Honeywell

MULTICS PROGRAMMERS' MANUAL – SUBROUTINES ADDENDUM C

SERIES 60 (LEVEL 68)

SOFTWARE

SUBJECT:

Additions and Changes to the Standard Multics Subroutines.

SPECIAL INSTRUCTIONS:

This is the third addendum to AG93, Revision 1, dated May 1975.

Insert the attached pages into the manual according to the collating instructions on the back of this cover. The following new subroutines have been added to Section II and do not contain change bars:

get_line_length_
get_temp_segments_
release_temp_segments_
send_mail_

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

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

SOFTWARE SUPPORTED:

Multics Software Release 4.0

DATE:

July 1976

ORDER NUMBER:

AG93C, Rev. 1

15828
2576
Printed in U.S.A.

COLLATING INSTRUCTIONS

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

Remove	Insert
iii through vi	iii through vi
1-1 through 1-4	1-1 through 1-4
2-7, 2-8	2-7, 2-8
2-25.1 through 2-28	2-25.1 through 2-28
2-69 through 2-72	2-69 through 2-72
2-81, 2-82	2-81, 2-82
2-84.1 through 2-88	2-84.1 through 2-88
2-99 through 2-101	2-99 through 2-101
	2-110.1 through 2-110.3
2-126.1 through 2-126.4	2-126.1 through 2-126.4
3-4.5, 3-4.6	3-4.5, 3-4.6
3-13 through 3-20.1	3-13 through 3-24

AG93C

File No.: 1L13

Honeywell

MULTICS PROGRAMMERS' MANUAL – SUBROUTINES ADDENDUM B

SERIES 60 (LEVEL 68)

SOFTWARE

SUBJECT:

Additions and Changes to the Standard Multics Subroutines.

SPECIAL INSTRUCTIONS:

This is the second addendum to AG93, Revision 1, dated May 1975.

Insert the attached pages into the manual according to the collating instructions on the back of this cover. The vfile_status_ subroutine is new and does not contain change bars. Throughout the rest of the manual, change bars in the margins indicate technical additions and changes; asterisks denote deletions. These changes will be incorporated into the next revision of the manual.

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

SOFTWARE SUPPORTED:

Multics Software Release [redacted] 3.1

DATE:

March 1976

ORDER NUMBER:

AG93B, Rev. 1

15009
1276
Printed in U.S.A.

COLLATING INSTRUCTIONS

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

Remove	Insert
iii through vi	iii through vi
1-3, 1-4	1-3, 1-4
2-9, 2-9.1	2-9, 2-9.1
2-81, 2-82 2-84.1, 2-84.2	2-81, 2-82 2-84.1, 2-84.2
2-87 through 2-90	2-87 through 2-90
2-93 through 2-100	2-93 through 2-100
2-125, 2-126	2-125, 2-126 2-126.1, 2-126.2 2-126.3, blank
3-13 through 3-16	3-13 through 3-16
3-17 through 3-20	3-17, 3-18 3-18.1, 3-18.2 3-19, 3-20 3-20.1, blank
A-1 through A-16	
B-1 through B-7	

Honeywell

MULTICS PROGRAMMERS' MANUAL – SUBROUTINES ADDENDUM A

SERIES 60 (LEVEL 68)

SOFTWARE

SUBJECT:

Additions and Changes to the Standard Multics Subroutines.

SPECIAL INSTRUCTIONS:

This is the first addendum to AG93, Revision 1, dated May 1975.

Insert the attached pages into the manual according to the collating instructions on the back of this cover. The following new subroutines and I/O modules have been added to Sections II and III respectively and do not contain change bars.

convert_authorization_
get_authorization_
get_max_authorization_
hcs_$create_branch_
hcs_$get_access_class
hcs_$get_access_class_seg
print_cobol_error_
rdisk_ (I/O module)
tape_ansi_ (I/O module)
tape_ibm_ (I/O module)

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

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

SOFTWARE SUPPORTED:

Multics Software Release 3.0

DATE:

September 1975

ORDER NUMBER:

AG93A, Rev. 1

13904D
1875
Printed in U.S.A.

COLLATING INSTRUCTIONS

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

Remove	Insert
iii through vi	iii through vi
1-1 through 1-4	1-1 through 1-4
2-9, 2-10	2-9, 2-9.1 2-9.2, 2-9.3 2-9.4, 2-10
2-21, 2-22	2-21, 2-22
	2-24.1, blank
2-25, 2-26	2-25, blank 2-25.1, 2-26
2-29 through 2-32	2-29 through 2-32
2-41 through 2-44	2-41, blank 2-41.1, 2-41.2 2-41.3, 2-42 2-43, 2-44
	2-52.1, 2-52.2
2-53 through 2-58	2-53, 2-54 2-54.1, 2-54.2 2-55 through 2-58
2-61, 2-62	2-61, 2-62
2-73, 2-74	2-73, 2-74
2-79, 2-84	2-79 through 2-84 2-84.1, 2-84.2
2-85, 2-86	2-85, 2-86
2-91, 2-92	2-91, blank 2-91.1, 2-92
2-99 through 2-102	2-99, 2-100 2-101, blank 2-101.1, 2-102
3-1, 3-2	3-1, 3-2
	3-4.1 through 3-4.6

File No.: 1L13

	3-10.1 through 3-10.58
3-11 through 3-17	3-11 through 3-20
A-1 through A-4	A-1 through A-4
A-7, A-8	A-7, A-8

August 20, 1975

Massachusetts Institute of Technology
Information Processing Center
Cambridge, Massachusetts 02139

Publications
39-483

This manual (MPM Volume 4, AG93-1) is part of a rather extensive revision of the Honeywell MPM. It is meant to replace Section 10 (Subroutines) of Part 2 of the M.I.T. MPM.

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

Subroutine	Reason*
active_fnc_err_	SWG
broadcast_	obsolete (ios_)
check_star_name_	SWG
condition_	obsolete
convert_binary_integer_	obsolete?
copy_acl_	AN67
copy_names_	AN67
copy_seg_	AN67
cu_	(most entry points moved to SWG)
cv_acl_	AN67
cv_bin_	SWG
cv_dec_	obsolete
cv_dir_acl_	AN67
cv_dir_mode_	AN67
cv_float_	obsolete
cv_mode_	AN67
cv_oct_	SWG
cv_userid_	AN67
decode_descriptor_	SWG
decode_entryname_	AN67
discard_output_	obsolete (ios_)
encipher_	AN51 (not included yet)

*SWG means moved to the Subsystem Writers' Guide (Vol. 5).

(over)

file_	obsolete (ios_)
find_condition_info_	SWG
get_default_wdir_	SWG
get_equal_name_	SWG
hcs_$del_dir_tree	SWG
hcs_$star_	SWG
ios_	obsolete
match_star_name_	SWG
move_	SWG
move_names_	SWG
nstd_	obsolete (ios_)
object_info_	SWG
parse_file_	AN51
plot_	GUS (not included yet)
read_list_	obsolete
reversion_	obsolete
signal_	SWG
stu_	SWG
suffixed_name_	AN67
syn	obsolete (ios_)
tape_	obsolete (ios_)
timer_manager_	SWG
total_cpu_time_	AN51 (not included yet)
tw_	obsolete (ios_)
unpack_system_code_	obsolete
write_list_	obsolete

Honeywell

MULTICS PROGRAMMERS' MANUAL
SUBROUTINES

SERIES 60 (LEVEL 68)

SUBJECT:

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

SPECIAL INSTRUCTIONS:

This manual is one of four manuals that constitute the <u>Multics Programmers´ Manual</u> (MPM).

<u>Reference Guide</u>	Order No. AG91
<u>Commands and Active Functions</u>	Order No. AG92
<u>Subroutines</u>	Order No. AG93
<u>Subsystem Writers´ Guide</u>	Order No. AK92

This manual supersedes AG93, Rev. 0, and its Addendum A. The manual has been extensively revised; therefore, marginal change indicators have not been included in this edition.

Appendix A of this document contains input/output (I/O) system information that properly belongs in Section IV of the MPM Reference Guide. In the interest of providing the information to Multics users at an early date, the I/O system discussion is contained in this document temporarily; most of the material in the appendix will be moved to the MPM Reference Guide when that document is next revised.

SOFTWARE SUPPORTED:

Multics Software Release 2.1

DATE:

May 1975

ORDER NUMBER:

AG93, Rev. 1

PREFACE

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

Document	Referred To In Text As
Reference Guide (Order No. AG91)	MPM Reference Guide
Commands and Active Functions (Order No. AG92)	MPM Commands
Subroutines (Order No. AG93)	MPM Subroutines
Subsystem Writers' Guide (Order No. AK92)	MPM Subsystem Writers' Guide

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

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

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

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

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

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

Several cross-reference facilities help locate information:

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

CONTENTS

Page

SECTION I

INTRODUCTION TO STANDARD SUBROUTINES

The subroutines described in this document are the basic set included in the standard Multics system. Many of the functions described here are also provided as runtime features of Multics-supported programming languages. The user is encouraged to use language-related facilities wherever possible.

Most local installations maintain a library of additional procedures that augment the standard repertoire. The user should consult the list of items in the Installation Maintained Library at the local installation. (Documentation of these procedures is supplied by the local installation.)

This section presents the subroutine repertoire, organized by function into the following categories:

Storage System, Utility Procedures
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 Procedures
Subroutine Call and Argument Procedures
Command Environment Utility Procedures
Input/Output System Procedures
Error Handling Procedures
Data Type Conversion Procedures
Miscellaneous Procedures

Section II provides a detailed description of all subroutines except the I/O modules, which are presented in Section III. The descriptions in both of these sections are presented alphabetically for ease of reference.

