1	1	
8	1	1	8	
9	1	27	351 + 54	(set_cs cp_cs)
10	1	27	54	
12	1	1	14 + 1	(call_ext_out_desc)
13	1	1	0 + 1	(return)
TOTAL			428 + 56	

 (END)

Name: program_interrupt, pi

The program_interrupt command allows the users of editors, subsystems and other interactive programs to interrupt those programs and re-enter them at known places.

When the user wants to interrupt a program he presses the quit button and types program_interrupt or pi after receiving a ready message. If the program thus interrupted is not prepared to accept the interrupt, the system will print a message of the form "no active handler for program_interrupt". Otherwise the interrupt is accepted and what happens next depends on the particular program that was interrupted.

Usage

program_interrupt

Note

To make use of the program interrupt facility, a program or subsystem must establish a condition handler for the condition "program_interrupt". When the user invokes the program_interrupt command, the handler established by the program or subsystem is invoked. For a discussion of conditions see the MPM Reference Guide sections, the Multics Condition Mechanism, and List of System Conditions and Default Handlers.

Example

The edm command has a handler for the "program_interrupt" condition which, when it is entered, stops whatever the editor is doing and looks for a request from the user's terminal. Thus, a user of edm who inadvertantly typed "p100" (to print 100 lines) could kill this printout by pressing the quit button and then typing program_interrupt.

 (END)

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-

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_stream stream_name -os stream_name	directs output from the progress command to be printed on the I/O stream, stream_name. The default stream 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. The default is -cput 10.
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.

3) pt is the percentage of total real time that the command was executing.

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:

8) pft is the number of page faults per second of virtual CPU time for the execution of the entire command.

Example

```
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)
```

 (END)

Name: qedx, qx

The qedx context editor can be used to create and edit ASCII segments in Multics. The editor is based on the QED editor as implemented by K. L. Thompson of Bell Telephone Laboratories. qedx is basically a subset of QED and is designed to provide the user with much of the power of QED and performance similar to the simpler edm editor also described in this manual.

Usage

qedx -inst_path- -arg1- ... -argn-

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. argi — is an optional argument that is appended as a separate line to the buffer "args"; arg1 becomes the first line in the buffer, and argn becomes the last. Arguments are used in conjunction with a macro specified by instr_path. See Initalization of Macros below.

If inst_path is provided, the editor executes the qedx requests contained in the specified segment and then interrogates the user's terminal for further requests. If inst_path is omitted, the editor immediatesly interrogates the user's terminal for the first 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 write-up. For the moment, we will restrict our discussion to the most straightforward use of the editor.

Once the qedx command 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, which allows the user to enter new ASCII text from his terminal until an appropriate escape 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.

   1a) Enter ASCII text lines into the buffer from the terminal.

   1b) Leave input mode by typing the appropriate escape 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 and 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.

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 and exit from the editor.

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.

Editor Requests

In the list given below, editor requests are divided into three categories: input requests, basic edit requests, and extended edit requests. The 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 are, in general, a little more difficult to learn to use and the discussion of these requests is postponed until later in this write-up. Note that the letter 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.

1. 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.

2. Basic Edit Requests

delete (d)	Delete specified line or lines from the buffer.
print (p)	Print specified line or lines on the terminal.
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.

3. Extended Edit Requests

execute (e)	Pass remainder of request line to the Multics command processor (i.e., escape to other commands).
line number (=)	Print line number of specified line.
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 (b)	Switch to specified buffer (i.e., change base of all subsequent editor operations to specified buffer).
move (m)	Move specified line or lines into a specified auxiliary buffer.
status (x)	Print on the terminal a summary of the status of all buffers currently in use.

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 with 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 line number;

2. Addressing relative to the "current" line;

3. Addressing by context.

In addition, a line address can be formed using a combination of the above techniques.

1. Addressing by Line Number

Each line in the buffer can be addressed by a decimal number 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 relocated 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.

2. Addressing Relative to the Current Line

The qedx editor maintains the notion of a "current" line that is addressable by using the character "." (period) to represent the address of the current line. 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 number 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 "." with a series of signed decimal numbers (e.g., .5+5-3 is interpreted as .+7).

3. 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 used as an address performs in a manner similar to the edm locate (l) request. 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 new line 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 a line containing "in" followed by any two characters followed by "to".
/in.*to/	Matches a line containing "in" and "to" in that order.
/¬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 (new line character only).

The special meanings of "/", "*", "$", "¬", and "." within a regular expression can be removed by preceding the special character with ¢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 (e.g., //). 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".

Note that the two uses of "." and "$" (as line numbers and as special characters in regular expressions) are distinguished by context.

4. 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 a line number is to appear in an address, it must be the first component of the address.

2. A 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.

3. A regular expression can follow an address specified relative to the current line number. For example, the address .-8/abc/ starts the search from 8 lines preceding the current line.

4. 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/ from 10 lines preceding 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/.

5. 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:

A1,A2

The pair of addresses specifies the series of lines starting with the line addressed by the address A1 through the line addressed by A2 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 point to the line addressed by A1 before the evaluation of A2 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 A1. As a further example, the address pair

```
/abc/;.+10
```

is equivalent to the address pair

```
/abc/,/abc/+10
```

6. 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.

6. "// undefined" - A null regular expression has been used and no previously typed regular expression is available.

Use of the Editor

1. 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 use 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 most editor requests

   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 most editor requests

$<request> for read requests

2. The 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 after the last line deleted.

3. 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 new line 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)

4. 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, i.e., 13 cannot be written as 1 3 (which 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.

5. Comments

The quotation mark character is reserved as the comment delimiter and is actually implemented as an editor request, the effect of which is to ignore the remainder of the request line. If the quotation mark is preceded by an address, the value of "." is set to that address.

6. The Locate Request

If an address is followed by a new line character, the value of "." is set to the addressed line and the line is printed on the user's terminal. For example, the request line

```
/¬start/
```

locates a line beginning with "start", sets the value of "." and prints the line.

7. 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 users using the qedx editor for the first time.

If the user inadvertently requests a large amount of terminal output from the editor and wishes to abort the output without abandoning the edit, he can hit the quit button on his terminal, and, after the quit response, he can re-enter the editor by invoking the program_interrupt (pi) command. 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 hitting the quit button, 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.

8. 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 new line 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 character of a new line. The ¢f escape sequence can be followed immediately with more editor requests on the same line. 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 before typing another request. Otherwise, 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 at the last line input from the terminal.

The special meaning of any of the escape sequences used by qedx (e.g., ¢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.

Input Requests

1. 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: Before

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

Request Sequence

```
     2a                  /x=/a
     q=r;      or        q=r;
     ¢f                  ¢f
```

After

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

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

2. 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 Before

```
     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
```

After

```
     a:  procedure;
         s=t;
         u=v;
"."->    w=z;
         end a;
```

3. 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

Before

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

Request Sequence

```
2i                  /x=/i
q=r;        or      q=r;
¢f                  ¢f
```

After

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

Note: 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 write-up under extended editor functions.

1. 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: Before

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

Request Sequence

```
     3,4d    or     /q=/,/s=/d
```

After

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

2. 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 last line to be printed).

Example: Contents of Buffer

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

Request

```
     2,4p     or      /x=/,/s=/p
```

Console Output

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

Note: The output from the print request can be aborted with the use of the quit button and program_interrupt command.

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.

Special Note: The quit request must be followed immediately by a new line character.

4. 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 new line character.

Default: r PATH is taken to mean $r PATH

Value of ".": Set to the last line read from the file.

Example: Before

```
     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;
```

After

```
     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.

5. 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.)

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:

```
Before:     The quick brown sox
Request:    s/sox/fox/
After:      The quick brown fox

Before:     xyzindex=q;
Request:    s/index/(&)/
After:      xyz(index)=q;

Before:     x=y
Request:    s/$/;/
After:      x=y;
```

6. Write (w)

The write request is used to write the addressed line or set of lines into a specified segment. If the segment does not already exist, a new segment is created with the specified name.

Format: ADR1,ADR2w -PATH-

PATH is the pathname of the segment the contents of which are to be replaced by the addressed lines in the buffer. The pathname can be preceded by any number of spaces and must be followed immediately by a new line character. If PATH is omitted, 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 or if a pathname (either the same one or a different one) is used with a read or write request a second time.

Default: w PATH is taken to mean 1,$w PATH

Value of ".": Unchanged

Example: Request

```
2,4w sam.pl1
```

Result

The second through fourth lines of the buffer replace the contents of the segment sam.pl1 in the user's working directory.

Extended Use of the Editor

The editor requests discussed up to this point comprise a basic subset sufficient for most applications. A user learning to use qedx for the first time might be well advised to stop at this point.

The editor requests remaining to be described are divided into two groups: those that require a knowledge of auxiliary buffers and those that do not. Discussion of the former group of requests is delayed until the section on auxiliary buffers. Discussion of the latter group of requests proceeds below.

1. 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 following the execute request in the request line are passed to the Multics command processor. The execute request can be followed by any legal Multics command line. However, due to a temporary restriction, the user should not invoke edm, bsys, or qedx while in qedx.

Format: e <command line>

Value of ".": Unchanged

Example: The request line

```
e print alpha.pl1
```

can be used to print a segment in the user's working directory.

The request line

```
e list; mail
```

lists the user's working directory and prints his mailbox (if any).

Note: If the user wishes to abort a command line issued with the execute request, he can hit quit and invoke the program_interrupt command to abort the command line and restore control to qedx.

2. Print Line Number (=)

There 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: Contents of Buffer

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

Request

```
/q;/=
```

Response

```
3
```

3. Global (g)

The global request is used in conjunction with some other request (e.g., print, delete, print line number). That request is to operate 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

Default: gX/REGEXP/ is taken to mean 1,$gX/REGEXP/

Value of ".": Set to ADR2 of request

Note: The character immediately following the request X is taken to be the regular expression delimiter and can be any character not appearing in REGEXP.

Example: Before

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

Request

```
1,$gd/q/
```

After

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

4. Exclude (v)

The exclude request is also used in conjunction with another request (e.g., print, delete, print line number). That request is to operate only on those lines addressed by the exclude request that do not 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

Default: vX/REGEXP/ is taken to mean 1,$vX/REGEXP/

Value of ".": Set to ADR2 of request

Note: The character immediately following the request X is taken to be the regular expression delimiter and can be any character not appearing in REGEXP.

Example: Before

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

Request

```
1,$v=/q/
```

Response

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 are usually enclosed in parentheses; for example, the buffer name Fred is typed as (Fred). However, for historical reasons, a buffer name consisting of a single character can be typed with or without the enclosing parentheses (e.g., "x" is taken to be "(x)").

1. The Change Buffer Request (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).

2. The Move Request (m)

The move request is used to move one or more lines from the current buffer to a specified auxiliary 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 first line after last line moved in current buffer. Set to line 0 in the specified auxiliary buffer.

Example:

```
Before:  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)

After:   Current Buffer          Buffer B

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

3. The Buffer Status Request (x)

The buffer status request is used to print on the user's terminal 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.

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)
 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.

4. 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 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.

¢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.

5. 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 move the text to be moved into an auxiliary buffer with a move request. For example, the request

```
1,5m(temp)
```

moves line 1 through 5 of the current buffer into the auxiliary buffer temp. 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 following the ¢b escape sequence is correct since it can be expected that the last line in buffer temp is terminated by a new line character that precedes the ¢f after the ¢b(temp) is expanded.

6. Defining Editor Cliches

Another common use for buffers is for the definition of frequently used editing sequences or "cliches". 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, the contents of buffer new contains the necessary editor requests and literal text to append four lines of text at any point in the current buffer.

Example: Contents of buffer new

```
a
if code ¬=0 then do;
   call error (code);
   return;
   end;
¢f
.-4,.1p
```

Usage

```
ADR¢b(new)
```

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.

7. 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 a pseudo-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: Contents of buffer read

```
e ioa_ Type
$a
¢r¢f
¢b(test)
¢b(read)
```

Contents of buffer test

```
s/¬¢c.$//
d
e ioa_ "Give me a segment name."
w ¢r
q
```

Explanation of the read buffer

1. The first request in read 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 under 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.

8. 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 multi-buffer 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 buffers read and test are initialized with append requests. Notice that all escape sequences to be placed into a buffer as literal text must be preceded by a ¢c escape sequence. Thus, the second line to be input to buffer "read" is input as

```
¢c¢r¢c¢f
```

to produce the following line in the target buffer.

```
¢r¢f
```

In addition to the above, the qedx editor can be invoked with more than one argument. Thus,

```
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 buffer exec and buffer args become:

exec	args
r ¢b(args)	path

and the request ¢b(exec) causes the segment with the pathname path to be read into buffer 0. At that point the editor 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 buffers exec and args 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 occurence of x the character y, write out the segment path and quit, returning to command level.

Notes

Since the name of the segment to be read in appears on the command line, this feature allows users to use abbreviations (see the write-up 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 may well be the result.

Command
Standard Service System
7/01/71

Name: ready, rdy

The ready command types out an up-to-date ready message giving the time of day, the processor time, and the page faults since the last ready message was typed.

Usage

ready

Note

See the MPM commands ready_on and ready_off.

(END)

Command
Standard Service System
7/01/71

Name: ready_off, rdf

The ready_off command allows the user to turn off the ready message typed on the console after the processing of each command line. The automatic typing is suspended until a ready_on command is given.

Usage

ready_off

Note

See the MPM commands ready and ready_on.

(END)

Command
Standard Service System
6/28/71

Name: ready_on, rdn

The ready_on command causes the ready message to be automatically typed on the console after each command line has been processed. The automatic printing is in effect until a ready_off command is called.

Usage

ready_on

Note

See the MPM commands ready and ready_off.

(END)

Command
Standard Service System
7/01/71

Name: release, rl

The release command causes the stack history which was preserved by a previous hold command to be released. That is, the Multics stack will be returned to a point immediately prior to the stack frame of the command which was being executed when the quit signal or unclaimed signal which led to the hold command occurred.

Usage

release -opt-

1) opt is an optional control argument which if "-all" or "-a" causes the stack history preserved by all previous hold commands (which have not already been released) to be released.

(END)

Name: rename, rn

The rename command replaces a specified segment, directory, or link name by a specified new name, without affecting any other names the entry may have.

Usage

rename path1 name1 ... pathn namen

1) pathi specifies the old name which is to be replaced; it may be a path name or an entry name.

2) namei specifies the new name which replaces the storage system entry name portion of pathi.

Notes

The star and equals conventions may be used.

The user's access mode with respect to the directory specified by pathi must contain the modify attribute; if the star convention is employed, the mode must also contain the status attribute.

The entry name namei must be unique in the directory specified by pathi; if a namei already exists in the directory, the user is interrogated as to whether he wishes the other entry which has that name to be deleted.

Example

rename alpha beta >sample_dir>gamma delta

causes alpha, in the user's working directory, to be renamed beta, and gamma, in the directory >sample_dir, to be renamed delta.

 (END)

Name: reorder_archive

This command provides a convenient way for ordering the contents of an archive segment, eliminating the need to extract the entire contents of an archive and then replace them in the order desired. This command will place specified components at the beginning of the archive, leaving any unspecified components in their original order at the end of the archive.

Usage

reorder_archive -ca1- ... path1 ... -can- ... -pathn-

1) cai may be chosen from the following list:

-ci if the command is to be driven from console input. (This is the default.)

-fi if the command is to be driven from a driving list.

2) pathi is the full path name of the archive segment to be reordered; the user need not supply the ".archive" suffix.

Notes

If no control arguments are specified, the default -ci (console input) is assumed.

When the command is invoked with the console input (-ci) control argument or with no control arguments, it will type "input for archive_name" where archive_name is the name of the archive file to be reordered. Component names are then input in the order desired, separated by carriage returns. The character "." terminates input. The character string ".*" causes the command to type back a "*". This feature can be used to make sure there are no input errors before typing ".". The character string ".q" will cause the command to terminate without reordering the archive.

The driving list (-fi control argument) must have the name "name.order" where "name.archive" is the name of the archive segment to be reordered. The order segment must be in the working directory. It consists of a list of component names in the order desired, separated by carriage returns. No "." is necessary to terminate the list. Any errors in the list (name not found in the archive file, name duplication) will cause the command to terminate without altering the archive.

The procedure creates a temporary segment named "ra_temp_.archive" in the user's process directory. This temporary segment is created once per process, and is truncated after it is copied into the directory specified by path_i. If the command cannot copy the temporary segment, it will attempt to save it and rename it to the name of the archive specified. There is an interval of time, while the command is copying the reordered archive into the user's directory, when a quit, not followed by a start, will result in the loss of the archive. Therefore, quits should be used judiciously.

The reorder_archive command will not operate upon archive segments containing more than 1000 components.

 (END)

Command
Development System
9/30/71

Name: reprint_error, re

This command allows the reprinting of an error message produced by a fault and saved by a "hold" request. The mode of the error message may be specified.

Usage

reprint_error -options-

1) options — may be in any order and are chosen from the following list of options:

-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 option may appear only once per command line.

-all, -a — prints messages corresponding to all existing sets of fault information.

-brief, -bf — prints the short form of the message.

-long, -lg — prints the long form of the message.

Notes

The default form of the message is the normal mode 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.

Examples

The following example illustrates some ways the reprint_error command might be used.

```
simf

Error while executing in ring 0:
Improper access by find_command_$find_command_|574
(>system_library_1>bound_command_loop_)
referencing >udd>m>Smith>dw>simf|202
(offset is relative to base of segment)
```

```
hold

setacl simf re

simf

Error:  Attempt by >udd>m>Smith>dw>simf|20
to reference through null pointer

hold

deh_test1$gate_error

Error:  Gate error by >udd>m>Smith>dw>deh_test1$gate_error|320
referencing >system_library_1>hcs_$initiate
Number of arguments expected = 7; number supplied = 3.

hold

reprint_error -all -brief
level 1:
Error:  gate_error
level 2:
Error:  simfault_000001
level 3:
Error:  accessviolation while in ring 0

reprint_error -long -depth 3
level 3:
Error while executing in ring 0:
Improper access by >system_library_1>bound_sss_active|63
(offset is relative to base of segment)
referencing >udd>m>Smith>dw>simf|202

The entry into ring 0 was by
find_command_$find_command_|574
(>system_library_1>bound_command_loop_|14076)
referencing >system_library_1>hcs_$make_ptr
(accessviolation condition)

reprint_error
level 1:
Error:  Gate error by >udd>m>Smith>dw>deh_test1$gate_error|320
referencing >system_library_1>hcs_$initiate
Number of arguments expected = 7; number supplied = 3.
```

 (END)

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 use this command to obtain information about other users' resource usage.

Usage

resource_usage -control_arg-

1) control_arg — provides the optional feature of selecting variable portions of available resource usage information. The valid control arguments follow:

-total, -tt — prints only total dollar figures including the user's dollar limit stop and his month-to-date spending (see Notes below).

-brief, -bf — prints the information selected by the -total control argument, and as well precedes this information with a header and follows 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 which 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.)

If no control argument is specified, the default action results in the selection of slightly less extensive resource usage information than that which is printed by the -long control argument; namely, all dollar charges are printed but resource usage expressed as time is not printed.

Notes

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 as only shifts or queues with accrued charges will be listed.

If no dollar limit stop has been set by a user's project administrator, the resource usage report will indicate this by the printing of "open" as the dollar limit entry.

Example

```
resource_usage -long

Doe.Example  Report from 04/01/73 1014.7 to 04/10/73  1345.8

  Month-To-Date Charge: $  292.61;
  Resource Limit:       $ 1000.00;

     Interactive Usage: $ 254.31; 19 logins, 7 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
```

 (END)*

Command
Development System
8/15/72

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 will fit on the line; the line is then justified by inserting extra blanks to make an even right margin. Up to twenty lines of header and footer 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 beginning with a period interspersed within the text, but not appearing 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 later be hand drawn, a separate segment can be created which contains reminders as to where each symbol should be placed. The user may define variables and cause expressions to be evaluated; in combination with the ability to refer to (and sometimes modify) variables connected with the workings of the runoff command, the user has extensive control over the processing of his text.

Usage

runoff pathname1 ... pathnamen -control_args-

1) pathnamei is the pathname of an input segment or multi-segment file (MSF) named entryname.runoff; however, the .runoff suffix may be supplied in the entryname. If more than one pathname is 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 may be chosen from the following list. Any control argument specified anywhere in the command invocation applies to all segments, and control arguments may be intermixed arbitrarily with segment names. Control arguments must be preceded by a minus sign.

-ball n, -bl n Output is converted to a form suitable for an n typeball on a unit equipped with a Selectric (R) 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 When this control argument is used, certain key characters in the output will by flagged 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 will appear with each flagged line, and reminder characters, enclosed by color-shift characters, will be substituted for the key characters. The default set of key and reminder characters corresponds to those unavailable with a 963 typeball, as follows:

left square bracket	<
right square bracket	>
left brace	(
right brace	)
tilde	t
grave accent	'

The key and reminder characters may 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 Output is prepared compatible with the device specified. This is usually used when the output is stored in a segment to be printed elsewhere. Suitable devices are consoles 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 will prevail.

If neither -device nor -ball was specified, the default device type is that from which the user is logged in; any unrecognized type is assumed to support the entire ASCII character set.

-from n, -fm n Printing starts 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_, which the user supplies, will be invoked to perform hyphenation when the next word to be output will not fit in the space remaining in a line (see Hyphenation Procedure Calling Sequence near the end of this document). Otherwise, no attempt will be made to hyphenate words.

-indent n, -in n When this control argument is used, the output will be indented n spaces from the left margin (default indentation is 0 except for "-device 202" (the default for -segment), which is 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
Page breaks in the output are suppressed.

-number, -nb Source line numbers will be printed in the left margin of the output; minimum indentation of 10 is forced.

-page n, -pg n This control argument changes the initial page number to n. All subsequent pages are similarly renumbered. Note that if the control word .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
The argument arg is assigned as a string to the internal variable "Parameter".

-pass n — The source segments will be processed n times to permit proper evaluation of expressions containing symbols which are defined at a subsequent point in the input. No output is produced until the last pass.

-segment, -sm — When this control argument is used, the output will be directed to the segment or MSF 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 will be approximately centered on the page when dprinted.

-stop, -sp — The runoff command will wait for a carriage return from the user before beginning typing and after each page of output.

-to n — Printing ends after the page numbered n.

-wait, -wt — The runoff command will wait 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 may take a parameter which is separated from the control word by one or more spaces. Lines which 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 will 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 NL character.

Terminology

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. Note that 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.

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.

A break insures that the text that follows will not be run together with the text before the break. The previous line is printed out as is, without padding.

Vertical spacing within the body of the text is controlled by the three commands .ss, .ds, and .ms. Single spacing, which is the default, is set by .ss, double spacing is set by .ds, and ".ms n" is used for multiple spacing. There are (n-1) blank lines between text lines for .ms.

A page eject insures that no text after the control line will be 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.

There are four margins on the page vertically. The first margin on the page is the number of blank lines above the first header, and is set by the .m1 control line. The second, set by .m2, concerns the number of lines between the last header and the first line of text. The third is between the last line of text and the first footer, set by .m3. The fourth is below the last footer, set by .m4. The default for the first and fourth margins is four lines; for the second and third, the default is two lines.

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.

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 may have up to twenty headers and twenty 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 .he (.fo). They are set separately by using .eh, .oh, .ef, and .of.

A header/footer definition control line has two arguments, the line number (denoted in the control line descriptions as "#"), and the title.

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 non-blank character after the line number. This character is taken to be the delimiting character, and may 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 may be null. Justification and centering of a header or footer line are derived from the line length and identation in effect at the time of the definition of the header or footer, and will be used whenever that line is output, regardless of the values at the time of use.

Omitting the title in the control line cancels the header or footer with that number, including its space on the page. A blank line in the header or footer may be achieved by a title consisting entirely of four delimiting characters. Omitting both number and title cancels all headers or footers of the type specified.

Expressions and Expression Evaluation

An expression may 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.

Operators in order of precedence:

¬ (bit-wise negation), - (unary)
*, /, ¢ (remainder)
+, - (binary)
=, <, >, ≠, ≤, ≥
(comparison operators, yield -1 (true) or 0 (false))
& (bit-wise AND)
| (bit-wise OR), ≡ (bit-wise equivalence)

Parentheses may be used for grouping.

Blanks are ignored outside of constants.

Octal numbers consist of "#" followed by a sequence of octal digits.

String constants are surrounded by the double quote character; certain special characters are defined by multiple character sequences beginning with the character *, as follows:

** yields	asterisk-character
*"	double-quote-character
*b	backspace character
*n	new-line character
*t	horizontal tab character
*s	space character
*cnnn	character whose decimal value is nnn (1 to 3 digits)

Concatenation of strings is performed by the juxtaposition of the strings involved, in order, left to right.

For positive i, k,

```
string_expression(i)
```

and

```
string_expression(i, k)
```

are equivalent to the PL/I substr built-in function references

```
substr(string_expression, i)
```

and

```
substr(string_expression, i, k)
```

respectively.

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.

Evaluation of substrings takes place after any indicated concatenations; string operations have higher precedence than all the binary operations.

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 which accept an "n" or "±n" argument.

Definition and Substitution of Variables

Variables may be defined by the use of the .sr control line; their values may be retrieved thereafter by a symbolic reference. Names of the variables are composed of the upper- and lower-case 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 which is to be its value, either arithmetic or string. Variables which 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 .cc).

Substitution of variables may 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 will be found at the end of this document); these 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 re-define an internal variable that is not explicitly modifiable will merely make it inaccessible to the user, but will cause no harm to the operation of the command.

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 will be 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 will be four lines of top margin, the headers, two blank lines, and then the text. If footers are used, there will be two lines skipped after the text, footers printed, and four lines of bottom margin.

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

This section gives a description of each of the control words which may 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 -
<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.

(blank line)
A blank line occurring in the text is treated as if it were a ".sp 1" control line.

.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 numeric variables 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. ".cc %" or .cc restores the percent sign as the special character.

.ce n Center: the next n text lines are centered. If n is missing, 1 is assumed. This control line implies ".ne n" (or ".ne 2n" if doublespacing) so that all lines centered will be on the same page. A break occurs.

.ch cd.. Characters: each occurrence of the character c will be replaced in the .chars segment 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 will not be flagged, and will occur 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 #. See the section entitled Terminology.

.eh # 'part1'part2'part3'
  Even header: this defines even page header line number #. See the section entitled Terminology.

.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 doublespacing) so that all equations will be on the same page. The format of the equations should be 'part1'part2'part3' just as in headers.

.ex text Execute: the remainder of the control line (text) is passed to the Multics command processor. Substitution of variables may 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 may be specified through the title in the .fh control line. This title is printed in the same manner as headers. 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. This is the default condition. Since right justification is also the default condition, getting a slightly even right margin without adjustment is accomplished by use of the .na control line. 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'
See the section entitled Terminology.

.fr c Footnote reset: this control line controls footnote numbering according to the argument c. Permitted values of this argument 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 will be ignored. If a footnote occurring near the bottom of a page will not fit on the page, as much as necessary will be 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 will be terminated and the offending line printed at the top of the succeeding 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'
```

See the section entitled Terminology.

.if f <expression>

Insert file: the segment with pathname f.runoff is inserted into the text at the point of the ".if f" request. The inserted segment may contain both text and control lines. No break occurs. The effect is as if the control line were replaced by the segment. Inserts may be nested to a maximum depth of 30. If a second argument is provided, it will be evaluated in the same fashion as the expression in .sr, and its value and type will be associated with the identifier "Parameter"; otherwise the value of "Parameter" remains unchanged (or undefined) (prior values of "Parameter" are 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. The default for n is 65 both initially and if n is omitted in the .ll control line. 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. The default is four lines. This control line is equivalent to

```
.m1 ±n
.m4 ±n
```

.mp ±n — Multiple pages: format the output text so that it prints on every nth page. This control line is valid only for output intended for the bulk printer. The default value is 1.

.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 above the first header is set to n lines, or changed by n if n is signed. The default is four lines.

.m2 ±n — Margin 2: the number of blank lines printed after the last header and before the first line of text is set to n, or changed by n if n is signed. The default is two lines.

.m3 ±n — Margin 3: the number of blank lines printed after the last line of text and before the first footer is set to n, or changed by n if n is signed. The default is two lines.

.m4 ±n — Margin 4: the margin below the last footer is set to n lines, or changed by n if n is signed. The default is four lines.

.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. The default value is 1.

.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 #. See the section entitled Terminology.

.oh # 'part1'part2'part3' — Odd header: this defines odd page header line number #. See the section entitled Terminology.

.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.

.pa ±n — Page: the current line is finished as is (ie 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.

.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, and the space is below the headers. This option may be used to allow for pictures and diagrams. If several .pi control lines are used, 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.

.pl ±n — Page length: the page length is set to n lines. The default is 66 lines. 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 stream "user_input"; this input line is then processed as if it had been encountered instead of the .rd control line. Thus it may be either a text line or a control line; a break occurs only if the replacement line is a control line which causes a break, or a text line beginning with one or more spaces, or a blank line.

.ro Roman numerals: numeric-to-string conversions required by an explicit .ur control line (q.v.) or in titles or equations will result in roman numerals in the evaluated text. This may be reset to arabic numerals (the default) by use of the .ar control line.

.rt Return: cease processing characters from the current input segment.

.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 argument may be used to leave out a page number for a figure. If n is not given 1 is assumed.

.sp n Space: 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. This control line causes a break.

.sr name <expression>
Set reference: associate the value of <expression> with the identifier name. The type of name will be set to the type of <expression> (either numeric or string); if the expression is not provided, or cannot be properly evaluated, a diagnostic message will be printed. name may be either a user-defined identifier or one of the built-in symbols which the user may 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 will translate 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 will not be split across a line, nor have padding inserted within it.)

.ts n Test: process the next input line if the value of n does not equal zero (false). The default value is 1.

.ty xxx — Type: write xxx (ie the rest of the control line) onto the stream "error_output". Substitution of variables may occur if the first or second character of xxx is "%".

.un n — Undent: the next output line is indented n spaces less than the current indentation. Adjustment, if in effect, will occur 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 (ie, the next output line will begin at the current left margin). This control line causes a break.

.ur text — Use reference: the remainder of the .ur control line (text) will be 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 may be another control line, including possibly another .ur).

.wt — Wait: read one line from the stream "user_input", and discard it (cf .rd).

.* — 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 and Control Line Formats

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 MSF "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.

Control line formats

The following conventions are used to specify arguments of control words:

c	character
cd	character pair
exp	expression (either numeric or string)
#	integer constant
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
(blank line)	yes		Equivalent to ".sp 1"
.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 file 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 xxx	no		Call command processor with "xxx"
.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 next 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

Request	Break	Default	Meaning
.li n	no	n=1	Next n lines treated as text
.ll ±n	no	n=65	Set line length to n
.ma ±n	no	n=4	Top and bottom margins set to 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
.nf	yes	off	Nofill; break after each input line
.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 "user_input" 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 non-zero
.ty xxx	no		Write "xxx" onto the stream "error_output"
.un n	yes	left margin	Indent next text line n spaces less
.ur text	no		Substitute values of variables in "text", and re-scan the line.

Request	Break	Default	Meaning
.wt	no		Read one line of text from "user_input" and discard it (for synchronization with console)
.*	no		Comment line; ignored
.	no		Comment line; ignored, but included in .chars output

Built-in Symbols

Only those symbols marked yes in the Set column may 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 which 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	".chars" file is being created (-character)
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)

Symbol	Set	Value
Fr		Footnote counter reset switch
From	yes	Control argument value (-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, -start)
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 may 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);
```

1) string — is the text word which 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 space (to allow for the hyphen), and may 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)

(END)*

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 path1 ... pathn -rf_args- -ear_args- -rfa_args-

1) pathi — is an absolute or relative path name specifying the segment to be processed by the runoff command. It need not specify the ".runoff" suffix, which must appear in the actual segment name, however. If more than one path name 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. See the MPM write-up for runoff.

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. See the MPM write-up for enter_abs_request.

4) rfa_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 not be queued for printing or deleted. Each output segment is formatted for printing on an IBM 2741 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 path names 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 path names can be mixed freely in the command line. All control arguments apply to all path names. An unrecognizable control argument causes the absentee request not to be submitted.

The runoff_abs command expects each segment to be processed to have the suffix ".runoff", whereas early versions of the runoff command accepted segments without such a suffix. If any of the input text segments cannot be found, no absentee request is submitted.

 (END)

Name: safety_sw_off, ssf

This command turns off the safety switch of a directory or a segment, thus permitting the segment or directory to be deleted. See the MPM Reference Guide section, Segment, Directory and Link Attributes, for a description of the safety switch.

Usage

safety_sw_off pathname1 ... pathnamen

1) pathnamei is the path name of the segment or directory which should have its safety switch turned off. If it is "-wd" or "-working_directory" or omitted, then the working directory is assumed. The star convention may be used.

Examples

safety_sw_off test.pl1 check.fortran

will turn off the safety switch of the segments test.pl1 and check.fortran

ssf *.temp_dir

will turn off the safety switch of all directories and segments with a two component name which ends in temp_dir.

 (END)

Name: safety_sw_on, ssn

This command turns on the safety switch of a directory or a segment, thus preventing deletion of the segment or directory. See the MPM Reference Guide section, Segment, Directory and Link Attributes, for a description of the safety switch.

Usage

safety_sw_on pathname1 ... pathnamen

1) pathnamei is the path name of the segment or directory which should have its safety switch on. If it is "-wd" or "-working_directory" or omitted then the working directory is assumed. The star convention may be used.

Examples

safety_sw_on *.pl1

will turn on the safety switch of all segments found in the working directory with two component names ending in .pl1.

ssn

will turn on the safety switch in the working directory.

 (END)

Names: set_bit_count, sbc

This command sets a specified bit count on a specified segment entry, and changes the bit count author for that entry to be the user who invoked the command.

Usage

set_bit_count path1 count1 ... pathn countn

1) pathi is the path name of the segment whose bit count is to be set. If pathi is a link, the branch linked to will have its bit count set.

2) counti is the bit count, in decimal, desired for pathi.

Notes

Setting the bit count on a directory is permitted, but several system modules will then regard the directory as a multi-segment file.

The user must have modify access on the directory containing the segment for which the bit count is to be set.

 (END)

Command
2/13/73

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 128 characters. An expanded command line is one obtained after all active strings have been processed.

Usage

set_com_line -size-

1) size is the new maximum expanded command line size. If size is not specified, the line size is restored to its default of 128 characters.

Notes

The get_com_line command prints on the user's terminal the current value of the maximum size of expanded command lines.

For a discussion of the command language (including the treatment of active strings), see the MPM Reference Guide section, The Command Language.

 (END)

Command
Standard Service System
11/03/70

Name: set_dartmouth_library, sdl

This command allows the user to specify a directory to be searched before the Dartmouth system library is searched in reference to Basic library programs. The library is searched whenever *** is appended to a program name within a Basic program.

Usage

set_dartmouth_library -pathname-

1) pathname is the pathname of the directory to be used as the user's library. If no pathname is given, the library pathname is set to null and no user library is searched.

Note that if a program in the user's library has the same name as a system library routine, the user's version is the one used.

(END)

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 Initial ACL entry. A directory Initial ACL contains the ACL entries to be placed on directories added to the directory. For a discussion of Initial ACLs, see the MPM Reference Guide section, Access Control.

Usage

set_iacl_dir pathname mode1 acname1 ... moden acnamen -ca-

1) pathname — specifies the directory in which the directory Initial ACL should be changed. If it is "-wd" or "-working_directory" then the working directory's Initial ACL is assumed. If an entry for acnamei already exists, then its mode is changed to modei, otherwise acnamei with modei is added to the ACL. The star convention may be used.

2) modei — is the mode associated with acnamei. It may 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 acnamei, "n", "", or "null" should be used for modei.

3) acnamei — is an access control name which is permitted modei to pathname. If the last modei has no acnamei following it, the user's name and project are assumed. acnamei must be of the form person.project.tag. If one or more of the components is missing, then they are assumed to be "*". Any component missing on the left must be delimited by periods. The periods to the right may be omitted.

4) ca — may be the control argument -ring (-rg). It may appear anywhere on the line, except between a mode and its associated acname, and affects the whole line. If present it must be followed by a digit, where user's ring $\leq$ digit $\leq$ 7, which specifies which ring's Initial ACL should be affected. If the control argument is not given the user's ring is assumed.

Examples

```
set_iacl_dir listings sm *
```

will change the mode or add an entry to the directory Initial ACL in the directory listings with the mode "sm" being given to *.*.* (everyone.)