Storage System, Utility Procedures

change_wdir_	changes user's current working directory
delete_	deletes segments and directories and unlinks links
expand_path_	converts relative pathname to absolute pathname
get_pdir_	returns pathname of process directory
get_temp_segments_	acquires temporary segments in the process directory
get_wdir_	returns pathname of current working directory
release_temp_segments_	returns temporary segments to the free pool
term_	removes a segment from the address space, unsnapping any subroutine linkage to it

Storage System, Access Control and Rings of Protection

convert_authorization_	converts an authorization back and forth between its binary and character-string representation
get_authorization_	returns authorization value of the process
get_group_id_	returns access control name of current useB"
get_max_authorization_	returns maximum authorization value of the process
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_$fs_get_mode	returns access control mode for a given segment relative to the current validation level
hcs_$get_access_class hcs_$get_access_class_seg	returns access class for a segment or a directory
hcs_$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

Storage System, Supervisor Entries for Manipulating Directories and Segments

hcs_$append_branch hcs_$append_branchx	creates a segment or a directory
hcs_$append_link	creates a directory link
hcs_$chname_file hcs_$chname_seg	adds, deletes, and changes names found in a directory
hcs_$create_branch_	creates a segment or directory, sets a number of attributes
hcs_$delentry_file hcs_$delentry_seg	deletes a single entry in a directory
hcs_$fs_move_file hcs_$fs_move_seg	moves contents of one segment to another
hcs_$make_seg	creates a new segment and then initiates it
hcs_$set_bc hcs_$set_bc_seg	sets the bit count of a segment
hcs_$status_	returns information about a given segment, directory, or link
hcs_$truncate_file hcs_$truncate_seg	truncates a file or segment to a given length

See also the "Storage System, Utility Procedures" category.

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 pathname 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_$initiate	initiates a segment and returns its segment number
hcs_$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 address space

See also the term_ and change_wdir_ subroutines in the "Storage System, Utility Procedures" category.

Clock and Timer Procedures

clock_	reads calendar clock
convert_date_to_binary_	converts an ASCII string to binary time
cpu_time_and_paging_	returns virtual CPU time used and paging activity of the process
date_time_	converts binary time to an ASCII string
decode_clock_value_	converts a binary time value into an ASCII string
virtual_cpu_time_	returns virtual CPU time used by this process

Command Environment Utility Procedures

cu_$arg_count	returns number of arguments supplied to the called procedure
cu_$arg_ptr	returns a pointer to a specified argument in current argument list
cu_$cp	calls the command processor to execute a command line
change_wdir_	changes user's current working directory
expand_path_	expands a relative pathname into an absolute pathname
get_pdir_	returns pathname of process directory
get_temp_segments_	acquires temporary segments in the process directory
get_wdir_	returns pathname of current working directory
release_temp_segments_	returns temporary segments to the free pool

Input/Output System Procedures

discard_	provides infinite sink for output (I/O module)
get_line_length_	returns the line length of an I/O switch
ioa_	produces formatted printed output
iox_	interfaces with the Multics I/O system
ntape_	supports I/O from/to files on tape (I/O module)
rdisk_	supports I/O from/to removable disk packs (I/O module)
record_stream_	maps stream calls into record calls or vice versa (I/O module)
syn_	makes one switch name equivalent to another (I/O module)

tape_ansi_	supports I/O from/to tapes written in proposed American National Standards Institute (ANSI) format (I/O module)
tape_ibm_	supports I/O from/to tapes written in IBM standard format (I/O module)
tty_	supports I/O from/to terminals (I/O module)
vfile_	supports I/O from/to segments and multisegment files in the storage system (I/O module)
vfile_status_	returns information about a storage system file supported by the vfile_ I/O module

Error Handling Procedures

com_err_	prints a standard status message for common errors
command_query_	handles questions generated by commands
print_cobol_error_	prints error messages produced by COBOL programs

Date Conversion Procedures

convert_date_to_binary_	converts ASCII string to binary clock reading
date_time_	convert clock reading to ASCII string
decode_clock_value_	converts a binary time value into an ASCII string

Miscellaneous Procedures

adjust_bit_count_	sets bit count of a segment to last nonzero character
get_process_id_	returns identification of current process
random_	returns random numbers
send_mail_	sends a message and an optional wakeup to a user
set_lock_	allows multiple processes to synchronize their use of shared data
unique_bits_	returns a unique bit string
unique_chars_	converts a unique bit string to a unique character string
user_info_	returns miscellaneous information about the current user

SECTION II

SUBROUTINE DESCRIPTIONS

This section contains descriptions of the Multics subroutines, presented in alphabetic order. Each description contains the name of the subroutine, discusses the purpose of the subroutine, lists the entry points, and describes the correct usage for each entry point. Notes and examples are included when deemed necessary for clarity. The discussion below briefly describes the context of the various divisions of the subroutine descriptions.

Name

The "Name" heading shows the acceptable name by which the subroutine is called. The name is usually followed by a discussion of the purpose and function of the subroutine and the results that may be expected from calling it.

Entry

Each "Entry" heading lists an entry point of the subroutine call. This heading may or may not appear in a subroutine description; its use is entirely dependent upon the purpose and function of the individual subroutine.

Usage

This part of the subroutine description first shows the proper format to use when calling the subroutine and then explains each element of the call. Generally, the format is shown in two parts: a declare statement that gives the number and describes (in PL/I notation) the arguments that can be used and the subroutine call line(s) that gives an example of correct use. Each variable element or control argument of the subroutine call is then explained. Arguments can be assumed to be required unless otherwise specified. Arguments that must be defined before calling the subroutine are identified as Input; those arguments defined by the subroutine are identified as Output.

Notes

Comments or clarifications that relate to the subroutine as a whole (or to an entry point) are given under the "Notes" heading.

Other Headings

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

Status Codes

The standard status codes returned by the subroutines are further identified, when appropriate, as either storage system or I/O system. For convenience, the most often encountered codes are listed in Appendix B in three categories: storage system, I/O system, and other. Certain codes have been included in the individual subroutine description if they have a special meaning in the context of that subroutine. The reader should not assume that the code(s) given in a particular subroutine description are the only ones that can be returned.

Treatment of Links

Generally, whenever the programmer references a link, the subroutine action is performed on the entry pointed to by the link. If this is the case, the only way the programmer can have the action performed on the link itself is if the subroutine has a chase switch and he sets the chase switch to 0.

Name: adjust_bit_count_

The adjust_bit_count_ subroutine performs the basic work of the adjust_bit_count command (described in the MPM Commands). The adjust_bit_count_ subroutine is called to find the last nonzero word or character of a segment or multisegment file 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(35));

     call adjust_bit_count_ (dir_name, entryname, char_sw, bit_count, code);
```

where:

1. dir_name — is the pathname of the containing directory. (Input)

2. entryname — is the entryname of the segment. (Input)

3. char_sw — is the character switch. (Input)
   "0"b adjusts to last bit of last nonzero word
   "1"b adjusts to last bit of last nonzero character

4. bit_count — is the computed bit count for the segment. If the value is less than 0, it indicates that no attempt to compute the count was made (code is nonzero). If the value is greater than or equal to 0, the computed value is correct, whether or not the bit count could be set. (Output)

5. code — is a standard status code. It is 0 if the operation was successful. (Output)

Name: change_wdir_

The change_wdir_ subroutine changes the user's current working directory to the directory specified.

Usage

```
declare change_wdir_ entry (char(168) aligned, fixed bin(35));

call change_wdir_ (path, code);
```

where:

1. path is the pathname of the directory that is to become the user's working directory. (Input)

2. code is a storage system status code. (Output)

Name: clock_

The clock_ subroutine reads the system clock and returns a fixed binary number equal to the number of microseconds since 0000 hours Greenwich mean time January 1, 1901. The returned time is suitable for input to the date_time_ or decode_clock_value_ subroutines, which convert the clock reading to an ASCII representation.

Usage

```
     declare clock_ entry returns (fixed bin(71));

     date_time = clock_ ();
```

where date_time is the number of microseconds since January 1, 1901, 0000 hours Greenwich mean time. (Output)

Name: com_err_

The com_err_ subroutine is the principal subroutine used by commands for printing error messages. It is usually called with a nonzero status code to report an unusual status condition. It may also be called with a code of 0 to report an error not associated with a status code.

Since this subroutine can be called with a varying number of arguments, it is not permissible to include a parameter attribute list in the declaration.

See also "Strategies for Handling Unusual Occurrences" in Section VI of the MPM Reference Guide.

Entry: com_err_

This entry point formats an error message and then signals the condition command_error. The default handler for this condition simply returns control to the com_err_ subroutine, which then writes the error message on the I/O switch error_output.

Usage

```
     declare com_err_ entry options (variable);

     call com_err_ (code, caller, control_string, arg1, ..., argn);
```

where:

1. code is a standard status code (fixed bin(35)). (Input)

2. caller is the name (char(*)) of the procedure calling the com_err_ subroutine. It can be either varying or nonvarying. (Input)

3. control_string is an ioa_ subroutine control string (char(*)). This argument is optional. See "Notes" below. (Input)

4. argi are ioa_ subroutine arguments to be substituted into control_string. These arguments are optional. (However, they can only be used if the control_string argument is given first.) See "Notes" below. (Input)

Notes

The error message prepared by the com_err_ subroutine has the following format:

```
caller:  system_message user_message
```

where:

1. caller — is the name of the program detecting the error.

2. system_message — is a standard message from the system data base error_table_ corresponding to the value of code. If code is equal to 0, no system_message is printed.

3. user_message — is constructed by the ioa_ subroutine from the control_string and argi arguments. If the control_string and argi arguments are not given, user_message is omitted.

If the com_err_ subroutine is passed a nonzero code that does not correspond to a standard format error table entry, the system message is of the form:

```
Code ddd.
```

where ddd is the decimal representation of code. The argument caller must not be null or blank; if it is, the handlers for command_error cannot identify the signalling procedure.

Entry: com_err_$suppress_name

The com_err_$suppress_name entry point should be used when the caller name and colon are not wanted. The caller name is still passed to the command_error condition handler. Otherwise, this entry point is the same as the com_err_ entry point.

Usage

```
declare com_err_$suppress_name entry options (variable);

call com_err_$suppress_name (code, caller, control_string, arg1, ...,
     argn);
```

where all of the arguments are the same as in the com_err_ entry point.

Name: command_query_

The command_query_ subroutine is the standard system procedure invoked to ask the user a question and to obtain an answer. It formats the question and then signals the condition command_question. See "List of System Conditions and Default Handlers" in Section VI of the MPM Reference Guide. The default handler for this condition simply returns control to the command_query_ subroutine, which writes the question on the I/O switch user_i/o. It then reads the I/O switch user_input to obtain the answer. Several options have been included in the commmand_query_ subroutine to support the use of a more sophisticated handler for the command_question condition.

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.

Usage

```
declare command_query_ entry options (variable);

call command_query_ (ptr, answer, caller, control_string, arg1, ..., argn);
```

where:

1. ptr is a pointer to the following structure. (Input)

```
dcl 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 code                     fixed bin(35),
      2 query_code               fixed bin(35);
```

where:

version — is the version number of this structure. The version number identifies the format of the structure. (Input)

yes_or_no_sw — indicates that an answer of a particular form is expected. (Input)
"0"b accepts any answer
"1"b does not return until a yes or no answer is read

suppress_name_sw — controls whether the name of the calling procedure appears in the question. See "Note" below. (Input)
"0"b includes name
"1"b omits name

code — is either the standard status code that prompted the question or 0. (Input)

query_code — is currently ignored. It is intended for use by specialized handlers for command_question. (Input)

2. answer — is the response (char(*) varying) read from the I/O switch user_input. Leading and trailing blanks plus the newline character have been removed. (Output)

3. caller — is the name (char(*)) of the calling procedure. It can be either varying or nonvarying. (Input)

4. control_string — is an ioa_ subroutine control string (char(*)). This argument is optional. See "Note" below. (Input)

5. argi — are ioa_ subroutine arguments to be substituted into control_string. These arguments are optional. (However, they can only be used if the control_string argument is given first.) See "Note" below. (Input)

Notes

The question prepared by the command_query_ subroutine has the format:

```
caller: message
```

The message is constructed by the ioa_ subroutine from the control_string and argi arguments. If the control_string and argi arguments are not given, the message portion of the question is omitted. If the suppress_name_sw switch is on, the name of the calling procedure and the colon are omitted.

If the user issues a quit signal before responding to the question and then invokes the program_interrupt command, the question is repeated. This feature is useful in case the original question was garbled.

Name: convert_authorization_

The convert_authorization_ subroutine is provided to convert an authorization in the Multics Access Isolation Mechanism (AIM) back and forth between its binary and character-string representations. Additional entries provide the ability to encode an authorization as a short character string for use in entrynames.

Entry: convert_authorization_$from_string

Usage

```
    declare convert_authorization_$from_string entry (bit(72) aligned, char(*),
         fixed bin(35));

    call convert_authorization_$from_string (authorization, string, code);
```

where:

1. authorization is the binary representation of string. (Output)
2. string is the character string to be converted. (Input)
3. code is a standard status code. (Output)

0	no errors in conversion
error_table_$ai_invalid_string	one or more namei is misspelled (See "Notes" below.)
error_table_$ai_above_allowed_max	no error in conversion; but the resulting authorization class is greater than the system_high authorization

Notes

The convert_authorization_$from_string entry point accepts a character string of the form:

name1,name2,...,namen

where namei represents the mnemonic for a sensitivity level or access category. This entry point converts this string to an encoded binary form suitable for storage in system tables and as input to the various modules that accept the binary form. The print_auth_names command (described in the MPM Commands) may be used to obtain a list of acceptable mnemonics.

If the string argument is null or system_low, the resulting authorization is level 0 and no categories. If the string is system_high, the system access_ceiling is returned (the maximum authorization or access class allowed).

Entry: convert_authorization_$to_string

This entry point accepts a binary form of an authorization and returns it as a printable string. This output string is suitable for input to the convert_authorization_$from_string entry point. Each level/category name has a maximum length of 32 characters.

Usage

```
     declare convert_authorization_$to_string entry (bit(72) aligned, char(*),
          fixed bin(35));

     call convert_authorization_$to_string (authorization, string, code);
```

where:

1. authorization is the binary representation of string. (Input)
2. string is the character string to be converted. (Output)
3. code is a standard status code. (Output)

0	no errors in conversion
error_table_$smallarg	supplied output is too short to hold the converted result (See "Note" below.)
error_table_$ai_invalid_binary	either the level number or category set is invalid; the resulting output is also invalid

Note

When the error_table_$smallarg code is returned, as much of the resulting conversion as fits in the output string is returned. However, since the results are not complete, they should not be used as input to the convert_authorization_$from_string entry point.

Entry: convert_authorization_$to_string_short

This entry point is identical to the convert_authorization_$to_string entry point, except that the short level/category names are returned. Each short name has a maximum length of eight characters. This output is also suitable for input to the convert_authorization_$from_string entry point.

Usage

```
     declare  convert_authorization_$to_string_short  entry  (bit(72)  aligned,
          char(*), fixed bin(35));

     call convert_authorization_$to_string_short (authorization, string,  code);
```

where authorization, string, and code are the same as above.

Entry: convert_authorization_$minimum

This entry point accepts an array of authorizations or access classes and a binary number indicating how many elements to process from the array. It returns an authorization/access class whose category set is the intersection of all input category sets and whose sensitivity level is the minimum of all input sensitivity levels. The returned value need not equal any of the input values.

Usage

```
     declare convert_authorization_$minimum entry (dim(*) bit(72) aligned, fixed
          bin, bit(72) aligned);

     call convert_authorization_$minimum (auth_array, n_elements, minimum_auth);
```

where:

1. auth_array are the input authorizations. (Input)

2. n_elements is the number of elements to be processed in the auth_array argument. (Input)

3. minimum_auth is the result. (Output)

Entry: convert_authorization_$encode

This entry point encodes an authorization or access class into a short character string, suitable for inclusion in entrynames. If the input authorization represents system_low, the returned string is blank.

Usage

```
declare convert_authorization_$encode entry (bit(72) aligned, char(*));

call convert_authorization_$encode (authorization, encoded_string);
```

where:

1. authorization is the input authorization. (Input)

2. encoded_string is a short string (maximum of 15 characters) that uniquely represents the input authorization/access class. (Output)

Entry: convert_authorization_$decode

This entry point takes the character string produced by the convert_authorization_$encode entry point and returns the original authorization or access class. The null string is converted to the system_low authorization.

Usage

```
declare convert_authorization_$decode entry (bit(72) aligned, char(*));

call convert_authorization_$decode (authorization, decoded_string);
```

where:

1. authorization is the decoded authorization. (Output)

2. decoded_string is a short string (maximum of 15 characters) that uniquely represents the input authorization/access class. (Input)

Name: convert_date_to_binary_

The convert_date_to_binary_ subroutine converts a character representation of a date and time into a 72-bit clock reading (see the clock_ subroutine). It accepts a wide variety of date and time forms, including the output of the date_time_ subroutine.

Usage

```
declare convert_date_to_binary_ entry (char(*), fixed bin(71),
     fixed bin(35));

call convert_date_to_binary_ (string, clock, code);
```

where:

1. string is the character representation of the clock reading desired. (See "Format of Clock Reading" below.) (Input)

2. clock is set to the computed clock value. It is unchanged in the event of an error. (Output)

3. code is a standard status code. It is either 0 (no errors) or error_table_$date_conversion_error. (Output) The latter is returned in all of the following cases:

   a. General syntax error.
   b. Unrecognized alphabetic field.
   c. Two or more dates, times, etc.
   d. Month without a date number.
   e. Year not in the twentieth century.
   f. Day of month does not exist (e.g., 35 March).
   g. Midnight and noon preceded by an hour other than 12.
   h. Minutes greater than 59.
   i. Seconds greater than 59.
   j. 24-hour time after 2400.0 specified.
   k. Zero hours in meridional time.
   l. Month greater than 12 in slash time.
   m. Minutes or seconds not two decimal places in length.
   n. Day of week and date conflict.
   o. Improper use of comma.
   p. 24-hour time less than three places in length.
   q. Improper use of offset.

Format of Clock Reading

The character representation of the clock reading has up to five parts (date, time, day-of-week, offset, and time zone), all of which are optional. They can 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 and must contain a whole word or an abbreviation made up of the first three letters of the word. That means that Jan and January are equivalent. No distinction is made between uppercase and lowercase. A numeric field consists of an integer of one or more decimal digits. In addition, there are four special characters: the slash (/), the period (.), the colon (:), and the comma (,). A blank must be used to separate two numeric fields. A blank is optional between an alphabetic and numeric field.

The five parts of the clock reading are as follows:

date
: is the day 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, 1975, then March 20 is equivalent to March 20, 1975, while March 12 is the same as March 12, 1976. There are three forms of the date, illustrated by the examples below:

```
16 March 1975 or 16 March

March 16, 1975 or March 16 1975 or March 16
(The comma is optional.)