```
sid -wd sa Jones.Faculty
```

will add to the directory Initial ACL in the working directory an entry with mode "sa" for Jones.Faculty.* if that entry does not exist; otherwise it will change the mode of the Jones.Faculty.* entry to "sa."

 (END)

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 Initial ACL entry. A segment Initial ACL contains the ACL entries to be placed on segments added to the directory. For a discussion of Initial ACLs see the MPM Reference Guide section, Access Control.

Usage

set_iacl_seg pathname mode1 acname1 ... moden acnamen -ca-

1) pathname — specifies the directory in which the segment Initial ACL should be changed. If it is "-wd" or "-working_directory" then the working directory's Initial ACL is assumed. If an entry for acnamei already exists, then its mode is changed to modei; otherwise acnamei with modei is added to the Initial ACL. The star convention may be used.

2) modei — is the mode associated with acnamei. It may consist of any or all of the letters "rew" (read, execute, write.) To specifically deny access to acnamei, "n", "", or "null" should be used for modei.

3) acnamei — is an access control name which is permitted modei to pathname. If the last modei has no acnamei following it the user's name and project are assumed. acnamei must be of the form person.project.tag. If one or more of the components is missing, then they are assumed to be "*". Any components missing on the left must be delimited by periods. The periods to the right may be omitted.

4) ca — may be the control argument -ring (-rg). It may appear anywhere on the line except between a mode and its associated acname, and affects the whole line. If present it must be followed by a digit, where user's ring $\leq$ digit $\leq$ 7, which specifies which ring's Initial ACL should be affected. If the control argument is not given, then the user's ring is assumed.

Examples:

```
set_iacl_seg test rew *
```

will change the mode or add an entry to the segment Initial ACL in the directory test, with the mode rew being given to *.*.* (everyone.)

```
sis -wd re Jones.Faculty -rg 5
```

will add to the segment Initial ACL for ring 5 in the working directory, an entry with mode re for Jones.Faculty.* if that entry does not exist; otherwise it will change the mode of the Jones.Faculty.* entry to "re".

Command
Development System
6/30/72

Name: set_search_dirs, ssd

The set_search_dirs command allows users to insert search directories after the working directory in the default search rules.

Usage

set_search_dirs arg1 ... argn

1) argi are the pathnames of the directories to be searched.

Notes

The current maximum number of arguments is thirteen.

See also set_search_rules and print_search_rules in the MPM.

Searching is expensive in machine time, so the fewer directories searched, the better.

(END)

Name: set_search_rules, ssr

The set_search_rules command allows the user to set his search rules to suit his individual needs with only minor restrictions.

Usage

set_search_rules pathname

1) pathname is the pathname of a segment containing the ASCII representation of the search rules.

Notes

The allowed search rules are absolute pathnames of directories to be searched and the following key words:

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 rule per line. A maximum of 21 search rules is allowed. Leading and trailing blanks are allowed but embedded blanks are not allowed.

See also print_search_rules and set_search_directories in the MPM.

Warning: searching is expensive in machine time so the fewer directories searched the better.

(END)

Name: setacl, sa

This command adds entries to an Access Control List (ACL) or modifies the access mode in an existing ACL entry of either a segment or a directory. See the MPM Reference Guide section, Access Control, for a discussion of ACLs.

Usage

setacl pathname mode1 acname1 ... moden -acnamen-

1) pathname — specifies the segment or directory for which the ACL should be changed. If it is "-wd" or "-working_directory", then the working directory is assumed. If an entry for acnamei already exists, then its mode is changed to modei, otherwise acnamei with modei is added to the ACL. The star convention may be used.

2) modei — is the mode associated with acnamei. For directories it may consist of any or all of the letters "sma" (status, modify, append) with the requirement that if "m" is given, "s" must also be given. For segments it may consist of any or all of the letters "rew" (read, execute, write.) To specifically deny access to acnamei, "n", "", or "null" should be used for modei.

3) acnamei — is an access control name which is permitted modei to pathname. If the last modei has no acnamei following it, the user's name and project are assumed. acnamei must be of the form person.project.tag. If one or more of the components is missing, then they are assumed to be "*". Any component missing on the left must be delimited by periods. The periods to the right may be omitted.

Examples

```
setacl *.pl1 rew *
```

will change the mode or add an entry to the ACL of every segment in the working directory that has a two component name with a second component pl1, giving the mode "rew" to *.*.* (everyone.)

```
sa -wd sm Jones.Faculty
```

will add to the ACL of the working directory an entry with mode "sm" for Jones.Faculty.* if that entry does not exist; otherwise it will change the mode of the Jones.Faculty.* entry to "sm."

 (END)

Command
Development System
11/04/70

Name: sort_file, sf

The sort_file command may be used to sort the lines in an ASCII file in ascending order according to the ASCII collating sequence. The resorted file replaces the previous contents of the specified file.

Usage

sort_file path

1) path specifies the pathname of an ASCII file to be sorted line by line. The resorted contents of the file will replace the previous contents of the file. (The length of the file will, of course, remain unchanged.)

Notes

Lines of unequal length are compared by assuming the shorter line to be padded on the right (after the new-line character) with following blanks.

The lines of the original file are resorted using temporary segments in the process directory. The original file is not modified until the last moment.

(END)

Command
Standard Service System
4/20/72

Name: start, sr

The start command is employed after the quit button has been pressed in order to resume execution of the user's process from the point of interruption. It may also be used to resume execution after an unclaimed signal, provided that the condition which caused the unclaimed signal either is innocuous or has been corrected. It restores the attachments of user_input, user_output, and error_output, and the mode of user_i/o to what they were at the time of the interruption, unless the -no_restore control argument is given (see below).

Usage

start -control_argument-

1) control_argument is either -no_restore or -nr. If present, it indicates that the standard I/O attachments should not be restored.

Notes

This command may be issued immediately after a quit signal. It may also be issued later, but only if a hold command was given immediately after the quit signal and no subsequent release command was given.

If there is no suspended computation to re-start, the command prints the message "start ignored".

See also the MPM Reference Guide section on The Multics Command Language Environment.

(END)

Name: status, st

The status command prints selected detailed file status information about the storage system entry specified.

Usage

status path1 ... pathn -control_arg1- ... -control_argn-

1) pathi — is the path name of the segment, directory, multi-segment file, or link for which status information is desired. The default path name is the working directory which may also be specified by "-wd". The star convention may be used.

2) control_argi — is chosen from the following list of control arguments. The control arguments may appear anywhere on the line and are in effect for the whole line.

The control arguments for segments and directories are:

-all, -a — all relevant information returned by hcs_$status_long; i.e., the type of entry, names, unique id, date used, date modified, date branch modified, date dumped, author, bit count author (if different from author), device, bit count, records used, current blocks (if different from records used), max length in words (if type is segment), safety switch (if it is on), user's mode and ring brackets, and copy switch (if it is on);

-date, -dt — all the dates on the entry; i.e., date used, date modified, date branch modified, date dumped;

-name, -nm — all the names on the entry;

-mode, -md — the user's mode, ring brackets and safety switch (if it is on);

-device, -dv — the device id;

-length, -ln — the bit count, the number of records used, the current blocks (if different from records used) and the max length in words (if type is segment);

-author, -at — the author of the entry and the bit count author (if different from author);

-type, -tp — the type of entry (segment, directory multi-segment file, link).

If no control argument is specified, the following information is given for segments and directories: names, type, date used, date modified, date branch modified, bit count, records used, user's mode.

The control arguments for links are:

-all, -a — all relevant information return by hcs_$status_long, i.e., the path name of the entry linked to, names, unique id, date link modified, date dumped, and the author of the link;

-date, -dt — all the dates, i.e., date link modified, date dumped; the path name of the entry linked to;

-name, -nm — all the names on the link;

-author, -at — the author of the link;

-type, -tp — the type of entry (link) and the path name of the entry linked to.

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 will be ignored for links.

Notes

Any zero-valued dates will not be printed.

Directories that have been used to implement multi-segment files will be labelled as such

Examples

1) status >Federal>Adams -all

```
names:      Washington
            Test_lss
            Adams
type:                           directory
unique id:                      764576046673
date used:                      01/27/73 1459.0 est Wed
date modified:                  01/27/73 1459.0 est Wed
branch modified:                11/19/72 1542.6 est Thu
author:                         Hamilton.Multics.a
bit count author:               Dumper.SysDaemon.a
device:                         DSU-190
bit count:                      0
records used:                   6
safety sw:                      on
mode:                           rew
ring brackets:                  5, 5
```

2) status -type -mode -date newtest.*

```
     >States>Washington>newtest.pl1

type:                           segment
date used:                      01/26/73 2145.0 est Tue
date modified:                  01/13/73 1630.0 est Wed
branch modified:                01/13/73 1626.7 est Wed
date dumped:                    01/14/73 0305.4 est Thu
mode:                           rew
ring brackets:                  4, 5, 5

     >States>Washington>newtest.list

type:                           link
links to:                       Federal>Jefferson>newtest.list
date link modified:             01/26/73 2139.3 est Tue
```

 (END)

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. It is most useful when recompiling procedures so that the new version may be invoked with no linkage complications. Therefore it is called automatically by the Multics compilers. The user may also call this command directly in order to test various versions of a procedure. Generally, the links to a segment are not reset unless that segment has a linkage section. However, they are always reset for the terminate_refname (tmr) and terminate_single_refname (tmsr) entries.

Usage

terminate name1 ... namen

1) namei is the path name of a segment to be terminated.

Entry: terminate_segno, tms

This entry allows termination by segment number.

Usage

terminate_segno segno1 ... segnon

1) segnoi is the segment number (in octal) of a segment to be terminated.

Entry: terminate_refname, tmr

This entry allows termination by reference name. The segment itself is terminated, not merely the particular reference name specified.

Usage

terminate_refname name1 ... namen

1) namei is the reference name of a segment 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 will not be terminated.

Usage

terminate_single_refname name1 ... namen

1) namei is the reference name to be terminated.

Notes

Caution must be exercised when using these commands as one may unintentionally terminate 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.

 (END)

Command
Development System
8/10/72

Name: trace_stack, ts

The track_stack command prints a detailed explanation of the current process's stack history in reverse order (most recent frame first). For each stack frame, all available information about the procedure which 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 is printed.

trace_stack 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 and the source line in which it occurred if possible.

Usage

trace_stack -control_args-

1) control_args may be selected from the following:

-brief, -bf Supress listing of arguments and handlers.

-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 fault. If such a frame is found, tracing will proceed backward from that point; otherwise, a comment is printed and tracing beings with the stack frame preceding trace_stack.

If a machine-conditions frame is found, track_stack repeats the system error message describing the fault. Unless "brief" mode was selected, trace_stack also prints the source line and faulting instruction, and a listing of the machine registers at the time the fault occurred.

The command then performs a backward trace of the stack, for n frames if the "-depth n" argument was specified, or until the beginning of the stack is reached.

For each stack frame, trace_stack prints the offset of the frame, the condition name if a fault occurred in the frame, and the identification of the procedure which 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 will be printed also.

The trace_stack command then attempts to locate and print the source line associated with the last instruction executed in the procedure which owns the frame (which is either a call forward, or a line which encountered a fault). 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 will print a comment explaining why.

Next, trace_stack prints the machine instruction last executed by the procedure which owns the current frame. If the machine instruction is a call on a pl1 operator, trace_stack will also print the name of the operator. If the instruction is a procedure call, trace_stack will suppress the octal of the machine instruction and print the name of the procedure being called.

Unless the output mode is "brief", trace_stack will next list the arguments supplied to the procedure which owns the current frame, and list any enabled condition, default and clean-up handlers established in the frame.

If the output mode is "long", trace_stack will then print an octal dump of the stack frame, with eight words per line.

Example:

After a fault which reenters the user environment and reaches command level, a user might type

```
hold

trace_stack
```


(END)

Command
11/14/72

Name: truncate, tc

This command will truncate a segment to a specified length, and reset the bit count accordingly. It resets the bit count author for the storage system entry to be the user who invoked the command. The segment may be specified by path name or segment number.

Usage

truncate -control_arg- segid length

1) -control_arg- if present, must be -name or -nm, indicating that the following segid is in fact a path name, although it may look like a number.

2) segid is either a path name or an octal segment number. A path name 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 in words to which the segment is to be truncated. If no length argument is provided, zero will be assumed.

Notes

The length argument designates the length of the segment after truncation. Thus,

truncate alpha 50

will truncate all of the segment alpha except the first 50 words (i.e., words 0 to 47). The bit count of the segment will be set to the truncated length.

If the segment is already shorter than the specified length, its length will be unchanged, but the bit count will be reset to the specified length.

 (END)

Name: unlink, ul

The unlink command deletes the specified link entry. For a discussion of links see the MPM Reference Guide section, Segment, Directory and Link Attributes.

Usage

unlink path1 ... pathn

1) pathi specifies a storage system link entry to be deleted.

Notes

The user must have modify access in the directory containing the link.

The star convention may be used.

The delete, deleteforce and delete_dir commands may be used to delete segment and directory entries.

 (END)

Name: v5basic

The v5basic command invokes the BASIC compiler to translate a segment containing BASIC source code. If the compile option is not specified, the compiled code is then executed.

Usage

v5basic source_name -option1- ... -optionn-

1) source_name — is the path name of the segment to be translated. The characters .basic may or may not appear as part of the path name. They must appear, however, on the segment itself.

2) optioni — is selected from the following list of options. The options may appear in any order.

-time n, -tm n — specifies a time limit of n CPU seconds where n is an integer. When the time limit is exceeded, execution stops, and the user is asked if he would like to continue execution. If he answers yes, a new timer is set giving the user the same amount of time.

-compile — indicates to compile the program and produce an object segment rather than immediately executing the code. The compiled object segment is saved in the user's working directory with the characters .dobj appended in place of .basic. The object segment is not a Multics standard object segment and can only be executed using the basic_run command.

-library, -lb — indicates that the Dartmouth library is to be searched for the source segment. No other directory is searched.

Notes

This implementation of BASIC is described in BASIC, Fifth Edition, published in 1970 by the Kiewit Computation Center, Dartmouth College, in Hanover, New Hampshire.

The following is a list of differences between the Dartmouth and Multics implementations of BASIC:

1) The Multics storage system conventions differ from those at Dartmouth. Therefore, if a user refers to a segment as

```
20 file #1:"alpha"
```

Multics will search for a segment named alpha in the user's working directory. If alpha is not found, the directory is searched for alpha.basic. If this is not found, the segment alpha is created.

2) The number sign (#) must be entered with an escape character preceding it to avoid the Multics interpretation as an erase character. The upward arrow character is entered as a circumflex on Multics.

The current version of the BASIC compiler is a proprietary program of Dartmouth College. It has been made available to users of the M.I.T. Information Processing Center with the permission of Dartmouth College. The BASIC compiler may not be used at other computer installations without permission of Dartmouth College.

 (END)

Command
Development System
9/27/71

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 all directories inferior to the starting node. The pathname of every directory in which the command line is executed is printed onto the user's console. Control arguments are provided to modify the behavior of the command (see the list below).

Usage

walk_subtree pathname "command line" -option1- ... -optionn-

1) pathname — is the starting node. This must be the first argument. If it is -wd, the working directory is assumed.

2) "command line" — is the command line to be executed. Note that 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) optioni — is chosen from the following list of options. These control arguments can appear in any order following the command line.

-first n, -ft n — makes n the first level in the file 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 lowest level in the file system hierarchy at which the command line is to be executed. The default is -last 99999, i.e., all levels.

-brief, -bf — suppresses the printing of the names of the directories in which the command line is executed. (This is not a default option.)

-bottom_up, -bu — causes execution of the command line to commence at the last level and to proceed upwards through the file system

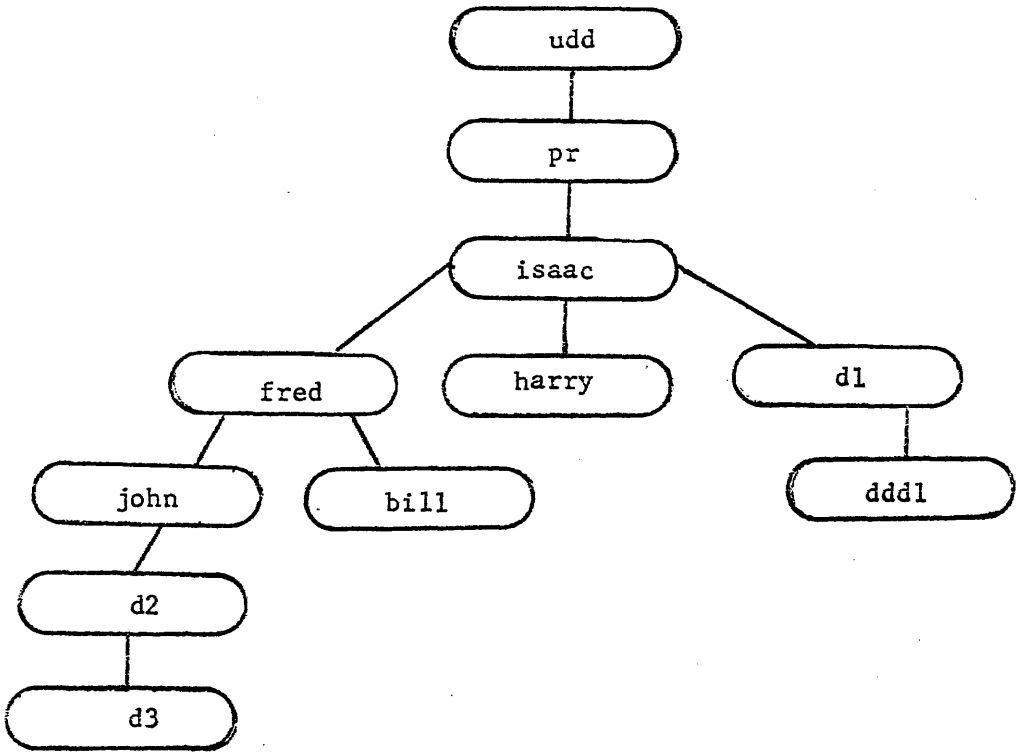
hierarchy until the first level is reached. In the default mode, execution begins at the highest (first) level and proceeds downward towards the lowest (last) level.

Notes

The walk_subtree command establishes a program_interrupt handler. If the user quits out of the walk_subtree command and immediately types pi (or program_interrupt), his working directory will be changed to the directory he was in when the walk_subtree command was typed.

Examples

Assume the following directory structure:



Assume that the user currently has working directory isaac.

1) walk_subtree >udd>pr>isaac "list *.list"

   will list all the segments with a second component of "list" in the directory isaac and all of its subdirectories.

2) walk_subtree >udd>pr>isaac "list *.list; dl *.list"

   will list and then delete all the segments with a second component name of "list" in the directory isaac and all of its subdirectories.

3) walk_subtree -wd "list *.list" -first 3

   executes the command line "list *.list" in the directories john, bill, ddd1, d2, d3.

4) walk_subtree >udd>pr>isaac "list *.list" -last 2

   executes the command line "list *.list" in the directories isaac, fred, harry, d1.

5) walk_subtree >udd>pr>isaac "list *.list" -last 4 -first 3

   executes the command line "list *.list" in the directories john, bill, ddd1, d2.

6) walk_subtree fred "setacl -wd rewa isaac.pr.*"

   executes the given command line first in the directory fred, then in john and in bill, then in d2, and then in d3.

7) walk_subtree fred "deleteacl -wd isaac" -bottom_up

   executes the given command line first in the directory d3, then in d2, then in john and in bill, and then in fred. Note that the -bottom_up option is essential in this case for the deleteacl command to succeed.

(END)*

Name: where, wh

The where command searches for a given reference name using the standard search rules and initiates the segment if found. It prints out the full path name of that segment, including its primary name. If the segment is not in the search path, an error message is printed. The segment will remain known to the process after the where command is invoked.

Usage

where name1 name2 ... namen

1) namei is a segment reference name of $\leq$ 32 characters.

Note

The primary name of a segment is the name which is first in the list of names on a storage system directory entry.

 (END)

Command
Standard Service System
8/21/72

Name: who

The who command determines the number, identification and partial 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, and the maximum load. (See the MPM write-up for how_many_users to print only the header.)

Usage

who -control_arg1- ... -control_argn- -arg1- ... -argn-

1) control_argi may 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, the time the system was brought up, and load information on absentee users.
-project, -pj	sorts the output by the project identification of each user.
-name, -nm	sorts the output by the name of each user.
-absentee, -as	lists only absentee users.
-brief, -bf	suppresses the printing of the header.

2) argi may be selected from the following list:

Name	lists only users with person name "Name".
.Proj	lists only users with project identification "Proj".
Name.Proj	lists only users with person name "Name" and project identification "Proj".

Notes

Absentee users are denoted by an asterisk (*) following "Name.Proj".

Up to twenty classes of selected users are permitted.

If the options -project or -name are omitted, the output is sorted on login time.

If an argi is specified, the header is suppressed even if the -long control argument is specified. The -long control argument will produce long information for each user listed.

Examples

1) Print default information.

```
who

Multics 17.6b, load 6.0/50.0; 5 users
Absentee users = 1/2

Backup.SysDaemon
IO.SysDaemon
Jones.Faculty
Doe.Work
Smith.Student*
```

2) Print long information for absentee users on the Student project (with no header).

```
who -absentee -long .Student

Absentee users = 1/2

08/21/72  0050.2 none 1.0  Smith.Student*
```

3) Print brief information for all users.

```
who -brief

Backup.SysDaemon
IO.SysDaemon
Jones.Faculty
Doe.Work
Smith.Student*
```


(END)

SUBROUTINES

This section contains, in alphabetic order, descriptions of all standard Multics subroutine calls. The user of this section will also want to refer to the Reference Guide section on the Multics Programming Environment, which contains a guide to the subroutines, organized by function.

The following conventions are used in subroutine descriptions:

1. An entry declaration, suitable for verbatim copying into a calling program, is provided. Using such a declaration is recommended practice, since it helps reduce errors.

2. Calling sequences are normally given for the PL/I language. Users of other languages should translate the sequences accordingly.

3. Following the description of each argument, the notation (Input) or (Output) indicates that the argument is passed to or comes from the subroutine, respectively.

4. I/O System Interface Modules are also alphabetically included in this section.

Note that subroutines can be distinguished from commands by name; generally, subroutines have names which end with a trailing underscore.

Subroutines not described in this section may possibly be listed in the Reference Guide sections on Obsolete Procedures or Internal Interfaces.

 (END)

Subroutine Call
7/5/73

Name: active_fnc_err_

The active_fnc_err_ subroutine is called by active functions when they detect unusual status conditions. It formats an error message (as described below) and then signals the condition active_function_error. The default handler for this condition prints the error message and then returns the user to command level. (See the MPM Reference Guide sections, Error Handling, and List of System Conditions and Default Handlers, for further information.)

Since this subroutine can be called with a varying number of arguments, it is not permissible to include a parameter attribute list in its declaration.

Usage

```
declare active_fnc_err_ entry options (variable);

call active_fnc_err_ (code, caller, control_string,
     arg1, ..., argn);
```

1) code — is the status code (fixed bin(35)) detected. (Input)

2) caller — is the name (char(*)) of the procedure calling active_fnc_err_. It can be either fixed or varying. (Input)

The remaining arguments are optional, as explained in Notes below.

3) control_string — is an ioa_ control string (char(*)). See the MPM write-up for the ioa_ subroutine. (Input)

4) argi — is an ioa_ format argument. See the MPM write-up for the ioa_ subroutine. (Input)

Notes

The error message prepared by active_fnc_err_ has the format "caller: system_message user_message". The system message is a particular message from error_table_ corresponding to the value of code. If code = 0, no system message is included. The user message is constructed by ioa_ from the control string and format arguments. If no control_string and format arguments are given, the user message is omitted.

 (END)

Subroutine Call
Development System
8/10/71

Name: adjust_bit_count_

This procedure performs the basic work of the adjust_bit_count command. This entry is called to find the last nonzero word or character of a segment and set the bit count accordingly.

Usage

```
declare adjust_bit_count_ entry (char(168) aligned,
        char(32) aligned, bit(1) aligned, fixed bin(24),
        fixed bin);

call adjust_bit_count_ (dn, en, char_sw, bit_count,
     code);
```

1) dn — is the directory pathname. (Input)

2) en — is the entry in the directory. (Input)

3) char_sw — is "0"b if adjustment is to be made to the last nonzero word; it is "1"b if to the last nonzero character. (Input)

4) bit_count — is the computed bit count for the segment. If the value is < 0, no attempt at computing the count was made (code will be nonzero). If the value is >= 0, the computed value is correct, regardless of whether the bit count could be set. (Output)

5) code — is 0 if the operation was successful. Any file system error code may be returned. (Output)

(END)

I/O System Interface Module
Development System
10/8/71

Name: broadcast_

The broadcast_ interface module, or broadcaster, is one means by which I/O system calls may fan out, i.e., a single I/O system call on a single stream can result in I/O operations being performed on several different devices. The broadcaster permits a single stream to be simultaneously attached to several other streams. Certain I/O system calls then issued to the first stream will result in similar calls to each of the object streams.

Note: Due to current limitations in the specifications of the I/O system, certain deficiencies exist in this release of the broadcaster. These deficiencies have to do with return arguments. The broadcaster performs I/O calls on several I/O streams and receives return values from each of these calls, e.g., status. However, the broadcaster itself has no way to return all of these return values to its caller. Currently, the broadcaster performs some mapping of these various values to determine the single value to be returned, and these mappings are indicated later in this document.

Usage

```
call ios_$attach (stream, "broadcast_", object_stream1, "",
     status);

call ios_$attach (stream, "broadcast_", object_stream2, "",
     status);
                          .
                          .
                          .

call ios_$attach (stream, "broadcast_", object_streamn, "",
     status);
```

For each object stream to be associated with the primary stream, an attach call, as indicated above, must be issued. All the various object streams will be simultaneously associated with the primary stream until detach calls are issued. Any I/O system calls listed below issued to the primary stream, except the attach and detach calls, will result in equivalent calls to each of the object streams.

I/O System Calls

The following I/O system calls are implemented by the broadcaster:

abort
attach
detach
resetwrite
write

The number of elements written, as returned by the write call, is the minimum of the number of elements written by each write call to an object stream.

Device Identification

Since the pseudo-device of the broadcast_ interface module is an object stream, any stream name is a permitted device identification. The object stream does not have to exist at attach time. However, any attempt to use a stream which broadcasts to a nonexistent object stream will result in an error status being returned.

Status

If any of the error codes in the status strings returned in I/O calls by the broadcaster to the object streams are nonzero, then one of these nonzero codes will be returned by the broadcaster. Otherwise, the error code portion of "status" returned by the broadcaster will be zero.

Modes

The broadcaster has no modes of its own; therefore I/O operations performed by the broadcaster on its object streams take on the modes of the individual streams.

Element Size

The broadcaster pays no attention to element size; therefore each object stream uses its own element size. Users must be careful that all object streams of a given broadcast stream have the same element size.

Synchronization

The broadcaster takes on the synchronization of its object streams. If all its object streams are write synchronous, the

broadcaster will also be write synchronous. If any of the object streams are in write asynchronous mode, then the broadcast stream will be in write asynchronous mode.

Detachment

The caller may specify that one or all of the object streams be detached. If the second argument to detach is a null string, all the object streams will be detached. If a particular object stream is specified as the second argument in the call to detach, only this stream will be detached. If a call to detach leaves no object stream associated with the primary stream, the primary stream will be deleted.

(END)

Subroutine Call
Standard Service System
02/16/71

Name: change_wdir_

The change_wdir_ subroutine changes the user's current working directory to the directory specified as its first argument.

Usage

```
declare change_wdir_ entry (char(168) aligned, fixed bin);

call change_wdir_ (directory_name, code);
```

1) directory_name is the pathname of the directory which will become the user's working directory. (Input)

2) code is a standard file system error code. See the MPM Reference Guide Section on "Miscellaneous Reference Data". (Output)

(END)

Name: check_star_name_

This procedure validates an entry name to insure that is has been formed according to the rules for constructing star names. These rules are given in the MPM Reference Guide section, Constructing and Interpreting Names. It also returns a status code that indicates whether the entry name contains asterisks or question marks, and whether it is a star name that matches every entry name.

Entry: check_star_name_$path

This entry point accepts an absolute path name as its input. It validates the final entry name in that path, as described above.

Usage

```
declare check_star_name_$path entry (char(*),
        fixed bin(35));

call check_star_name_$path (pathname, code);
```

1) pathname is the absolute path name whose final entry name is to be validated. Trailing spaces in the path name character string are ignored. (Input)

2) code is one of the following status codes: (Output)

0 the entry name is valid, and does not contain stars or question marks.

1 the entry name is valid, and does contain stars or question marks.

2 the entry name is valid, and is a star name that matches every entry name. This means that the entry name is either "**", or "*.**", or "**.*".

error_table_$badstar
the entry name is invalid. It violates one or more of the rules for constructing star names.

Entry: check_star_name_$entry

This entry point accepts, as input, the entry name to be validated.

Usage

```
declare check_star_name_$entry entry (char(*),
        fixed bin(35));

call check_star_name_$entry (entryname, code);
```

1) entryname is the entry name to be validated. Trailing spaces in the entry name character string are ignored. (Input)

2) code is as above. (Output)

Notes

Refer to the MPM write-up for the hcs_$star_ subroutine to see how to get a list of directory entries that match a given star name.

Refer to the MPM write-up for the match_star_name_ subroutine to see how to compare an entry name with a given star name.

 (END)

Subroutine Call
2/16/73

Name: clock_

The clock_ procedure reads the system clock. The time returned is a fixed bin(71) number equal to the number of microseconds since January 1, 1901, 0000 hrs GMT. It is suitable for input to the date_time_ subroutine which converts the time to an ASCII representation.

Usage

```
declare clock_ entry returns (fixed bin(71));

date_time = clock_ ();
```

1) date_time is the number of microseconds since January 1, 1901, 0000 hrs GMT. (Output)

Note

Since the "leap second" declared by the National Bureau of Standards on June 30, 1972, the value returned by clock_ understates the time since January 1, 1901, 0000 hrs GMT by one second. As a result, conversion routines which ignore the "leap second" will give correct answers for times since midnight of June 30, 1972, and will be one second high for times before that date.

 (END)

Subroutine Call
11/16/72

Name: com_err_

This is the principal error message printing subroutine. It should be called to report any unusual status condition, e.g., when a nonzero status code is returned.

Since this procedure can be called with a different number of arguments, it is not permissible to include a parameter attribute list in the declaration.

See also the MPM Reference Guide section, Strategies for Handling Unusual Occurrences.

Entry: com_err_

This procedure formats an error message (as described below) and then signals the condition command_error. The default handler for this condition simply returns control to com_err_ which then writes the error message on the stream "error_output".

Usage

```
declare com_err_ entry options (variable);

call com_err_ (code, caller, control_string, arg1 ... argn);
```

1) code — is a status code (fixed bin (35)) as returned from system entry points, etc. If code = 0, no system message is printed. (Input)

2) caller — is the name of the procedure (character(*)) calling com_err_. It may be either fixed or varying. (Input)

The remaining arguments are optional, as explained in the Notes below.

3) control_string — is an ioa_ control string (character(*)). See the MPM description of ioa_. (Input)

4) argi — is an ioa_ format argument. See the MPM description of ioa_. (Input)

Notes

The error message prepared by com_err_ has the format "caller: system_message user_message". The system message is a particular message from error_table_ corresponding to the value of code. If code = 0, no system message is included. The user

message is constructed by ioa_ from the control string and format arguments. If no control_string or format arguments are given, the user message is omitted.

Entry: com_err_$suppress_name

This is the recommended entry to use when the caller name and colon are not wanted because it still allows a meaningful caller name to be passed to the command_error condition handler. Otherwise, it is the same as the com_err_ entry.

Usage

```
declare com_err_$suppress_name entry options (variable);

call com_err_$suppress_name (code, caller, control_string,
     arg1 ... argn);
```

The description of the arguments is the same as for com_err_. The argument "caller" must not be null or blank because condition handlers for command_error need to know who signalled them.

Notes

If a nonzero code is provided which does not correspond to an error_table_ entry, the system message will be of the form "Code ddd not found in error_table_", where ddd is the decimal respresentation of code; if code is negative the message will be of the form "I/O Status ooo" where ooo is the 12-digit unsigned octal representation of code (this form is intended for use by I/O System device interface modules (DIMs) which wish to return hardware error status to their callers; the actual interpretation of the value is dependent on the physical device and the individual DIM).

 (END)

Subroutine Call
Development System
2/15/72

Name: command_query_

This subroutine is the standard system procedure invoked to ask a question and obtain an answer from the user. It formats the question (as described below) and then signals the condition command_question. See the MPM Reference Guide section, List of System Conditions and Default Handlers. The default handler for this condition simply returns control to command_query_ which writes the question on the stream user_i/o. It then reads the stream user_input to obtain the answer. Several options have been included in command_query_ to support the use of a more sophisticated handler for the command_question condition.

Since this procedure can be called with different numbers of arguments, it is not permissible to include a parameter attribute list in the declaration.

Usage

```
declare command_query_ entry options (variable);

call command_query_ (p, answer, caller, control_string,
     arg1 ... argn);
```

1) p — is a pointer to the structure below. (Input)

```
declare 1 query_info aligned,
          2 version fixed bin init(2),
          2 yes_or_no_sw bit(1) unaligned,
          2 suppress_name_sw bit(1) unaligned,
          2 status_code fixed bin(35),
          2 query_code fixed bin(35);
```

(1) version — is the version number of this structure. (Input)

(2) yes_or_no_sw — if "1"b, command_query_ will not return until a yes or no answer is read. (Input)

(3) suppress_name_sw — if "1"b, the name of the calling procedure will be omitted from the question. See Notes below. (Input)

(4) status_code	is the status code which prompted the question; otherwise it should be zero. (Input)
(5) query_code	is currently ignored. It is intended for use by specialized handlers for command_question. (Input)
2) answer	is the response (character(*) varying) read from user_input. Leading and trailing blanks plus the "new line" character have been removed. (Output)
3) caller	is the name (character(*)) of the calling procedure. It may be either varying or nonvarying. (Input)

The remaining arguments are optional as explained in the Notes.

4) control_string	is an ioa_ control string (character(*)). See the MPM description of ioa_. (Input)
5) argi	is an ioa_ format argument. See the MPM description of ioa_. (Input)

Notes

The question prepared by command_query_ has the format "caller: message". If suppress_name_sw is on, then the caller name will be omitted from the question. The message is constructed by ioa_ from the control string and format arguments. If no control string and therefore no format arguments are given, the message portion of the question is omitted.

(END)

Subroutine Call
Standard Service System
2/25/72

Name: condition_

This procedure establishes a handler for a condition in the calling block activation. If a handler for the specified condition is currently established in the calling block activation, it will be overridden.

A description of the condition mechanism is given in the MPM Reference Guide section on The Multics Condition Mechanism.

Usage

```
declare condition_ entry (char(*), entry);

call condition_ (name, handler);
```

1) name is the name of the condition for which the handler is to be established. (Input)

2) handler is the handler to be invoked when the condition is raised. (Input)

Notes

The condition names unclaimed_signal and cleanup are obsolete special condition names and should not be used.

The PL/I on statement and the condition_ subroutine must not be invoked during the same block activation in order to establish a handler for the same condition.

In order to explicitly revert a handler established by a call to condition_, the reversion_ (see the MPM subroutine) procedure must be called. The PL/I revert statement must not be used for this purpose.

In PL/I Version 2, when a call to condition_ appears within the scope of a begin block or internal procedure of a procedure, the no_quick_blocks option must be specified in the procedure statement of that procedure. The no_quick_blocks option is a nonstandard feature of the Multics PL/I language and, therefore, programs using it may not be transferable to other systems.

(END)

Subroutine Call
Development System
8/23/71

Name: convert_binary_integer_

This procedure performs conversion from binary integers to a character string representation either in the octal base or the decimal base. It contains entries to handle double precision integers.

The string representation is returned as an appropriate varying character string with no included blanks and an assumed decimal (binary) point at the right. If the argument is negative, the first character of the returned value will be a minus sign. Leading zeros will be omitted.

Entry: convert_binary_integer_$decimal_string

This entry converts a single precision binary integer to its decimal string representation.

Usage

```
declare convert_binary_integer_$decimal_string entry
        (fixed bin(35)) returns (char(12) varying);

string = convert_binary_integer_$decimal_string (number);
```

1) number is a binary integer to be converted. (Input)

2) string is the representation of "number" in the decimal base. (Output)

Entry: convert_binary_integer_$octal_string

This entry converts a single precision binary integer to its octal string representation.

Usage

```
declare convert_binary_integer_$octal_string entry
        (fixed bin(35)) returns (char(13) varying);

string = convert_binary_integer_$octal_string (number);
```

1) number is a binary integer to be converted. (Input)

2) string is the representation of "number" in the octal base. (Output)

Entry: convert_binary_integer_$long_decimal_string

This entry converts a double precision binary integer to its decimal string representation.

Usage

```
declare convert_binary_integer_$long_decimal_string entry
        (fixed bin(71)) returns (char(23) varying);

string = convert_binary_integer_$long_decimal_string
         (long_num);
```

1) long_num is a double precision binary integer to be converted. (Input)

2) string is the representation of long_num in the decimal base. (Output)

Entry: convert_binary_integer_$long_octal_string

This entry converts a double precision binary integer to its octal string representation.

Usage

```
declare convert_binary_integer_$long_octal_string entry
        (fixed bin(71)) returns (char(25) varying);

string = convert_binary_integer_$long_octal_string
         (long_num);
```

1) long_num is a double precision binary integer to be converted. (Input)

2) string is the representation of long_num in the octal base. (Output)

(END)

Subroutine Call
8/10/73

Name: convert_date_to_binary_

The convert_date_to_binary_ subroutine converts a character representation of a date and time into the 72-bit system clock format. It accepts a wide variety of date and time forms, including the output of date_time_ (see the data_time_ subroutine write-up in the MPM).

Usage

```
declare convert_date_to_binary_ entry (char(*),
        fixed bin(71), fixed bin(35));

call convert_date_to_binary_ (string, clock, code);
```

1) string is the character representation of the clock reading desired. It has up to five parts (date, time, day-of-week, offset, and time zone), all of which are optional. They may appear only once and in any order. If all of them are omitted, the current time is returned. Each part can be made up of alphabetic fields, numeric fields, and special characters. An alphabetic field is made up of letters. The whole word or an abbreviation made up of the first three letters must be supplied. That means that Jan and January are equivalent. No distinction is made between upper and lower case. A numeric field consists of an integer of one or more decimal places. In addition, there are four special characters: the slash (/); the period (.); the colon (:); and the comma (,). Blanks are necessary to separate two numeric fields or two alphabetic fields. They are optional between alphabetic and numeric fields.

The five parts of the clock reading are:

date This is the date of the year. The year is optional, and, if omitted, is assumed to be the year in which the date will occur next. That is, if today is March 16, 1971, then March 20 is equivalent to March 20, 1971, while March 12 is the same as March 12, 1972. There are three forms of the date, illustrated by the examples below:

16 March 1971 or 16 March

March 16, 1971 or March 16 1971 or March 16 (Note that the comma is optional.)

3/16/71 or 3/16

time

This is the time of day. If omitted, it is assumed to be the current time. It has two basic formats, 24-hour and meridional time. The 24-hour time format consists of a four digit number hhmm, where hh is the hour, and mm is the minutes, followed by a period, and an optional decimal fraction of a minute field. Also acceptable are hours, minutes, and an optional seconds field separated by colons. The minutes and seconds fields must be two digits in length each.

Examples of 24-hour time are:

1545.

1545.715

15:45:08

Meridional time must end with a meridional designator (i.e., am, pm, noon (or n), midnight (or m)). If it is not preceded by a time, midnight (0000.0) is indicated by the alphabetic fields m or midnight, and noon (1200.0) is indicated by n or noon. The designator may be preceded by an hour, an hour-colon-minutes time, or an hour-colon-minutes-colon-seconds time. The minutes and seconds fields, if present, must be two digits in length.

Examples of meridional time are:

midnight

5 am

5:45 am

11:07:30 pm

day-of-week This field is the day of the week (i.e., Monday, Tuesday, etc.). If the day of the week is present along with a date, the date must fall on that day of the week or else a status code is returned. If a date is not present, the first day of the week after the current date is used; that means that Tuesday is interpreted as next Tuesday.

offset This field specifies an amount of time to be added to the clock value specified by the other fields. Offsets may be specified in any and all of the following units:

seconds	(second, sec)
minutes	(minute, min)
hours	(hour)
days	(day)
weeks	(week)
months	(month)

Only one occurrence of each unit can be present, each preceded by an integer. The singular version can only be used with 1, the plural for any other value. Note that if the offset field is the only field present, the offset is added to the current time.

If the month offset results in a nonexistent date (e.g., "Jan 31 3 months" would yield April 31), the last date of the resulting month is used (e.g., April 30). The month offset is applied before the other offsets and must not be abbreviated nor used with the zone field.

Examples of offset fields:

1 hour 5 minutes (an hour and five minutes from now)

Monday 6 am 2 weeks (two weeks from next Monday)

zone — This is the starting time zone to be used in making the conversion to GMT. It currently may be any of the following:

GMT (Greenwich mean time)
EST (eastern standard time)
EDT (eastern daylight time)
CST (central standard time)
CDT (central daylight time)
MST (mountain standard time)
MDT (mountain daylight time)
PST (Pacific standard time)
PDT (Pacific daylight time)

or the current time zone used by the system.

If omitted, the current time zone used by the system is assumed.

Note that if the date and day of the week are not present, the time returned is the next instance of that time after (or equal to) the current time. For example, if it is currently 3 pm, April 15, then 2 pm means 2 pm on the 16th, while 7 pm means 7 pm on the 15th (i.e., tonight).

2) clock — is set by convert_date_to_binary_ to the computed clock value. It is returned unchanged in the event of an error.

3) code — is a standard Multics status code. It is either zero (no errors), or error_table_$date_conversion_error. The nonzero value is returned in all of the following cases:

1) General syntax error.
2) Unrecognized alphabetic field.
3) Two or more dates, times, etc.
4) Month without a date number.
5) Year not in the twentieth century.
6) Date of month does not exist (e.g., 35 March).
7) Midnight and noon preceded by an hour other than 12.
8) Minutes greater than 59.
9) Seconds greater than 59.
10) 24-hour time after 2400.0 specified.
11) Zero hours in meridional time.

12) Month greater than 12 in slash time.
13) Minutes or seconds not two decimal places in length.
14) Day of week and date conflict.
15) Improper use of comma.
16) 24-hour time less than three places in length.
17) Improper use of offset.

Entry: convert_date_to_binary_$relative

This entry is similar to convert_date_to_binary_, except that the clock reading returned is computed relative to an input clock time rather than the current clock time. Thus the clock reading returned for the string "March 26" is the clock reading for the first March 26 following the input clock time, rather than the clock reading for the first March 26 following the present clock time. Given a 72-bit clock time to use, this entry converts a character representation of a date and time to the equivalent 72-bit system clock representation.

```
declare convert_date_to_binary_$relative entry
        (char(*), fixed bin(71), fixed bin(71),
        fixed bin(35));

call convert_date_to_binary_$relative (string, clock,
     clock_in, code);
```

1) string is as above. (Input)

2) clock is the clock time determined by string relative to clock_in. (Output)

3) clock_in is the clock time relative to which string is converted into a clock time. (Input)

4) code is as above. (Output)

Examples of Input

March 23

17 May 1974 EST 8:30 pm

03/28/71 2252.9 EST Sun

 (END)

Miscellaneous Call
Standard Service System
09/16/70

Entry: copy_acl_

This procedure copies the access control list from one segment to another. The ACL on the target segment is emptied before the new list is added.

Usage

```
declare copy_acl_ entry (char(*), char(*), char(*),
        char(*), char(*), bit(1) aligned, fixed bin(17));

call copy_acl_ (dir1, en1, dir2, en2, errsw, code);
```

1) dir1 is the directory in which the original segment is found. (Input)

2) en1 is a name on the original segment. (Input)

3) dir2 is the directory in which the target segment is found. (Input)

4) en2 is a name already on the target segment. (Input)

5) errsw indicates whether the error indicated by "code" occurred on the original segment ("0"b) or on the target segment ("1"b). (Output)

6) code is a standard file system error code. (Output)

(END)

Subroutine Call
Standard Service System
3/15/72

Name: copy_names_

This procedure copies all the names from one segment to another. Name duplications are handled by the standard system name duplication handling procedure.

Usage

```
declare copy_names_ entry (char(*), char(*), char(*),
        char(*), char(*), bit(1) aligned, fixed bin);

call copy_names_ (dir1, en1, dir2, en2, caller, errsw,
     code);
```

1) dir1 is the absolute path name of the directory in which the original segment is found. (Input)

2) en1 is a name on the original segment. (Input)

3) dir2 is the absolute path name of the target segment's directory. (Input)

4) en2 is a name already on the target segment. (Input)

5) caller is the name of the calling procedure. It is used by the standard system name duplication handling procedure. (Input)

6) errsw indicates which segment the error indicated by the code argument occurred in. It is "0"b if the error was on the original segment and "1"b if on the target segment. (Output)

7) code is a standard storage system status code. (Output)

Note

If a name duplication occurs due to another segment having the same name as the one copied, the status code error_table_$namedup is returned. Otherwise, if a name duplication occurs due to name being copied into a segment having the same name, the status code error_table_$segnamedup is returned.

(END)

Subroutine Call
Standard Service System
3/28/72

Name: copy_seg_

This procedure produces a copy of a Multics nondirectory segment. The new segment is created with rewa access for the creator.

Usage

```
declare copy_seg_ entry (char(*), char(*), char(*),
        char(*), char(*), bit(1) aligned, fixed bin);

call copy_seg_ (dir1, en1, dir2, en2, caller,
     errsw, code);
```

1) dir1 is the absolute path name of the directory in which the original segment is to be found. (Input)

2) en1 is a name on the original segment. (Input)

3) dir2 is the absolute path name of the directory in which the copy is to be created. (Input)

4) en2 is the name to be given the new segment. (Input)

5) caller is the name of the calling procedure. It is the procedure name that will be printed in status messages and questions. (Input)

6) errsw indicates which segment the error reported via the code argument occurred in. It is "0"b if the error was on the original segment and "1"b if on the target segment. (Output)

7) code is a standard system status code. (Output)

Notes

The following status codes may be of special interest to the caller of copy_seg_:

error_table_$dirseg
error_table_$moderr
error_table_$namedup
error_table_$sameseg

Note that other status codes may also be returned.

Entry: copy_seg_$no_message

This entry point performs the same task as copy_seg_ with the exception that the messages "Bit count inconsistent with current length..." and "Current length is not the same as records used..." are suppressed.

Usage

```
declare copy_seg_$no_message entry (char(*), char(*), char(*),
        char(*), char(*), bit(1) aligned, fixed bin);

call copy_seg_$no_message (dir1, en1, dir2, en2,
     caller, errsw, code);
```

Same arguments as above.

(END)

Subroutine Call
Development System
6/30/72

Name: cpu_time_and_paging_

This procedure returns the total CPU time used by the calling process since it was created as well as two measures of the paging activity of the process.

Usage

```
declare cpu_time_and_paging_ entry (fixed bin, fixed bin(71),
        fixed bin);

call cpu_time_and_paging_ (pf, time, pp);
```

1) pf is the total number of page faults taken by the calling process. (Output)

2) time is the total cpu time used by the calling process. (Output)

3) pp is the total number of pre-pagings for the calling process. (Output)

(END)

Name: cu_

The cu_ (command utility) module provides several short subroutines coded in machine language which provide functions not directly available in the PL/I language. Although these subroutines are designed primarily for the use of command writers, many may prove useful to Multics users and subsystem developers.

Entry: cu_$arg_count

The arg_count entry may be used by any procedure to determine the number of arguments with which it was called.

Usage

```
declare cu_$arg_count entry (fixed bin);

call cu_$arg_count (nargs);
```

1) nargs is the number of arguments passed to the caller of arg_count by his caller. (Output)

Entry: cu_$arg_ptr

The arg_ptr entry is designed for use by a command or subroutine which is callable with a varying number of arguments, each of which is an adjustable length unaligned character string (i.e., declared char(*)). This entry returns a pointer to and the length of a specified character string argument.

The command or subroutine which uses this entry must be called with data descriptors for its arguments. Otherwise, the returned value of arglen will be zero. If the argument specified by argno is not a character string, arglen will be the value of the "size" field of the descriptor (the rightmost 18 bits for old descriptors, the rightmost 24 bits for new descriptors). This entry must not be called from an internal procedure which has its own stack frame (because arg_ptr does not check for a display pointer).

Usage

```
declare cu_$arg_ptr entry (fixed bin, ptr, fixed bin,
        fixed bin(35));

call cu_$arg_ptr (argno, argptr, arglen, code);
```

1) argno — is an integer specifying the number of the desired argument. (Input)

2) argptr — is a pointer to the unaligned character string argument specified by argno. (Output)

3) arglen — is the length (in characters) of the argument specified by argno. (Output)

4) code — is an error status code. The code may be one of the following: 0 (normal return) or error_table_$noarg (the argument specified by argno does not exist). If error_table_$noarg is returned, the values of argptr and arglen are undefined. (Output)

Entry: cu_$arg_ptr_rel

Some PL/I procedures may wish to reference arguments passed to other procedures. This entry permits a procedure to reference arguments in any specified argument list.

Usage

```
declare cu_$arg_ptr_rel entry (fixed bin, ptr, fixed bin,
        fixed bin(35), ptr);

call cu_$arg_ptr_rel (argno, argptr, arglen, code, ap);
```

1 - 4) are the same as for cu_$arg_ptr.

5) ap — is a pointer to the argument list from which this argument is being extracted. This pointer may be determined by calling cu_$arg_list_ptr in the program whose argument list is to be processed and then passing it to the program wanting to look at the argument list. (Input)

Entry: cu_$arg_list_ptr

It is sometimes desirable to design a PL/I procedure to accept a variable number of arguments of varying data types (e.g., ioa_). In these cases, the PL/I procedure must be able to interrogate its argument list directly to determine the number, type, and location of each argument. The arg_list_ptr entry is designed for use in such cases and returns a PL/I pointer to its caller's argument list.

Usage

```
declare cu_$arg_list_ptr entry (ptr);

call cu_$arg_list_ptr (ap);
```

1) ap is a pointer to the callers' argument list. (Output)

Entry: cu_$af_arg_count

This entry assumes it has been called by an active function. It returns to its caller the number of arguments passed to the caller by its caller, not including the active function return argument. If the caller has not been invoked as an active function a status code is returned.

Usage

```
declare cu_$af_arg_count entry (fixed bin, fixed bin(35));

call cu_$af_arg_count (nargs, code);
```

1) nargs is the number of input arguments passed to the caller. (Output)

2) code may be one of the following:

0 - caller was called as an active function;

error_table_$nodescr - no argument descriptors were passed to the caller or an incorrect argument list header was encountered;

error_table_$not_act_fnc - the caller was not invoked as an active function. (Output)

Note

This entry and the two following entries, cu_$af_arg_ptr and cu_$af_return_arg, have been provided so that active functions need not have knowledge of the mechanism for returning arguments programmed into them.

Entry: cu_$af_arg_ptr

This entry assumes it has been called by an active function. It operates in the same fashion as cu_$arg_ptr, except that it verifies that the caller was invoked as an active function, and does not allow the return argument to be accessed. That is, if the return argument happens to be the i-th argument in the actual argument list, and one asks cu_$af_arg_ptr for the i-th argument, it will return the (i+1)st argument (if any). If the (i+1)st argument does not exist a "no argument" status code will be returned. In practice, the return argument is currently always the last one, but use of this entry and the following entry allows the active function to be independent of the position of the return argument in the argument list. (See Note under cu_$af_arg_count above.)

Usage

```
declare cu_$af_arg_ptr entry (fixed bin, ptr, fixed bin,
        fixed bin(35));

call cu_$af_arg_ptr (argno, argptr, arglen, code);
```

1) argno — is the number of the desired argument. (Input)

2) argptr — is the same as for cu_$arg_ptr, except that it is set to the null value if any error is encountered. (Output)

3) arglen — is the same as for cu_$arg_ptr, except that it is set to 0 if any error is encountered. (Output)

4) code — is the same as for cu_$af_arg_count, except that error_table_$noarg may also be returned, meaning that the argno-th input argument was not present. (Output)

Entry: cu_$af_return_arg

This entry assumes it has been called by an active function. It makes the active function's return argument available as described in Note below. It is provided to permit writing of active functions which accept an arbitrary number of arguments. (See Note under cu_$af_arg_count above).

Usage

```
declare cu_$af_return_arg entry (fixed bin, ptr, fixed bin,
        fixed bin(35));

declare return_string char (max_length) varying based
        (rtn_string_ptr);

call cu_$af_return_arg (nargs, rtn_string_ptr,
     max_length, code);
```

1) nargs is as in cu_$af_arg_count. (Output)

2) rtn_string_ptr is a pointer to the varying string return argument of the active function. (Output)

3) max_length is the maximum length of the varying string pointed to by rtn_string_ptr. (Output)

4) code is as in cu_$af_arg_count. (Output)

Note

An active function which takes an arbitrary number of arguments makes use of this entry to return a value as follows. It calls the entry to get a pointer to the return argument and its maximum length. It declares the based varying string, return_string, as described above. It then merely assigns its return value to return_string.

Entry: cu_$stack_frame_ptr

The stack_frame_ptr entry returns a pointer to its caller's stack frame.

Usage

```
declare cu_$stack_frame_ptr entry (ptr);

call cu_$stack_frame_ptr (sp);
```

1) sp is a pointer to the caller's stack frame. (Output)

Entry: cu_$stack_frame_size

The stack_frame_size entry returns the size (in words) of its caller's stack frame.

Usage

```
declare cu_$stack_frame_size entry (fixed bin);

call cu_$stack_frame_size (size);
```

1) size is the size (in words) of the caller's stack frame. (Output)

Entry: cu_$gen_call

The gen_call entry is used to generate a standard call to a specified procedure with a specified argument list. This call is designed for cases in which a PL/I procedure has explicitly built an argument list from its input data. The principal use of this entry is by command processors which call a command with an argument list built from a command line input from a terminal.

Usage

```
declare cu_$gen_call entry (ptr, ptr);

call cu_$gen_call (proc_ptr, ap);
```

1) proc_ptr is a pointer specifying the procedure entry point to be called. (Input)

2) ap is a pointer to the argument list to be passed to the called procedure. (Input)

Entry: cu_$ptr_call

The ptr_call entry is used to call a procedure, the name of which is not known at compilation time. This entry is similar to gen_call with the exception that the argument list of the procedure to be called is known at compilation time rather than constructed at execution time.

Usage

```
declare cu_$ptr_call entry;

call cu_$ptr_call (proc_ptr, arg1, ..., argn);
```

1) proc_ptr is a pointer to the procedure entry point to be called. (Input)

2) argi is an argument (which may be of any type) to be passed to the procedure specified by proc_ptr. (Input)

Entry: cu_$set_cp

Some standard commands (e.g., edm) allow the user to type normal command lines even when the user is not currently at command level. When one of these commands recognizes a command line, the command line is passed to the command processor for processing. In order to allow use of these standard commands in a closed subsystem environment (e.g., Limited Service System), these commands (edm, etc.) do not call the command processor directly but call the cp (command processor) entry point in cu_ which will pass control to the procedure entry point defined as the current command processor. The set_cp entry point allows a subsystem developer to replace the standard command processor with a procedure of his own. This mechanism can be used to insure that the subsystem remains in full control and still allow subsystem users the use of many standard commands.

Usage

```
declare cu_$set_cp entry (ptr);

call cu_$set_cp (proc_ptr);
```

1) proc_ptr is a pointer to the procedure entry point to which control is passed upon receiving a call to cu_$cp. (See below.) If proc_ptr is null, cu_$cp will call the standard command processor. (Input)

Entry: cu_$cp

The cp entry is called by any standard command which recognizes normal Multics command lines. When a Multics command line is recognized by one of these commands, a call is made to cu_$cp to pass the command line to the currently defined command processor for processing. The contents of the command line are destroyed.

Usage

```
declare cu_$cp entry (ptr, fixed bin, fixed bin(35);

call cu_$cp (line_ptr, line_len, code);
```

1) line_ptr is a pointer to the beginning of an aligned character string containing a command line to be processed. (Input)

2) line_len — is the length of the command line in characters. (input)

3) code — is an error status code. If code is non-zero, an error has been detected in the command processor. However, the caller of cp is not expected to print a diagnostic at this time since it can be expected that the command processor has already done so. (Output)

Entry: cu_$get_cp

This entry returns to the caller a pointer to the procedure currently being invoked in a call to cu_$cp. A null pointer is returned if the default standard command processor is being invoked by cu_$cp.

Usage

```
declare cu_$get_cp entry (ptr);

call cu_$get_cp (proc_ptr);
```

1) proc_ptr — is a pointer to the procedure entry point to which control is passed upon receiving a call to cu_$cp. (Output)

Entry: cu_$set_cl

The Standard Service System provides a set of procedures to handle any error conditions which may be signalled within a process (see signal_, condition_ and reversion_). The standard error handlers attempt to type an understandable diagnostic and call a procedure to reenter command level. Reentering command level is done for a Standard Service user via a call to get_to_cl_$unclaimed_signal. However, in order to allow use of the standard error handling procedures in a closed subsystem environment, the error handlers do not call get_to_cl_$unclaimed_signal directly but call the cl (command level) entry point in cu_ which will pass control to the procedure entry point currently defined by the last call to cu_$set_cl. If cu_$set_cl has never been called in the process, control will be passed to get_to_cl_$unclaimed_signal on a call to cu_$cl.

Usage

```
declare cu_$set_cl entry (ptr);

call cu_$set_cl (proc_ptr);
```

1) proc_ptr is a pointer to the procedure entry to be called by the standard error handlers after printing a diagnostic message. If proc_ptr is null, get_to_cl_$unclaimed_signal will be called. (Input)

Entry: cu_$cl

The cl entry is called by all standard error handlers after printing a diagnostic message. This entry will pass control to the procedure specified by the last call to set_cl. If no such procedure has been specified (the normal case for the Standard Service System), control is passed to get_to_cl_$unclaimed_signal which reenters command level.

Usage

```
declare cu_$cl entry;

call cu_$cl;
```

There are no arguments.

Entry: cu_$get_cl

This entry returns to the caller a pointer to the procedure entry currently being invoked by a call to cu_$cl. If a null pointer is returned, then the default call, to get_to_cl_$unclaimed_signal, is made by cu_$cl.

Usage

```
declare cu_$get_cl entry (ptr);

call cu_$get_cl (proc_ptr);
```

1) proc_ptr is a pointer to the procedure entry being called by the standard error handlers after printing a diagnostic message. (Output)

Entry: cu_$level_set

The level_set entry is used to change the current protection ring validation level. This entry is useful for procedures which must distinguish the periods of time when the procedure is acting in behalf of itself (i.e., its own ring) and when it is acting in behalf of another procedure which may be in an inferior (i.e., lower privilege) protection ring.

Usage

```
declare cu_$level_set entry (fixed bin);

call cu_$level_set (level);
```

1) level specifies the new protection validation level and must be greater than or equal to the current ring number. (Input)

Entry: cu_$level_get

The level_get entry is used to obtain the current ring validation level. This entry is normally used prior to a call to level_set to save the current validation level.

Usage

```
declare cu_$level_get entry (fixed bin);

call cu_$level_get (level);
```

1) level is the current validation level. (Output)

 (END)

Subroutine Call
9/13/73

Name: cv_acl_

This subroutine, given a pointer to an Access Control List (ACL), and the array index of an entry in the ACL, formats the entry for printing, returning the formatted string to the caller. Options allow suppression of the mode, process class identifier, and error message subfields. Refer to the MPM Reference Guide section, Access Control, for a discussion of access control and to the subroutine writeup for hcs_$add_acl_entries for a description of an ACL structure.

The formatted ACL entry which is returned has the form:

```
rew     User.Project.*     error_table_ message
```

Usage

```
declare cv_acl_ entry (ptr, fixed bin, char(*), fixed bin,
        bit(*));

call cv_acl_ (acl_ptr, index, string, len, options);
```

1) acl_ptr — is a pointer to the ACL array, one of which is to be formatted. (Input)

2) index — is the array index of the ACL entry to be formatted. (Input)

3) string — is the resultant formatted string containing the mode, process class indentifier, and error message as described above. (Output)

4) len — is the number of significant characters in string. (Output)

5) options — are control bits allowing suppression of various parts of the output: (Input)

   bit 1 on - include mode
   bit 2 on - include error_table_ message associated with the status code in the ACL entry
   bit 3 on - suppress process class identifier.

 (END)

Subroutine Call
10/11/72

Name: cv_bin_

The cv_bin_ procedure converts a binary integer (of any base) to a twelve-character ASCII string.

Usage

```
declare cv_bin_ entry (fixed bin, char(12), fixed bin);

call cv_bin_ (n, string, base);
```

1) n — is the binary integer to be converted. (Input)

2) string — is the character string in which to return the ASCII representation. (Output)

3) base — is the base to use in converting the binary integer. (e.g., base = 10 for decimal integers). (Input)

Entry: cv_bin_$dec

This entry converts a binary integer of base 10 to a twelve-character ASCII string.

Usage

```
declare cv_bin_$dec entry (fixed bin, char (12));

call cv_bin_$dec (n, string);
```

1) n — is the binary integer to be converted. (Input)

2) string — is the character string in which to return the ASCII representation. (Output)

Entry: cv_bin_$oct

This entry converts a binary integer of base 8 to a twelve-character ASCII string.

Usage

```
declare cv_bin_$oct entry (fixed bin, char(12));

call cv_bin_$oct (n, string);
```

1) n is the binary integer to be converted. (Input)

2) string is the character string in which to return the ASCII representation. (Output)

 (END)

Subroutine Call
Standard Service System
5/9/72

Name: cv_dec_

This procedure takes an ASCII representation of a decimal integer and returns the fixed binary(35) representation of that number.

Usage

```
declare cv_dec_ entry (char(*)) returns (fixed bin(35));

a = cv_dec_ (string);
```

1) string is the string to be converted. It must be non-varying. (Input)

2) a is the result of the conversion. (Output)

Note

The syntax of the string is a sequence of digits preceded by an optional plus or minus sign. Leading and trailing blanks are ignored.

Entry: cv_dec_check_

This entry point differs from cv_dec_ only in that a code is returned indicating the possibility of a conversion error. It may be called as shown because the segment is multiply named.

Usage

```
declare cv_dec_check_ entry (char(*), fixed bin)
        returns (fixed bin(35));

a = cv_dec_check_ (string, code);
```

1) string is the string to be converted. It must be nonvarying. (Input)

2) code =0 if no error occurred; otherwise it is the index of the character that terminated the conversion. (Output)

3) a is the result of the conversion. (Output)

(END)

Subroutine Call
9/13/73

Name: cv_dir_acl_

This subroutine, given a pointer to a directory Access Control List (ACL), and the array index of a directory ACL entry, formats the entry for printing, returning the formatted string to the caller. Options allow suppression of the mode, process class identifier, and error message subfields. Refer to the MPM Reference Guide section, Access Control, for a discussion of access control and the subroutine writeup for hcs_$add_dir_acl_ entries for a description of a directory ACL structure.

The formatted directory ACL entry that is returned has the form:

```
sma  User.Project.*  error_table_ message
```

Usage

```
declare cv_dir_acl_ entry (ptr, fixed bin, char(*),
        fixed bin, bit(*));

call cv_dir_acl_ (acl_ptr, index, string, len, options);
```

1) acl_ptr is a pointer to the ACL array, one of which is to be formatted. (Input)

2) index is the array index of the ACL entry to be formatted. (Input)

3) string is the resultant formatted string, containing the mode, process class identifier and error message as described above. (Output)

4) len is the number of significant characters in string. (Output)

5) options are control bits allowing suppression of various parts of the output: (Input)

   bit 1 on - include mode
   bit 2 on - include error_table_ message associated with the code in the directory ACL entry
   bit 3 on - suppress process class identifier.

 (END)

Subroutine Call
8/20/73

Name: cv_dir_mode_

This procedure converts a character string representation of access mode directory attributes (e.g., "sma", "null", "n", "") to the proper bit string for insertion into an Access Control List (ACL) entry. Mode characters in the input string can be in any order, and embedded blanks are ignored. See the MPM Reference Guide section, Access Control, for a description of access mode attributes.

Usage

```
declare cv_dir_mode_ entry (char(*), bit(*), fixed bin(35));

call cv_dir_mode_ (chars, bits, code);
```

1) chars is the character string to be converted (e.g., "sma"). (Input)

2) bits is the mode bit string corresponding to chars (e.g., "111"b). (Output)

3) code is a status code that has the value zero or error_table_$bad_acl_code. (Output)

 (END)

Subroutine Call
Development System
03/01/71

Name: cv_float_

This procedure converts a character string representation of a floating point number into a single precision floating point representation. If an illegal character is encountered, its index in the string is returned and the number is set to 0.0e0.

Usage

```
declare cv_float_ entry (char(*), fixed bin, float bin(27));

call cv_float_ (string, code, fnum);
```

1) string is the character representation of the number. (Input)

2) code is the index in string of the first illegal character, if one was found; otherwise it is zero. (Output)

3) fnum is the number in floating point form. (Output)

Entry: cv_float_double_

This entry is similar to cv_float_ except that the number returned is in double precision.

Usage

```
declare cv_float_double_ entry (char(*), fixed bin,
        float bin(63));

call cv_float_double_ (string, code, fnum);
```

Same arguments as above.

Note

The input string should look like a PL/1 floating or fixed point constant. It does not need either an explicit exponent or a decimal point.

(END)

Subroutine Call
8/20/73

Name: cv_mode_

This procedure converts a character string representation of access mode segment attributes (e.g., "rew", "null", "n", or "") to the proper bit string for insertion into an Access Control List (ACL) entry. Mode characters in the input string can be in any order, and embedded blanks are ignored.

Usage

```
declare cv_mode_ entry (char(*), bit(*), fixed bin(35));

call cv_mode_ (chars, bits, code);
```

1) chars is the character string to be converted (e.g., "rew"). (Input)

2) bits is the mode bit string corresponding to chars (e.g., "111"b). (Output)

3) code is a status code that has the value zero or error_table_$bad_acl_code. (Output)

 (END)

Subroutine Call
Standard Service System
8/18/71

Name: cv_oct_

This procedure takes an ASCII representation of an octal integer and returns the fixed binary(35) representation of that number. It may be called as shown because the segment is multiply named.

Usage

```
declare cv_oct_ entry (char(*) returns (fixed bin(35));

a = cv_oct_ (string);
```

1) string is the string to be converted. It must be non-varying. (Input)

2) a is the result of the conversion. (Output)

Entry: cv_oct_check_

This entry differs from cv_oct_ only in that a code is returned indicating the possiblity of a conversion error. It may be called as shown because the segment is multiply named.

Usage

```
declare cv_oct_check_ entry (char(*), fixed bin)
        returns (fixed bin(35));

a = cv_oct_check_ (string, code);
```

1) string is the string to be converted. It must be non-varying. (Input)

2) code =0 if no error occurred; otherwise it is the index of the character that terminated the conversion. (Output)

3) a is the result of the conversion. (Output)

(END)

Subroutine Call
8/20/73

Name: cv_userid_

This procedure, given an unnormalized process class identifier (e.g., "Multics"), returns a normalized identifier (e.g., "*.Multics.*"). See the MPM Reference Guide section, Access Control, for a discussion of process class identifiers.

Usage

```
declare cv_userid_ entry (char(*)) returns (char(32));

normal_id = cv_userid_(arg);
```

1) arg is the unnormalized process class identifier to be converted. (Input)

2) normal_id is the resultant normalized identifier. If it has fewer than 32 characters, it is padded on the right with blanks. (Output)

 (END)

Subroutine Call
Standard Service System
03/08/71

Name: date_time_

The date_time_ procedure converts a system clock reading to ASCII. It takes as an argument a clock reading (see clock_), fixed binary(71), and returns as an argument a 24 character nonvarying ASCII string. (If the caller declares the length to be less than 24, the string is truncated on the right; if greater than 24, the string is padded on the right with blanks.) The string format is "mm/dd/yy hhmm.m zzz www" where www is a three letter abbreviation of the day of the week. Clock readings not corresponding to dates in the twentieth century (after 1/1/1901) are converted as "01/01/01 0000.0". The clock reading is assumed to be in microseconds relative to January 1, 1901, Greenwich Mean Time (GMT). The time returned is local standard time.

Usage

```
declare date_time_ entry (fixed bin(71), char(*));

call date_time_ (time, string);
```

1) time is the clock reading. (Input)

2) string is the ASCII string equivalent of time. (Output)

Entry: date_time_$fstime

This entry point performs the same function as the above entry point but accepts a 36 bit time (as used internally in the file system) as input.

Usage

```
declare date_time_$fstime entry (fixed bin(35), char(*));

call date_time_$fstime (time, string);
```

Same arguments as above.

(END)

Subroutine Call
Development System
11/1/71

Name: decode_clock_value_

Given a calendar clock value, decode_clock_value_ will return the month, the day of the month, the year, the time of day, and the day of week it represents. In addition, the current time zone, used in the calculation, is returned.

Usage

```
declare decode_clock_value_ entry (fixed bin(71), fixed bin,
        fixed bin, fixed bin, fixed bin(71), fixed bin,
        char(3) aligned);

call decode_clock_value_ (clock, month, dom, year, tod, dow,
     zone);
```

1) clock is the clock value to be decoded. (Input)

2) month is the month (January = 1, December = 12). (Output)

3) dom is the day of the month. (Output)

4) year is the year. (Output)

5) tod is the number of microseconds since midnight (the time of day). (Output)

6) dow is the day of the week (Monday = 1, Sunday = 7). (Output)

7) zone is the current time zone. (Output)

(END)

Subroutine Call
Development System
7/28/71

Name: decode_descriptor_

This procedure extracts information from argument descriptors. It should be called by any procedure wishing to handle variable length or variable type argument lists. It can process both the old descriptor format used by Version 1 PL/1 and the new format used by Version 2 PL/1 and Fortran. The following type codes are used:

Type	Datum Type
1	real fixed binary short
2	real fixed binary long
3	real float binary short
4	real float binary long
5	complex fixed binary short
6	complex fixed binary long
7	complex float binary short
8	complex float binary long
9	real fixed decimal
10	real float decimal
11	complex fixed decimal
12	complex float decimal
13	pointer
14	offset
15	label
16	entry
17	structure
18	area

19	bit string
20	varying bit string
21	character string
22	varying character string
23	file

Usage

```
declare decode_descriptor_ entry (ptr, fixed bin,
        fixed bin, bit(1) aligned, fixed bin, fixed bin,
        fixed bin);

call decode_descriptor_ (pt, n, type, packed, ndims, size,
     scale);
```

1) pt — points either directly at the descriptor to be decoded or at the argument list in which the descriptor appears. (Input)

2) n — controls which descriptor is decoded. If n is 0, pt points at the descriptor to be decoded; otherwise, pt points at the argument list header and the nth descriptor will be decoded. (Input)

3) type — will be set to the data type specified by the descriptor. Type codes appearing in the old form of descriptor will be mapped into the new codes given earlier. The value 0 will be returned if an illegal type code is found in the old format descriptor. The value -1 will be returned if descriptors are not present in the argument list or if the nth descriptor does not exist. (Output)

4) packed — will be set to "1"b if the data item is packed. If an old format descriptor specifies a string, the value "1"b will be returned; otherwise, the value "0"b will be returned for old descriptors. (Output)

5) ndims — will be set to the number of dimensions specified by the descriptor. This value will be 0 or 1 for an old form of descriptor. (Output)

6) size — will be set to the arithmetic precision, string size, or number of structure elements of the datum. This value will be 0 if an old form of

descriptor specifies a structure. (Output)

7) scale — will be set to the scale of an arithmetic value. This value will be 0 for an old form of descriptor. (Output)

(END)

Subroutine Call
8/20/73

Name: decode_entryname_

This procedure, given a procedure entry point name of the form "a$b", returns the reference name and entry point portions separately; i.e., "a" and "b". If "a" is supplied, "a" and "a" are returned. If "a$" is supplied, the entry point portion is blank on return.

Usage