3/16/75 or 3/16
```

time
: 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 represents hours, and mm represents minutes), followed by a period. This number (hhmm.) may be followed by an optional decimal fraction-of-a-minute field. Also acceptable are hours, minutes, and seconds fields separated by colons (the seconds field is optional). The minutes and seconds fields must each be two digits in length. 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)). Midnight and noon can be indicated by simply giving the meridional designator. The designator can be preceded by time expressed as hours, hours:minutes, or hours:minutes:seconds. The minutes and seconds fields, if present, must each be two digits in length. Examples of meridional time are:

```
midnight
5 am
5:45 am
11:07:30 pm
```

day-of-week
: is the day of the week (e.g., Monday). 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 standard nonzero status code is returned. If a date is not present, the next occurance of that day (after the current date) is used; that means that Tuesday is interpreted as the next Tuesday.

offset

is an amount of time to be added to the clock value specified by the other fields. Offsets can 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. 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 are:

1 hour 5 minutes	(an hour and five minutes from now)
Monday 6 am 2 weeks	(two weeks from the next occurrence of Monday 6:00 am)

zone

is the time zone to be used in making the conversion to Greenwich mean time. It currently can be any of the following:

GMT, gmt (Greenwich mean time)
EST, est (eastern standard time)
EDT, edt (eastern daylight time)
CST, cst (central standard time)
CDT, cdt (central daylight time)
MST, mst (mountain standard time)
MDT, mdt (mountain daylight time)
PST, pst (pacific standard time)
PDT, pdt (pacific daylight time)

or the current time zone used by the system (this is the default).

Note

If the date and day of the week portions of the string argument 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).

Examples

```
March 23
17 May 1975 EST 8:30 pm
03/28/75  2252.9 est Fri
```

Entry: convert_date_to_binary_$relative

This entry point is similar to the convert_date_to_binary_ entry point, 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 current clock time. Given a 72-bit clock time to use, this entry point converts a character representation of a date and time to the equivalent 72-bit clock reading.

Usage

```
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);
```

where:

1. string is the same as above. (Input)
2. clock is the computed clock value relative to the clock_in argument. (Output)
3. clock_in is the clock time used to compute the clock value. (Input)
4. code is the same as above. (Output)

Name: cpu_time_and_paging_

This procedure returns the virtual 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, pd_faults);
```

where:

1. pf — is the total number of page faults taken by the calling process. (Output)

2. time — is the virtual CPU time (in microseconds) used by the calling process. (Output)

3. pd_faults — is the total number of page faults from the paging device for the calling process. (Output)

Name: cu_

The cu_ (command utility) subroutine provides several short entry points that provide functions not directly available in the PL/I language. Although these entry points are designed primarily for the use of command writers, many may prove useful to Multics users and subsystem developers. (Most of the entry points to the cu_ subroutine are described in the MPM Subsystem Writers' Guide.)

Entry: cu_$arg_count

The cu_$arg_count entry point can 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);
```

where nargs is the number of arguments passed to the caller of cu_$arg_count. (Output)

Entry: cu_$arg_ptr

The cu_$arg_ptr entry point is used by a command or subroutine that can be called with a varying number of arguments, each of which is an adjustable length unaligned character string (i.e., declared char(*)). This entry point returns a pointer to a specified character-string argument and also returns the length of this argument.

Usage

```
     declare cu_$arg_ptr entry (fixed bin, ptr, fixed bin,  fixed bin (35));

     call cu_$arg_ptr (arg_no, arg_ptr, arg_len, code);
```

where:

1. arg_no is an integer specifying the number of the desired argument. (Input)

2. arg_ptr is a pointer to the unaligned character-string argument specified by arg_no. (Output)

3. arg_len — is the length (in characters) of the argument specified by arg_no. (Output)

4. code — is a standard status code. (Output) The code can be one of the following:

0	normal return
error_table_$noarg	argument specified by arg_no does not exist (if error_table_$noarg is returned, the values of arg_ptr and arg_len are undefined)

Note

The command or subroutine that uses this entry point must be called with data descriptors for its arguments. Otherwise, the returned value of arg_len is 0. If the argument specified by arg_no is not a character string, arg_len is the value of the "size" field of the descriptor (the rightmost 24 bits). This entry point must not be called from an internal procedure that has its own stack frame or from within a begin block (because cu_$arg_ptr does not check for a display pointer).

Entry: cu_$cp

Some standard Multics commands (e.g., edm, described in the MPM Commands) permit the user to escape from them to execute other commands. In this case, the escapable command passes the execute request line to the command processor. The cu_$cp entry point is called by any standard command that recognizes other Multics command lines. When a Multics command line is recognized, a call is made to cu_$cp to pass the command line to the currently defined command processor for processing.

Usage

```
declare cu_$cp entry (ptr, fixed bin, fixed bin(35));

call cu_$cp (line_ptr, line_len, code);
```

where:

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 a standard status code. (Output) It can be one of the following:

   0 normal return

   nonzero an error has been detected; however, the caller of the cu_$cp entry point is not expected to print a diagnostic at this time since it can be expected that the command processor has already done so

Name: date_time_

The date_time_ subroutine converts a system clock reading to ASCII representation. The clock reading is assumed to be in microseconds relative to January 1, 1901, 0000 Greenwich mean time. The time returned is local standard time.

Usage

```
declare date_time_ entry (fixed bin(71), char(*));

call date_time_ (time, string);
```

where:

1. time — is the clock reading. See the clock_ subroutine. (Input)

2. string — is the ASCII string equivalent of time. (Output) The string format is:

   MM/DD/YY hhmm.m zzz www

   where:

   MM/DD/YY identifies the month, day, and year using two characters per field

   hhmm.m is hours and minutes (including tenths of minutes) given in 24-hour time

   zzz is a three-letter abbreviation for the time zone

   www is a three-letter abbreviation for the day of the week

Notes

If the string declared by the caller has a length less than 24, the string is truncated on the right; if greater than 24, the string is padded on the right with blanks.

Clock readings not corresponding to dates in the twentieth century (before 01/01/1901 or after 12/31/2000) are converted as "01/01/01 0000.0".

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 storage system (file system), as input.

Usage

```
     declare date_time_$fstime entry (bit(36) aligned, char(*));

     call date_time_$fstime (time, string);
```

where:

1. time is a 36-bit internal storage system time. (Input)

2. string is the same as above. (Output)

Name: decode_clock_value_

The decode_clock_value_ subroutine takes a given system clock reading and returns the month, the day of the month, the year, the time of day, the day of the week, and the local time zone.

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);
```

where:

1. clock is the system clock value to be decoded. (Input)

2. month is the month (January = 1, ..., December = 12). (Output)

3. dom is the day of the month, e.g., 1 to 31. (Output)

4. year is the year, e.g., 1975. (Output)

5. tod is the time of day (number of microseconds since midnight). (Output)

6. dow is the day of the week (Monday = 1, ..., Sunday = 7). (Output)

7. zone is a three-character lowercase abbreviation of the current time zone used by the system. (Output)

Name: delete_

The delete_ subroutine deletes segments, directories, and multisegment files and unlinks links. If the segment, directory, or multisegment file to be deleted is protected (i.e., the safety switch is on), the subroutine requires user verification before attempting to remove the protection. There are two entry points: one called with a pathname, the other with a pointer to a segment. Both have a set of switches that specify the actions to be taken by the subroutine. If the specified entry is a segment, it is terminated using the term_ subroutine. In general, users should call the delete_ subroutine to delete segments, directories, and multisegment files, rather than directly addressing entry points in hcs_.

Entry: delete_$path

This entry point is called with the pathname of the segment, directory, multisegment file, or link to be deleted.

Usage

```
     declare delete_$path entry (char(*), char(*), bit(6), char(*), fixed
          bin(35));

     call delete_$path (dir_name, entryname, switches, caller, code);
```

where:

1. dir_name   is the pathname of the containing directory. (Input)

2. entryname   is the entryname of the segment, directory, multisegment file, or link. (Input)

3. switches   are six switches that specify the actions to be taken. The switches must be given in the order listed below. (Input)

   force_sw
   - "1"b deletes the entry even if it is protected
   - "0"b examines the next switch

   question_sw
   - "1"b asks the user if a protected entry should be deleted if the force_sw is off ("0"b); if the user gives a negative response, the subroutine returns the code error_table_$action_not_performed. If force_sw is on ("1"b) and the entryname argument is the name of a directory, delete_ prints a message for the first entry under the directory that cannot be deleted
   - "0"b deletes the entry without interrogating the user; if unable to delete the entry, the subroutine returns an appropriate storage system status code

directory_sw
- "1"b deletes directories
- "0"b examines the next switch; if the entryname argument refers to a directory and the directory_sw switch is "0"b, the subroutine returns the code error_table_$dirseg

segment_sw
- "1"b deletes segments and multisegment files
- "0"b examines the next switch; if the entryname argument refers to a segment or multisegment file and the segment_sw switch is "0"b, the subroutine returns the code error_table_$nondirseg

link_sw
- "1"b deletes (i.e., unlinks) links
- "0"b examines the next switch; if the entryname argument refers to a link and the link_sw switch is "0"b, the subroutine returns the code error_table_$not_a_branch

chase_sw
- "1"b "chases" the link and deletes the segment the link points to, if the link_sw is also "1"b
- "0"b no action

4. caller is the name of the calling procedure, to be used when questions are asked. (Input)

5. code is a storage system status code. (Output)

Entry: delete_$ptr

The delete_$ptr entry point is similar to the delete_$path entry point, except that the caller has a pointer to the actual segment to be deleted. Directories, multisegment files, and links cannot be deleted with the delete_$ptr entry point. The directory_sw, link_sw, and chase_sw switches are not examined by this entry point, but must be present.

Usage

```
declare delete_$ptr entry (ptr, bit(6), char(*), fixed bin(35));

call delete_$ptr (seg_ptr, switches, caller, code);
```

where:

1. seg_ptr is a pointer to the segment to be deleted. (Input)
2. switches are the same as above. (Input)
3. caller is the same as above. (Input)
4. code is the same as above. (Output)

Name: expand_path_

The expand_path_ subroutine expands a relative pathname into an absolute pathname. See "Constructing and Interpreting Names" in Section I of the MPM Commands.

Usage

```
declare expand_path_ entry (ptr, fixed bin, ptr, ptr, fixed bin(35));

call expand_path_ (pathp, pathl, dir_namep, entrynamep, code);
```

where:

1. pathp
   is a pointer to the pathname to be expanded. It must point to a nonvarying character string, which may be aligned or unaligned. (Input)

2. pathl
   specifies the length of the pathname. If the value is 0, the pathname is assumed to be that of the current working directory. (Input)

3. dir_namep
   is a pointer to a character string in which either the directory portion of the pathname or the entire pathname is stored. (See below.) It is assumed that dir_namep points to an aligned character string that is 168 characters long. (Input)

4. entrynamep
   is a pointer to a string in which the entryname portion of the pathname is to be stored. If entrynamep is null, then the entire pathname is stored in the string pointed to by dir_namep. It is assumed that entrynamep points to an aligned character string that is 32 characters long. (Input)

5. code
   is a standard status code. (Output) It may have the following values:

   error_table_$badpath
   bad syntax in the pathname

   error_table_$dirlong
   the directory pathname is longer than 168 characters

   error_table_$entlong
   the entryname is longer than 32 characters

   error_table_$lesserr
   too many less-than characters (<) in the pathname

   error_table_$pathlong
   the absolute pathname is longer than 168 characters

Examples

In all of the following examples, assume that the user's current working directory is >udd>Alpha>Day and that dir_name and entryname stand for the strings pointed to by dir_namep and entrynamep, respectively.

Input (pathname)	Output		
	if entrynamep is null	otherwise	
	dir_name	dir_name	entryname
work	>udd>Alpha>Day>work	>udd>Alpha>Day	work
<	>udd>Alpha	>udd	Alpha
<<	>udd	>	udd
<<Beta	>udd>Beta	>udd	Beta
<<Beta>Jones	>udd>Beta>Jones	>udd>Beta	Jones
>udd>Gamma>Smith	>udd>Gamma>Smith	>udd>Gamma	Smith

Name: get_authorization_

The get_authorization_ subroutine returns the authorization value of the process.

Usage

```
     declare get_authorization_ entry returns (bit(72) aligned);

     authorization = get_authorization_ ();
```

where authorization is the returned authorization. (Output)

This page intentionally left blank.

Name: get_group_id_

The get_group_id_ subroutine returns the 32-character access identifier of the process in which it is called. The access identifier is of the form:

```
     Person_id.Project_id.tag
```

Usage

```
     declare get_group_id_ entry returns (char(32));

     user_id = get_group_id_ ();
```

where user_id contains the access identifier that is returned to the user. It is a left-justified character string, padded with trailing blanks. (Output)

Entry: get_group_id_$tag_star

This entry point returns the access identifier of its caller with the instance component replaced by an asterisk (*).

Usage

```
     declare get_group_id_$tag_star entry returns (char(32));

     user_id = get_group_id_$tag_star ();
```

where user_id is the same as above.

This page intentionally left blank.

Name: get_line_length_

The get_line_length_ subroutine returns the line length currently in effect on a given I/O switch. If the line length is not available (for any reason), a status code is returned, and a default line length is returned.

Entry: get_line_length_$stream

This entry point returns the line length of a given I/O switch, identified by name.

Usage

```
     declare get_line_length_$stream  entry  (char(*),  fixed  bin(35))  returns
          (fixed bin(17));

     line_length = get_line_length_$stream (switch_name, code);
```

where:

1. switch_name — is the name of the switch whose line length is desired. If switch_name is null, the user_output I/O switch is assumed. (Input)

2. code — is a standard status code. (Output)

3. line_length — is the line length of switch_name. (Output)

Entry: get_line_length_$switch

This entry point returns the line length of a given I/O switch, identified by pointer.

Usage

```
     declare get_line_length_$switch  entry (ptr, fixed bin(35)) returns  (fixed
          bin(17));

     line_length = get_line_length_$switch (switch_ptr, code);
```

where:

1. switch_ptr is a pointer to the I/O control block of the switch whose line length is desired. If switch_ptr is null, the user_output I/O switch is assumed. (Input)

2. code is as above. (Output)

3. line_length is as above. (Output)

Name: get_max_authorization_

The get_max_authorization_ subroutine returns the maximum authorization of the process. (See "Access Control" in Section III of the MPM Reference Guide.)

Usage

```
     declare get_max_authorization_ entry returns (bit(72) aligned);

     max_authorization = get_max_authorization_ ();
```

where max_authorization is the returned maximum authorization. (Output)

Name: get_pdir_

The get_pdir_ subroutine returns the absolute pathname of the user's process directory. For a discussion of process directories, see "Storage System Directory Hierarchy" in Section III of the MPM Reference Guide.

Usage

```
     declare get_pdir_ entry returns (char(168));

     process_dir = get_pdir_ ();
```

where process_dir contains the absolute pathname of the user's process directory. It is assigned a left-justified character string, padded with trailing blanks. (Output)

Name: get_process_id_

The get_process_id_ subroutine returns the 36-bit identifier of the process in which it is called. The identifier is generated by the system when the process is created.

Usage

```
     declare get_process_id_ entry returns (bit(36));

     proc_id = get_process_id_ ();
```

where proc_id contains the 36-bit identifier of the process. (Output)

This page intentionally left blank.

Name: get_temp_segments_

The get_temp_segments_ subroutine puts temporary segments in the process directory for whatever purpose the caller may have. The segments returned to the caller are zero-length.

A free pool of temporary segments is associated with each user process. The pool concept makes it possible to use the same temporary segment more than once during the life of a process. Reusing temporary segments in this way avoids the cost incurred in creating a segment each time one is needed.

Usage

```
declare get_temp_segments_ entry (char (*), (*) ptr, fixed bin (35));

call get_temp_segments_ (program_name, ptrs, code);
```

where:

1. program_name is the name of the program requesting temporary segments. (Input)

2. ptrs is an array of returned pointers to the requested temporary segments. (Output)

3. code is a standard system status code. (Output)

Notes

This subroutine assigns temporary segments to its caller. It creates new temporary segments and adds them to the free pool if there currently are not enough available to satisfy the request. The temporary segments are created in the process directory with a unique name including the temp.xxxx suffix, where xxxx is the segment number of the segment in octal. See the description of the release_temp_segments_ subroutine for a description of how to return temporary segments to the free pool.

The number of segments returned to the caller is determined by the bounds of the ptrs array above.

Name: get_wdir_

The get_wdir_ subroutine returns the absolute pathname of the user's current working directory. For a discussion of working directories, see "Storage System Directory Hierarchy" in Section III of the MPM Reference Guide.

Usage

```
     declare get_wdir_ entry returns (char(168));

     working_dir = get_wdir_ ();
```

where working_dir contains the absolute pathname of the user's current working directory. (Output)

Name: hcs_$add_acl_entries

The hcs_$add_acl_entries entry point adds specified access modes to the access control list (ACL) of the specified segment. If an access name already appears on the ACL of the segment, its mode is changed to the one specified by the call.