```
dcl decode_entryname_ entry (char(*), char(32), char(32));

call decode_entryname_ (name, rname, ename);
```

1) name is an entry point name; e.g., "a$b". (Input)

2) rname is the reference name portion of name; e.g., "a". If it has fewer than 32 characters, it is padded on the right with blanks. (Output)

3) ename is the entry point portion of name; e.g., "b". If it has fewer than 32 characters it is padded on the right with blanks. (Output)

 (END)

Subroutine Call
Development System
7/14/72

Name: delete_

The delete_ subroutine deletes segments and unlinks links. It asks questions if the segment to be deleted is protected, and attempts to remove the protection. It has two entries: one is called with a path name, the other with a pointer to a segment. Both have a set of switches which tell delete_ what to do. If the specified entry is a segment, it is terminated using the term_ subroutine.

Entry: delete_$path

This entry is called with the path name of the segment or link to be deleted.

Usage

```
declare delete_$path entry (char(*), char(*), bit(6),
        char(*), fixed bin(35));

call delete_$path (dname, ename, switches, caller, code);
```

1) dname is the directory in which the entry to be deleted resides. (Input)

2) ename is the entry within the specified directory to be deleted. (Input)

3) switches specifies the actions to be taken by this routine. It consists of six switches in the order listed below. (Input)

   force_sw If force_sw is "1"b, then delete_ will automatically attempt to delete the entry even if it is protected; if "0"b, the next switch is examined.

   question_sw If question_sw is "1"b, delete_ will ask the user if the entry should be deleted. A negative response will cause delete_ to return the status code error_table_$action_not_performed. If question_sw is "0"b, delete_ will just return if it is unable to delete the entry, with an appropriate storage system status code.

   dirctory_sw delete_ will only delete directories if this switch is "1"b. If the path name specified refers to a directory, and this switch is "0"b, delete_

will return with a status code of error_table_$dirseg.

segment_sw delete_ will delete segments only if this switch is "1"b. Multi-segment files are considered to be segments. If the path name specified refers to a segment, and this switch is "0"b, delete_ will return a status code of error_table_$nondirseg.

link_sw delete_ will delete (i.e. unlink) links only if this switch is "1"b. If the path name specified is a link, and this switch is "0"b, delete_ will return a status code of error_table_$not_a_branch.

chase_sw If link_sw is "1"b, and chase_sw is "1"b, delete_ will "chase" the link, and delete the segment that it points to.

4) caller is the name of the calling procedure, to be used when questions are asked. (Input)

5) code is a status code. (Output)

Entry: delete_$ptr

The ptr entry is similar to the path entry, except that the caller has a pointer to the actual segment to be deleted. The directory_sw, link_sw, and chase_sw are not examined by this entry, but must be present.

Usage

```
declare delete_$ptr entry (ptr, bit(6), char(*), fixed
        bin(35));

call delete_$ptr (segp, switches, caller, code);
```

1) segp is a pointer to the segment to be deleted. (Input)

The other arguments are as above.

(END)

I/O System Interface Module
Development System
9/30/71

Name: discard_output_

This I/O System Interface Module (IOSIM) provides a means by which output may be discarded. Any output written to a stream attached via the discard_output_ module will be discarded, i.e., no operation is performed. All implemented I/O system calls will return information indicating successful completion of the operation.

Usage

```
call ios_$attach (stream_name, "discard_output_", "",
     "", status);
```

Subsequent to the above attach call and until a corresponding detach call is made, all data written on stream_name will be discarded.

I/O System Calls

abort
attach
detach
resetwrite
write

Device Identification

Since no device or pseudo-device is involved, the device identification must be "".

Status

Only standard Multics error codes are returned as the first part of the status string.

Detachment

No special action is permitted when detaching.

(END)

Subroutine Call
8/16/73

Name: encipher_, decipher_

This subroutine enciphers and deciphers an array of double words. The caller supplies a 72-bit key that is used to generate an initial encoding, and subsequent keys are generated from the enciphered text.

Entry: encipher_

This entry point enciphers an array.

Usage

```
declare encipher_ entry (fixed bin(71), (*)fixed bin(71),
        (*)fixed bin(71), fixed bin);

call encipher_ (key, input, output, lth);
```

1) key is the input key. Any 72-bit key is appropriate and produces some enciphering of the array. (Input)

2) input is the array to be enciphered. (Input)

3) output is the enciphered array. (Output)

4) lth is the length of the input and output arrays in fixed bin(71) elements. (Input)

Entry: decipher_

This entry point deciphers an array previously produced by encipher_.

Usage

```
declare decipher_ entry (fixed bin(71), (*)fixed bin(71),
        (*)fixed bin(71), fixed bin);

call decipher_ (key, input, output, lth);
```

1) key is as above and should be the same as the key used to encipher the array if the original array is to be reproduced. (Input)

2) input is the enciphered array to be deciphered. (Input)

3) output is the deciphered array. (Output)

4) lth is as above. (Input)

 (END)

Subroutine Call
Standard Service System
03/08/71

Name: expand_path_

The expand_path_ procedure expands a relative pathname into an absolute pathname.

See MPM Reference Guide Section 2 on "The Multics Programming Environment".

Usage

```
declare expand_path_ entry (ptr, fixed bin, ptr, ptr,
        fixed bin);

call expand_path_ (pnamep, pnamel, dirp, enamep, code);
```

Below pname is used to denote the string pointed to by pnamep.

1) pnamep — is a pointer to the pathname to be expanded. It may point to an unaligned string. (Input)

2) pnamel — specifies the length of pname. If 0, pname is assumed to be the current working directory. (Input)

3) dirp — is a pointer to a string in which either the directory pathname derived from pname or the entire pathname derived from pname will be stored. (See 4, below.) It is assumed that dirp points to an aligned character string of length 168. (Input)

4) enamep — is a pointer to a string in which the entry name derived from pname is to be stored. If enamep = null, then the entire pathname derived from pname will be stored in dir. It is assumed that enamep points to an aligned character string of length 32. (Input)

5) code — is an error code. (Output)

Error codes that can be returned are:

1) error_table_$badpath — bad syntax in the pathname.

2) error_table_$dirlong — the directory pathname is longer than 168 characters.

3) error_table_$entlong — the entry name is longer than 32 characters.

4) error_table_$lesserr — too many "<"'s in the pathname.

5) error_table_$pathlong — the final pathname (directory name || entry name) is longer than 168 characters.

Examples

In all of the following examples, assume that the user's current working directory is >udd>dir>dl and that dir and ename stand for the string pointed to by dirp and enamep, respectively.

Input (pname)	Output: if enamep is null	Output: otherwise	
	dir	dir	ename
fred	>udd>dir>dl>fred	>udd>dir>dl	fred
<	>udd>dir	>udd	dir
<<	>udd	>	udd
<<george	>udd>george	>udd	george
<<george>harry	>udd>george>harry	>udd>george	harry
>udd>jack>bill	>udd>jack>bill	>udd>jack	bill

(END)

I/O System Interface Module
11/6/72

Name: file_

This pseudodevice interface module enables a procedure to use segments and multi-segment files as if they were I/O devices. For the sake of brevity, this document shall use the term "file" to mean a segment or multi-segment file. A file is made to appear as an I/O device by issuing an attach call as illustrated below. Once the attach call has been made, data may be read from or written to the file by the usual I/O read and write operations. Information on the I/O system can be found in the MPM Reference Guide section, Use of the Input and Output System.

Usage

```
call ios_$attach (stream_name, "file_", file_name, mode,
     status);
```

This call causes the file with absolute path name file_name to be attached as a pseudodevice via the stream stream_name. All subsequent read or write calls to stream_name will cause data to be input from or output to the specified file. The same file should not be attached to more than one stream or by more than one process simultaneously.

Permitted I/O System Calls

attach
detach
getdelim
getsize
read
seek
setdelim
setsize
tell
write

Device Identifiers

The device identifier used in a call to attach must be the path name of the file to be used as a pseudodevice.

Modes

Only read and write modes are allowed by this IOSIM. The mode string in the attach call conforms to the conventions described in the MPM Reference Guide section, Use of the Input Output System. The mode of a particular attachment may not be changed after the attach call.

Status

The first half of the status string is a standard Multics status code with zero indicating no error. In the second half of the status string, the stream name detached bit being on indicates the stream has been detached; the end of data bit being on indicates the last element has been read, or, more precisely, that the "read" reference pointer is beyond the "last" reference pointer.

Element Size

Any element size up to 2,359,296 is permitted. This is the size of a 64K segment. The default element size is 9.

Read Delimiters and Break Characters

Any element can be a read delimiter subject to the following general restrictions. Read delimiters may not be established if the element size is greater than 72 or if an element could potentially span a segment boundary (i.e., the element size does not evenly divide the maximum segment size). For element sizes less than or equal to 9, any number of elements may be established as read delimiters. For element sizes greater than 9, the maximum number of permitted read delimiters is the integral part of 720 divided by the element size. Note that changing the element size causes all currently established read delimiters to be discarded since the delimiters no longer make sense with a new element size. The default read delimiter is the "new line" character. Break characters are not implemented.

Reference Pointers

The reference pointers first, read, write, last, and bound are all implemented. The reference pointer "first" always has a value of zero and cannot be modified. The reference pointer "read" is initially set to the value of "first", while the reference pointer "write" is set to the value of "last". The pointer "last" is set to the last element of the file as indicated by the bit count.

Synchronization

The file_ I/O System Interface Module (IOSIM) operates in read synchronous, write synchronous mode only.

Detaching

Detaching causes the bit length attribute of the file to be updated to the new value if the length of the file has been modified. No special actions are permitted during detachment.

 (END)

Subroutine Call
10/31/73

Name: find_condition_info_

This procedure is given a pointer to a stack frame being used when a condition occurred and returns information relevant to that condition.

Usage

```
declare find_condition_info_ entry (ptr, ptr,
        fixed bin(35));

call find_condition_info_ (sp, cip, code);
```

1) sp is a pointer to a stack frame being used when a condition occurred. It is normally the result of a call to find_condition_frame_; or if null, the most recent condition frame is used. (Input)

2) cip is a pointer to the following structure in which information is returned. (Input)

```
declare 1 cond_info aligned,
          2 mcptr ptr,
          2 version fixed bin,
          2 condition_name char(32) varying,
          2 infoptr ptr,
          2 wcptr ptr,
          2 loc_ptr ptr,
          2 flags aligned,
            3 crawlout bit(1) unaligned,
            3 pad1 bit(35) unaligned,
          2 pad2 bit(36) aligned,
          2 user_loc_ptr ptr,
          2 pad(4) bit(36) aligned;
```

1) mcptr if not null, points to the machine conditions. Machine conditions are described in the MPM Reference Guide section, The Multics Condition Mechanism.

2) version is the version number of this structure (currently = 1).

3) condition_name is the condition name.

4) infoptr points to the info structure if there is one; otherwise it is null. The info structures for various

conditions are described in the MPM Reference Guide section, List of System Conditions and Default On Unit Actions.

5) wcptr — is a pointer to machine conditions describing a fault that caused control to leave the current ring. This occurs when the condition described by this structure was signalled from a lower ring and, before the condition occurred, the current ring was left because of a fault. Otherwise, it is null.

6) loc_ptr — is a pointer to the location where the condition occurred. If crawlout = "1"b, this points to the last location in the current ring before the condition occurred.

7) crawlout — if "1"b, indicates that the condition occurred in a lower ring but that it could not be adequately handled there.

8) pad1 — is currently unused.

9) pad2 — is currently unused.

10) user_loc_ptr — is a pointer to the most recent non-support location before the condition occurred. If the condition occurred in a support procedure (e.g., a PL/I support routine), this makes it possible to locate the user call that preceded the condition.

11) pad — is currently unused.

3) code — is a standard system status code. It is nonzero when sp does not point to a condition frame or, if sp is null, when no condition frame can be found. (Output)

 (END)

Subroutine Call
Development System
11/30/71

Name: get_default_wdir_

This function returns the path name of the user's current default working directory.

Usage

```
declare get_default_wdir_ entry returns (char(168) aligned);

default_wdir = get_default_wdir_;
```

1) default_wdir is the path name of the user's current default working directory. (Output)

(END)

Subroutine Call
9/28/73

Name: get_equal_name_

This procedure accepts an entry name and an equal name as its input, and constructs a target name by substituting components or characters from the entry name into the equal name, according to the Multics equal convention. Refer to the MPM Reference Guide section, Constructing and Interpreting Names, for a description of the equal convention and for the rules used to construct and interpret equal names.

Usage

```
declare get_equal_name_ entry (char(*), char(*), char(32),
        fixed bin(35));

call get_equal_name_ (entryname, equal_name, target_name,
     code);
```

1) entryname is the entry name from which the target is to be constructed. Trailing blanks in the entry name character string are ignored. (Input)

2) equal_name is the equal name from which the target is to be constructed. Trailing blanks in the equal name character string are ignored. (Input)

3) target_name is the target name that is constructed. (Output)

4) code is one of the following status codes: (Output)

0 the target name was constructed properly.

error_table_$bad_equal_name
the equal name has a bad format.

error_table_$badequal
there was no letter or component in the entry name that corresponds to a percent character (%) or an equal sign (=) in the equal name.

error_table_$longeql
the target name to be constructed would be longer than 32 characters.

Note

If the error_table_$badequal status code is returned, then a target_name is returned in which null character strings are used to represent the missing letter or component of entryname.

If the error_table$longeql status code is returned, then the first 32 characters of the target name to be constructed are returned as target_name.

The entryname argument which is passed to get_equal_name_ can also be used as the target_name argument, as long as the argument has a length of 32 characters.

 (END)

Name: get_group_id_

The get_group_id_ subroutine returns to the user the 32-character access identifier of the process in which it is called.

Usage

```
declare get_group_id_ entry returns (char(32) aligned);

char_string = get_group_id_ ();
```

1) char_string is a left-justified character string padded with trailing blanks, which contains the access identifier. (Output)

Entry: get_group_id_$tag_star

This entry point returns the access identifier of its caller with the instance component replaced by "*".

Usage

```
declare get_group_id_$tag_star entry returns
        (char(32) aligned);

char_string = get_group_id_$tag_star ();
```

1) char_string is as above.

 (END)

Name: get_pdir_

This subroutine returns the path name of the user's process directory. For a discussion of process directories, see the MPM Reference Guide section, The Storage System Directory Hierarchy.

Usage

```
declare get_pdir_ entry returns (char(168) aligned);

char_string = get_pdir_ ();
```

1) char_string is a left-justified character string, padded with trailing blanks, which contains the path name of the user's process directory. (Output)

 (END)

Name: get_process_id_

The get_process_id_ subroutine returns to the user the 36-bit identifier of the process in which it is called.

Usage

```
declare get_process_id_ entry returns (bit(36));

bit_string = get_process_id_ ();
```

1) bit_string is the 36-bit identifier of the process. (Output)

 (END)

Subroutine Call
Standard Service System
02/16/71

Name: get_wdir_

This function returns the pathname of the user's current working directory.

Usage

```
declare get_wdir_ entry returns (char(168) aligned);

working_dir = get_wdir_;
```

1) working_dir is the pathname of the user's current working directory. (Output)

(END)

Subroutine Call
2/12/73

Name: hcs_$add_acl_entries

This subroutine, given a list of Access Control List (ACL) entries, will add the given ACL entries, or change their modes if a corresponding entry already exists, to the ACL of the specified segment.

Usage

```
declare hcs_$add_acl_entries entry (char(*), char(*),
        ptr, fixed bin, fixed bin(35));

call hcs_$add_acl_entries (dirname, ename, acl_ptr,
     acl_count, code);
```

1) dirname — is the directory portion of the path name of the segment in question. (Input)

2) ename — is the entry name portion of the path name of the segment in question. (Input)

3) acl_ptr — points to a user-filled segment_acl structure. See Notes below. (Input)

4) acl_count — contains the number of ACL entries in the segment_acl structure. See Notes below. (Input)

5) code — is a standard status code. (Output)

Notes

The following structure is used:

```
dcl 1 segment_acl (acl_count) aligned based (acl_ptr),
    2 access_name char(32),
    2 modes bit(36),
    2 zero_pad bit(36),
    2 status_code fixed bin(35);
```

1) access_name — is the access name (in the form person.project.tag) which identifies the processes to which this ACL entry applies.

2) modes — contain the modes for this access name. The first three bits correspond to the modes read, execute, and write. The remaining bits must be zero.

3) zero_pad must contain zero. (This field is for use with extended access.)

4) status_code is a standard status code for this ACL entry only.

If code is returned as error_table$argerr then the offending ACL entries in segment_acl will have status_code set to an appropriate error and no processing will have been performed.

If the segment is a gate (see the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings)), then if the validation level is greater than Ring 1, then only access names that contain the same project as the user, and "SysDaemon" and "sys_control" projects will be allowed. If the ACL to be added is in error then no processing will be performed and the code error_table_$invalid_project_for_gate will be returned.

 (END)

Subroutine Call
2/13/73

Name: hcs_$add_dir_acl_entries

This subroutine, given a list of Access Control List (ACL) entries, will add the given ACL entries, or change their directory modes if a corresponding entry already exists, to the ACL of the specified directory.

Usage

```
declare hcs_$add_dir_acl_entries entry (char(*), char(*),
        ptr, fixed bin, fixed bin(35));

call hcs_$add_dir_acl_entries (dirname, ename, acl_ptr,
     acl_count, code);
```

1) dirname — is the path name of the directory superior to the one in question. (Input)

2) ename — is the entry name of the directory in question. (Input)

3) acl_ptr — points to a user-filled dir_acl structure. See Notes below. (Input)

4) acl_count — contains the number of entries in the dir_acl structure. See Notes below. (Input)

5) code — is a standard status code. (Output)

Notes

The following structure is used:

```
declare 1 dir_acl (acl_count) aligned based (acl_ptr),
          2 access_name char(32),
          2 dir_modes bit(36),
          2 status_code fixed bin(35);
```

1) access_name — is the access name (in the form person.project.tag) which identifies the process to which this ACL entry applies.

2) dir_modes — contains the directory modes for this access name. The first three bits correspond to the modes status, modify, and append. The remaining bits must be zero.

3) status_code — is a standard status code for this ACL entry only.

If code is returned as error_table_$argerr then the offending ACL entries in the dir_acl structure will have status_code set to an appropriate error and no processing will have been performed.

 (END)

Subroutine Call
3/19/73

Name: hcs_$append_branch

This entry creates a segment in the specified directory, initiates the segment's Access Control List (ACL) by copying the Initial ACL for segments found in the directory, and adds the user to the segment's ACL with the mode specified. ACLs and Initial ACLs are described in the MPM Reference Guide section, Access Control.

Usage

```
declare hcs_$append_branch entry (char(*), char(*),
        fixed bin (5), fixed bin (35));

call hcs_$append_branch (dirname, entryname, mode, code);
```

1) dirname is the path name of the directory in which segname is to be placed. (Input)

2) entryname is the entry name of the segment to be created. (Input)

3) mode is the user's access mode; see Notes below. (Input)

4) code is a standard storage system status code. (Output)

Notes

Append (a) access mode is required in the directory dirname to add an entry to that directory.

A number of attributes of the segment are set to default values.

1) Ring brackets are set to the user's current validation level. See the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

2) The user ID is set to the name and project of the user, with the instance tag set to *.

3) The copy switch is set to 0.

4) The bit count is set to 0.

See the MPM write-up for hcs_$append_branchx to create a storage system entry with other values than the defaults listed above.

The mode argument is a fixed binary number where the desired mode is encoded with one access mode specified by each bit. For segments the modes are:

read	8-bit (i.e., 01000b)
execute	4-bit (i.e., 00100b)
write	2-bit (i.e., 00010b)

For directories, the modes are:

status	8-bit (i.e., 01000b)
modify	2-bit (i.e., 00010b)
append	1-bit (i.e., 00001b)

The unused bits are reserved for unimplemented attributes and must be zero. For example, rw access in bit form is 01010b, and is 10 in fixed binary form.

 (END)

Subroutine Call
3/20/73

Name: hcs_$append_branchx

This entry creates either a subdirectory or a segment in the specified directory. (The entry point is really nothing more than an extended and more general form of hcs_$append_branch.) If a subdirectory is created then the subdirectory's access control list (ACL) is initiated by copying the Initial ACL for directories that is stored in the specified directory; otherwise the segment's ACL is initialized by copying the Initial ACL for segments. The input userid and mode (See Usage below) are then added to the ACL of the subdirectory or segment.

Usage

```
declare hcs_$append_branchx entry (char(*), char(*), fixed
        bin (5), (3) fixed bin (6), char(*), fixed bin (1),
        fixed bin (1), fixed bin (24), fixed bin (35));

call hcs_$append_branchx (dirname, entryname, mode, rings,
     userid, dirsw, copysw, bitcnt, code);
```

1) dirname is the path name of the directory in which entryname is to be placed. (Input)

2) entryname is the name of the segment or subdirectory to be created. (Input)

3) mode is the user's access mode; see Notes below. (Input)

4) rings are the new segment's or subdirectory's ring brackets; see the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings). (Input)

5) userid is the user's access control name of the form Person.Project.Tag. (Input)

6) dirsw is the branch's directory switch (= 1 if a directory is being created; = 0 otherwise). (Input)

7) copysw is the segment copy switch (= 1 if a copy is wanted whenever the segment is initiated; = 0 if the original is wanted). (Input)

8) bitcnt is the segment's length (in bits). (Input)

9) code is a standard storage system status code. (Output)

Notes

Append (a) access mode is required in the directory dirname to add an entry to that directory.

The mode argument is a fixed binary number where the desired mode is encoded with one access mode specified by each bit. For segments the modes are:

read	8-bit (i.e., 01000b)
execute	4-bit (i.e., 00100b)
write	2-bit (i.e., 00010b)

For directories, the modes are:

status	8-bit (i.e., 01000b)
modify	2-bit (i.e., 00010b)
append	1-bit (i.e., 00001b)

Note that if modify access is given for a directory, then status must also be given; i.e., 01010b. The unused bits are reserved for unimplemented attributes and must be zero. For example, rw access in bit form is 01010b, and is 10 in fixed binary form.

Name: hcs_$append_link

This subroutine is provided to create a link in the storage system directory hierarchy to some other directory entry in the hierarchy. For a discussion of links see the MPM Reference Guide section Segment, Directory and Link Attributes.

Usage

```
declare hcs_$append_link entry (char(*), char(*) char(*),
        fixed bin(35));

call hcs_$append_link (dir_name, link_name, path, code);
```

1) dir_name is the directory path name in which the link is to be created. (Input)

2) link_name is the entry name of the link to be created. (Input)

3) path is the path name of the segment to which link_name is to point. (Input)

4) code is a standard storage system status code. (Output)

Notes

The user must have the append attribute with respect to the directory in which the link is being created.

The entry pointed to by the link need not exist at the time the link is created.

The subroutines hcs_$append_branch and hcs_$append_branchx may be used to create a segment or directory entry in the storage system hierarchy.

 (END)

Name: hcs_$chname_file

This subroutine changes an entry name on a storage system entry specified by path name. If an old (i.e., already existing) name is specified, it is deleted from the entry; if an new name is specified it is added. Thus, if only an old name is specified, the effect is to delete a name; if only a new name is specified, the effect is to add a name; and if both are specified, the effect is to rename the entry.

Usage

```
declare hcs_$chname_file entry (char(*), char(*),
        char(*), char(*), fixed bin(35));

call hcs_$chname_file (dir_name, entry_name, oldname,
     newname, code);
```

1) dir_name is the path name of the directory in which the entry to be manipulated is found. (Input)

2) entry_name is the name of the entry to be manipulated. (Input)

3) oldname is the name to be deleted from the entry. It may be a null character string ("") in which case no name is to be deleted. If oldname is null, then newname must not be null. (Input)

4) newname is the name to be added to the entry. It must not already exist in the directory on this or another entry. It may be a null character string ("") in which case no name is added. If it is null, then oldname must not be the only name on the entry. (Input)

5) code is a standard storage system status code. It may have the values:

   error_table_$nonamerr
   error_table_$namedup
   error_table_$segnamedup (Output)

Notes

The subroutine hcs_$chname_seg performs the same function, given a pointer to the segment instead of its path name.

The user must have the modify attribute with respect to the directory in question.

Examples

Assume that the entry >my_dir>alpha exists and that it also has the entry name beta. Then the following calls to hcs_$chname_file would have the effects described.

```
call hcs_$chname_file (">my_dir", "alpha", "beta", "gamma",
     code);
```

This call would change the entry name beta to gamma.

```
call hcs_$chname_file (">my_dir", "gamma", "gamma", "",
     code);
```

This call would remove the entry name gamma. Note that any entry name may be used in the second argument position.

```
call hcs_$chname_file (">my_dir", "alpha", "", "delta",
     code);
```

This call would add the entry name delta. The entry now has the names alpha and delta.

 (END)

Subroutine Call
2/13/73

Name: hcs_$chname_seg

This subroutine changes an entry name on a storage system segment, given a pointer to the segment. If an old (i.e., already existing) name is specified, it is deleted from the entry; if a new name is specified, it is added. Thus, if only an old name is specified, the effect is to delete a name; if only a new name is specified, the effect is to add a name; and if both are specified, the effect is to rename the entry.

Usage

```
declare hcs_$chname_seg entry (ptr, char(*), char(*),
        fixed bin(35));

call hcs_$chname_seg (seg_ptr, oldname, newname, code);
```

1) seg_ptr is a pointer to the segment whose name will be changed. (Input)

2) oldname is the name to be deleted from the entry. It may be a null character string ("") in which case no name is to be deleted. If oldname is null, then newname must not be null. (Input)

3) newname is the name to be added to the entry. It must not already exist in the directory on this or another entry. If may be a null character string ("") in which case no name is added. If it is null, then oldname must not be the only name on the entry. (Input)

4) code is a standard storage system status code. It may have the values:

   error_table_$namedup
   error_tab_$nonemerr
   error_table_$segnamedup (Output)

Notes

The subroutine hcs_$chname_file performs the same function, given the directory and entry names of the segment instead of the pointer.

The user must have the modify attribute with respect to the directory in question.

Examples

Assume that the user has a pointer, seg_ptr, to a segment which has two entry names, alpha and beta. Then the following calls to hcs_$chname_seg would have the effects described.

```
call hcs_$chname_seg (seg_ptr, "beta", "gamma", code);
```

This call would change the entry name beta to gamma.

```
call hcs_$chname_seg (seg_ptr, "gamma", "", code);
```

This call would remove the entry name gamma.

```
call hcs_$chname_seg (seg_ptr, "", "delta", code);
```

This call would add the entry name delta. The entry now has the names alpha and delta.

 (END)

Subroutine Call
2/21/73

Name: hcs_$del_dir_tree

This subroutine deletes a subtree of the storage system hierarchy, given the path name of a directory. All segments, links and directories inferior to that directory are deleted including the contents of any inferior directories. The specified directory is not itself deleted; to delete it, see the MPM write-ups for hcs_$delentry_file and hcs_$delentry_seg.

Usage

```
declare hcs_$del_dir_tree entry (char(*), char(*),
        fixed bin(35));

call hcs_$del_dir_tree (parent_name, dir_name, code);
```

1) parent_name is the path name of the parent directory of the directory whose subtree is to be deleted. (Input)

2) dir_name is the entry name of the directory whose subtree is to be deleted. (Input)

3) code is a standard storage system status code.

Note

The user must have the status and modify attributes with respect to the specified directory and the safety switch must be off in that directory. If the user does not have status and modify attributes on inferior directories, hcs_$del_dir_tree will provide them.

If an entry in an inferior directory gives the user access only in a ring lower than his validation level, that entry will not be deleted and no further processing will be done on the subtree. For those users who need to know about rings, they are discussed in the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

 (END)

Subroutine Call
3/15/73

Name: hcs_$delentry_file

This subroutine, given a directory name and an entry name, deletes the given entry from its parent directory. If the entry is a segment the contents of the segment are deleted first. If the entry specifies a directory which contains entries the status code error_table_$fulldir is returned and hcs_$del_dir_tree must be called to remove the contents of the directory.

Usage

```
declare hcs_$delentry_file (char(*), char(*),
        fixed bin(35));

call hcs_$delentry_file (dirname, ename, code);
```

1) dirname is the parent directory name. (Input)

2) ename is the entry name to be deleted. (Input)

3) code is a standard storage system status code. (Output)

Notes

The subroutine hcs_$delentry_seg performs the same function, given pointer to the segment instead of the pathname.

The user must have modify permission with respect to dirname. If ename specifies a segment or directory rather than a link, the safety switch of the segment or directory must be off. For a temporary period the user must have write permission with respect to the segment or modify permission with respect to the directory being deleted.

 (END)

Subroutine Call
3/14/73

Name: hcs_$delentry_seg

This subroutine, given the pointer to a segment, deletes the corresponding entry from its parent directory. If the entry is a segment the contents of the segment are deleted first. If the entry specifies a directory which contains entries the status code error_table_$fulldir is returned and hcs_$del_dir_tree must be called to remove the contents of the directory.

Usage

```
declare hcs_$delentry_seg (ptr, fixed bin(35));

call hcs_$delentry_seg (sp, code);
```

1) sp is the pointer to the segment to be deleted. (Input)

2) code is a standard storage system status code. (Output)

Notes

The subroutine hcs_$delentry_file performs the same function, given the directory and entry names of the segment instead of the pointer.

The user must have modify permission with respect to the segment's parent directory. The safety switch of the segment must be off. For a temporary period the user must have write permission with respect to the segment.

 (END)

Subroutine Call
2/12/73

Name: hcs_$delete_acl_entries

This subroutine is called to delete specified entries from an Access Control List (ACL) for a segment.

Usage

```
declare hcs_$delete_acl_entries entry (char(*), char(*),
        ptr, fixed bin, fixed bin(35));

call hcs_$delete_acl_entries (dirname, ename, acl_ptr,
     acl_count, code);
```

1) dirname is the directory portion of the path name of the segment in question. (Input)

2) ename is the entry name portion of the path name of the segment in question. (Input)

3) acl_ptr points to a user-filled delete_acl structure. See Notes below. (Input)

4) acl_count contains the number of ACL entries in the delete_acl structure. See Notes below. (Input)

5) code is a standard status code. (Output)

Notes

The following structure is used:

```
declare 1 delete_acl (acl_count) aligned based (acl_ptr),
          2 access_name char(32),
          2 status_code fixed bin(35);
```

1) access_name is the access name (in the form of person.project.tag) which identifies the ACL entry to be deleted.

2) status_code is a standard status code for this ACL entry only.

If code is returned as error_table_$argerr then the offending ACL entries in the delete_acl structure will have status_code set to an appropriate error and no processing will have been performed.

If an access name cannot be matched to one existing on the segment's ACL then the status code of that ACL entry is set to error_table_$user_not_found, processing continues to the end of the delete_acl structure and code is returned as zero.

 (END)

Subroutine Call
2/13/73

Name: hcs_$delete_dir_acl_entries

This subroutine is used to delete specified entries from an Access Control List (ACL) for a directory. The delete_acl structure used by this subroutine is described in the MPM write-up for hcs_$delete_acl_entries.

Usage

```
declare hcs_$delete_dir_acl_entries entry (char(*),
        char(*), ptr, fixed bin, fixed bin(35));

call hcs_$delete_dir_acl_entries (dirname, ename, acl_ptr,
     acl_count, code);
```

1) dirname is the path name of the directory superior to the one in question. (Input)

2) ename is the entry name of the directory in question. (Input)

3) acl_ptr points to a user-filled delete_acl structure. (Input)

4) acl_count is the number of ACL entries in the delete_acl structure. (Input)

5) code is a standard status code. (Output)

Note

The status code is interpreted as described in hcs_$delete_acl_entries.

 (END)

Subroutine Call
3/8/73

Name: hcs_$fs_get_mode

This subroutine returns the access mode of the user, at the current validation level, with respect to a specified segment. For a discussion of access modes, see the MPM Reference Guide section, Access Control.

Usage

```
declare hcs_$fs_get_mode entry (ptr, fixed bin(5),
        fixed bin(35));

call hcs_$fs_get_mode (segptr, mode, code);
```

1) segptr is an pointer to the segment in question. (Input)

2) mode is the mode (see Notes below). (Output)

3) code is a standard storage system status code. (Output)

Notes

The mode and ring brackets for the segment in the user's address space are used in combination with the user's current validation level to determine the mode the user would have if he accessed this segment. For a discussion of ring brackets and validation level, see the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

The mode argument is a fixed binary number where the desired mode is encoded with one access mode specified by each bit. For segments the modes are:

read	8-bit (i.e., 01000b)
execute	4-bit (i.e., 00100b)
write	2-bit (i.e., 00010b)

For directories, the modes are:

status	8-bit (i.e., 01000b)
modify	2-bit (i.e., 00010b)
append	1-bit (i.e., 00001b)

Note that if modify access is given for a directory, then status must also be given; i.e., 01010b). The high-order bit is reserved for an unimplemented attribute and must be zero. For example, rw access in bit form is 01010b, and is 10 in fixed binary form.

 (END)

Subroutine Call
2/28/73

Name: hcs_$fs_get_path_name

This entry, given a pointer to a segment, returns a path name for the segment, with the directory and entry name portions of the path name separated. The entry name returned is the primary name on the entry; see the MPM Reference Guide section, Segment, Directory and Link Attributes for a discussion of primary names.

Usage

```
declare hcs_$fs_get_path_name entry (ptr, char(*),
        fixed bin, char(*), fixed bin(35));

call hcs_$fs_get_path_name (segptr, dirname, ldn, ename,
     code);
```

1) segptr — is a pointer to the segment in question. (Input)

2) dirname — is the path name of the directory superior to the segment pointed to by segptr. If the length of the path name to be returned is greater than the length of dirname, the path name will be truncated. To avoid this problem, the length of dirname should be 168 characters. (Output)

3) ldn — is the number of nonblank characters in dirname. (Output)

4) ename — is the primary entry name of the segment pointed to by segptr. If the length of the entry name to be returned is greater than the length of ename, the entry name will be truncated. To avoid this problem, the length of ename should be 32 characters. (Output)

5) code — is a standard storage system status code. (Output)

 (END)

Subroutine Call
Standard Service System
8/24/71

Name: hcs_$fs_get_ref_name

This entry point returns a specified (i.e., first, second, etc.) reference name for a specified segment.

Usage

```
declare hcs_$fs_get_ref_name entry (ptr, fixed bin,
        char(*), fixed bin);

call hcs_$fs_get_ref_name (segptr, count, rname, code);
```

1) segptr — is a pointer to the segment in question. (Input)

2) count — specifies which reference name is to be returned. See Notes. (Input)

3) rname — is the desired reference name. (Output)

4) code — is a standard file system status code. (Output)

Notes

If "count" = 1, the name by which the segment has most recently been made known will be returned. If "count" = 2, the second most recently added name is returned and so on. If "count" is larger than the total number of names, the name by which the segment was originally made known is returned and "code" is set to error_table_$ref_count_too_big.

See the MPM Reference Guide Section on naming conventions.

(END)

Subroutine Call
2/16/73

Entry: hcs_$fs_get_seg_ptr

Given a reference name of a segment, hcs_$fs_get_seg_ptr returns a pointer to the base of the segment. For a discussion of reference names, see the MPM Reference Guide section, Constructing and Interpreting Names.

Usage

```
declare hcs_$fs_get_seg_ptr entry (char(*), ptr,
        fixed bin(35));

call hcs_$fs_get_seg_ptr (rname, segptr, code);
```

1) rname is the reference name of a segment for which a pointer is to be returned. (Input)

2) segptr is a pointer to the base of the segment. (Output)

3) code is a standard status code. (Output)

Note

If the reference name is accessible from the user's current validation level, segptr is returned pointing to the segment; otherwise, it is null. The user who needs to know about rings and validation levels can find a discussion of them in the Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

 (END)

Subroutine Call
2/27/73

Name: hcs_$fs_move_file

This subroutine moves the data associated with one segment in the storage system hierarchy to another segment given the path names of the segments in question. The old segment remains, with a zero length.

Usage