Usage

```
     declare hcs_$add_acl_entries entry (char(*), char(*), ptr, fixed bin,
          fixed bin(35));

     call hcs_$add_acl_entries (dir_name, entryname, acl_ptr, acl_count, code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)

2. entryname is the entryname of the segment. (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 storage system status code. (Output)

Notes

The following structure is used for segment_acl:

```
     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);
```

where:

1. access_name is the access name (in the form Person_id.Project_id.tag) that identifies the processes to which this ACL entry applies.

2. modes contains the modes for this access name. The first three bits correspond to the modes read, execute, and write. The remaining bits must be 0's. For example, rw access is expressed as "101"b.

3. zero_pad must contain the value zero. (This field is for use with extended access and can only be used by the system.)

4. status_code is a storage system status code for this ACL entry only.

If code is returned as error_table_$argerr, then the erroneous ACL entries in the segment_acl structure have status_code set to an appropriate error code. No processing is performed.

If the segment is a gate (see "Intraprocess Access Control--Rings" in Section III of the MPM Reference Guide) and if the validation level is greater than ring 1, then access is given only to names that contain the same project as the user or to the SysDaemon project. If the ACL to be added is in error, no processing is performed and the subroutine returns the code error_table_$invalid_project_for_gate.

Name: hcs_$add_dir_acl_entries

The hcs_$add_dir_acl_entries entry point adds specified directory access modes to the access control list (ACL) of the specified directory. If an access name already appears on the ACL of the directory, its mode is changed to the one specified by the call.

Usage

```
declare hcs_$add_dir_acl_entries entry (char(*), char(*), ptr, fixed bin,
     fixed bin(35));

call hcs_$add_dir_acl_entries (dir_name, entryname, acl_ptr, acl_count,
     code);
```

where:

1. dir_name — is the pathname of the containing directory. (Input)

2. entryname — is the entryname of the directory. (Input)

3. acl_ptr — points to a user-filled dir_acl structure. See "Notes" below. (Input)

4. acl_count — contains the number of ACL entries in the dir_acl structure. See "Notes" below. (Input)

5. code — is a storage system status code. (Output)

Notes

The following structure is used for dir_acl:

```
dcl 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);
```

where:

1. access_name — is the access name (in the form Person_id.Project_id.tag) that 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 0's. For example, status permission is expressed as "100"b.

3. status_code — is a storage system status code for this ACL entry only.

If code is returned as error_table_$argerr, then the erroneous ACL entries in the dir_acl structure have status_code set to an appropriate error code. No processing is performed.

Name: hcs_$append_branch

The hcs_$append_branch entry point creates a segment in the specified directory, initializes the segment's access control list (ACL) by adding *.SysDaemon.* with a mode of rw and adding the initial ACL for segments found in the containing directory, and adds the user to the segment's ACL with the mode specified. ACLs and initial ACLs are described in "Access Control" in Section III of the MPM Reference Guide.

Usage

```
     declare  hcs_$append_branch  entry  (char(*),  char(*),  fixed bin(5),
          fixed bin(35));

     call hcs_$append_branch (dir_name, entryname, mode, code);
```

where:

1. dir_name   is the pathname of the containing directory. (Input)

2. entryname   is the entryname of the segment. (Input)

3. mode   is the user's access mode. See "Notes" below. (Input)

4. code   is a storage system status code. (Output)

Notes

Append permission on the containing directory is required to add a segment to that directory.

A number of attributes of the segment are set to default values as follows:

1. Ring brackets are set to the user's current validation level. See "Intraprocess Access Control (Rings)" in Section III of the MPM Subsystem Writers' Guide.

2. The User_id of the ACL entry specifying the given mode is set to the Person_id and Project_id of the user, with the instance tag set to an asterisk (*).

3. The copy switch in the branch is set to 0.

4. The bit count is set to 0.

See the description of the hcs_$append_branchx entry point to create a storage system entry with values other 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	the 8-bit is 1 (i.e., 01000b)
execute	the 4-bit is 1 (i.e., 00100b)
write	the 2-bit is 1 (i.e., 00010b)

The unused bits are reserved for unimplemented attributes and must be 0. For example, rw access is 01010b in binary form, and 10 in decimal form.

Name: hcs_$append_branchx

The hcs_$append_branchx entry point creates either a subdirectory or a segment in a specified directory. It is an extended and more general form of hcs_$append_branch. If a subdirectory is created, then the subdirectory´s access control list (ACL) is initialized by adding *.SysDaemon.* with a mode of sma and adding the initial ACL for directories that is stored in the containing directory; otherwise the segment´s ACL is initialized by adding *.SysDaemon.* with a mode of rw and adding the initial ACL for segments. The input User_id and mode are then merged to form an ACL entry that is added to the ACL of the subdirectory or segment.

Usage

```
     declare hcs_$append_branchx  entry  (char(*),  char(*),  fixed bin(5),
          (3) fixed bin(3),  char(*), fixed bin(1), fixed bin(1), fixed bin(24),
          fixed bin(35));

     call hcs_$append_branchx (dir_name, entryname, mode,  rings,  user_id,
          dir_sw, copy_sw, bit_count, code);
```

where:

1. dir_name — is the pathname of the containing directory . (Input)

2. entryname — is the entryname of the segment or subdirectory. (Input)

3. mode — is the user´s access mode. See "Notes" below. (Input)

4. rings — is a three-element array that specifies the ring brackets of the new segment or subdirectory. See "Intraprocess Access Control (Rings)" in Section III of the MPM Subsystem Writers´ Guide. (Input)

5. user_id — is an access control name of the form Person_id.Project_id.tag. (Input)

6. dir_sw — is the branch´s directory switch. (Input)
   1 if a directory is being created
   0 if a segment is being created

7. copy_sw — is the value of the copy switch to be placed in the branch. See "Segment, Directory, and Link Attributes" in Section III of the MPM Reference Guide for an explanation of the copy switch. (Input)

8. bit_count — is the segment length (in bits). (Input)

9. code — is a storage system status code. (Output)

Notes

Append permission is required on the containing directory 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 the 8-bit is 1 (i.e., 01000b)
execute the 4-bit is 1 (i.e., 00100b)
write the 2-bit is 1 (i.e., 00010b)

For directories, the modes are:

status the 8-bit is 1 (i.e., 01000b)
modify the 2-bit is 1 (i.e., 00010b)
append the 1-bit is 1 (i.e., 00001b)

If modify permission 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 0. For example, rw access is 01010b in binary form, and 10 in decimal form.

Name: hcs_$append_link

The hcs_$append_link entry point 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 "Segment, Directory, and Link Attributes" in Section III of the MPM Reference Guide.

Usage

```
     declare hcs_$append_link entry (char(*), char(*), char(*), fixed  bin(35));

     call hcs_$append_link (dir_name, entryname, path, code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)

2. entryname is the entryname of the link. (Input)

3. path is the pathname of the directory entry to which the entryname argument points. The pathname may be a maximum of 168 characters. (Input)

4. code is a storage system status code. (Output)

Notes

Append permission is required in 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 hcs_$append_branch and hcs_$append_branchx entry points can be used to create a segment or directory entry in the storage system hierarchy.

Name: hcs_$chname_file

The hcs_$chname_file entry point changes the entryname on a specified storage system entry. If an already existing name (an old 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_file entry (char(*), char(*), char(*), char(*),
     fixed bin(35));

call hcs_$chname_file (dir_name, entryname, oldname, newname, code);
```

where:

1. dir_name — is the pathname of the containing directory. (Input)

2. entryname — is the entryname of the segment, directory, multisegment file, or link. (Input)

3. oldname — is the name to be deleted from the entry. It can be a null character string ("") in which case no name is 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 can 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 storage system status code. (Output) It can have the values:

error_table_$nonamerr	attempting to delete the only name of a directory entry
error_table_$namedup	attempting to add a name that exists on another entry
error_table_$segnamedup	attempting to add a name that already exists on this entry

Notes

The hcs_$chname_seg entry point performs a similar function using a pointer to a segment instead of its pathname.

The user must have modify permission on the directory containing the entry whose name is to be changed.

Examples

Assume that the entry >my_dir>alpha exists and that it also has the entryname beta. Then the following sequence of calls to hcs_$chname_file would have the effects described.

```
     call hcs_$chname_file (">my_dir", "alpha", "beta", "gamma", code);
```

The above call changes the entryname beta to gamma. The entry now has the names alpha and gamma.

```
     call hcs_$chname_file (">my_dir", "gamma", "gamma", "",  code);
```

The above call removes the entryname gamma. Either alpha or gamma could be used in the second argument position. The entry now has only the name alpha.

```
     call hcs_$chname_file (">my_dir", "alpha", "", "delta",  code);
```

The above call adds the entryname delta. The entry now has the names alpha and delta.

Name: hcs_$chname_seg

The hcs_$chname_seg entry point changes an entryname on a segment, if a pointer to the segment is given. If an already existing name (an old 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);
```

where:

1. seg_ptr is a pointer to the segment whose name is to be changed. (Input)

2. oldname is the name to be deleted from the entry. It can 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. It can 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 storage system status code. (Output) It can have the values:

error_table_$nonamerr	attempting to delete the only name of a directory entry
error_table_$namedup	attempting to add a name that exists on another entry
error_table_$segnamedup	attempting to add a name that already exists on this entry

Notes

The hcs_$chname_file entry point performs the same function if the pathname of the segment is given instead of a pointer.

The user must have modify permission on the directory containing the segment whose name is to be changed.

Examples

Assume that the user has a pointer, seg_ptr, to a segment that has two entrynames, alpha and beta. Then the following sequence of calls to hcs_$chname_seg would have the effects described.

```
     call hcs_$chname_seg (seg_ptr, "beta", "gamma", code);
```

The above call changes the entryname beta to gamma.

```
     call hcs_$chname_seg (seg_ptr, "gamma", "", code);
```

The above call removes the entryname gamma.

```
     call hcs_$chname_seg (seg_ptr, "", "delta", code);
```

The above call adds the entryname delta. The entry now has the names alpha and delta.

This page intentionally left blank.

Name: hcs_$create_branch_

The hcs_$create_branch_ subroutine creates either a subdirectory or a segment in the specified directory. (This entry point is an extended and more general form of the hcs_$append_branchx subroutine.) 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 initiated by copying the initial ACL for segments. The access_name and mode items from the create_branch_info structure (see "Notes" below) are then added to the ACL of the created subdirectory or segment.