```
declare hcs_$fs_move_file entry (char(*), char(*),
        fixed bin(2), char(*), char(*), fixed bin(35));

call hcs_$fs_move_file (from_dir, from_entry, at_sw, to_dir,
     to_entry, code);
```

1) from_dir — is the path name of the directory in which from_entry resides. (Input)

2) from_entry — is the entry name of the segment from which data is to be moved. (Input)

3) at_sw — see Notes below. (Input)

4) to_dir — is the path name of the directory in which to_entry resides. (Input)

5) to_entry — is the entry name of the segment to which data is to be moved. (Input)

6) code — is a standard storage system status code. It may have the value error_table_$no_move if either entry is not a segment, or one of the values described in Notes below.

Notes

The input argument at_sw is a 2-indicator switch which directs the procedure to use certain options. The two options specified are append option and truncate option. If the append option (high-order bit) is on, then append to_entry to to_dir if it does not already exist. If the append option is off and the destination entry can not be found the status code error_table_$noentry is returned.

If the truncate option (low-order bit) is on, to_entry is truncated if it is not zero length. Otherwise (i.e., if the option is off and the length of to_entry is not zero) the status code error_table_$clnzero is returned. In both of the cases where the move is not completed, the procedure will attempt to return the data to the original segment.

The subroutine hcs_$fs_move_seg performs the same function given pointers to the segments in question instead of path names.

 (END)

Subroutine Call
2/28/73

Name: hcs_$fs_move_seg

This subroutine moves the data associated with one segment in the hierarchy to another segment, given pointers to the segments in question. The old segment remains, with a zero length.

Usage

```
declare hcs_$fs_move_seg entry (ptr, ptr, fixed bin(1),
        fixed bin(35));

call hcs_$fs_move_seg (from_ptr, to_ptr, trunsw, code);
```

1) from_ptr is the pointer to the segment from which data is to be moved. (Input)

2) to_ptr is the pointer to the target segment. (Input)

3) trunsw if equal to 1, then truncate the segment specified by to_ptr (if it is not already zero-length) before performing the move;
if equal to 0, then reflect the status code error_table_$clnzero if that segment is not already zero-length. (Input)

4) code is a standard storage system status code. Besides the value given under trunsw above, it may also have the value error_table_$no_move. (Output)

Note

The subroutine hcs_$fs_move_file performs the same function given the path names of the segments in question instead of the pointers.

 (END)

Subroutine Call
3/12/73

Name: hcs_$initiate

This subroutine is used to search for a segment, make a copy of it if the copy switch so indicates, and make the segment or its copy known to the process. The reference name specified is entered in the address space of the process and a pointer to the segment is returned. If segsw is on, then the segment pointer is input and the segment is made known with that segment number.

Usage

```
declare hcs_$initiate entry (char(*), char(*), char(*),
        fixed bin(1), fixed bin(2), ptr, fixed bin(35));

call hcs_$initiate (pname, ename, rname, segsw, copysw,
     segptr, code);
```

1) pname — is the path name of the directory containing the segment. (Input)

2) ename — is the entry name of the segment. (Input)

3) rname — is the reference name. If it is zero in length, the segment is initiated by a null name. (Input)

4) segsw — is the reserved segment switch:

   = 0 if no segment number has been reserved;
   = 1 if a segment number was reserved. (Input)

5) copysw — is the copy switch:

   = 0 if it is desired to go by the setting in the directory entry;
   = 1 if no copy is wanted;
   = 2 if a copy is always wanted. (Input)

6) segptr — is a pointer to the segment. (It is Input if segsw = 1. Otherwise, it is Output.)

7) code — is a standard storage system status code. (Output)

Notes

The user must have non-null access on the segment ename in order to initiate it.

If ename cannot be initiated, a null pointer is returned for segptr and the returned value of code indicates the reason for failure. If ename is already known to the user's process, code is returned as error_table_$segknown and the the argument segptr will contain a valid pointer to ename. If ename is not already known, and no problems are encountered, segptr will contain a valid pointer and code will be zero.

 (END)

Subroutine Call
3/12/73

Name: hcs_$initiate_count

This subroutine, given a path name and a reference name, causes the segment defined by the path name (or a copy of it, depending upon the copysw option) to be made known by the given reference name. A segment number is assigned and returned as a pointer and the bit count of the segment is returned.

Usage

```
declare hcs_$initiate_count entry char(*), char(*), char(*),
        fixed bin(24), fixed bin(2), ptr, fixed bin(35));

call hcs_$initiate_count (pname, ename, rname, bitcount,
     copysw, segptr, code);
```

1) pname — is the directory portion of the path name of the segment in question. (Input)

2) ename — is the entry name portion of the path name of the segment in question. (Input)

3) rname — is the desired reference name. If it is zero in length, the segment is initiated by a null name. (Input)

4) bitcount — is the bit count of the segment. (Output)

5) copysw — is the copy switch:

   = 0 if it is desired to go by the setting in the hierarchy entry;
   = 1 if no copy is wanted;
   = 2 if a copy is always wanted. (Input)

6) segptr — is a pointer to the segment in question. (Output)

7) code — is a standard storage system status code. (Output)

Notes

The user must have non-null access on the segment ename in order to initiate it.

If ename cannot be initiated, a null pointer is returned for segptr and the returned value of code indicates the reason for failure. If ename is already known to the user's process, code is returned as error_table_$segknown and the argument segptr will contain a valid pointer to ename. If ename is not already known, and no problems are encountered, segptr will contain a valid pointer and code will be zero.

See also the MPM Reference Guide section, Constructing and Interpreting Names.

 (END)

Subroutine Call
2/15/73

Name: hcs_$list_acl

This subroutine is used to either list the entire Access Control List (ACL) of a segment or to return the access modes from specified entries. The segment_acl structure used by this subroutine is described in the MPM write-up for hcs_$add_acl_entries.

Usage

```
declare hcs_$list_acl entry(char(*), char(*), ptr, ptr,
        ptr, fixed bin, fixed bin(35));

call hcs_$list_acl (dirname, ename, area_ptr, area_ret_ptr,
     acl_ptr, acl_count, code)
```

1) dirname — is the directory portion of the path name of the segment in question. (Input)

2) ename — is the entry name portion of the path name of the segment in question. (Input)

3) area_ptr — points to an area into which the list of ACL entries is to be allocated. (Input)

4) area_ret_ptr — points to the start of the allocated list of ACL entries. (Output)

5) acl_ptr — if area_ptr is null then acl_ptr is assumed to point to an ACL structure, segment_acl, into which mode information is to be placed for the access names specified in that same structure. (Input)

6) acl_count — is the number of entries in the ACL structure identified by acl_ptr (Input); or is set to the number of entries in the segment_acl structure allocated in the area pointed to by area_ptr, if area_ptr is not null. (Output)

7) code — is a standard status code. (Output)

Note

If acl_ptr is used to obtain modes for specified access names (rather than obtaining modes for all access names on a segment), then each ACL entry will either have a zero code and will contain the segment's mode or will have code set to error_table_$user_not_found and will contain a zero mode.

Subroutine Call
2/13/73

Name: hcs_$list_dir_acl

This subroutine is used to either list the entire Access Control List (ACL) of a directory or to return the access modes for specified entries. The dir_acl structure described in hcs_$add_dir_acl_entries is used by this subroutine.

Usage

```
declare hcs_$list_dir_acl entry (char(*), char(*), ptr,
        ptr, ptr, fixed bin, fixed bin(35));

call hcs_$list_dir_acl (dirname, ename, area_ptr,
     area_ret_ptr, acl_ptr, acl_count, code);
```

1) dirname — is the path name of the directory superior to the one in question. (Input)

2) ename — is the entry name of the directory in question. (Input)

3) area_ptr — points to an area into which the list of ACL entries is to be allocated. (Input)

4) area_ret_ptr — points to the start of the list of the ACL entries. (Output)

5) acl_ptr — if area_ptr is null then acl_ptr is assumed to point to an ACL structure, dir_acl, into which mode information is to be placed for the access names specified in that same structure. (Input)

6) acl_count — is either the number of entries in the ACL structure identified by acl_ptr (Input); or if area_ptr is not null, then it is set to the number of entries in the dir_acl structure that has been allocated. (Output)

7) code — is a standard status code. (Output)

Note

If acl_ptr is used to obtain modes for specified access names (rather than obtaining modes for all access names on a segment), then each ACL entry will either have a zero code and will contain the directory's mode or will have code set to error_table_$user_not_found and will contain a zero mode.

 (END)

Subroutine Call
Standard Service System
1/17/72

Name: hcs_$make_ptr

This entry, when given a segment name and an entry point name, returns a pointer to a segment entry point. It uses the search rules to find the required segment.

Usage

```
declare hcs_$make_ptr entry (ptr, char(*), char(*), ptr,
        fixed bin);

call hcs_$make_ptr (caller_ptr, seg_name, entry_point_name,
     entry_point_ptr, error_code);
```

1) caller_ptr is a pointer to the "calling procedure" (see Notes below). (Input)

2) seg_name is the name of segment to be located. (Input)

3) entry_point_name is the name of the entry point to be located. (Input)

4) entry_point_ptr is the pointer to the segment entry point specified by seg_name and entry_point_name. (Output)

5) error_code is the returned error code. (Output)

Notes

The directory in which the procedure pointed to by caller_ptr is located is used as the calling directory for the standard search rules. If it is null, then rule 1 of the standard search rules is skipped. See the MPM Reference Guide section on The System Libraries and Search Rules. The standard usage is to have caller_ptr null.

The seg_name and entry_point_name arguments are nonvarying character strings of length $\leq$ 32. They need not be aligned and may be blank padded.

If a null string is given for the entry_point_name argument, then a pointer to the base of the segment is returned. In either case, the segment seg_name is made known to the process with the reference name seg_name. If an error was encountered upon return, the entry_point_ptr argument is null and a nonzero error code is given.

To invoke the procedure entry point pointed to by entry_point_ptr, use cu_$gen_call or cu_$ptr_call. (See cu_ in the MPM.)

Example

The following PL/I statements will generate a call to the procedure fred$foo passing as arguments the integer 17, the pointer p, and the character string "treat".

```
call hcs_$make_ptr (null, "fred", "foo", ep, code);

call cu_$ptr_call (ep, 17, p, "treat");
```


(END)

Subroutine Call
Standard Service System
2/16/72

Name: hcs_$make_seg

This procedure creates a segment in a specified directory with a specified entry name. Once the segment is created, it is made known to the process by a call to hcs_$initiate and a pointer to the segment is returned to the caller. If the segment already exists, an error code is returned. However, a pointer to the segment is still returned.

Usage

```
declare hcs_$make_seg entry (char(*), char(*), char(*),
        fixed bin(5), ptr, fixed bin);

call hcs_$make_seg (dirname, entry, rname, mode
     segptr, code);
```

1) dirname is the directory in which to create the segment. (Input)

2) entry is the entry name of the segment to be created. (Input)

3) rname is the desired reference name, or "". (Input)

4) mode specifies the mode for this user. See Notes in hcs_$append_branch for more information on modes. (Input)

5) segptr is a pointer to the created segment. (Output)

6) code is a standard file system status code. (Output)

Notes

If dirname is null, the process directory is used. If the entry argument is null, a unique name is provided. The rname argument is passed directly to hcs_$initiate and should normally be null.

See also the MPM Reference Guide section on Constructing and Interpreting Names.

(END)

Subroutine Call
2/13/73

Name: hcs_$replace_acl

This subroutine replaces an entire Access Control List (ACL) for a segment with a user-provided ACL, and can optionally add an entry for *.SysDaemon.* with mode rw to the new ACL. The segment_acl structure described in hcs_$add_acl_entries is used by this subroutine.

Usage

```
declare hcs_$replace_acl entry (char(*), char(*), ptr,
        fixed bin, bit(1), fixed bin(35));

call hcs_$replace_acl (dirname, ename, acl_ptr, acl_count,
     no_sysdaemon_sw, code);
```

1) dirname — is the directory portion of the path name of the segment in question. (Input)

2) ename — is the entry name portion of the path name of the segment in question. (Input)

3) acl_ptr — points to the user supplied segment_acl structure that is to replace the current ACL. (Input)

4) acl_count — is the number of entries in the segment_acl structure. (Input)

5) no_sysdaemon_sw — if "0"b, then a *.SysDaemon.* rw entry will be put on the segment's ACL after the existing ACL has been deleted and before the user supplied segment_acl entries are added; if "1"b, then only the user-supplied segment_acl will replace the existing ACL. (Input)

6) code — is a standard status code. (Output)

Notes

If acl_count is zero then the existing ACL will be deleted and only the action indicated by no_sysdaemon_sw will be performed (if any). In the case when acl_count is greater than zero, processing of the segment_acl entries is performed top to bottom, allowing later entries to overwrite previous ones if the access_name parts are identical.

If the segment is a gate (see the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings)) and if the validation level is greater than Ring 1, and only access names that contain the same project as the user, and "SysDaemon" and "sys_control" projects will be allowed. If the replacement ACL is in error then no processing will be performed and the code error_table_$invalid_project_for_gate will be returned.

 (END)

Subroutine Call
2/15/73

Name: hcs_$replace_dir_acl

This subroutine replaces an entire Access Control List (ACL) for a directory with a user-provided ACL, and can optionally add an entry for *.SysDaemon.* with mode sma to the new ACL. The dir_acl structure described in hcs_$add_dir_acl_entries is used by this subroutine.

Usage

```
declare hcs_$replace_dir_acl entry (char(*), char(*), ptr,
        fixed bin, bit(1), fixed bin(35));

call hcs_$replace_dir_acl (dirname, ename, acl_ptr,
     acl_count, no_sysdaemon_sw, code);
```

1) dirname — is the path name of the directory superior to the one in question. (Input)

2) ename — is the entry name of the directory in question. (Input)

3) acl_ptr — points to a user-supplied dir_acl structure that is to replace the current ACL. (Input)

4) acl_count — is the number of entries in the dir_acl structure. (Input)

5) no_sysdaemon_sw — if "0"b, then a *.SysDaemon.* sma entry will be put on the directory's ACL after the existing ACL has been deleted and before the user-supplied dir_acl entries are added; if "1"b, then only the user-supplied dir_acl will replace the existing ACL. (Input)

6) code — is a standard status code. (Output)

Note

If acl_count is zero then the existing ACL will be deleted and only the action indicated by no_sysdaemon_sw will be performed (if any). In the case when acl_count is greater than zero, processing of the dir_acl entries is performed top to bottom, allowing later entries to overwrite previous ones if the access_name parts are identical.

 (END)

Subroutine Call
3/19/73

Name: hcs_$set_bc

This subroutine sets the bit count of a segment in the storage system, given a path name. It also sets that segment's bit count author to be the user who called it.

Usage

```
declare hcs_$set_bc entry (char(*), char(*),
        fixed bin(24), fixed bin(35));

call hcs_$set_bc (dirname, ename, bit_count, code);
```

1) dirname is the directory name of the segment whose bit count is to be changed. (Input)

2) ename is the entry name of the segment whose bit count is to be changed. (Input)

3) bit_count is the new bit count of the segment. (Input)

4) code is a standard storage system status code. (Output)

Notes

The user must have write permission with respect to the segment, but does not need write permission with respect to the parent directory.

The subroutine hcs_$set_bc_seg performs the same function, when a pointer to the segment is provided rather than a path name.

 (END)

Subroutine Call
3/19/73

Name: hcs_$set_bc_seg

This subroutine sets the bit count of a segment in the storage system, given the pointer to the segment. It also sets that segment's bit count author to be the user who called it.

Usage

```
declare hcs_$set_bc_seg entry (ptr, fixed bin(24),
        fixed bin(35));

call hcs_$set_bc_seg (segptr, bitcount, code);
```

1) segptr is a pointer to the segment whose bit count is to be changed. (Input)

2) bitcount is the new bit count of the segment. (Input)

3) code is a standard storage system status code. (Output)

Note

The user must have write permission with respect to the segment, but does not need write permission with respect to the parent directory.

The subroutine hcs_$set_bc performs the same function, when provided with a path name of a segment rather than the pointer.

 (END)

Name: hcs_$star_

This subroutine is the star convention handler for the storage system. (See The Star Convention in the MPM Reference Guide section, Constructing and Interpreting Names.) It is called with a directory name, and an entry name containing components which may be * or **. The directory is searched for all entries which match the given entry name. Information about these entries is returned in a structure. If the entry name is **, information on all entries in the directory is returned.

Status permission is required with respect to the directory to be searched.

The main entry returns the storage system type and all names which match the given entry name. (See hcs_$star_list_ below to obtain more information about each entry.)

Usage

```
declare hcs_$star_ entry (char(*), char(*), fixed bin(2),
        ptr, fixed bin, ptr, ptr, fixed bin(35));

call hcs_$star_ (dirname, star_name, select_sw,
     areap, ecount, eptr, nptr, code);
```

1) dirname is the path name of the directory to be searched. (Input)

2) star_name is the entry name which may contain asterisks. (Input)

3) select_sw = 1 if information is to be returned about link entries only;

= 2 if information is to be returned about segment entries only;

= 3 if information is to be returned about all entries. (Input)

4) areap is a pointer to the area in which information is to be returned. If the pointer is null, ecount is set to the total number of selected entries. See Notes immediately below. (Input)

5) ecount is a count of the number of entries which match the entry name. (Output)

6) eptr is a pointer to the allocated structure in which information on each entry is returned. (Output)

7) nptr is a pointer to the allocated array of all the entry names in this directory which match star_name. See Notes immediately below. (Output)

8) code is a standard storage system status code. See Status Codes below. (Output)

Notes

Even if areap is null, ecount is set to the total number of entries in the directory which match star_name. The setting of select_sw determines whether ecount is the total number of link entries, the total number of segment entries or the total number of all entries.

If areap is not null, the following structure is allocated in the user-supplied area:

```
declare 1 entries (ecount) aligned based (eptr),
        (2 type bit(2),
         2 nnames bit(16),
         2 nindex bit(18)) unaligned;
```

1) type specifies the storage system type of entry:

0 ("00"b) = link,
1 ("01"b) = nondirectory segment,
2 ("10"b) = directory segment.

2) nnames specifies the number of names for this entry which match star_name.

3) nindex specifies the offset in the array of names (pointed to by nptr) for the first name returned for this entry.

All of the names which are returned for any one entry are stored consecutively in an array of all the names, allocated in the user-specified area. The first name for any one entry begins at the offset nindex in the array.

```
declare names (total_names) char(32) aligned based (nptr);
```

where total_names is the total number of names returned.

It should be noted that the user must provide an area large enough for this subroutine to store the requested information.

Entry: hcs_$star_list_

This entry returns more information about the selected entries.

Usage

```
declare hcs_$star_list_ entry (char(*), char(*),
        fixed bin(3), ptr, fixed bin, fixed bin, ptr, ptr,
        fixed bin(35));

call hcs_$star_list_ (dirname, star_name, select_sw,
     areap, seg_count, link_count, eptr, nptr, code);
```

1) dirname is as above. (Input)

2) star_name is as above. (Input)

3) select_sw =1 if information is to be returned about link entries only;

=2 if information is to be returned about segment entries only;

=3 if information is to be returned about all entries;

=5 if information is to be returned about link entries only, including the path name associated with each link entry;

=7 if information is to be returned about all entries, including the path name associcated with each link entry. (Input)

4) areap is a pointer to the area in which information is to be returned. If the pointer is null, seg_count and link_count are set to the total number of selected entries. See Notes immediately below. (Input)

5) seg_count is a count of the number of segments and directories which match the entry name. (Output)

6) link_count is a count of the number of links which match the entry name. (Output)

7) eptr is as above. (Output)

8) nptr is a pointer to the allocated array in which selected entry names and path names associated with link entries are stored. (Output)

9) code is as above. (Output)

Notes

Even if areap is null, seg_count is set to the total number of segments and directories which match star_name, if information on segments is requested. If information on links is requested, link_count is the total number of links which match star_name.

The following structure is allocated in the user-supplied area, if areap is not null:

```
declare entries (count) bit(144) aligned based (eptr);
```

where count = seg_count + link_count.

For each unit of the array, one of two structures will be found. Which structure should be used may be determined by the type item which is located at the base of each structure. It should be noted that the first three items in each structure are identical to the structure returned by hcs_$star_.

The following structure is used if the entry is a segment or a directory:

```
declare 1 branches aligned based (eptr),
        (2 type bit(2),
         2 nname bit(16),
         2 nindex bit(18),
         2 dtm bit(36),
         2 dtu bit(36),
         2 mode bit(5),
         2 pad bit(13),
         2 records bit(18)) unaligned;
```

1) type is as above.

2) nname is as above.

3) nindex is as above.

4) dtm is the date and time the segment or directory was last modified.

5) dtu is the date and time the segment or directory was last used.

6) mode is the current user's access to the segment or directory. See the MPM write-up of hcs_$append_branch for a description of modes.

7) pad is unused space in this structure.

8) records is the number of 1024-word records of secondary storage which have been assigned to the segment or directory.

The following structure is used if the entry is a link:

```
declare 1 links aligned based (eptr),
        (2 type bit(2),
         2 nname bit(16),
         2 nindex bit(18),
         2 dtm bit(36),
         2 dtd bit(36),
         2 pathname_len bit(18),
         2 pathname_index bit(18)) unaligned;
```

1) type is as above.

2) nname is as above.

3) nindex is as above.

4) dtm is the date and time the link entry was last modified.

5) dtd is the date and time the link entry was last dumped.

6) pathname_len is the number of significant characters in the pathname assocated with the link.

7) pathname_index is the offset in the array of names for the link pathname. See below.

If the path name associated with each link entry was requested, the path name will be placed in the names array and will occupy six units of this array. The offset of the first unit is specified by pathname_index in the links array. The length of the path name is given by pathname_len in the links array.

Status Codes

If no match with star_name was found in the directory, code will be returned as error_table_$nomatch.

If star_name contained illegal syntax with respect to the star convention, code will be returned as error_table_$badstar.

If the user did not provide enough space in the area to return all the requested information, code will be returned as error_table_$notalloc. In this case the total number of entries (for hcs_$star_) or the total number of segments and the total number of links (for hcs_$star_list_) will be returned, to provide an estimate of the space required.

 (END)

Name: hcs_$status_

This subroutine consists of a number of hardcore, user-callable, storage system entry points which return various items of information about a specified hierarchy entry.

The main entry point (hcs_$status_) returns the most often needed information about a specified entry. (See hcs_$status_long below.)

Usage

```
declare hcs_$status_ entry (char(*), char(*), fixed bin(1),
        ptr, ptr, fixed bin(35));

call hcs_$status_ (dirname, entry, chase, eptr, nareap,
     code);
```

1) dirname is the directory portion of the path name of the entry in question. (Input)

2) entry is the entry name portion of the path name of the entry in question. (Input)

3) chase
=0: if the entry is a link, return link information;
=1: if the entry is a link, return information about the entry to which it points. (Input)

4) eptr is a pointer to the structure in which information is returned. See Notes immediately below. (Input)

5) nareap is a pointer to the area in which names are returned. If the pointer is null, no names are returned. See Notes immediately below. (Input)

6) code is a storage system status code. See Access Requirements below. (Output)

Notes

The argument eptr points to the following structure if the entry is a segment or directory:

```
declare 1 branch based (eptr) aligned,
        (2 type bit(2),
         2 nnames bit(16),
         2 nrp bit(18),
         2 dtm bit(36),
         2 dtu bit(36),
         2 mode bit(5),
         2 pad1 bit(13),
         2 records bit(18)) unaligned;
```

1) type specifies the type of entry:

```
0 ("00"b) = link;
1 ("01"b) = segment;
2 ("10"b) = directory.
```

2) nnames specifies the number of names for this entry.

3) nrp is a relative pointer (relative to the base of the segment containing the user-specified free storage area) to an array of names.

4) dtm contains the date and time the segment was last modified.

5) dtu contains the date and time the segment was last used.

6) mode contains the mode of the segment with respect to the current user. See the MPM write-up of hcs_$append_branch for a description of modes.

7) pad1 is unused space in this structure.

8) records contains the number of 1024-word records of secondary storage which has been assigned to the segment.

The argument eptr points to the following structure if the entry is a link:

```
declare 1 link based (eptr) aligned,
        (2 type bit(2),
         2 nnames bit(16),
         2 nrp bit(18),
         2 dtem bit(36),
         2 dtd bit(36),
         2 pnl bit(18),
         2 pnrp bit(18)) unaligned;
```

1) type as above.

2) nnames as above.

3) nrp as above.

4) dtem contains the date and time the link was last modified.

5) dtd contains the date and time the link was last dumped.

6) pnl specified the length in characters of the link path name.

7) pnrp is a relative pointer (relative to the base of the segment containing the user-specified free storage area) to the link path name.

Note that the user must provide the storage space required by the above structures. The status entry point merely fills them in.

If nareap is not null, entry names are returned in the following structure allocated in the user-specified area.

```
declare names (nnames) char(32) aligned based (np);
```

where np = ptr (nareap, eptr->entry.nrp).

The first name in this array is defined as the primary name of the entry.

Link path names are returned in the following structure allocated in the user-specified area.

```
declare pathname char(pnl) aligned based (lp);
```

where lp = ptr (nareap, eptr->link.nrp);

Note that the user allocates the area and it must be large enough to accommodate a reasonable number of names.

Access Requirements

The user must have status permission on the parent directory to obtain complete information.

If the user lacks status permission but does have non-null access to a segment, the following per-segment attributes may be returned: type, effective access, bit count, records and current length. In this instance if the entry point hcs_$status_ or hcs_$status_long is called, the status code error_table_$no_s_permission is returned to indicate that incomplete information has been returned.

Entry: hcs_$status_minf

This subroutine returns the bit count and entry type given a directory and entry name. The access required to use this subroutine is status permission on the directory or non-null access to the entry.

Usage

```
declare hcs_$status_minf entry (char(*), char(*), fixed
        bin(1), fixed bin(2), fixed bin(24), fixed bin(35));

call hcs_$status_minf (dirname, entry, chase, type, bitcnt,
     code);
```

1) dirname is as above. (Input)

2) entry is as above. (Input)

3) chase is as above. (Input)

4) type specifies the type of entry:

   0 = link;
   1 = segment;
   2 = directory. (Output)

5) bitcnt is the bit count. (Output)

6) code is as above. (Output)

Entry: hcs_$status_mins

This subroutine returns the bit count and entry type given a pointer to the segment. The access required to use this subroutine is status permission on the directory or non-null access on the segment.

Usage

```
declare hcs_$status_mins entry (ptr, fixed bin(2),
        fixed bin(24), fixed bin(35)):
```

```
call hcs_$status_mins (segptr, type, bitcnt, code);
```

1) segptr is a pointer to the segment about which information is desired. (Input)

2) type is as above. (Output)

3) bitcnt is as above. (Output)

4) code is as above. (Output)

Entry: hcs_$status_long

This subroutine returns most user-accessible information about a specified entry. The access required to use this subroutine is the same as that required by hcs_$status_ and described in Access Requirements above.

Usage

```
declare hcs_$status_long entry (char(*), char(*),
        fixed bin(1), ptr, ptr, fixed bin(35));

call hcs_$status_long (dirname, entry, chase, eptr, nareap,
     code);
```

Arguments are as above.

Notes

The argument eptr points to the same structure as before if the entry is a link. It points to the following structure if the entry is a segment or directory:

```
declare 1 branch based (eptr) aligned,
         (2 type bit(2),
          2 nnames bit(16),
          2 nrp bit(18),
          2 dtm bit(36),
          2 dtu bit(36),
          2 mode bit(5),
          2 pad1 bit(13),
          2 records bit(18),
          2 dtd bit(36),
          2 dtem bit(36),
          2 pad2 bit(36),
          2 curlen bit(12),
          2 bitcnt bit(24),
```

```
        2 did bit(4),
        2 pad3 bit(4),
        2 copysw bit(9),
        2 pad4 bit(9),
        2 rbs (0:2) bit(6),
        2 uid but(36)) unaligned;
```

1-8) are as described above in the structure for segments and directories returned by hcs_$status_.

9) dtd is the date and time the segment was last dumped.

10) dtem is the date and time the branch was last modified.

11) pad2 is unused space in this structure.

12) curlen is the current length of the segment in units of 1024-word records.

13) bitcnt is the bit count associated with the segment.

14) did specifies the secondary storage device (if any) on which the segment currently resides.

15) pad3 is unused space in this structure.

16) copysw contains the setting of the segment copy switch.

17) pad4 is unused space in this structure.

18) rbs contains the ring brackets of the segment.

19) uid is the segment unique identifier.

 (END)

Subroutine Call
2/20/73

Name: hcs_$terminate_file

This subroutine, given the path name of a segment in the current process, removes all the reference names of that segment and then removes the segment from the address space of the process. For a discussion of reference names, see the MPM Reference Guide section, Contructing and Interpreting Names.

Usage

```
declare hcs_$terminate_file entry (char(*), char(*),
        fixed bin(1), fixed bin(35));

call hcs_$terminate_file (dir_name, entry_name, rsw, code);
```

1) dir_name is the directory portion of the path name of the segment in question. (Input)

2) entry_name is the entry name portion of the path name of the segment in question. (Input)

3) rsw is the reserved segment switch. If equal to 1, the segment number should be saved in the reserved segment list; if equal to 0, the segment number should not be saved. (Input)

4) code is a standard storage system status code. (Output)

Notes

The subroutine hcs_$terminate_seg performs the same operation given a pointer to a segment instead of a path name; hcs_$terminate_name and hcs_$terminate_noname terminate a single reference name.

The subroutine term_ performs the same operation as hcs_$terminate_file.

In fact, only those reference names are removed for which the ring level associated with the name is greater than or equal to the validation level of the process. If the user needs to concern himself with rings, he should refer to the MPM Subsystem Writers' Guide section, Intraprocess Access Control (Rings).

 (END)

Subroutine Call
2/20/73

Name: hcs_$terminate_name

This subroutine terminates one reference name from a segment. If it is the only reference name for that segment, the segment is removed from the address space of the process. For a discussion of reference names see the MPM Reference Guide section, Constructing and Interpreting Names.

Usage

```
declare hcs_$terminate_name entry (char(*), fixed bin(35));

call hcs_$terminate_name (ref_name, code);
```

1) ref_name is the reference name to be terminated. (Input)

2) code is a standard storage system status code. (Output)

Note

The subroutine hcs_$terminate_noname terminates a null reference name from a specified segment; hcs_$terminate_file and hcs_$terminate_seg completely terminate a segment given its path name or segment number, respectively.

The subroutine term_$single_refname performs the same operation as hcs_$terminate_name.

 (END)

Name: hcs_$terminate_noname

This subroutine terminates a null reference name from the specified segment. If this is the segment's only reference name, the segment is removed from the address space of the process. This entry is used to clean up after initiating a segment by a null name; see also the MPM write-up for hcs_$initiate. For a discussion of reference names, see the MPM Reference Guide section, Constructing and Interpreting Names.

Usage

```
declare hcs_$terminate_noname entry (ptr, fixed bin(35));

call hcs_$terminate_noname (segptr, code);
```

1) segptr is a pointer to the segment in question. (Input)

2) code is a standard storage system status code. (Output)

Note

The subroutine hcs_$terminate_name terminates a specified non-null reference name; hcs_$terminate_file and hcs_$terminate_seg completely terminate a segment given its path name or segment number, respectively.

 (END)

Subroutine Call
2/20/73

Name: hcs_$terminate_seg

This subroutine, given a pointer to a segment in the current process, removes all the reference names of that segment and then removes the segment from the address space of the process. For a discussion of reference names, see the MPM Reference Guide section, Constructing and Interpreting Names.

Usage

```
declare hcs_$terminate_seg entry (ptr, fixed bin(1),
        fixed bin(35));

call hcs_$terminate_seg (segptr, rsw, code);
```

1) segptr is a pointer to the segment to be terminated. (Input)

2) rsw is the reserved segment switch. If equal to 1, the segment number should be saved in the reserved segment list; if equal to 0, the segment number should not be saved. (Input)

3) code is a standard storage system status code. (Output)

Notes

The subroutine hcs_$terminate_file performs the same operation given the path name of a segment instead of a pointer; hcs_$terminate_name and hcs_$terminate_noname terminate a single reference name.

The subroutine term_$segptr performs the same operation as hcs_$terminate_seg.

In fact, only those reference names are removed for which the ring level associated with the name is greater than or equal to the validation level of the process. If the user needs to concern himself with rings, he should refer to the MPM Subsystem Writer's Guide section, Intraprocess Access Control (Rings).

 (END)

Subroutine Call
3/19/73

Name: hcs_$truncate_file

This subroutine, given a pathname, truncates a segment to a specified length. If the segment is already shorter than the specified length, no truncation is done. The effect of truncating a segment is to store zeros in the words beyond the specified length.

Usage

```
declare hcs_$truncate_file entry (char(*), char(*),
        fixed bin, fixed bin(35));

call hcs_$truncate_file (dirname, ename, length, code);
```

1) dirname is the directory portion of the path name of the segment in question. (Input)

2) ename is the entry portion of the path name of the segment in question. (Input)

3) length is the new length (decimal) of the segment in words. (Input)

4) code is a standard storage system error code. (Output)

Notes

The subroutine hcs_$truncate_seg performs the same function when given a pointer to the segment instead of the path name. See also the restrictions discussed in that write-up under Notes.

 (END)

Subroutine Call
3/19/73

Name: hcs_$truncate_seg

This subroutine, given a pointer, truncates a segment to a specified length. If the segment is already shorter than the specified length, no truncation is done. The effect of truncating a segment is to store zeros in the words beyond the specified length.

Usage

```
declare hcs_$truncate_seg entry (ptr, fixed bin,
        fixed bin(35));

call hcs_$truncate_seg (segptr, length, code);
```

1) segptr is a pointer to the segment to be truncated. Only the segment number portion of the pointer is used. (Input)

2) length is the new length (decimal) of the segment in words. (Input)

3) code is a standard storage system status code. (Output)

Notes

The write attribute is required with respect to the segment.

A directory may not be truncated.

The implementation is such that pages will be thrown away starting from the next page after the word number length and the remainder of the last page will be zeroed.

The subroutine hcs_$truncate_file performs the same function when given the pathname of the segment instead of the pointer.

 (END)

Subroutine Call
7/11/73

Name: ioa_

This procedure is used for formatting, according to a control string (see Notes), character strings, fixed binary numbers, floating numbers, and pointers into complete character string form. Entry points are provided that write the formatted string out on the stream "user_output", on a specified stream, or which return the expanded string. The expanded string cannot exceed 256 characters when the line is to be written out.

Since this procedure can be called with a varying number of arguments, it is not permissible to include a parameter attribute list in the declaration of the various entry points.

Entry: ioa_

The ioa_ entry reformats the input data and writes the resultant character string out on the stream "user_output" with a new line character added at the end.

Usage

```
declare ioa_ entry options (variable);

call ioa_ (control_string, arg1, ..., argn);
```

1) control_string — is a character string (char(*)) specifying the output format for the data. See Notes below. (Input)

2) argi — is the variable to be placed in the ith argument position of the control string. (Input)

Entry: ioa_$nnl

This entry is identical to ioa_ except that no new line character is added.

Usage

```
declare ioa_$nnl entry options (variable);

call ioa_$nnl (control_string, arg1, ..., argn);
```

Arguments are as above.

Entry: ioa_$ioa_stream

This entry reformats the input data, appends a new line character, and writes the resultant character string out on the specified output stream.

Usage

```
declare ioa_$ioa_stream entry options (variable);

call ioa_$ioa_stream (stream_name, control_string,
     arg1, ..., argn);
```

1) stream_name is the name (char(*)) of the output stream desired. (Input)

2) is as above. (Input)

3) is as above. (Input)

Entry: ioa_$ioa_stream_nnl

This entry is identical to ioa_$ioa_stream except that no new line character is added.

Usage

```
declare ioa_$ioa_stream_nnl entry options (variable);

call ioa_$ioa_stream_nnl (stream_name, control_string,
     arg1, ..., argn);
```

Arguments are as above.

Entry: ioa_$rs

This entry reformats the data but instead of writing the string out, it returns the new expanded string as well as the significant length of the new string, in characters. Note that the returned string argument should be declared large enough to allow for expansion extremes. It can be a varying or nonvarying string, aligned or unaligned. If it is nonvarying, it is padded with blanks if the expanded string does not completely fill it, unless the entries ioa_$rsnp or ioa_$rsnpnnl were called (see below). Expansion stops when the end of the returned string argument is reached.

Usage

```
declare ioa_$rs entry options (variable);

call ioa_$rs (control_string, return_string, length,
     arg1, ..., argn);
```

1) control_string is as above. (Input)

2) return_string is the returned formatted string (char(*)). See Notes below. (Output)

3) length is the number (fixed bin) of significant characters in return_string. (Output)

4) argi is as above. (Input)

Entry: ioa_$rsnnl

This entry is identical to ioa_$rs except that no new line character is added at the end.

Usage