Usage

```
declare hcs_$create_branch_ entry (char(*), char(*), ptr,  fixed  bin(35));

call hcs_$create_branch_ (dir_name, entryname, info_ptr, code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)

2. entryname is the entryname of the segment or subdirectory to be created. (Input)

3. info_ptr is a pointer to the information structure described below. (Input)

4. code is a storage system status code. (Output)

Notes

The user must have append permission on the containing directory to add an entry to that directory.

The pointer info_ptr points to a structure of the following form:

```
dcl 1 create_branch_info  aligned,
    2 version             fixed bin,
    2 switches            unaligned,
      3 dir_sw            bit(1) unaligned,
      3 copy_sw           bit(1) unaligned,
      3 chase_sw          bit(1) unaligned,
      3 priv_upgrade_sw   bit(1) unaligned,
      3 mbz1              bit(32) unaligned,
        2 mode                bit(3) unaligned,
        2 mbz2                bit(33) unaligned,
        2 rings               (3) fixed bin(3),
        2 access_name         char(32),
        2 bitcnt              fixed bin(24),
        2 quota               fixed bin(18),
        2 access_class        bit(72);
```

where:

1. version — is a number representing the version of the create_branch_info structure being used. The structure described above is version 1.

2. dir_sw — controls whether a directory or nondirectory segment is to be created.
"1"b   create a directory segment
"0"b   create a nondirectory segment

3. copy_sw — is the created segment's copy switch.
"1"b   make a copy whenever the segment is initiated
"0"b   do not make a copy--use original

4. chase_sw — allows creation through links.
"1"b   chase entryname if it is a link and create the desired segment in the final directory
"0"b   do not chase links

5. priv_upgrade_sw — allows creation of upgraded ring 1 nondirectory segments (i.e., with an access class higher than the containing directory's access class). The use of this switch is limited to ring 1 programs, and it should normally be "0"b.

6. mbz1 — must be (32)"0"b.

7. mode — is the ACL mode desired for access_name. The meanings of the bits are as follows. For directory segments:
"100"b   status
"010"b   modify
"001"b   append

For nondirectory segments:
"100"b   read
"010"b   execute
"001"b   write

8. mbz2 — must be (33)"0"b.

9. rings — are the desired ring brackets; see "Intraprocess Access Control--Rings" in Section III of the MPM Reference Guide.

10. access_name — is the access control name of the form Person_id.Project_id.tag to be added to the ACL.

11. bitcnt — is the segment's length (in bits).

12. quota — is the desired quota to be moved to the directory created. (It must be 0 for nondirectory segments.) If access_class is not equal to the access class of dir_name, quota must be greater than 0.

13. access_class — is the desired access class of the directory. For nondirectory segments, access_class must be equal to the access class of dir_name unless the priv_upgrade_sw switch is set. (See the hcs_$get_access_class subroutine.)

Name: hcs_$delentry_file

The hcs_$delentry_file entry point, given a directory name and an entryname, deletes the given entry from its containing directory. This entry may be a segment, a directory, or a link. If the entry is a segment, the contents of the segment are truncated first. If the entry specifies a directory that contains entries, the code error_table_$fulldir is returned and hcs_$del_dir_tree must be called to remove the contents of the directory. See the description of hcs_$del_dir_tree in the MPM Subsystem Writers' Guide. Generally, programmers should use the delete_ subroutine rather than this entry point in order to ensure that their address space is properly cleaned up.

Usage

```
declare hcs_$delentry_file (char(*), char(*), fixed bin(35));

call hcs_$delentry_file (dir_name, entryname, code);
```

where:

1. dir_name   is the pathname of the containing directory. (Input)

2. entryname   is the entryname of the segment, directory, or link. (Input)

3. code   is a storage system status code. (Output)

Notes

The hcs_$delentry_seg entry point performs the same function on a segment, given a pointer to the segment instead of the pathname.

The user must have modify permission on the containing directory. If entryname specifies a segment or directory (but not a link), the safety switch of the entry must be off.

Name: hcs_$delentry_seg

The hcs_$delentry_seg entry point, given a pointer to a segment, deletes the corresponding entry from its containing directory. The contents of the segment are truncated first. Generally, programmers should use the delete_ subroutine rather than this entry point in order to ensure that their address space is properly cleaned up.

Usage

```
     declare hcs_$delentry_seg (ptr, fixed bin(35));

     call hcs_$delentry_seg (seg_ptr, code);
```

where:

1. seg_ptr is the pointer to the segment to be deleted. (Input)

2. code is a storage system status code. (Output)

Notes

The hcs_$delentry_file entry point performs the same function, given the pathname of the segment instead of the pointer.

The user must have modify permission on the containing directory. The safety switch of the segment must be off.

Name: hcs_$delete_acl_entries

The hcs_$delete_acl_entries entry point 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 (dir_name, entryname, acl_ptr, acl_count,
     code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the segment. (Input)
3. acl_ptr points to a user-filled delete_acl structure. See "Notes" below. (Input)
4. acl_count is the number of ACL entries in the delete_acl structure. See "Notes" below. (Input)
5. code is a storage system status code. (Output)

Notes

The following is the delete_acl structure:

```
dcl 1 delete_acl (acl_count)       aligned based (acl_ptr),
    2 access_name                  char(32),
    2 status_code                  fixed bin(35);
```

where:

1. access_name is the access name (in the form of Person_id.Project_id.tag) that identifies the ACL entry to be deleted.
2. status_code is a storage system status code for this ACL entry only.

If code is returned as error_table_$argerr, then the erroneous ACL entries in the delete_acl structure have status_code set to an appropriate error code. No processing is performed.

If an access name cannot be matched to a name already on the segment's ACL, then the status_code for that ACL entry in the delete_acl structure is set to error_table_$user_not_found. Processing continues to the end of the delete_acl structure and code is returned as 0.

Name: hcs_$delete_dir_acl_entries

The hcs_$delete_dir_acl_entries entry point is used to delete specified entries from an access control list (ACL) for a directory. The delete_acl structure used by this subroutine is discussed in the description of the hcs_$delete_acl_entries entry point.

Usage

```
     declare   hcs_$delete_dir_acl_entries   entry   (char(*),   char(*),   ptr,
          fixed bin, fixed bin(35));

     call hcs_$delete_dir_acl_entries (dir_name, entryname, acl_ptr, acl_count,
          code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)

2. entryname is the entryname of the directory. (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 storage system status code (see "Note" below). (Output)

Note

The storage system status code is interpreted as described in the hcs_$delete_acl_entries entry point.

Name: hcs_$fs_get_mode

The hcs_$fs_get_mode entry point returns the access mode of the user on a specified segment at the current validation level. For a discussion of access modes, see "Access Control" in Section III of the MPM Reference Guide.

Usage

```
declare hcs_$fs_get_mode entry (ptr, fixed bin(5), fixed bin(35));

call hcs_$fs_get_mode (seg_ptr, mode, code);
```

where:

1. seg_ptr is a pointer to the segment whose access mode is to be returned. (Input)
2. mode is the access mode returned (see "Notes" below). (Output)
3. code is a 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 "Intraprocess Access Control (Rings)" in Section III of the MPM Subsystem Writers' Guide.

The mode argument is a fixed binary number where the desired mode is encoded with one access mode specified by each bit. The modes are:

read	the 8-bit is 1 (i.e., 01000b)
execute	the 4-bit is 1 (i.e., 00100b)
write	the 2-bit is 1 (i.e., 00010b)

The unused bits are reserved for unimplemented attributes and must be 0. For example, rw access is 01010b in binary form, and 10 in decimal form.

Name: hcs_$fs_get_path_name

The hcs_$fs_get_path_name entry point, given a pointer to a segment, returns a pathname for the segment, with the directory and entryname portions of the pathname separated. The entryname returned is the primary name on the entry. See "Segment, Directory, and Link Attributes" in Section III of the MPM Reference Guide 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 (seg_ptr, dir_name, ldn, entryname, code);
```

where:

1. seg_ptr is a pointer to the segment. (Input)

2. dir_name is the pathname of the containing directory. If the length of the pathname to be returned is greater than the length of dir_name, the pathname is truncated. To avoid this problem, the length of dir_name should be 168 characters. (Output)

3. ldn is the number of nonblank characters in dir_name. (Output)

4. entryname is the primary entryname of the segment. If the length of the entryname to be returned is greater than the length of entryname, the entryname is truncated. To avoid this problem, the length of entryname should be 32 characters. (Output)

5. code is a storage system status code. (Output)

Name: hcs_$fs_get_ref_name

The hcs_$fs_get_ref_name entry point returns a specified (i.e., first, second, etc.) reference name for a specified segment. See "Constructing and Interpreting Names" in Section I of the MPM Commands.

Usage

```
     declare hcs_$fs_get_ref_name entry (ptr, fixed bin, char(*), fixed
          bin(35));

     call hcs_$fs_get_ref_name (seg_ptr, count, ref_name, code);
```

where:

1. seg_ptr is a pointer to the segment whose reference name is sought. (Input)

2. count specifies which reference name is to be returned, where 1 is the name by which the segment has most recently been made known, 2 is the next most recent name, etc. (Input)

3. ref_name is the desired reference name. (Output)

4. code is a storage system status code. (Output)

Note

If the count argument 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_name_count_too_big.

Name: hcs_$fs_get_seg_ptr

The hcs_$fs_get_seg_ptr entry point, given a reference name of a segment, returns a pointer to the base of the segment. For a discussion of reference names, see "Constructing and Interpreting Names" in Section I of the MPM Commands.

Usage

```
     declare hcs_$fs_get_seg_ptr entry (char(*), ptr,  fixed bin(35));

     call hcs_$fs_get_seg_ptr (ref_name, seg_ptr, code);
```

where:

1. ref_name is the reference name of a segment for which a pointer is to be returned. (Input)

2. seg_ptr is a pointer to the base of the segment. (Output)

3. code is a storage system status code. (Output)

Note

If the reference name is accessible from the user´s current validation level, seg_ptr is returned pointing to the segment; otherwise, it is null. For more information on rings and validation levels refer to "Intraprocess Access Control (Rings)" in Section III of the MPM Subsystem Writers´ Guide.

Name: hcs_$fs_move_file

The hcs_$fs_move_file entry point moves the data associated with one segment in the storage system hierarchy to another segment given the pathnames of the segments in question. The old segment remains, but 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);
```

where:

1. from_dir
   is the pathname of the directory in which from_entry resides. (Input)

2. from_entry
   is the entryname of the segment from which data is to be moved. (Input)

3. at_sw
   is a 2-bit append/truncate switch. (Input)

   append (first bit)
   - 0 if to_entry does not exist, the code error_table_$noentry is returned
   - 1 if to_entry does not exist, it is created

   truncate (second bit)
   - 0 if to_entry is not a zero length segment, the code error_table_$clnzero is returned
   - 1 if to_entry is not a zero length segment, it is truncated before moving

4. to_dir
   is the pathname of the directory in which to_entry resides. (Input)

5. to_entry
   is the entryname of the segment to which data is to be moved. (Input)

6. code
   is a storage system status code. It can 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 hcs_$fs_move_seg entry point performs the same function given pointers to the segments in question instead of pathnames.