```
declare ioa_$rsnnl entry options (variable);

call ioa_$rsnnl (control_string, return_string, len,
     arg1, ..., argn);
```

Arguments are as above.

Entry: ioa_$rsnp

This entry is identical to ioa_$rs except that the end of return_string is not padded with blanks.

Usage

```
declare ioa_$rsnp entry options (variable);

call ioa_$rsnp (control_string, return_string, len,
     arg1, ..., argn);
```

Arguments are as above.

Entry: ioa_$rsnpnnl

This entry is identical to ioa_$rsnp except no new line character is added at the end.

Usage

```
declare ioa_$rsnpnnl entry options (variable);

call ioa_$rsnpnnl (control_string, return_string, len,
     arg1, ..., argn);
```

Arguments are as above.

Notes

All entries require a control string argument. This is a character string which may or may not contain control characters within it. If no control characters occur, the string is merely returned at the rs entry points or written out at the other entry points. If control characters exist, they govern the conversion of successive additional arguments which are expanded into the appropriate characters and inserted into the resultant string. The control characters are indicated by the circumflex character (^) and may take an optional field width (n or d). The possibilities are:

^d	or	^nd	decimal
^o	or	^no	octal
^f	or or or	^n.df ^nf ^.df	floating
^e	or	^de	exponential
^a	or	^na	ASCII
^p			pointer
^w	or	^nw	full word octal
^|			insert form feed (new page) character
^/	or	^n/	insert new line character

^-	or ^n-	insert horizontal tab
^x	or ^nx	insert blank character
^A		acc string
^R		insert red ribbon shift character
^B		insert black ribbon shift character
^^		insert circumflex character

where n expresses the field width. n can be a decimal integer constant or the letter v. If n is the letter v, the next argi specifies the field width and the following argument, arg(i+1), is the datum to be converted, if required. d is a decimal integer specifying the number of digits to the right of the decimal point.

^d assumes a fixed binary input and converts this to decimal. If a number appears between the circumflex and the d, the decimal character string is placed right justified in a field as wide as the specified number and padded with blanks.

^o assumes a fixed binary input and converts it to octal form. It is similar to d.

^f assumes a floating input. The following cases can occur:

^f An attempt is made to output nine significant digits. If the field width exceeds 13, the number is converted to exponential form. The field does not contain any blanks.

^nf The decimal point is placed at the extreme right in the field and no fraction part appears. High-order zeros are converted into blanks.

^.df The minimum field width is d+2 to accommodate a fraction part d digits long and the decimal point. If necessary, the field is extended to accommodate the integer part or minus sign or both.

^n.df If the number cannot be accommodated in this field, it is converted to exponential form. High-order zeroes are converted into blanks.

^e assumes a floating input. d, if not specified, is assumed to be eight. The number is converted to the following format:

$$\pm.x_1 x_2 \text{------} x_d \; e\pm y_1 y_2$$

^a (ASCII) assumes a varying or nonvarying character string as input. If there is no field width, trailing blanks are stripped. If the field width is specified, the string is left adjusted within the field and padded with trailing blanks up to the specified field width. If the length of the string after trailing blanks have been removed is greater than the specified field size, the field size is ignored.

^p assumes a pointer as input and expands it to a character string of the form "276|13640". If a bit offset is present, it is printed as a decimal number immediately following the word offset in the form "276|13640(27)". No field width option is accepted.

^w (word) assumes a fixed binary number as input and converts it to a 12-character octal number padded on the left with leading zeros. The field width is accepted, but is ignored if less than 12.

^| causes a form feed (new page) character to be inserted into the expanded string. No field width is accepted.

^/ causes a new line character to be inserted into the expanded string. n specifies repetition.

^- causes a horizontal tab character to be inserted into the expanded line. n specifies repetition.

^x causes a blank character to be inserted into the expanded line. n specifies repetition.

^A assumes that the current argument is a pointer to a character string in acc (ASCII Character with Count) format. The character string is inserted into the expanded line. No field width is accepted.

^^ causes a circumflex character ( ) to be inserted into the string.

^R causes a red ribbon shift character to be inserted into the string.

^B causes a black ribbon shift character to be inserted into the string.

Any character other than those cited following a circumflex is ignored. To write a bit string out, it should first be converted to fixed mode and then an octal format used. If a number does not fit in a specified field width, the field width is expanded so that the complete number is printed.

If no arguments remain to be converted, the circumflex is merely copied into the output field.

Examples

1) call ioa_ ("This is ^a the third of ^a", "Mon", "July");

Result: This is Mon the third of July.

2) call ioa_ ("date ^d/^d/^d, time ^d:^d", 6, 20, 69,
2014, 36);

Result: date 6/20/69, time 2014:36

3) call ioa_ ("overflow at ^p", ptr);

Result: overflow at 271|4671

4) call ioa_ (^6o^14w^14w^14w", no, word1, word2, word3);

Result: 014100 014114214300 00000014000 111000101104

5) call ioa_ ("^a");

Result: a

6) call ioa_ ("a=^f^x^.3f", a, a);

Result: a=123.456789 123.457

Name: ios_

This write-up describes the generalized I/O system function calls. The reader is cautioned that the descriptions contained in this document state the general purpose of each call as put forth in the I/O system specifications and that the result of a particular call to a device is dependent on that device and its associated software. However, any deviations of a particular device from the specifications stated here are enumerated in the MPM write-up describing the software for that device. These descriptions are labeled I/O System Interface Modules (IOSIMs). The reader should also refer to the specific IOSIM write-ups to see which I/O system function calls are implemented for that device since only a limited number of these calls are usually implemented for each device. Users should also see the MPM Reference Guide section, Input and Output Facilities, for further information.

Generic Arguments

Rather than reproduce the descriptions of arguments that are common to several function calls under the description of each function call, they are given here.

1) stream_name	is a string of 32 characters or less that identifies the stream upon which this call is to be performed. A stream usually identifies a particular device and the control software, the Device Interface Module (DIM), for that device. (Input)
2) type	is a string of 32 characters or less that identifies the control software, the DIM, for a type of device. A list of system supported types is given in the MPM Reference Guide section, Available Input and Output Facilities, and each type is described in the MPM Subroutines section as an IOSIM. (Input)
3) device/stream_name	is the identifier of a particular device, pseudodevice, or stream name upon which an I/O operation can be performed. (Input)
4) mode	describes characteristics related to an attachment (e.g., readable, writeable,

etc.). The modes permitted for a particular device type are described in the IOSIM write-up for that device type. (Input)

5) status — is a bit string returned by an I/O call containing information about the success of that call. (See the MPM Reference Guide section, Use of the Input and Output System.) This bit string must be aligned. (Output)

6) disposal — indicates special action to be taken when a device is detached. (Input)

7) gmode — is a string of 128 characters or less containing an encoding of the mode of an attachment. The addition of new modes in the future could make the maximum length of gmode even greater. (Output)

8) smode — is the synchronization mode. The mode can be synchronous ("s") or asynchronous ("a"). (Input)

9) limit — is the maximum number of elements of write-behind or read-ahead data permissible in asynchronous mode. (Input)

10) oldstatus — is the status string of a previous transaction. This string must be aligned. (Input)

11) request — is a special request to the I/O system. The requests appropriate to a device type are described in the IOSIM write-up for that device type. (Input)

12) argptr — is a pointer to a data structure containing information relevant to a special request. (Input)

13) element_size — is the current element size (i.e., the number of bits in an element) for read or write calls. (Input/Output)

14) workspace — is a pointer to the buffer space for data to be read into or written from. The buffer space must be aligned.

(Input)

15) offset — is the number of elements from the beginning of the workspace at which to start reading or writing data. (Input)

16) nelem — is the number of elements requested to be read or written. (Input)

17) nelemt — is the number of elements actually read or written. (Output)

18) nbreaks — is the number of break characters in breaklist. (Input/Output)

19) breaklist — is an array of break characters. This is an unaligned bit string array each element of which is element_size bits long. (Input/Output)

20) nreads — is the number of read delimiters in readlist. (Input/Output)

21) readlist — is an array of read delimiters. This is an unaligned bit string array each element of which is element_size bits long. (Input/Output)

22) pointer_name_1 — is a string of 32 characters or less specifying a reference pointer that identifies a particular position in the data referred to by the stream (e.g., the write pointer points to the next element to be written). For a description of reference pointers, see the MPM Reference Guide section, Use of the Input and Output System. (Input/Output)

23) pointer_name_2 — is the same as pointer_name_1. (Input/Output)

Entry: ios_$write_ptr

The write_ptr call is a specialized form of the write call (see below). The number of elements specified by nelem in the buffer area pointed to by workspace starting at the element specified by offset is written on the stream "user_output". Since the stream "user_output" is normally associated with the

user's terminal, a write_ptr call usually results in the specified elements being typed on the user's terminal. The write_ptr call should be used in preference to the write call when writing on the stream "user_output" due to the greater efficiency of the write_ptr call.

Usage

```
declare ios_$write_ptr entry (ptr, fixed bin, fixed bin);

call ios_$write_ptr (workspace, offset, nelem);
```

Entry: ios_$read_ptr

The read_ptr call is a specialized form of the read call (see below). The number of elements specified by nelem are attempted to be read from the stream "user_input" into the buffer area pointed to by workspace. If a read delimiter is encountered as one of the elements being read, reading ceases with this element. Therefore, the number of elements read in is either nelem or up to the first read delimiter, whichever comes first, and this number is returned in nelemt. Since the stream "user_input" is normally associated with the user's terminal, a read_ptr call usually results in the specified elements being read from the terminal. The read_ptr call should be used in preference to the read call when reading from the stream "user_input" due to the greater efficiency of the read_ptr call.

Usage

```
declare ios_$read_ptr entry (ptr, fixed bin, fixed bin);

call ios_$read_ptr (workspace, nelem, nelemt);
```

Entry: ios_$write

The write call attempts to write from the buffer area pointed to by workspace, starting offset elements from the beginning of the buffer area, the requested number (nelem) of elements onto the stream stream_name. The number of elements actually written is returned in nelemt and indications of the status of the transaction are returned in status.

Usage

```
declare ios_$write entry (char(*), ptr, fixed bin,
        fixed bin, fixed bin, bit(72) aligned);
```

```
call ios_$write (stream_name, workspace, offset, nelem,
     nelemt, status);
```

Entry: ios_$read

The read call attempts to read into the buffer area pointed to by workspace, starting offset elements from the beginning of the buffer area, the requested number (nelem) of elements from the stream stream_name. If a read delimiter is encountered as one of the elements being read, reading ceases with this element. Therefore, the number of elements read in is either nelem or up to the first read delimiter, whichever comes first, and this number is returned in nelemt. Indications of the status of the transaction are returned in the status argument.

Usage

```
declare ios_$read entry (char(*), ptr, fixed bin,
        fixed bin, fixed bin, bit(72) aligned);

call ios_$read (stream_name, workspace, offset, nelem,
     nelemt, status);
```

Entry: ios_$attach

The attach call associates the given stream_name with a device or stream name, device/stream_name, as a particular type (DIM). All subsequent read or write operations performed upon the stream identified by stream_name result in data being transferred from or to the device or stream identified by device/stream_name and this transfer is performed by the DIM (control software) identified by the type argument. This association remains in force until removed by a detach call (see below). The mode argument specifies how attributes of this attachment differ from the default. If mode is "", then the default attributes for the specified device are used.

Usage

```
declare ios_$attach entry (char(*), char(*), char(*),
        char(*), bit(72) aligned);

call ios_$attach (stream_name, type, device/stream_name,
     mode, status);
```

Entry: ios_$detach

The detach call deletes all associations established by an attach call between stream_name and device/stream_name. If

device/stream_name is "", then all associations for stream_name are deleted. The disposal argument indicates any special action to be taken on detachment. If disposal is "", the appropriate default action for the IOSIM involved is taken.

Usage

```
declare ios_$detach entry (char(*), char(*), char(*),
        bit(72) aligned);

call ios_$detach (stream_name, device/stream_name,
     disposal, status);
```

Entry: ios_$resetwrite

The resetwrite call is used to delete unused write-behind data collected by the I/O system as a result of write-behind associated with the stream stream_name.

Usage

```
declare ios_$resetwrite entry (char(*), bit(72) aligned);

call ios_$resetwrite (stream_name, status);
```

Entry: ios_$resetread

The resetread call is used to delete unused read-ahead data collected by the I/O system as a result of read-ahead associated with the stream stream_name.

Usage

```
declare ios_$resetread entry (char(*), bit(72) aligned);

call ios_$resetread (stream_name, status);
```

Entry: ios_$abort

The abort call causes all outstanding transactions on the stream stream_name to be aborted. The oldstatus argument should be set to ""b.

Usage

```
declare ios_$abort entry (char(*), bit(72) aligned,
        bit(72) aligned);

call ios_$abort (stream_name, oldstatus, status);
```

Entry: ios_$order

The order call is used to issue a specialized request on the stream stream_name. The argptr argument points to a data structure containing arguments relevant to the particular request. Allowable requests depend upon the type (the DIM) associated with the steam stream_name. The order requests appropriate to a device type are described in the IOSIM write-up for that device type. The order call is used to perform specialized I/O operations when no generalized I/O call is available. Note that although argptr is an input argument, the structure to which it points can contain both input and output information.

Usage

```
declare ios_$order entry (char(*), char(*), ptr, bit(72)
        aligned);

call ios_$order (stream_name, request, argptr, status);
```

Entry: ios_$getsize

The getsize call returns the current element size associated with the stream stream_name.

Usage

```
declare ios_$getsize entry (char(*), fixed bin,
        bit(72) aligned);

call ios_$getsize (stream_name, element_size, status);
```

Entry: ios_$setsize

The setsize call sets the element size for subsequent calls on the stream stream_name.

Usage

```
declare ios_$setsize entry (char(*), fixed bin,
        bit(72) aligned);

call ios_$setsize (stream_name, element_size, status);
```

Entry: ios_$getdelim

The getdelim call returns the read delimiters and breaks currently in effect on the stream stream_name. The breaks are

returned as an array of elements in breaklist and nbreaks is set to the number of breaks. The read delimiters are returned as an array of elements in readlist and nreads is set to the number of read delimiters. If either array is not large enough to contain all of the returned delimiters or breaks, as many as possible are returned.

Usage

```
declare ios_$getdelim entry (char(*), fixed bin,
        (*) bit (element_size), fixed bin,
        (*) bit (element_size), bit(72) aligned);

call ios_$getdelim (stream_name, nbreaks, breaklist,
     nreads, readlist, status);
```

Entry: ios_$setdelim

The setdelim call establishes elements to delimit data read by subsequent read calls on the stream stream_name. The argument breaklist is an array of breaks (containing nbreak elements), each serving simultaneously as an interrupt, canonicalization and erase-and-kill delimiter. Breaks are meaningful only on character oriented devices. The argument readlist is an array of read delimiters (containing nreads elements). Read delimiters cause subsequent read calls to cease reading at the first read delimiter element. The new delimiters established by this call are in effect until superseded by a subsequent setdelim call.

Usage

```
declare ios_$setdelim entry (char(*), fixed bin,
        (*) bit (element_size), fixed bin,
        (*) bit (element_size), bit(72) aligned);

call ios_$setdelim (stream_name, nbreaks, breaklist,
     nreads, readlist, status);
```

Entry: ios_$seek

The seek call sets the reference pointer specified by pointer_name_1 to the value of the pointer specified by pointer_name_2 plus the value of a signed offset, offset. pointer_name_1 and pointer_name_2 can be "read", "write", "last", or "bound". The read pointer indicates the next element to be read, the write pointer the next element to be written, the first pointer the first element of data associated with this stream, the last pointer the last element of data, and the bound pointer the element beyond which data cannot grow. The seek call is used to truncate; e.g., call ios_$seek (stream_name, "last", "last",

-40, status); or to set the bound of data; e.g., call ios_$seek (stream_name, "bound", "last", 27, status); in addition to its more traditional usage involving the read and write pointer; e.g., call ios_$seek (stream_name, "read", "write", -2, status);. The read and write pointers are also set as a result of read and write calls, respectively. All relative offsets between reference pointers are in terms of numbers of elements.

Usage

```
declare ios_$seek entry (char(*), char(*), char(*),
        fixed bin, bit(72) aligned);

call ios_$seek (stream_name, pointer_name_1,
     pointer_name_2, offset, status);
```

Entry: ios_$tell

The tell call returns the value of the pointer specified by pointer_name_1 as an offset, with respect to pointer_name_2. The arguments pointer_name_1, pointer_name_2, and offset have the same meaning as in the seek call. As an example, the call can be used to obtain the bound of the data by call ios_$tell (stream_name, "bound", "first", offset, status);.

Usage

```
declare ios_$tell entry (char(*), char(*), char(*),
        fixed bin, bit(72) aligned);

call ios_$tell (stream_name, pointer_name_1,
     pointer_name_2, offset, status);
```

Entry: ios_$changemode

The mode of an attachment describes certain characteristics related to the attachment (e.g., readable, writeable, linear, formatted, etc.). The changemode call permits mode changes to be invoked for the given stream stream_name that modify the mode of the attachment. The gmode argument is set to the mode of the attachment prior to this call.

Usage

```
declare ios_$changemode entry (char(*), char(*), char(*),
        bit(72) aligned);

call ios_$changemode (stream_name, mode, gmode, status);
```

Entry: ios_$readsync

For the given stream_name, the readsync call sets the read synchronization mode, smode, of subsequent read calls. This mode is either synchronous or asynchronous. Synchrony implies that the read operation is performed in its entirety during a read call. Asynchrony implies that read-ahead is possible to the extent permitted by the limit argument, which is the desired maximum number of elements which can be read ahead. The default mode is asynchronous.

Usage

```
declare ios_$readsync entry (char(*), char(1), fixed bin,
        bit(72) aligned);

call ios_$readsync (stream_name, smode, limit, status);
```

Entry: ios_$writesync

For the given stream_name, the writesync call sets the write synchronization mode, smode, of subsequent write calls. The mode is either synchronous or asynchronous. Synchrony implies that the write operation is performed in its entirety during a write call. Asynchrony implies that write-behind is possible to the extent permitted by the limit argument, which is the desired maximum number of elements which can be written behind. The default mode is asynchronous.

Usage

```
declare ios_$writesync entry (char(*), char(1), fixed bin,
        bit(72) aligned);

call ios_$writesync (stream_name, smode, limit, status);
```

Subroutine Call
9/28/73

Name: match_star_name_

This procedure implements the Multics storage system star convention by comparing an entry name with a name containing stars or question marks (called a star name). Refer to the MPM Reference Guide section, Constructing and Interpreting Names, for a description of the star convention and a definition of acceptable star name formats.

Usage

```
declare match_star_name_ entry (char(*), char(*),
        fixed bin(35));

call match_star_name_ (entry_name, star_name, code);
```

1) entry_name is the entry name to be compared with the star name. Trailing spaces in the entry name are ignored. (Input)

2) star_name is the star name it is to be compared with. Trailing spaces in the star name are ignored. (Input)

3) code is a status code that can have one of the values:

0 the entry name matches the star name.

error_table_$nomatch
the entry name does not match the star name.

error_table_$badstar
the star name does not have an acceptable format. (Output)

Notes

Refer to the MPM write-up for the hcs_$star_ subroutine to see how to list the directory entries that match a given star name.

Refer to the MPM write-up for the check_star_name_ subroutine to see how to validate a star name.

 (END)

Miscellaneous Call
Standard Service System
09/17/70

Name: move_names_

This procedure moves all the entry names, except the one used to designate the original segment, from one segment to another. Name duplications are handled by nd_handler_.

Usage

```
declare move_names_ entry (char(*), char(*), char(*),
        char(*), char(*), bit(1) aligned, fixed bin(17));

call move_names_ (dir1, en1, dir2, en2, caller, errsw,
     code);
```

1) dir1 is the directory in which the original segment is found. (Input)

2) en1 is a name on the original segment. (Input)

3) dir2 is the target segment's directory. (Input)

4) en2 is a name already on the target segment. (Input)

5) caller is the name of the calling procedure; it is used in calls to nd_handler_. (Input)

6) errsw indicates which segment the error indicated by "code" occurred on; it is set to "0"b if the error was on the original segment and to "1"b if on the target. (Output)

7) code is a standard File System status code.

Note

If a name duplication occurs and the conflicting name is not deleted, then the code "error_table_$namedup" is returned to

the caller. The names that occur after the conflicting name are processed.

(END)

I/O System Interface Module
3/20/73

Name: nstd_

This procedure is an I/O system Device Interface Module (DIM) used to control operation of magnetic tapes. (Note that this is the nonstandard tape DIM; Multics standard tapes are defined by and dealt with through the tape_ DIM.) The subroutine nstd_ is not directly called by a user program. Instead, the user provides the name nstd_ in a call to the I/O system attach entry. He then accomplishes the I/O operations by calling standard I/O system entry points which are independent of the interface module in use. Further information on the I/O system may be found in the MPM Reference Guide sections on Input and Output Facilities. Details on the I/O system call syntax may be found in the module description of ios_. This write-up explains how nstd_ interprets the standard I/O system calls.

Usage

call ios_$attach (stream, "nstd_", reel, mode, status);

1) stream — is the name of the I/O stream to be attached. (Input)

2) "nstd_" — specifies the nonstandard tape DIM. (Input)

3) reel — is the reel identifying message which will be passed on to the operator in the mount message. (Input)

4) mode — is r for reading, and w for writing, or rw for reading and writing. If the mode is r, the mount message will specify no write ring. All other modes will cause the mount message to specify a write ring. (Input)

5) status — is a status indicator. See the section below on Returned Status. (Output)

Permitted I/O System Calls

The following I/O system calls are implemented by this DIM:

attach
changemode
detach
getsize
order
read
write

Order Requests

The following order requests are implemented by this DIM:

binary	set hardware mode to binary (this is the default).
bcd	set hardware mode to binary coded decimal (BCD).
nine	set hardware mode to nine track (the default is seven track).
back	backspace one record.
eof	write end of file (EOF).
saved_status	return last hardware status. The result will be written into the location pointed to by the pointer argument of the ios_$order call as bit(12) aligned.
err_count	set error retry count. This is used to change the error retry count which has an initial value of 10 and which controls the number of times a tape operation encountering an error will be retried before being reflected to the caller. If the pointer argument of the ios_$order call is null, the error retry count will be set to 0 (i.e., errors will be passed directly to the caller with no retry attempts). If the pointer argument is not null, it must point to a fixed bin(17) error count which is nonnegative and less than 100.
request_status	issue a request status command to the tape controller. The status will be written into the location pointed to by the pointer argument of the ios_$order call as bit(12) aligned.
forward_record	forward space one record.
forward_file	forward space to an EOF mark. The tape stops just past the EOF mark.
backspace_file	backward space to an EOF mark. The tape stops just before the EOF mark; i.e., a subsequent read will encounter the EOF mark.

erase	erase tape.
high	set high density (this is the default).
low	set low density.
protect	set write inhibit regardless of the presence of a write permit ring in the tape reel. The tape unit will remain write inhibited until the tape is detached.
unload	rewind tape and unload (done automatically when the tape is detached).
rewind	rewind the tape to the load point.
fixed_record_length	allow the DIM to operate asynchronously, reading up to six physical records at a time. The pointer argument must point to a fixed bin(17) number indicating the record size. Subsequent read and write calls continue to pass one physical record.

Returned Status

The first half of the status string may contain either standard Multics status codes or hardware status. If the latter, the first bit will be 1 and the rightmost 12 bits of the first half of the status string will hold hardware status as described in Table I below.

Detaching

When a tape is detached, it will be rewound and unloaded and the drive will be freed for attachment. No other types of detachment are permitted.

Element Size

Only an element size of thirty-six is permitted.

Buffer Size

The maximum number of words which may be transmitted on a read or write call is 1632.

Status Codes

The following status codes may be returned by nstd_:

error_table_$bad_index
error_table_$buffer_big
error_table_$ionmat
error_table_$no_backspace
error_table_$no_device
error_table_$not_attached
error_table_$undefined_order_request

See also the MPM Reference Guide section, List of System Status Codes and Meanings, for more information.

Notes

All order requests and the changemode call reset the fixed_record_length state after writing the current set of buffers, if any. The changemode call does not reposition the tape; it is the user's responsibility to do so.

Table 1

	Major Status	Substatus
Peripheral Subsystem Ready	000000	
Write Protected		000XX1
Positioned on Leader		000X1X
Nine Track Tape Unit		0001XX
Device Busy	000001	
Device Attention	000010	
Write Inhibit		0XXXX1
No Such Tape Unit		0XXX1X
Tape Unit in Standby		0XX1XX
Tape Unit Check		0X1XXX
Blank Tape on Write		01XXXX
Device Data Alert	000011	
Transfer Timing Alert		000001
Blank Tape on Read		XXXX1X
Transmission Parity Alert		XXX1XX
Lateral Parity Alert		XX1XXX
Longitudinal Parity Alert		X1XXXX
End of Tape (EOT) Mark		1XXXXX
Bit During Erase		XXXX11
End of File	000100	
EOF Mark (Seven Track)		001111
EOF Mark (Nine Track)		011100
Data Alert Condition		111111
Single Character Record		XXXXXX
Command Reject	000101	
Invalid Operation Code		XXXXX1
Invalid Device Code		XXXX1X
Parity on I/O		XXX1XX
Positioned on Leader		XX1XXX
Read After Write		X1XXXX
Nine Track Error		1XXXXX
Program Load Termination	000111	
Peripheral Subsystem Busy	001000	

 (END)

Subroutine Call
Development System
8/21/72

Name: object_info_

This procedure returns structural and identifying information extracted from an object segment. It has three entry points returning progressively larger increments of information. All three entry points have identical calling sequences, the only distinction being the amount of information returned in the info structure described below.

Entry: object_info_$brief

This entry only returns the structural information necessary in order to be able to locate the object's four sections.

Usage

```
declare object_info_$brief entry (ptr, fixed bin(24), ptr,
        fixed bin(35);

call object_info_$brief (segp, bc, infop, code);
```

1) segp is a pointer to the base of the object segment. (Input)

2) bc is the bit count of the object segment. (Input)

3) infop is a pointer to the info structure in which the object information is returned. (Input)

4) code is a standard Multics status code. (Output)

Entry: object_info_$display

This entry returns, in addition to the $brief information, all the identifying data required by certain object display commands, such as print_link_info.

Usage

```
declare object_info_$display entry (ptr, fixed bin(24), ptr,
        fixed bin(35);

call object_info_$display (segp, bc, infop, code);
```

1-4) as above. (Input/Output)

Entry: object_info_$long

This entry returns, in addition to the $brief and $display information, the data required by the Multics binder.

Usage

```
declare object_info_$long entry (ptr, fixed bin(24), ptr,
        fixed bin(35);

call object_info_$long (segp, bc, infop, code);
```

1-4) as above. (Input/Output)

Information Structure

The info structure is as follows:

```
declare 1 info aligned,
          2 version_number fixed bin,
          2 textp ptr,
          2 defp ptr,
          2 linkp ptr,
          2 symbp ptr,
          2 bmapp ptr,
          2 tlng fixed bin,
          2 dlng fixed bin,
          2 llng fixed bin,
          2 slng fixed bin,
          2 blng fixed bin,
          2 format,
            3 old_format bit(1) unaligned,
            3 bound bit(1) unaligned,
            3 relocatable bit(1) unaligned,
            3 procedure bit(1) unaligned,
            3 standard bit(1) unaligned,
            3 gate bit(1) unaligned,
          2 call_delimiter fixed bin,

/*This is the limit of the $brief info structure.*/

          2 compiler char(8) aligned,
          2 compile_time fixed bin(71),
          2 userid char(32) aligned,
          2 cvers aligned,
            3 offset bit(18) unaligned,
            3 length bit(18) unaligned,
          2 comment,
            3 offset bit(18) unaliged,
            3 length bit(18) unaligned,
```

```
          2 source_map fixed bin,

     /*This is the limit of the $display info structure.*/

          2 rel_text ptr,
          2 rel_def ptr,
          2 rel_link ptr,
          2 rel_symbol ptr,
          2 text_boundary fixed bin,
          2 static_boundary fixed bin,
          2 default_truncate fixed bin,
          2 optional_truncate fixed bin;

     /*This is the limit of the $long info structure.*/
```

1) version_number — is the version number of the structure (currently = 1).

2) textp — is a pointer to the base of the text section.

~~3) defp — is a pointer to the base of the~~ definition section.

4) linkp — is a pointer to the base of the linkage section.

5) symbp — is a pointer to the base of the symbol section.

6) bmapp — is a pointer to the break map.

7) tlng — is the length (in words) of the text section.

8) dlng — is the length (in words) of the definition section.

9) llng — is the length (in words) of the linkage section.

10) slng — is the length (in words) of the symbol section.

11) blng — is the length (in words) of the break map.

12) old_format — is "1"b if this segment is in the old format; otherwise it is "0"b.

13) bound	is "1"b if this is a bound object segment; otherwise it is "0"b.
14) relocatable	is "1"b if the object is relocatable; otherwise it is "0"b.
15) procedure	is "1"b if it is a procedure; is "0"b if it is nonexecutable data.
16) standard	is "1"b if this is a standard object segment; otherwise it is "0"b.
17) gate	is "1"b if this is a procedure generated in the gate format; otherwise it is "0"b.
18) call_delimiter	is the call delimiter value if this is a gate procedure.

This is the limit of the $brief info structure.

19) compiler	is the name of the compiler which generated this object segment.
20) compile_time	is the date and time this object was generated.
21) userid	is the access id of the user in whose behalf this object was generated.
22) cvers.offset	is the offset (in words), relative to the base of the symbol section, of the aligned variable length character string which describes the compiler version used.
23) cvers.length	is the length (in characters) of the compiler version string.
24) comment.offset	is the offset (in words), relative to the base of the symbol section, of the aligned variable length character string containing some compiler generated comment.
25) comment.length	is the length (in characters) of the comment string.
26) source_map	is the offset (relative to the base of the symbol section) of the source map.

This is the limit of the $display info structure.

27) rel_text — is a pointer to the object's text section relocation information.

28) rel_def — is a pointer to the object's definition section relocation information.

29) rel_link — is a pointer to the object's linkage section relocation information.

30) rel_symbol — is a pointer to the object's symbol section relocation information.

31) text_boundary — partially defines the beginning address of the text section. The text must begin on an integral multiple of some number, e.g., 0 mod 2, 0 mod 64; this is that number.

32) static_coundary — is analogous to text_boundary for ~~internal static.~~

33) default_truncate — is the offset (in words), relative to the base of the symbol section, starting from which the symbol section may be truncated to remove nonessential information (e.g., relocation information).

34) optional_truncate — is the offset (in words), relative to the base of the symbol section, starting from which the symbol section may be truncated to remove unwanted information (e.g., the compiler symbol tree).

This is the limit of the $long info structure.

(END)

Subroutine Call
Development System
05/04/71

Name: parse_file_

The parse_file_ module provides a facility for parsing an ASCII text into symbols and break characters. It is recommended for occasionally used text scanning applications. In applications where speed or frequent use are important, in-line PL/I code to do parsing is recommended instead.

A restriction of the procedure is that the text to be parsed be an aligned character string.

The initialization entry points, parse_file_init_name and parse_file_init_ptr, both save a pointer to the text to be scanned and a character count in internal static storage. Thus, one text only can be parsed at one time.

Entry: parse_file_$parse_file_init_name

This entry initializes the module given a directory and an entry name. It gets a pointer to the desired segment and saves it for subsequent calls in internal static.

Usage

```
declare parse_file_$parse_file_init_name entry (char(*),
        char(*), ptr, fixed bin);

call parse_file_$parse_file_init_name (dir, entry, p,
     code);
```

1) dir is the directory name portion of the pathname of the segment to be parsed. (Input)

2) entry is the entry name of the segment to be parsed. (Input)

3) p is a pointer to the segment. (Output)

4) code is an error code. It is zero if the segment is initiated. If nonzero, the segment cannot be initiated. It can return any code from hcs_$initiate except error_table_$segknown.

Entry: parse_file_$parse_file_init_ptr

This entry initializes the parse_file_ module with a supplied pointer and character count. It is used in cases where a pointer to the segment to be parsed is already available.

Usage

```
declare parse_file_$parse_file_init_ptr entry (ptr,
        fixed bin);

call parse_file_$parse_file_init_ptr (p, cc);
```

1) p is a pointer to a segment or an aligned character string. (Input)

2) cc is the character count of the ASCII text to be scanned. (Input)

Entry: parse_file_$parse_file_set_break

Break characters may be defined by use of this entry. Normally, all nonalphanumeric characters are break characters (including blank and new line).

Usage

```
declare parse_file_$parse_file_set_break entry (char(*));

call parse_file_$parse_file_set_break (cs);
```

1) cs is a control string. Each character found in cs will be made a break character. (Input)

Entry: parse_file_$parse_file_unset_break

This entry renders break characters as normal alphanumeric characters.

Usage

```
declare parse_file_$parse_file_unset_break entry (char(*));

call parse_file_$parse_file_unset_break (cs);
```

1) cs is a control string each character of which will be made a nonbreaking character. (Input)

Entry: parse_file_

The text file is scanned and the next break character or symbol is returned. Comments enclosed by /* and */ are skipped over.

Usage

```
declare parse_file_ entry (fixed bin, fixed bin,
        fixed bin(1), fixed bin(1));

call parse_file_ (ci, cc, break, eof);
```

1) ci is an index to the first character of the symbol or break character. (The first character of the text is considered to be character 1.) (Output)

2) cc is the number of characters in the symbol. (Output)

3) break is set to 1 if the returned item is a break character; otherwise it is 0. (Output)

4) eof is set to 1 if the end of text has been reached; otherwise it is 0. (Output)

Entry: parse_file_$parse_file_ptr

This entry is identical to parse_file_ except that a pointer (with bit offset) to the break character or the symbol is returned instead of a character index.

Usage

```
declare parse_file_$parse_file_ptr entry (ptr, fixed bin,
        fixed bin(1), fixed bin(1));

call parse_file_$parse_file_ptr (p, cc, break, eof);
```

1) p is a pointer to the symbol or the break character. (Output)

2-4) are the same as above. (Output)

Entry: parse_file_$parse_file_cur_line

The current line of text being scanned is returned to the caller. This entry is useful in printing diagnostic error messages.

Usage

```
declare parse_file_$parse_file_cur_line entry
        (fixed bin, fixed bin);
```

```
call parse_file_$parse_file_cur_line (ci, cc);
```

1-2) are the same as in parse_file_ above.

Entry: parse_file_$parse_file_line_no

The current line number of text being scanned is returned to the caller. This entry is useful in printing diagnostic error messages.

Usage

```
declare parse_file_$parse_file_line_no entry (fixed bin);

call parse_file_$parse_file_line_no (cl);
```

1) cl is the number of the current line. (Output)

Examples

Suppose the file zilch in the directory dir contains the following text:

```
name: foo; /*foo program*/

path_name: >bar;

linkage;

end;

fini;
```

The following calls could be made to initialize the parsing of zilch:

```
call parse_file_$parse_file_init_name (dir, zilch,
     p, code);

call parse_file_$parse_file_unset_break (">_");

declare atom char (cc) unaligned based (p);
```

Subsequent calls to parse_file_ptr would then yield the following:

atom	break	eof
name	0	0
:	1	0
foo	0	0
;	1	0
path_name	0	0
:	1	0
>bar	0	0
;	1	0
linkage	0	0
;	1	0
end	0	0
;	1	0
fini	0	0
;	1	0
-	-	1

(END)

Subroutine Call
Development System
5/25/72

Name: plot_

The procedure plot_ is a user interface to the Multics Graphics System. It creates a two dimensional graph from input data for use with Multics display terminals. The graph created is a Cartesian graph, scaled so as to permit maximum coverage of the screen, and labeled in convenient increments to facilitate reading. This routine can be made to plot either with vectors connecting the data points, with a specified character displayed at each point plotted, or both. It also has facilities that enable the user to append a new plot over the one being currently displayed (in which case the new plot is scaled to match the old one), to suppress the grid (in which case only the left-most and lowest lines are displayed, with tic marks at increments), and to direct that the graph be scaled equally in both directions.

For a more extensive description of graphics facilities, see the MPM Reference Guide section Graphics Support on Multics.

Usage

```
declare plot_ entry ((*)float bin, (*)float bin, fixed bin,
        fixed bin, char(1));

call plot_ (x, y, xydim, vec_sw, symbol);
```

1) x is an array of x coordinates of points to be plotted. (Input)

2) y is an array of y coordinates of points to be plotted. (Input)

3) xydim is the number of elements in the x and y array pairs. (Input)

4) vec_sw = 1 if the vectors but no symbol are desired;
= 2 if the symbol and vectors are desired;
= 3 if the symbol but no vectors are desired. (Input)

5) symbol is the symbol to be plotted at each point. (Input)

Notes

It is possible, by repetitive calls to plot_, to display any set of graphs on top of one another. All graphs after the first graph will be scaled to the scale of the first. A call to plot_ will erase the screen only if there was a call to either plot_$init or plot_$initf prior to it. The only exception is

that the first call to plot_ in a process will always erase the screen whether or not plot_$init or plot_$initf have been called. Default values for options are dotted grid, automatic scaling, no labels, and linear-linear plot.

Entry: plot_$init

This entry allows the user to set parameters controlling the type of plotting performed. The parameters specify the type of graph desired (log-log, linear, etc.), the type of grid desired (if any), and whether or not plot_ is to scale both axes equally. A call to this entry also ensures that the next call to plot_ will erase the screen.

Usage

```
declare plot_$init entry (char(*), char(*), fixed bin,
        float bin, fixed bin, fixed bin);

call plot_$init (xlabel, ylabel, type, base, grid_sw,
     eq_scale_sw);
```

1) xlabel — is the label desired along the x axis. (Input)

2) ylabel — is the label desired down the y axis. (Input)

3) type
= 1 for linear-linear plot;
= 2 for log-linear plot (log on x axis);
= 3 for linear-log plot (log on y axis);
= 4 for log-log plot. (Input)

4) base — is the logarithm base (for logarithmic plots). (Input)

5) grid_sw
= 0 if tic marks and values are desired;
= 1 if dotted grid and values are desired;
= 2 if solid grid and values are desired;
= 3 if no grid or values are desired. (Input)

6) eq_scale_sw
= 0 if normal scaling is desired;
= 1 if the plot is to be scaled equally in both directions. (Input)

Entry: plot_$initf

This entry is similar to plot_$init but is callable from FORTRAN.

Usage

```
integer xlabel (<n>), ylabel (<n>), xlng, ylng, type,
        grid_sw, eq_scale_sw

real base

equate initf, plot_$initf

call initf (xlabel, xlng, ylabel, ylng, type, base,
     grid_sw, eq_scale_sw);
```

1) xlabel — is the label desired along the x axis. (Input)

2) xlng — is the length (in characters) of xlabel. (Input)

3) ylabel — is the label desired down the y axis. (Input)

4) ylng — is the length (in characters) of ylabel. (Input)

5) type
= 1 for linear-linear plot;
= 2 for log-linear plot (log on x axis);
= 3 for linear-log plot (log on y axis);
= 4 for log-log plot. (Input)

6) base — is the logarithm base (for logarithmic plots). (Input)

7) grid_sw
= 0 if tic marks and values are desired;
= 1 if dotted grid and values are desired;
= 2 if solid grid and values are desired;
= 3 if no grid or values are desired. (Input)

8) eq_scale_sw
= 0 if normal scaling is desired;
= 1 if the plot is to be scaled equally in both directions. (Input)

Entry: plot_$scale

This entry allows a user to set his own scaling by allowing him to specify the extent of the axes in the x and y directions. If this scaling feature is desired, this entry must be called before any call to plot_ (i.e., immediately after a call to plot_$init or plot_$initf); otherwise, it is ignored.

Usage

```
declare plot_$scale entry (float bin, float bin,
        float bin, float bin);

call plot_$scale (xmin, xmax, ymin, ymax);
```

1) xmin is the desired low bound of the x axis. (Input)

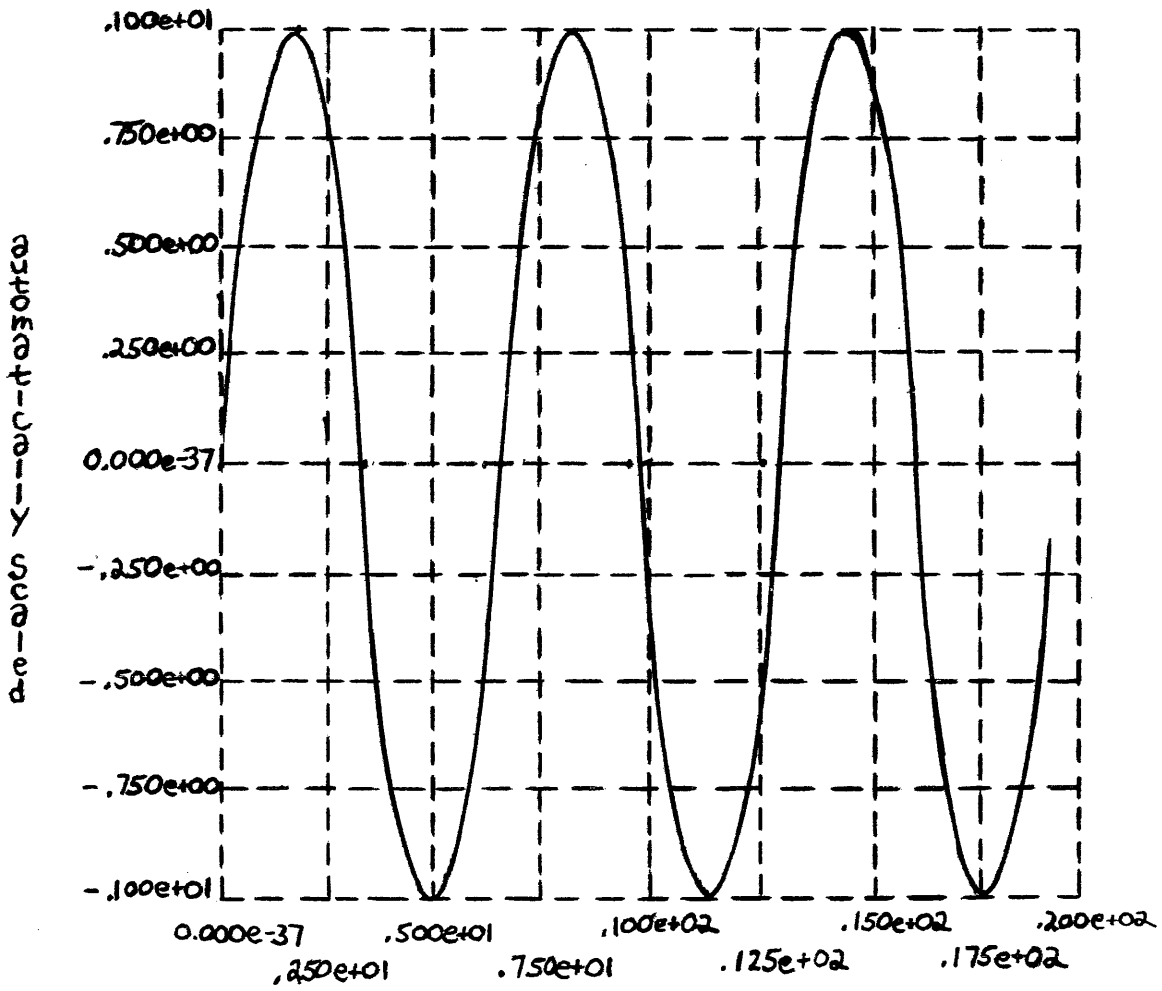
2) xmax is the desired high bound of the x axis. (Input)

3) ymin is the desired low bound of the y axis. (Input)

4) ymax is the desired high bound of the y axis. (Input)

Example

```
p_sin:  proc;

declare x(180) float bin,
        y(180) float bin,
        i fixed bin,
        pi float bin static internal initial (3.14159e0),
        three_cyc float bin,
        plot_ entry ((*)float bin, (*)float bin, fixed bin,
              fixed bin, char(1)),
        plot_$init entry (char(*), char(*), fixed bin,
                  float bin, fixed bin, fixed bin),
        (sin, float) builtin;

three_cyc = 6e0*pi/180e0;

do i = 1 to 180;
        x(i) = three_cyc * float (i-1);
        y(i) = sin (x(i));
        end;

call plot_$init ("this is a sine curve", "automatically
     scaled",1,0e0,1,0);

call plot_ (x, y, 180, 1, "");

return;
end;
```



(END)*

Subroutine Call
Development System
7/30/71

Name: random_

The procedure random_ is a random number generator with entry points which, given an input seed, generate a pseudo-random variable with a uniform, exponential, or normal distribution. The seed is an optional input argument; if it is not included in the call, an internal static variable is used and updated.

For one set of entry points, each call to random_ produces a single random number. To obtain a sequence of random numbers with the desired distribution, repeated calls are made, each time using the value of the seed, returned from a call, as the input value of the seed for the next call in the sequence.

There is an additional set of entry points which return an array with a sequence of random numbers. The first element of the array is generated from the input seed and the last element corresponds to the returned value of the seed. In addition, for the uniform and normal distributions, there are entry points which produce the antithetic random variables, either singly or as a sequence. For any given seed, the random variable produced is negatively correlated with that produced at the corresponding entry.

Entry: random_$uniform

The entry point random_$uniform generates a random number 0.0 < random_no < 1.0. The sequence of random numbers has a uniform distribution on the interval zero to one.

Usage

```
declare random_$uniform entry (float bin(27));

call random_$uniform (random_no);
```

or

```
declare random_$uniform entry (fixed bin(35), float
        bin(27));

call random_$uniform (seed, random_no);
```

1) seed is the optional (see Notes) seed that is used to generate the random number. The value of seed is modified by this entry. The value returned is the seed that is used to generate the next random number of the sequence. The

value of seed must be a nonzero positive integer. (Input/Output)

2) random_no — is the random number that is generated. (Output)

Entry: random_$uniform_seq

This entry point returns an array of random numbers from the uniform sequence.

Usage

```
declare random_$uniform_seq entry (float bin(27),
        fixed bin);

call random_$uniform_seq (array, array_size);
```

or

```
declare random_$uniform_seq entry (fixed bin(35),
        float bin(27), fixed bin);

call random_$uniform_seq (seed, array, array_size);
```

1) seed — is the optional (see Notes) seed used to generate array. See References (2). The value returned corresponds to the random number returned as array (array_size). (Input/Output)

2) array (n) — is an array of the generated random numbers where n is greater than or equal to array_size. (Output)

3) array_size — specifies the number of random variables to be returned in array. (Input)

Entry: random_$uniform_ant

This entry point generates a uniformly distributed random number, random_ant, that is negatively correlated with random_no produced by the entry random_$uniform. For any particular value of the seed, (random_ant + random_no) = 1.0.

Usage

```
declare random_$uniform_ant entry (float bin(27));
```

```
call random_$uniform_ant (random_ant);
```

or

```
declare random_$uniform_ant entry (fixed bin(35), float
        bin(27));

call random_$uniform_ant (seed, random_ant);
```

1) seed — is the optional (see Notes) seed used to generate the random number. (Input/Output)

2) random_ant — is the random number that is generated. (Output)

Entry: random_$uniform_ant_seq

The entry point random_$uniform ant_seq returns an array, ant_array, of uniformly distributed random numbers that are negatively correlated with the array produced by random_$uniform_seq. For any particular value of the seed, (ant_array(i) + array(i)) = 1.0, for i between one and array_size.

Usage

```
declare random_$uniform_ant_seq entry (float bin(27),
        fixed bin);

call random_$uniform_ant_seq (ant_array, array_size);
```

or

```
declare random_$uniform_ant_seq entry (fixed bin(35), float
        bin(27), fixed bin);

call random_$uniform_ant_seq (seed, ant_array, array_size);
```

1) seed — is the optional seed used. (Input/Output)

2) ant_array (n) — is the array of generated random numbers where n is greater than or equal to array_size. (Output)

3) array_size — is the number of values returned in ant_array. (Input)

Entry: random_$normal

The entry point random_$normal generates a random number, -6.0 < random_no < 6.0. The sequence of random numbers has an approximately normal distribution with a mean of zero and a variance of one. The random number is formed by taking the sum of twelve successive random numbers from the uniformly distributed sequence and then adjusting the sum for a mean of zero.

Usage

```
declare random_$normal entry (float bin(27));

call random_$normal (random_no);
```

or

```
declare random_$normal entry (fixed bin(35), float
        bin(27));

call random_$normal (seed, random_no);
```

Same arguments as above.

Entry: random_$normal_seq

The entry point random_$normal_seq generates a sequence, of length array_size, of random variables with an approximately normal distribution.

Usage

```
declare random_$normal_seq entry (float bin(27),
        fixed bin);

call random_$normal_seq (array, array_size);
```

or

```
declare random_$normal_seq entry (fixed bin(35), float
        bin(27), fixed bin);

call random_$normal_seq (seed, array, array_size);
```

Same arguments as above.

Entry: random_$normal_ant

The entry point random_$normal_ant generates a random number, random_ant, that is negatively correlated with random_no produced by the entry random_$normal. For any particular value of the seed, (random_ant + random_no) = 0.0.

Usage

```
declare random_$normal_ant entry (float bin(27));

call random_$normal_ant (random_ant);
```

or

```
declare random_$normal_ant entry (fixed bin(35), float
        bin(27));

call random_$normal_ant (seed, random_ant);
```

Same arguments as above.

Entry: random_$normal_ant_seq

The entry point random_$normal_ant_seq generates a sequence, of length array_size, of random variables with approximately normal distribution. These variables are negatively correlated with those produced by the entry point random_$normal_seq.

Usage

```
declare random_$normal_ant_seq entry (float bin(27),
        fixed bin);

call random_$normal_ant_seq (ant_array, array_size);
```

or

```
declare random_$normal_ant_seq entry (fixed bin(35),
        float bin(27), fixed bin);

call random_$normal_ant_seq (seed, ant_array, array_size);
```

Same arguments as above.

Entry: random_$exponential

The entry point random_$exponential generates a positive random number. The sequence of random numbers has an exponential

distribution with a mean of one. The random number is generated by taking successive random numbers from the uniformly distributed sequence and applying the VonNeumann method (see References (2)) for generating an exponential distributed random variable.

Usage

```
declare random_$exponential entry (float bin(27));

call random_$exponential (random_no);
```

or

```
declare random_$exponential entry (fixed bin(35), float
        bin(27));

call random_$exponential (seed, random_no);
```

Same arguments as above.

Entry: random_$exponential_seq

The entry point random_$exponential_seq produces an array of exponentially distributed random variables.

Usage

```
declare random_$exponential_seq entry (float bin(27),
        fixed bin);

call random_$exponential_seq (array, array_size);
```

or

```
declare random_$exponential_seq entry (fixed bin(35), float
        bin(27), fixed bin);

call random_$exponential_seq (seed, array, array_size);
```

Same arguments as above.

Entry: random_$get_seed

The entry point random_$get_seed is used to obtain the current value of the internal seed (see Notes).

Usage

```
declare random_$get_seed entry (fixed bin(35));

call random_$get_seed (seed_value);
```

1) seed_value is the current value of the internal seed. (Output)

Entry: random_$set_seed

The entry point random_$set_seed is used to set the value of the internal seed. This internal seed is used as the seed for the next call to any random_ entry point in which the optional argument seed is not provided (see Notes).

Usage

```
declare random_$set_seed entry (fixed bin(35));

call random_$set_seed (seed_value);
```

1) seed_value is the value to which the internal seed is set. seed_value must be a nonzero positive integer. (Input)

Notes

All non-optional arguments must be included in the call, even if only the value of some are of interest. For all entry points (except random_$set_seed and random_$get_seed), if the optional parameter seed is not provided in the call, an internal seed is used and updated in exactly the same manner as a seed provided by the caller. This internal seed is maintained as an internal static variable. At the beginning of a user's process, it has a default value or 4084114320. Its value is changed only by calls to random_$set_seed or by calls to other entry points in which the optional parameter seed is not included.

If the value of a seed is zero, the new value of the seed and the random numbers will be zero. If the value of a seed is negative, the low order 35 bits of the internal representation will be used as the seed; if nonzero, a valid value will be returned for the seed and the random numbers. A given seed will always produce the same random number from any given entry point. Since all entry points use the same basic method for computing the next seed, the distribution of the sequence produced by calls to any given entry point will be maintained, although the input seed used may have been produced by a call to a different entry

point. In other words, the user need keep only a single value of the next seed even though he calls more than one of the entry points. However, in general, the different entry points will, for any given input seed, produce different values for the next seed.

On the other hand, the user may generate independent streams of random numbers by beginning each stream with separate initial seeds and maintaining separate values for the next seed.

The uniformly distributed random number sequence is generated using the Tausworth method (see References (1) and (3)). The algorithm, in terms of abstract registers A and B, is described below.

The parameter n is one less than the number of used bits per word (for Multics, use n = 35). The parameter m is the amount of shift (for Multics, m = 2).

a) Let register A initially contain the previous random number in bit positions 1 to n with zero in the sign bit (position 0).

b) Copy register A into register B and then right-shift register B m places.

c) Exclusive-or register A into register B and also store the result back into register A. (Registers A and B now have bits for the new random number in positions m + 1 to n, but still contain bits from the old n bit random number in position 1 through m.)

d) Left-shift register B (n-m) positions. (This places m bits for the new random number in positions 1 to m of register B and zero bits in positions m + a through n.)

e) Exclusive-or register B into register A and zero out register A's sign bit. (Register A now contains all n bits of the new random number.)

f) To obtain a random number between 0.0 and 1.0, we divide the n bit integer in register A by 2**n. The contents of register A must be saved for use in generating the next random number.

In random_, a word is considered as being 36 bits long including the sign bit. This gives rise to a 35 bit integer random number. Since in Multics, a floating point number has a 27 bit mantissa, this means different seeds may produce the same floating point value; however, the interval between identical

values of the integer seed is equal to the cycle length of the integer random number generator. In random_, a shift of 2 is used, which gives a cycle of (2**35)-1 (see References (1)). The essence of the assembly language code used by random_ is given below.

```
equ     shift, 2          use a shift of 2
ldq     seed              seed into the Q register
qrl     shift             shift the seed right
ersq    seed              exclusive-or to the seed
ldq     seed              put result in the Q register
qls     35-shift          shift left
erq     seed              exclusive-or the previous result
anq     =o377777777777    save only 35 bits
stq     seed              return the value of the seed
lda     seed              load the integer value
lde     0b25, du          convert to floating point
fad     =0., du           normalize the floating point
fst     random_no         return a random number
```

References

1) Golomb, S. W., Welch, L. R., Goldstein, R. M., and Hales, A. W., Shift Register Sequences, Holden-Day, 1967, p. 97.

2) Taub, A. H., John VonNeumman, Collected Works, V, Pergamon Press, 1963, p. 770.

3) Whittleself, John R. B., "A Comparison of the Correlation Behavior of Random Number Generators for the IBM 360", Communications of the ACM, 11, 9, September, 1968.

(END)

Subroutine Call
7/11/73

Name: read_list_

This procedure is used to read and convert free-formatted input data. It reads a series of input values from the I/O stream "user_input" and interprets each in terms of the data type of the corresponding variable in the calling sequence. (See Notes below for acceptable data types and break characters.) If an input value is discovered to have incorrect syntax, a comment is printed asking that the input value (and any following it) be retyped.

If the line read contains fewer values than were requested in the calling sequence, by default read_list_ types out a message of the form:

n more input values expected

and attempts to read another line from the stream "user_input". This does not occur before the first input line.

Since this procedure can be called with a varying number of arguments, it is not permissible to include a parameter attribute list in the declaration of the various entry points.

Usage

```
declare read_list_ entry options (variable);

call read_list_ (v1, v2, ..., vn);
```

1) vi is any variable of the calling program and can be any scalar quantity. (Output)

Entry: read_list_$no_prompt

This entry works exactly as read_list_ does except that there is no prompting if the line read contains fewer values than were requested in the calling sequence. Instead, read_list_ merely attempts to read another line until all requested values have been supplied.

Usage

```
declare read_list_$no_prompt entry options (variable);

call read_list_$no_prompt (v1, v2, ..., vn);
```

1) vi is as above. (Output)

Entry: read_list_$prompt

This entry works as read_list_ does except that alternate arguments are taken to be prompting messages to be typed on the stream "user_output". No other prompting is done.

Usage

```
declare read_list_$prompt entry options (variable);

call read_list_$prompt (p1, v1, p2, v2, ..., pn, vn);
```

1) pi is a character string to be typed on the user's terminal. It is typed out via ioa_$nnl, so there is no new line after a message unless the last two characters of the string are "¬/". (See the MPM subroutine write-up for ioa_.) If any prompting argument is a null character string, that prompting message is not typed. If the number of arguments is odd, the last (unmatched) prompting argument is taken as a message to be typed after reading the last variable. If the user anticipates input by typing several input values on one line, intermediate prompting messages are omitted. (Input)

2) vi is as above. (Output)

Entry: read_list_$scan_string

This entry works as read_list_ does except that instead of reading input from the user's terminal it considers its first argument to be a character string to be scanned and converted as though it had been read from "user_input". Subsequent argument values are supplied from the contents of the first argument.

Usage

```
declare read_list_$scan_string entry options (variable);

call read_list_$scan_string (scan_string, nret,
     v1, v2, ..., vn);
```

1) scan_string is a character string (char(*)) to be scanned as though it had been read from "user_input". (Input)

2) nret is a fixed bin(17) number containing, upon return, the number of arguments actually supplied in scan_string. The reason for nret is that there is no prompting and no way to ask for more values if there weren't enough to begin with. A value of nret less than 0 indicates a syntax error, with the absolute value of nret indicating which argument was bad. (The arguments following that one are not processed.) Counting for nret starts with v1 as argument one. (Output)

Notes

Input conversion is currently implemented for the following Multics PL/I data types using the rules for the formation of PL/I constants:

single precision fixed point

double precision fixed point

single precision floating point

double precision floating point

nonvarying character string

varying character string

nonvarying bit string

varying bit string

pointer

Character strings can be typed without enclosing quotation marks if they don't contain any of the delimiters listed below.

A fixed point number can be typed in octal format if it is followed by the letter "o".

Floating point constants can be typed without an explicit exponent or decimal point.

Input pointer values are typed as follows:

d1 means a pointer to segment number d1, word offset 0, bit offset 0.

d1|d2 means a pointer to segment number d1, word offset d2, bit offset 0.

d1|d2(d3) means a pointer to segment number d1, word offset d2, bit offset d3.

where d1 and d2 are octal integers and d3 is a decimal integer.

Fixed point constants can be typed in binary form; e.g., 1011b.

Typed character or bit strings which are too large or too small for the space declared in the calling program are truncated or padded, respectively, according to the usual PL/I language rules.

The delimiters allowed in the input line are space (ϐ), tab (HT), comma (,), and the new-line character (NL). Successive input values can be separated by any number of blanks and/or tabs. Other combinations of delimiters have the following meanings:

,NL	=	NL
,,	=	no new input value for corresponding argument
NL,	=	,,
NLϐ,	=	,,
NLHT,	=	,,
ϐNL	=	NL

 (END)

Subroutine Call
Standard Service System
2/25/72

Name: reversion_

This procedure causes the handler currently established for the given condition in the calling block activation to be disestablished. If no handler for the given condition is established in the calling block activation, no action is taken. A description of the condition mechanism is given in the MPM Reference Guide section on The Multics Condition Mechanism.

Usage

```
declare reversion_ entry (char(*));

call reversion_ (name);
```

1) name is the name of the condition for which the handler is to be disestablished. (Input)

Notes

The condition names unclaimed_signal and cleanup are obsolete special condition names and should not be used.

A call to reversion_ must be used only to revert a handler established by a call to condition_ (see the MPM subroutine). reversion_ must not be used to revert a handler established by a PL/I on statement.

In PL/I Version 2, when a call to reversion_ appears within the scope of a begin block or internal procedure of a procedure, the no_quick_blocks option must be specified in the procedure statement of that procedure. The no_quick_blocks option is a nonstandard feature of the Multics PL/I language and, therefore, programs using it may not be transferable to other systems.

(END)

Subroutine Call
10/31/73

Name: signal_

This procedure signals the occurrence of a given condition. A description of the condition mechanism and the way in which a handler is invoked by signal_ is given in the MPM Reference Guide section, The Multics Condition Mechanism.

Usage

```
declare signal_ entry (char(*), ptr, ptr);

call signal_ (name, mcptr, info_ptr);
```

1) name is the name of the condition to be signalled. (Input)

2) mcptr points to the machine conditions at the time the condition was raised. This argument is for use by system programs only in order to signal hardware faults. In user programs, this argument should be null if a third argument is supplied. This argument is optional. (Input)

3) info_ptr points to information relating to the condition being raised. The structure of the information is dependent upon the condition being signalled; however, conditions raised with the same name should provide the information in the same structure. Important: all structures must begin with a standard header. The structures provided with system conditions are described in the MPM Reference Guide section, System Conditions and Default On Unit Actions. The format for the header is described in the MPM Reference Guide section, The Multics Condition Mechanism. This argument is intended for use in signalling conditions other than hardware faults. This argument is also optional. (Input)

Notes

If signal_ returns to its caller, indicating that the handler has returned to it, the calling procedure should retry the operation that caused the condition to be signalled.

The PL/I signal statement differs from the signal_ subroutine in that the above parameters cannot be provided in the signal statement.

 (END)

Subroutine Call
Standard Service System
09/23/70

Name: stu_

The stu_ (Symbol Table Utility) procedure provides a number of entry points for retrieving information from the segment symbol table section of an object segment. Multics compilers will produce a segment symbol table only when explicitly instructed to do so (e.g., with the use of the PL/I table option).

Entry: stu_$find_header

This entry, given an ascii name and a pointer to any location in a (possibly bound) object segment, searches the given segment for the symbol table header corresponding to the given name.

Usage

```
declare stu_$find_header entry (ptr, char(32) aligned,
        fixed bin(22)) returns (ptr);

header_pt = stu_$find_header (seg_pt, name, bc);
```

1) seg_pt — points at any location in the object segment. (Input)

2) name — is the ascii name of the program whose symbol header is to be found (same as the name of the segment if the object segment is not bound). (Input)

3) bc — is the bit count of the object segment; if zero, find header will determine the bit count itself. (Input)

4) header_pt — will point to the header if it could be found or will be null if the header could not be found. (Output)

Notes

Since determining the bit count of a segment is relatively expensive, the user should provide the bit count if he has it available (i.e., as a result of a call to hcs_$initiate_count).

If seg_pt is not null, decode_object is called; the symbol header is assumed to start at word 0 of the symbol segment. If seg_pt is null, hcs_$make_ptr is used to locate the "symbol_table" segdef in the segment with the given name.

Entry: stu_$find_block

This entry point, given a pointer to the symbol table header of a procedure, searches for a procedure symbol block corresponding to a given block in the object program.

Usage

```
declare stu_$find_block entry (ptr, char(*) aligned)
        returns (ptr);

block_pt = stu_$find_block (header_pt, name);
```

1) header_pt — points at a symbol table header. (Input)

2) name — is the ascii name of the symbol block to be found. The name of a symbol block is the same as the first name written on a procedure statement. (Input)

3) block_pt — will point to the symbol block if found or will be null if the block could not be found. (Output)

Entry: stu_$get_block

This entry point, given a pointer to the stack frame corresponding to an active PL/1 procedure or begin block, returns pointers to the symbol table header and symbol block associated with the procedure or begin block. Null pointers will be

returned if the stack frame does not belong to a PL/1 program or if the PL/1 program does not have a symbol table.

Usage

```
declare stu_$get_block entry (ptr, ptr, ptr);

call stu_$get_block (stack_pt, header_pt, block_pt);
```

1) stack_pt — points at an active stack frame. (Input)

2) header_pt — will be set to point at the symbol table header. (Output)

3) block_pt — will be set to point at the symbol block. (Output)

Entry: stu_$find_symbol

This entry point, given a pointer to the symbol block corresponding to a procedure or begin block, searches for the symbol node associated with a specified variable name. If the name is not found in the given block, the parent block is searched. This is repeated until the name is found or the root block of the symbol structure is reached, in which case a null pointer is returned.

Usage

```
declare stu_$find_symbol entry (ptr, char(*) aligned, fixed
        bin) returns (ptr);

symbol_pt = stu_$find_symbol (block_pt, name, steps);
```

1) block_pt — points at the symbol block where the search is to begin. (Input)

2) name — is the ascii name of the symbol to be found. name may be a completely qualified structure name (i.e., "a.b.c"), in which case the symbol node for the lowest level item will be found. (Input)

3) steps — will be set to the number of steps along the parent chain that were taken before the symbol was found. steps will be 0 if the symbol was found in the given block. (Output)

4) symbol_pt — will point to the symbol node if found or will be null if the symbol could not be found in any block. (Output)

Entry: stu_$decode_value

This entry point is called to decode values stored in a symbol node (see Reference Data Section of this manual).

Usage

```
declare stu_$decode_value entry (fixed bin(35), ptr, ptr,
        fixed bin) returns (fixed bin(35));

value = stu_$decode_value (v, stack_pt, ref_pt, code);
```

1) v — is the value from the symbol node to be decoded. (Input)

2) stack_pt — is a pointer to the active stack frame for the procedure or begin block corresponding to the symbol block in which the symbol node whose value to be decoded appears. (Input)

3) ref_pt — is the value of the reference pointer if the variable corresponding to the symbol node is based. (Input)

4) code — will be set to 0 if the value was successfully decoded and to 1 if the value could not be decoded. (Output)

5) value — will be the decoded value if code = 0. (Output)

Entry: stu_$get_reference

This entry point, given a pointer to the symbol node corresponding to a PL/1 based variable, attempts to return the value of the pointer variable that appeared in the based declaration (i.e., the value of "p" in "dcl a based (p);"). A null pointer will be returned if the declaration does not have the proper form or if the value of the pointer could not be determined.

Usage

```
declare stu_$get_reference entry (ptr, ptr) returns (ptr);

ref_pt = stu_$get_reference (symbol_pt, stack_pt);
```

1) symbol_pt — points at the symbol node for the based variable. (Input)

2) stack_pt — points at the active stack frame for the procedure or begin block corresponding to the symbol block in which the symbol node is found. (Input)

3) ref_pt — will be set to the value of the pointer variable or will be null if the value could not be determined. (Output)

Notes

A null pointer will be returned for any one of a number of reasons. Some of these are:

1) the based variable was declared as

```
dcl a based;
```

2) the pointer base does have an active stack frame.

Entry: stu_$get_address

This entry point, given a pointer to a symbol node, an active stack frame and a vector of subscripts, determines the address of the specified variable.

Usage

```
declare stu_$get_address entry (ptr, ptr, ptr, ptr) returns
        (ptr);

add_pt = stu_$get_address (symbol_pt, stack_pt, ref_pt,
    subs_pt);
```

1) symbol_pt — points at the symbol table node. (Input)

2) stack_pt — points at the active stack frame for the PL/1 procedure or begin block corresponding to the symbol block in which the symbol node is found. (Input)

3) ref_pt — is the value of the reference pointer to be used if the symbol node corresponds to a based variable. If ref_pt is null, get_address will call get_reference to determine the value of the pointer appearing in the original declaration. (Input)

4) subs_pt — points at a vector of single precision fixed point subscripts. The number of subscripts is assumed to match the number required by the declaration. This argument may be null if the symbol node does not correspond to an array. (Input)

5) add_pt — will point to the full bit address of the variable corresponding to the symbol node or will be null if the address could not be determined. (Output)

Notes

Pointers to the text or linkage segment, if required, are obtained from the specified stack frame.

Entry: stu_$get_addr

This entry point, given a pointer to a symbol node, a vector of information pointer and a pointer to a vector of subscripts, determines the address of the specified variable.

Usage

```
declare stu_$get_addr entry (ptr, (3)ptr, ptr, ptr) returns
        (ptr);

add_pt = stu_$get_addr (symbol_pt, info, ref_pt, sub_pt);
```

1) symbol_pt — points at the symbol table node. (Input)

2) info — is an array of pointers. info(1) points at the active stack frame; info(2) points at the linkage section; info(3) points at the text segment. (Input)

3) ref_pt — is the value of the reference pointer to be used if the symbol node corresponds to a based variable. If ref_pt is null, get_address will call get_reference to determine the value of the pointer appearing in the original declaration. (Input)

4) subs_pt — points at a vector of single precision fixed point subscripts. The number of subscripts is assumed to match the number required by the declaration. This argument may be null if the symbol node does not correspond to an array. (Input)

5) add_pt — will point to the full bit address of the variable corresponding to the symbol node or will be null if the address could not be determined. (Output)

Entry: stu_$get_line_no

This entry, given a pointer to a symbol table block and an offset in the text segment corresponding to the block, determines the line number, starting location, and number of instructions in the source statement containing the specified instruction.

Usage

```
declare stu_$get_line_no entry (ptr, fixed bin(18), fixed
        bin(18), fixed bin(18)) returns (fixed bin(18));

line_no = stu_$get_line_no (block_pt, offset, start, num);
```

1) block_pt — points at the block node. (Input)

2) offset — is the offset of an instruction in the text segment. (Input)

3) start — will be the offset in the text segment of the first instruction generated for the source line containing the specified instruction, or

will be -1 if the line could not be found. (Output)

4) num — will be the number of instructions generated by the specified source line. (Output)

5) line_no — will be the line number of the source statement which generated the given instruction. (Output)

Notes

1) If the source program contains include files, all line numbers refer to the expanded source segment.

2) No distinction is made between several statements occurring on the same source line. "start" will be the starting location of the code generated for the first statement on the line and "num" will be the total length of all the statements on the line.

Entry: stu_$get_location

This entry, given a pointer to a symbol table block and the line number of a source statement in the block, returns the location in the text segment of the first instruction generated by the specified source line.

Usage

```
declare stu_$get_location entry (ptr, fixed bin(18))
        returns (fixed bin(18));

offset = stu_$get_location (block_pt, line_no);
```

1) block_pt — points at the block node. (Input)

2) line_no specifies the source line number. (Input)

3) offset will be the offset in the text segment of the first instruction generated by the given line, or will be -1 if no instructions were generated by the given line. (Output)

Example

We will illustrate the use of some of the procedures documented above by presenting a sample procedure which is called with

stack_pt a pointer to the stack frame of PL/I procedure

symbol an ascii string giving the name of a user symbol in the PL/I program

and subs_pt a pointer to an array of integers giving subscript values.

The procedure will determine the address, data type, and size of the specified symbol. If any errors occur, the returned address will be null.

```
%;
example:  proc (stack_pt, symbol, subs_pt, size) returns (ptr);

          declare stack_pt     pt,
                  symbol       aligned char(*),
                  subs_pt      ptr,
                  size         fixed bin(35);

          declare (header_pt, block_pt,  symbol_pt,  ref_pt,  sp,
                      add_pt) ptr,

                  (i, steps, code) fixed bin,

                  stu_$get_block entry (ptr, ptr, ptr),
```

```
          stu_$find_symbol entry (ptr,  char(*)  aligned,
              fixed bin) returns (ptr),

          stu_$get_address entry  (ptr,  ptr,  ptr,  ptr)
              returns (ptr),

          stu_$decode_value entry  (fixed  bin(35),  ptr,
              ptr, fixed bin) returns (fixed bin(35));

declare 1  frame    based,
        2  skip(32) fixed,
        2  display  ptr;

% include symbol_node;

          /* determine header and block pointers */

          call   stu_$get_block   (stack_pt,   header_pt,
              block_pt);

          if block_pt = null then return (null);

          /* search for specified symbol */

          symbol_pt = stu_$find_symbol (block_pt, symbol,
              steps);

          if symbol_pt = null then return (null);

          /* determine stack frame of block owning symbol
              */

          sp = stack_pt;
          do i = 1 to steps;
             sp = sp→ frame.display;
             end;
```

```
/* determine address of symbol */
ref_pt = null;
add_pt  =  stu_$get_address   (symbol_pt,   sp,
    ref_pt, subs_pt);

if add_pt = null then return (null);

/* determine size */

size = symbol_pt -> symbol_node.size;

if size < 0
then do;
     size  =   stu_$decode_value   (size,   sp,
           ref_pt, code);
     if code > = 0 then return (null);
     end;

return (add_pt);
end example;
```

(END)

Subroutine Call
8/27/73

Name: suffixed_name_

This subroutine handles suffixed storage system entry names. It provides an entry point that creates a properly-suffixed name from a user-supplied name that may or may not include a suffix, an entry point that changes the suffix on a user-supplied name that may or may not include the original suffix, and an entry point that finds a segment, a directory, or a multisegment file whose name matches a user-supplied name that may or may not include a suffix. It is intended to be used by commands that deal with segments with a standard suffix, but that do not require the user to supply the suffix in the command arguments.

Entry: suffixed_name_$find

This entry point attempts to find a directory entry whose name matches a user-supplied name that may or may not be properly suffixed. This directory entry can be a segment, another directory, or a multi-segment file (MSF).

Usage

```
declare suffixed_name_$find entry (char(*), char(*),
        char(*), char(32) aligned, fixed bin(2),
        fixed bin(5), fixed bin(35));

call suffixed_name_$find (directory, name, suffix, entry
     type, mode, code);
```

1) directory is the name of the directory in which the entry is to be found. (Input)

2) name is the name that has been supplied by the user, and that may or may not be properly suffixed. (Input)

3) suffix is the suffix that is supposed to be on name. It should not contain a leading period. (Input)

4) entry is a properly-suffixed version of name. It is returned even if the directory entry, directory>entry, does not exist. (Output)

5) type is a switch indicating the type of directory entry that was found: (Output)

   0 -- no entry was found.
   1 -- a segment was found.
   2 -- a directory was found.

3 -- a multi-segment file was found.

6) mode is the caller's access mode to the directory entry that was found. See the MPM writeup of hcs_$append_branch for a description of mode. Note that the caller's assecc mode to the MSF directory is returned for a multisegment file. (Output)

7) code is one of the following status codes: (Output)

0 -- the search was successful.

error_table_$noentry -- no directory entry that matches name was found.

error_table_$no_info -- no directory entry that matches name was found, and futhermore, the caller does not have status permission to the directory.

error_table_$incorrect_access -- a directory entry that matches name was found, but the caller has null access to this entry, and to the directory containing this entry.

error_table_$entlong -- the properly-suffixed name that was made is longer than name.

Entry: suffixed_name_$make

This entry point makes a properly-suffixed name out of a name supplied by the user that may or may not be properly suffixed.

Usage

```
declare suffixed_name_$make entry (char(*), char(*),
        char(32) aligned, fixed bin(35));

call suffixed_name_$make (name, suffix, priper_name, code);
```

1) name is as above. (Input)

2) suffix is as above. (Input)

3) proper_name is a properly-suffixed version of name.

4) code is one of the following status codes: (Output)

0 -- proper name was made successfully,

error_table_$entlong -- the properly-suffixed name that was made is longer that proper_name. proper_name contains only a part of the properly-suffixed name.

Entry: suffixed_name_$new_suffix

This entry point creates a name with a new suffix by changing the (possibly existing) suffix on a user-supplied name to the new suffix. If there is no suffix on the user-supplied name, then the new suffix is merely appended to the user-supplied name.

Usage

```
declare suffixed_name_$new_suffix entry (char(*), char(*),
        char(*), char(32), fixed bin(35));

call suffixed_name_$new_suffix (name, suffix, new_suffix,
     new_name, code);
```

1) name is as above. (Input)

2) suffix is the suffix that may or may not already be on name. (Input)

3) new_suffix is the new suffix. (Input)

4) new_name is the name that was created. If name ends with .suffix, then .new_suffix replaces .suffix in new_name. Otherwise, new_name if formed by appending .new_suffix to name. (Output)

5) code is one of the following status codes: (Output)

0 -- new_name was made successfully.

error_table_$entlong -- the properly-suffixed new name is longer than new_name. new_name contains only part of the properly-suffixed new name.

Note

If error_table_$no_s_permission is encountered during the processing for suffixed_name_$find, it is ignored and is not returned in the status code.

 (END)

I/O System Interface Module
Standard Service System
9/8/71

Name: syn

The synonym interface module provides a means by which two stream names can be made equivalent. Two stream names may be synonymized by the following call:

```
call ios_$attach (stream1, "syn", stream2, "", status);
```

Once the above call has been executed, all I/O system calls, with the exception of the attach and detach calls, will be redirected to stream2 until the synonymization is dissolved by a call to detach. In other words, all such I/O calls on either stream will have identical results and are, therefore, synonymous. The synonym interface module has been heavily optimized. It is therefore recommended that in cases where I/O devices are repeatedly detached and reattached within a process, that this detaching and reattaching be performed with a synonymized stream rather than detaching and reattaching the devices themselves.

I/O System Calls

All I/O system calls are implemented by the synonym module.

Device Identification

The pseudo-device upon which the synonym module operates is simply a stream, therefore any stream name is a legitimate device identifier.

Detachment

Detachment results in the dissolution of the synonymization of the associated stream. No other action can be specified in the detach call.

Notes

Due to the importance of the synonym module, it has been given several special properties.

The second stream name, stream2, in the call to attach does not have to exist at attach call time. The attachment will be completed anyway and, unless stream2 is subsequently attached to some device, an error will result when a call is made upon stream1.

Since all calls, except attach and detach, made to a stream attached via the synonym module are simply forwarded to the object or attached to stream, the synonym module has no modes, synchronization, element sizes, delimiters, breaks, reference pointers, or order calls of its own. It simply takes on the properties of the stream to which it is attached. The mode argument in the call to attach is ignored by the synonym module.

Currently, when an attempt is made to attach a stream that is already attached via the synonym module, the synonym attachment will be automatically detached. This special feature is only temporary and will be removed. Users are advised not to take advantage of it.

(END)

Name: tape_

This procedure is the I/O system Device Interface Module (DIM) used to access magnetic tapes written in Multics standard tape format. (See the MPM Reference Guide section, Multics Standard Magnetic Tape Format.) It is not called directly by the user's program. Instead, the user provides the name tape_ in a call to the I/O system attach entry. He then accomplishes the I/O operations by calling standard I/O system entry points that are independent of the interface module being used. Further information on the I/O system can be found in the MPM Reference Guide section on Input and Output Facilities. Details about the I/O system call syntax can be found in the module description of ios_.

Notes

Tape label records and end of reel records are written by the attach/detach functions automatically. Blocking of data into 256-word blocks is done internally by the read/write functions so that any number of words of data can be transmitted in a single call.

Permitted I/O System Calls

attach
detach
getsize
order
read
seek
write

Device Identifiers

This DIM accepts any character string of length 6 as a device identifier for a tape. The character string to be used should be agreed upon with Multics Operations when the tape is signed out.

Status Indications

Only standard Multics status codes are returned in the first half of the status string. The following status codes can be returned by tape_:

error_table_$argerr
error_table_$bad_index
error_table_$bad_label

error_table_$bad_mode
error_table_$bad_processid
error_table_$bad_ring_brackets
error_table_$blank_tape
error_table_$buffer_big
error_table_$data_improperly_terminated
error_table_$dev_offset_out_of_bounds
error_table_$device_end
error_table_$device_limit_exceeded
error_table_$device_parity
error_table_$improper_data_format
error_table_$invalid_read
error_table_$invalid_write
error_table_$io_still_assnd
error_table_$ionmat
error_table_$mount_not_ready
error_table_$no_device
error_table_$no_message
error_table_$no_room_for_dsb
error_table_$no_sys
error_table_$not_attached
error_table_$redundant_mount
error_table_$too_many_buffers
error_table_$undefined_order_request
error_table_$undefined_ptrname
error_table_$unimplemented_ptrname

Currently, the only meaningful bits in the second half of the status argument are the physical_end_of_data bit, the logical_end_of_data bit, and the stream_name_detached bit. See the MPM Reference Guide section, Use of the Input and Output System, for more information.

The physical_end_of_data bit is currently turned on during writing when the end of a physical tape reel is reached, indicating that no more data can be written on the reel. Checking this bit is the standard way of telling whether the end of the reel has been reached. It is turned on during reading if a normal tape termination is reached and it occurs at the physical end of reel. (A tape is normally terminated if it was properly detached when written, i.e., if it contains a valid end of reel record.)

The logical_end_of_data bit has no significance for writing. During reading, this bit is turned on when the logical end of data on the tape, i.e., the last relevant record, has been reached; the situation is further described by the status code portion of the status argument. For example, if the tape was not normally terminated, but the data is otherwise good, the code error_table_$data_improperly_terminated is returned.

The stream_name_detached bit is turned on when the tape and the stream are detached from the process; this can happen only during attach or detach requests.

Currently there are some temporary deviations from the standards described above with respect to the logical_end_of_data bit. It is turned on during writing, but should be ignored. When it is turned on during reading, a status code of zero eventually will mean normal termination, although currently error_table_$device_end is returned in this case.

Modes

The only modes accepted by the attach call to tape_ are r (read) and w (write). Any other mode specification, including rw, results in refusal to attach the stream.

Element Size

Only an element size of 36 is permitted.

Order Requests

The following order requests are implemented by this DIM:

error_count — is permitted only for tapes attached for writing. It writes out all currently filled write-behind buffers and returns the count of the total number of rewrite attempts in a fixed binary(35) number pointed to by the argptr argument of the ios_$order call.

rewind — is obsolete and will be removed eventually. The seek request should be used instead.

Break Characters and Read Delimiters

None of these are used by tape_. All transmission is controlled by the nelem argument of the read/write call. The only nonerroneous case where nelemt differs from nelem is for end of data reading.

Seek and Tell Calls

Seek, as currently implemented, is very limited and intended for rewinding only. pointer_name_1 can be either read or write, pointer_name_2 must be "first", and offset must be 0. (See the MPM subroutine write-up for ios_$seek.) This causes the tape to be rewound without unloading and readied for use in the same mode as before. To change mode, the tape must first be detached, then reattached in the other mode.

Tell is not currently implemented.

Synchronization

This DIM operates only in read asynchronous, write asynchronous mode.

Detaching

A tape is unloaded when detached.

Notes

There are some aspects of the Multics standard tape format that are not currently implemented by tape_. There is no provision for handling multireel logical tapes and the synchronous indicator of a record header is never set. The padding bit pattern word used is a word consisting of all ones.

Subroutine Call
2/20/73

Name: term_

The procedure term_ terminates a segment. That is, it does the work of removing a segment from the caller's address space and his combined linkage section. It unsnaps links to the terminated routine and removes references to it (if any) from the command processor's memory. For the term_ entry, links to the segment are not unsnapped unless the segment has a linkage section.

Usage

```
declare term_ entry (char(*) aligned, char(*) aligned,
        fixed bin(35));

call term_ (dirpath, ename, code);
```

1) dirpath is the path name of the parent directory of the segment to be terminated. (Input)

2) ename is the entry name of the segment to be terminated. (Input)

3) code is a standard status code. (Output)

Entry: term_$refname

This entry allows termination of a segment by reference name rather than path name. For this entry, links to the segment are always unsnapped even if it has no linkage section.

Usage

```
declare term_$refname entry (char(*) aligned,
        fixed bin(35));

call term_$refname (refname, code);
```

1) refname is the reference name of the segment to be terminated. (Input)

2) code is a standard status code. (Output)

Entry: term_$seg_ptr

This entry allows termination of a segment referenced by a pointer. Links are not unsnapped unless the segment has a linkage section.

Usage

```
declare term_$seg_ptr entry (ptr, fixed bin(35));

call term_$seg_ptr (segptr, code);
```

1) segptr is a pointer to the segment to be terminated. (Input)

2) code is a standard status code. (Output)

Entry: term_$unsnap

This is a special entry identical to the nomakeunknown entry execpt that links to the segment are always unsnapped even if it has no linkage section.

Usage

```
declare term_$unsnap entry (ptr, fixed bin(35));

call term_$unsnap (segptr, code);
```

Arguments are as above.

Entry: term_$single_refname

This entry allows termination of a single reference name. The segment is not terminated unless the specified reference name was the only reference name by which it was known. Links to the segment are always unsnapped even if it has no linkage section.

Usage

```
declare term_$single_refname (char(*) aligned,
        fixed bin(35));

call term_$single_refname(refname, code);
```

1) refname is the reference name to be terminated. (Input)

2) code is a standard status code. (Output)

Notes

The possible status codes returned are:

1) error_table_$invalidsegno;

2) error_table_$seg_unknown;

3) error_table_$nolot;

4) error_table_$loterr.

The subroutine hcs_$terminate_file performs the same operation as does term_, but provides an additional option; hcs_$terminate_seg is the same as term_$seg_ptr with an additional option; and hcs_$terminate performs the same operation as term_$single_refname.

 (END)

Subroutine Call
Development System
03/29/71

Name: timer_manager_

The timer manager is provided to fulfill a specialized need of certain sophisticated programs. A user should be familiar with Interprocess Communication in Multics and the pitfalls of writing programs which may run asynchronously within a process. These pitfalls may be avoided by using only the timer_manager_$sleep entry.

The timer manager allows many cpu usage timers and real time timers to be used simultaneously by a process. The caller may specify for each timer whether a wakeup is to be issued or a specified procedure is to be called when the timer goes off.

Generic Arguments

1) channel — is the name of the event channel (fixed binary(71)) over which a wakeup is desired. Two or more timers may be running simultaneously, all of which may, if desired, issue a wakeup on the same event channel.

2) routine — is a procedure entry point (entry) that will be called when the timer goes off. The routine will be called this way:

```
declare routine entry (ptr, char(*));

call routine (mcptr, name);
```

1) mcptr — is a pointer to a structure containing the machine conditions at the time of the process interrupt. (input)

2) name — is alrm for a real time timer and is cput for a cpu timer. (input)

(See signal_ for a full description of these arguments.) Two or more timers may be running simultaneously, all of which may, if desired, call the same routine.

3) time — is the time (fixed binary(71)) at which the wakeup or call is desired.

4) flags is a two bit string (bit(2)) which determines in just what way time is to be interpreted. The high order bit indicates whether it is an absolute or a relative time. The low order bit indicates whether it is in units of seconds or microseconds. Absolute real time is time since January 1, 1901, 0000 hours Greenwich Mean Time (GMT), i.e., the time returned by clock_. Absolute cpu time is total time used by the process, i.e., the time returned by hcs_$get_usage_values. Relative time is time from when timer_manager_ was called.

"11"b means relative seconds
"10"b means relative microseconds
"01"b means absolute seconds
"00"b means absolute microseconds

Entry: timer_manager_$sleep

This entry point causes the process to go blocked for a period of real time. Other timers that are active will continue to be processed whenever they ring; however, this routine will not return until the real time has been passed.

Usage

```
declare timer_manager_$sleep entry (fixed bin(71), bit(2));

call timer_manager_$sleep (time, flags);
```

The time is always real time; however, it may be relative or absolute, seconds or microseconds, as explained in the Generic Arguments.

Entry: timer_manager_$alarm_call

This entry sets up a real time timer that will call the routine specified when the timer goes off.

Usage

```
declare timer_manager_$alarm_call entry (fixed bin(71),
        bit(2), entry);

call timer_manager_$alarm_call (time, flags, routine);
```

Entry: timer_manager_$alarm_call_inhibit

This entry sets up a real time timer that will call the routine specified with all interrupts inhibited when the timer goes off. When the interrupt is returned from, interrupts will be re-enabled. If the interrupt is not returned from, interrupts will not be re-enabled.

Usage

```
declare timer_$manager_$alarm_call_inhibit entry (fixed
        bin(71), bit(2), entry);

call timer_manager_$alarm_call_inhibit (timer, flags,
     routine);
```

Entry: timer_manager_$alarm_wakeup

This entry sets up a real time timer that will issue a wakeup on the event channel specified when the timer goes off. The event message passed is alarm___.

Usage

```
declare timer_manager_$alarm_wakeup entry (fixed bin(71),
        bit(2), fixed bin(71));

call timer_manager_$alarm_wakeup (time, flags, channel);
```

Entry: timer_manager_$cpu_call

This entry sets up a cpu timer that will call the routine specified when the timer goes off.

Usage

```
declare timer_manager_$cpu_call entry (fixed bin(71),
        bit(2), entry);

call timer_manager_$cpu_call (time, flags, routine);
```

Entry: timer_manager_$cpu_call_inhibit

This entry sets up a cpu timer that will call the routine specified with all interrupts inhibited when the timer goes off. When the interrupt is returned from, interrupts will be re-enabled. If the interrupt is not returned from, interrupts will not be re-enabled.

Usage

```
declare timer_manager_$cpu_call_inhibit entry (fixed
        bin(71), bit(2), entry);

call timer_manager_$cpu_call_inhibit (time, flags,
     routine);
```

Entry: timer_manager_$cpu_wakeup

This entry sets up a cpu timer that will issue a wakeup on the event channel specified when the timer goes off. The event message passed is cpu_time.

Usage

```
declare timer_manager_$cpu_wakeup entry (fixed bin(71),
        bit(2), fixed bin(71));

call timer_manager_$cpu_wakeup (time, flags, channel);
```

Entry: timer_manager_$reset_cpu_call

This entry turns off all cpu timers that will call the routine specified when they go off.

Usage

```
declare timer_manager_$reset_cpu_call entry (entry);

call timer_manager_$reset_cpu_call (routine);
```

Entry: timer_manager_$reset_cpu_wakeup

This entry turns off all cpu timers that will issue a wakeup on the event channel specified when they go off.

Usage

```
declare timer_manager_$reset_cpu_wakeup entry (fixed
        bin(71));

call timer_manager_$reset_cpu_wakeup (channel);
```

Entry: timer_manager_$reset_alarm_call

This entry turns off all real time timers that will call the routine specified when they go off.

Usage

```
declare timer_manager_$reset_alarm_call entry (entry);

call timer_manager_$reset_alarm_call (routine);
```

Entry: timer_manager_$reset_alarm_wakeup

This entry turns off all real time timers that will issue a wakeup on the event channel specified when they go off.

Usage

```
declare timer_manager_$reset_alarm_wakeup entry (fixed
        bin(71));

call timer_manager_$reset_alarm_wakeup (channel);
```

Notes

For most uses of timer_manager_, a cleanup condition handler should be set up that will reset all the timers that might be set by a system of programs. This way, if the system is aborted and released, any timers set up by the system will be reset instead of going off at undesired times.

In order to be used, timer_manager_ must be established as the condition handler for the conditions alrm and cput. This is done automatically by the standard Multics environment. Subsystems which do not use the standard environment should make the following calls when establishing their environment:

```
call condition_ ("alrm", timer_manager_$alarm_interrupt);

call condition_ ("cput",
     timer_manager_$cpu_timer_interrupt);
```

(END)

Subroutine Call
Development System
6/13/72

Name: total_cpu_time_

This procedure returns the total CPU time used by the calling process since it was created. The time includes time spent handling page faults, segment faults, and bound faults for the calling process as well as time spent handling any system interrupt which occurred while the calling process was executing.

Usage

```
declare total_cpu_time_ entry returns (fixed bin (71));

time = total_cpu_time_();
```

1) time is the total CPU time, in microseconds used by the calling process. (Output)

(END)

I/O System Interface Module
10/3/73

Name: tw_

This procedure is the I/O System Interface Module (IOSIM) used to control operation of a console typewriter. It is not directly called by a user program. Instead, the user provides the name tw_ in a call to the I/O system attach entry. He then accomplishes the I/O operations by calling standard I/O system entry points that are independent of the interface module in use. Further information on the I/O system can be found in the MPM Reference Guide section on I/O and details of the I/O system call syntax can be found in the subroutine description for ios_. This write-up explains how the subroutine tw_ interprets the standard I/O system calls.

Usage

```
call ios_$attach (stream_name, "tw_", ttychan, mode,
     status);
```

The normal sequence of process initialization for a local dialup user includes a call to attach the user's typewriter through tw_ to the stream "user_i/o". Thus, the casual user need not concern himself about performing the attach call himself unless he is performing some special operation such as attaching a second typewriter to his process. However, the remainder of this section might still be of interest since it details the way in which the module tw_ interprets the standard I/O system calls directed to the stream "user_i/o" and any other streams that are attached to "user_i/o".

Notes

This IOSIM supports all devices used as local consoles in the Multics system. It expects the device to have both an input (usually a keyboard) and an output (a printer or a video display) device. The devices currently supported include the IBM 1050, IBM 2741, Bell model 33, 35, 37, and 38 Teletypes, ARDS, GE Terminet 300, Datel 30, and any devices presenting an interface to the system equivalent to any of the above.

Most data presented to the IOSIM should be standard Multics character strings (sequences of 7-bit Multics character codes right adjusted in 9-bit fields). Conversion to the character code of the device is performed automatically by the IOSIM. The exceptions occur when in raw input or output mode (see Modes below) or when in graphic input or output mode when communicating with graphic devices. In normal operation, the IOSIM performs

standard Multics canonicalization and erase-and-kill processing as described in the MPM Reference Guide section, Typing Conventions. Tab positions on all terminals are assumed to be placed in every tenth character position and automatically replace spaces, when appropriate, on output.

Part of the Multics quit mechanism is included in this IOSIM. When the key or keys on the terminal designated as the quit key are activated, the IOSIM immediately aborts and discards input or output currently being performed on this terminal and echoes a new line character. It then sends a quit interrupt to the process that currently owns this terminal. If the process signalled is in the standard Multics command environment, the computation in progress is suspended and the process returns to command level.

Permitted I/O System Calls

The following I/O system calls are implemented by this IOSIM:

abort
attach
changemode
detach
getsize
order
read
resetread
resetwrite
write

Device Identifiers

Terminals handled by this IOSIM are usually connected to the system via telephone lines; therefore the identifiers presented to the IOSIM at attach time correspond to channels connected to particular lines rather than to the actual device. Therefore, the same terminal can have different device identifiers in different terminal sessions. Channel identifiers are character strings of up to six characters.

Modes

The following modes can be specified in calls to attach and changemode.

erkl — specifies that erase-and-kill processing is to be performed on input. (Default is on.)

can — indicates that standard canonicalization is to be performed. (Default is on.)

rawi — indicates that the data specified is to be read from the device directly without any conversion or processing. (Default is off.)

rawo — indicates that data is to be written to the device directly without any conversion or processing. (Default is off.)

tabs — indicates that tabs are to be inserted in output in place of spaces when appropriate. (Default is off for model 33, 35 and 38 teletypes; default is on for all other terminal types).

edited — causes printing of characters for which there is no defined Multics equivalent on the device referenced to be suppressed. If edited mode is off, the 9-bit octal representation of the character is printed. (Default is off.)

esc — enables escape processing (see the MPM Reference Guide section, Typing Conventions) on all input read from the device. (Default is on.)

red — specifies that red and black shifts are to be sent to the terminal. (Default is off for GE terminet 300s and for all terminals without an answerback identifier; default is on for all other terminals.)

crecho — specifies that a carriage return is to be echoed when a line feed is typed. (Default is off; this mode is only functional with model 33, 35, 37, and 38 Teletypes and with GE Terminet 300s.)

lln — specifies the length in character positions of a console line. If an attempt is made to output a line longer than this length, the

excess characters are placed on the next line. (Default line length is 130 for IBM 1050s, 125 for IBM 2741s, 88 for TTY37s, 118 for Terminet 300s, for ARDS, 74 for TTY33s and TTY35s, and 125 for TTY38s.)

pln — specifies the length in lines of a page. When an attempt is made to exceed this length, an ARDS "DEL" character is printed; when the user types an erase character, the output continues with the next page. This mode is functional only for ARDS terminals. (Default page length is 50 for ARDS.)

hndlquit — specifies that when a quit is detected, a new line character is echoed and a resetread of the associated stream is performed. (Default is on.)

default — is a shorthand for erkl, can, ^rawi, ^rawo, and esc. The settings for other modes are not affected.

Returned Status

Only standard Multics error codes are returned as the first half of the status string. The first half of the status string being nonzero indicates an error. At present, none of the bits in the second half of the status string are meaningful.

Order Requests

The following order requests are implemented by this DIM:

hangup — causes the telephone line connection of the terminal to be disconnected, if possible.

listen — cause a wakeup to be sent to the process if the line associated with this device ID is dialed up.

info — causes information about the device to be returned. The pointer argument should point to the following structure that is filled in by the call.

```
declare 1 info_structure aligned,
          2 id char(4) unaligned,
          2 reserved char(8) unaligned,
          2 tw_type fixed bin;
```

1) id is the identifier of the specific device as told to Multics by the device when the device is initialized.

2) reserved is space reserved for compatibility purposes.

3) tw_type identifies the type of device:

1 = IBM 1050;
2 = IBM 2741 (with M.I.T. modifications);
3 = Teletype model 37;
4 = Terminet 300;
5 = ARDS;
6 = IBM 2741 (standard);
7 = Teletype models 33 or 35;
8 = Teletype model 38.

quit_enable causes quit processing to be enabled for this device. (Quit processing is initially disabled.)

quit_disable causes quit processing to be disabled for this device.

start causes a wakeup to be signalled on the event channel associated with this device. This request is used to restart processing on a device whose wakeup may have been lost or discarded.

printer_off causes the printer mechanism of the console to be temporarily disabled if it is physically possible for the terminal to do so.

printer_on causes the printer mechanism of the terminal to be re-enabled.

Element Size

Only an element size of nine is permitted.

Break Characters and Read Delimiters

The only permitted break character and read delimiter is the new line character. There is currently an implementation restriction such that new line characters that have been typed in as escape sequences are not recognized as read delimiters.

Synchronization

This DIM operates only in read asynchronous, write asynchronous, workspace synchronous mode. The limits of read-ahead and write-behind are determined by the IOSIM at call time and are dependent upon the load on the IOSIM by all of its users on the system and, therefore, can vary from call to call.

 (END)

Subroutine Call
3/19/73

Name: unique_bits_

This subroutine returns a unique bit string useful as an identifier. It is obtained by reading the system clock which returns the number of microseconds elapsed since January 1, 1901, 0000 hrs GMT. The bit string is unique among all bit strings obtained in this manner in the history of a Multics installation.

Usage

```
declare unique_bits_ entry returns (bit(70));

bit_string = unique_bits_();
```

1) bit_string is the unique bit string. (Output)

 (END)

Subroutine Call
2/15/73

Name: unique_chars_

The procedure unique_bits_ provides the user with a source of bit-string identifiers guaranteed to differ from all other identifiers generated by that procedure. The procedure unique_chars_ provides a character string representation of such a unique bit string.

Usage

```
declare unique_chars_ entry (bit(*)) returns (char(15));

char_string = unique_chars_(bits);
```

1) char_string is a unique character string. (Output)

2) bits is a bit string of up to 70 bits. (Input)

Notes

If the bits argument is less than 70 bits in length, unique_chars_ pads it with zeros on the right to produce a 70-bit string. If the bits argument equals zero, unique_chars_ calls unique_bits_ to obtain a unique bit string. Note that if the bits argument is supplied (non-zero) and is not a unique bit string, the character string returned by unique_chars_ cannot be guaranteed to be unique.

The first character in the character string produced is always ! (exclamation point) to identify the string as a unique identifier. The remaining 14 characters, forming the unique identifier, are alphanumeric. All vowels are omitted to avoid accidently forming a name likely to be chosen by a person.

 (END)

Subroutine Call
Development System
4/13/72

Name: unpack_system_code_

This procedure must be used to convert certain packed status codes, returned by system procedures using nonstandard status codes, into Multics standard status codes, which may be used in calling com_err_ or command_query_, or which may be compared with codes defined in the standard status code table. (See the MPM Reference Guide sections on Strategies for Handling Unusual Occurrences and List of System Status Codes and Meanings. See also the MPM Subroutines section for the write-ups of com_err_ and command_query_.)

unpack_system_code_ converts bit strings of length greater than 12 and less than 36 which have been returned by such procedures as hcs_$acl_add, hcs_$acl_delete, hcs_$acl_list, and hcs_$acl_replace to fixed binary(35) aligned values. (See the MPM Subroutines section for the aforementioned procedures.)

Those procedures which use nonstandard packed status codes will be phased out in the future, at which time all status codes will be stored in standard form. unpack_system_code_ will no longer be required then.

Usage

```
declare unpack_system_code_ entry (bit(*) unaligned)
        returns (fixed bin(35) aligned);

code = unpack_system_code_ (pack_code);
```

1) pack_code is a packed code. For an example of a packed code, see the document on hcs_$acl_add. (Input)

2) code is the standard status code corresponding to the packed value given. (Output)

Notes

This procedure will only work for packed codes returned by system subroutines. There is no way to pack user-defined standard status codes into less than 36 bits.

(END)

Subroutine Call
4/5/73

Name: user_info_

This procedure allows the user to obtain information concerning his login session.

Entry: user_info_

This entry returns the user's login name, project id, and account id.

Usage

```
declare user_info_ entry (char(*), char(*), char(*));

call user_info_ (name, proj, acct);
```

1) name is the user's name from the login line (maximum of 22 characters). (Output)

2) proj is the user's project ID (maximum of 9 characters). (Output)

3) acct is the user's account ID (maximum of 32 characters). (Output)

Entry: user_info_$whoami

This is the same as the user_info_ entry point. The added name is for mnemonic convenience.

Usage

```
declare user_info_$whoami entry (char(*), char(*),
        char(*));

call user_info_$whoami (name, proj, acct);
```

Arguments are as above.

Entry: user_info_$login_data

This entry returns useful information about how the user logged in.

Usage

```
declare user_info_$login_data entry (char(*), char(*),
        char(*), fixed bin, fixed bin, fixed bin,
        fixed bin(71), char(*));

call user_info_$login_data (name, proj, acct, anon, stby,
     weight, time_login, login_word);
```

1) name is as above. (Output)

2) proj is as above. (Output)

3) acct is as above. (Output)

4) anon = 1 if the user is an anonymous user. (Output)

5) stby = 1 if the user is standby (i.e., may be preempted). (Output)

6) weight = 10 times the user's weight. (Output)

7) time_login is the time the user logged in. It is expressed as a calendar clock reading in microseconds. (Output)

8) login_word is login or enter, reflecting which command the user logged in by. (Output)

Entry: user_info_$usage_data

This entry returns user usage data.

Usage

```
declare user_info_$usage_data entry (fixed bin,
        fixed bin(71), fixed bin(71), fixed bin(71));

call user_info_$usage_data (nproc, old_cpu,
     time_login, time_create);
```

1) nproc is the number of processes created for this login session. (Output)

2) old_cpu is the CPU time used by previous processes in the login session. (Output)

3) time_login is the time of login. (Output)

4) time_create is the time the process was created. (Output)

Entry: user_info_$homedir

This entry returns the path name of the user's initial working directory.

Usage

```
declare user_info_$homedir entry (char(*));

call user_info_$homedir (hdir);
```

1) hdir is the path name of the user's home directory (maximum of 64 characters). (Output)

Entry: user_info_$responder

This entry returns the name of the user's login responder.

Usage

```
declare user_info_$responder entry (char(*));

call user_info_$responder (resp);
```

1) resp is the name of the user's login responder (maximum of 64 characters). (Output)

Entry: user_info_$tty_data

This entry returns information about the user's process' terminal.

Usage

```
declare user_info_$tty_data entry (char(*), fixed bin,
        char(*));

call user_info_$tty_data (idcode, type, channel);
```

1) idcode is the terminal ID code of the user's (maximum of 4 characters). (Output)

2) type is the type of terminal:

0 = absentee process or network user;
1 = IBM 1050;
2 = IBM 2741 (with MIT modifications);
3 = Teletype Model 37;
4 = Terminet 300;
5 = ARDS;
6 = IBM 2741 (unmodified);
7 = Teletype Model 33, Model 35. (Output)

3) channel is the channel identification (maximum of 8 characters). (Output)

Entry: user_info_$logout_data

This entry returns the event channel over which logout is to be signalled and the process ID to which the signal is to be directed.

Usage

```
declare user_info_$logout_data entry (fixed bin(71),
        bit(36) aligned);

call user_info_$logout_data (logout_channel, logout_pid);
```

1) logout_channel is the event channel over which logouts are to be signalled. (Output)

2) logout_pid is the process ID of the answering service. (Output)

Entry: user_info_$absin

This entry returns the path name of the absentee input segment for an absentee job. For an interactive user it is returned as blanks.

Usage

```
declare user_info_$absin entry (char(*));
```

```
call user_info_$absin (path);
```

1) path is the path name of the absentee input segment. (Output)

Entry: user_info_$absout

This entry returns the path name of the absentee output segment for an absentee job. For an interactive user it is returned as blanks.

Usage

```
declare user_info_$absout entry (char(*));

call user_info_$absout (path);
```

1) path is the path name of the absentee output segment. (Output)

Entry: user_info_$limits

This entry returns the per-user limit values established by the user's project administrator, and the user's spending against the limits.

If a limit is specified as open, the limit value returned is 1.0e37.

Usage

```
declare user_info_$limits entry (float bin, float bin,
        fixed bin(71), fixed bin, (0:7) float bin,
        float bin, float bin, (0:7) float bin);

call user_info_$limits (mlim, clim, cdate, crf, shlim,
     msp, csp, shsp);
```

1) mlim is the month limit in dollars. (Output)

2) clim is the cutoff limit in dollars. (Output)

3) cdate is the cutoff date. (Output)

4) crf — is the cutoff refresh code. This indicates what will happen at the cutoff date:

0 - permanent cutoff
1 - add one day
2 - add one month
3 - add one year
4 - add one calendar year
5 - add one fiscal year. (Output)

5) shlim — is the array of shift limits in dollars. (Output)

6) msp — is the month-to-date spending in dollars. (Output)

7) csp — is the spending against the cutoff limit in dollars. (Ouput)

8) shsp — is the spending against shift limits in dollars. (Output)

Note

All entries which take more than one argument will count their arguments and not attempt to return more values than there are arguments.

 (END)

Subroutine Call
7/9/73

Name: write_list_

This procedure is used to print out on the user's terminal the values of internal variables with two spaces between succeeding values. The data to be written out must be one of the five types: real decimal fixed-point number, real decimal floating-point number, bit-string, character-string, or pointer.

Since this procedure can be called with a varying number of arguments, it is not permissible to include a parameter attribute list in the declaration of the various entry points.

Usage

```
declare write_list entry options (variable);

call write_list_ (arg1, arg2, ..., argn);
```

1) argi is any variable of one of the five types listed above. (Output)

Entry: write_list_$nnl

This entry is identical in usage and output to write_list_, except that no new line character is appended to the output string.

Usage

```
declare write_list_$nnl entry options (varible);

call write_list_$nnl (arg1, arg2, ..., argn);
```

Arguments are as above.

Notes

The maximum number of arguments is 64.

The data type of each argument is obtained from the descriptor, and conversion takes place to transform the value of each argument to its appropriate character string representation on the terminal.

A real decimal floating-point number is printed in E format, with 8 decimal places for a single precision number and 19 decimal places for a double precision number.

A character string is printed without the enclosing quotes.

Example

The following procedure produces the output indicated below.

```
x:   proc;
     declare a float, b char(5), c bit(3), d fixed bin;
     declare s float, m fixed bin;
     declare write_list_ external entry options (variable);
     .
     call write_list_ (a,b,c,d);
     .
     call write_list_ ("x=", s, "m=", m);
     .
     end x;
```

The two lines printed on the terminal are:

```
ßß0.17500000e-04ßßnameßßß"101"bßß-5
ßßx=ßß-0.21200000e+03ßßm=ßß3192
```

 (END)

INDEX

This Index covers only Part II of the manual, namely the Reference Guide sections 1 to 8, and the command and subroutine write-ups.

The Index is organized around the numerically ordered Reference Guide sections and the alphabetically ordered commands and subroutine write-ups, rather than by page number. Thus, for example, the entry for bulk input and output might read:

```
bulk I/O
   3.4
   4.4
   dprint
   dpunch
```

The first two items under bulk I/O refer to the Reference Guide sections 3.4 and 4.4, and the last two to the write-ups for the dprint and dpunch commands. They are referenced in the order that they appear in this manual. Note that command names can normally be distinguished from subroutines by the trailing underscore in the segment name of subroutines.

Some entries are of the form:

```
I/O (bulk)
   see bulk I/O
```

For simplicity of usage, these entries always refer to other places in the Index, never to normal Reference Guide, command or subroutine write-ups.

Some entries are followed by information within parentheses. This information serves to explain the entry by giving a more complete name or the name of the command under which the actual entry can be found. For example:

```
e (enter)
listnames (list)
```

In addition to this Index, other indexes to information are:

1) MPM Table of Contents
   - lists names of commands and subroutines with write-up issue dates
   - lists commands and subroutines documented under other write-ups; e.g., console_output: see file_output

2) Reference Guide Section 1.1: The Multics Command Repertoire
   - lists commands by function

3) Reference Guide Section 2.1: The Multics Subroutine Repertoire
   - lists subroutines by function

4) Reference Guide Section 8.3: Obsolete Procedures

(continued)

(END)