The code error_table_$no_move is returned if:

1. The user does not have rw access to to_entry.
2. The user does not have read access to from_entry.
3. The max_length of to_entry is less than the length of from_entry.
4. There is not enough quota in to_dir to perform the move.

Name: hcs_$get_access_class_seg

The hcs_$get_access_class_seg subroutine, given a pointer, returns the access class of that pointer's corresponding segment.

Usage

```
     declare hcs_$get_access_class_seg entry (ptr, bit(72) aligned, fixed
          bin(35));

     call hcs_$get_access_class_seg (seg_ptr, access_class, code);
```

where:

1. seg_ptr is the pointer to the segment. (Input)
2. access_class is the access class of the segment. (Output)
3. code is a storage system status code. (Output)

Name: hcs_$initiate

The hcs_$initiate entry point, given a pathname and a reference name, causes the segment defined by the pathname to be made known and the given reference name initiated. If the reserved segment switch is on, then the segment pointer is input and the segment is made known with that segment number. In this case, the user supplies the initial segment number. If the reserved segment switch is off, a segment number is assigned and returned as a pointer.

Usage

```
declare hcs_$initiate entry (char(*), char(*), char(*), fixed bin(1),
     fixed bin(2), ptr, fixed bin(35));

call hcs_$initiate (dir_name, entryname, ref_name, seg_sw, copy_ctl_sw,
     seg_ptr, code);
```

where:

1. dir_name — is the pathname of the containing directory. (Input)

2. entryname — is the entryname of the segment. (Input)

3. ref_name — is the reference name. If it is zero length, the segment is initiated with a null reference name. (Input)

4. seg_sw — is the reserved segment switch. (Input)
   0 if no segment number has been reserved
   1 if a segment number was reserved

5. copy_ctl_sw — specifies whether or not a copy of the segment is generated. (Input)
   0 create a copy of the specified segment (in the process directory) if the segment has its copy switch on (1)
   1 do not create a copy even if the segment has its copy switch on
   2 create a copy even if the segment has its copy switch off

6. seg_ptr — is a pointer to the segment. (Input or Output)
   Input if seg_sw is on (1)
   Output if seg_sw is off (0)

7. code — is a storage system status code. (Output)

system-defined ceiling. If entryname is not already known, and no problems are encountered, seg_ptr contains a valid pointer and code is 0. If ref_name has already been initiated in the current ring, the code is returned as error_table_$namedup and the seg_ptr argument contains a valid pointer to the segment already initiated. If the seg_ptr argument contains a nonnull pointer, the bit_count argument is set to the bit count of the segment to which seg_ptr points.

Name: hcs_$list_acl

The hcs_$list_acl entry point is used either to list the entire access control list (ACL) of a segment or to return the access modes of specified ACL entries. The segment_acl structure used by this entry point is discussed in the description of hcs_$add_acl_entries.

Usage

```
declare hcs_$list_acl entry (char(*), char(*), ptr, ptr, ptr, fixed bin,
     fixed bin(35));

call hcs_$list_acl (dir_name, entryname, area_ptr, area_ret_ptr, acl_ptr,
     acl_count, code);
```

where:

1. dir_name — is the pathname of the containing directory. (Input)

2. entryname — is the entryname of the segment. (Input)

3. area_ptr — points to an area in which the list of ACL entries, which make up the entire ACL of the segment, is allocated. If area_ptr is null, then the user wants access modes for certain ACL entries; these will be specified by the structure pointed to by acl_ptr (see below). (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 points to an ACL structure, segment_acl, into which mode information is placed for the access names specified in that same structure. (Input)

6. acl_count — is the number of entries in the ACL structure. (Input or Output)
   Input — is the number of entries in the ACL structure identified by acl_ptr
   Output — is the number of entries in the segment_acl structure allocated in the area pointed to by area_ptr, if area_ptr is not null

7. code — is a storage system status code. (Output) *

Note

If acl_ptr is used to obtain modes for specified access names (rather than for all access names on a segment), then each ACL entry in the segment_acl structure either has status_code set to 0 and contains the segment's mode or has status_code set to error_table_$user_not_found and contains a mode of 0.

This page intentionally left blank.

Name: hcs_$list_dir_acl

The hcs_$list_dir_acl entry point is used either to list the entire access control list (ACL) of a directory or to return the access modes for specified entries. The dir_acl structure described in hcs_$add_dir_acl_entries is used by this entry point.

Usage

```
declare hcs_$list_dir_acl entry (char(*), char(*), ptr, ptr, ptr,
     fixed bin, fixed bin(35));

call hcs_$list_dir_acl (dir_name, entryname, area_ptr, area_ret_ptr,
     acl_ptr, acl_count, code);
```

where:

1. dir_name — is the pathname of the containing directory. (Input)

2. entryname — is the entryname of the directory. (Input)

3. area_ptr — points to an area in which the list of ACL entries, which make up the entire ACL of the directory, is allocated. If area_ptr is null, then the user wants access modes for certain ACL entries; these will be specified by the structure pointed to by acl_ptr (see below). (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 points to an ACL structure, dir_acl, into which mode information is placed for the access names specified in that same structure. (Input)

6. acl_count — is the number of entries in the ACL structure. (Input or Output)
   - Input — is the number of entries in the ACL structure identified by acl_ptr
   - Output — is the number of entries in the dir_acl structure allocated in the area pointed to by area_ptr, if area_ptr is not null

7. code — is a storage system status code. (Output)

Note

If acl_ptr is used to obtain modes for specified access names (rather than for all access names on a directory), then each ACL entry in the dir_acl structure either has status_code set to 0 and contains the directory's mode or has status_code set to error_table_$user_not_found and contains a mode of 0.

This page intentionally left blank.

Name: hcs_$make_ptr

The hcs_$make_ptr entry point, when given a reference name and an entry point name, returns a pointer to a specified entry point. If the reference name has not yet been initiated, the search rules are used to find a segment with a name the same as the reference name. The segment is made known and the reference name initiated.

Usage

```
     declare hcs_$make_ptr entry (ptr, char(*), char(*), ptr, fixed bin(35));

     call hcs_$make_ptr (ref_ptr, entryname, entry_point_name,  entry_point_ptr,
          code);
```

where:

1. ref_ptr is a pointer to the segment that is considered the referencing procedure. See "Notes" below. (Input)

2. entryname is the entryname of the segment. (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 entryname and entry_point_name. (Output)

5. code is a storage system status code. (Output)

Notes

The directory in which the segment pointed to by ref_ptr is located is used as the referencing directory for the standard search rules. If ref_ptr is null, then the standard search rule specifying the referencing directory is skipped. See "System Libraries and Search Rules" in Section III of the MPM Reference Guide. Normally ref_ptr is null.

The entryname and entry_point_name arguments are nonvarying character strings with a length of up to 32 characters. They need not be aligned and can 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 any case, the segment identified by entryname is made known to the process with the entryname argument initiated as a reference name. If an error is encountered upon return, the entry_point_ptr argument is null and an 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 the description of the cu_ subroutine in the MPM Subsystem Writers´ Guide.)

Name: hcs_$make_seg

The hcs_$make_seg entry point creates a segment with a specified entryname in a specified directory. Once the segment is created, it is made known to the process and a pointer to the segment is returned to the caller. If the segment already exists or is already known, a nonzero 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(35));

     call hcs_$make_seg (dir_name, entryname, ref_name, mode, seg_ptr, code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)

2. entryname is the entryname of the segment. (Input)

3. ref_name is the desired reference name or a null character string (""). (Input)

4. mode specifies the mode for this user. See "Notes" in the description of hcs_$append_branch for more information on modes. (Input)

5. seg_ptr is a pointer to the created segment. (Output)

6. code is a storage system status code. (Output) It may be one of the following:

   error_table_$namedup if the specified segment already exists or the specified reference name has already been initiated

   error_table_$segknown if the specified segment is already known

Notes

If dir_name is null, the process directory is used. If the entryname is null, a unique name is generated. The segment is made known and the reference name, ref_name, is initiated.

See also "Constructing and Interpreting Names" in Section I of the MPM Commands.

Name: hcs_$replace_acl

The hcs_$replace_acl entry point replaces an entire access control list (ACL) for a segment with a user-provided ACL, and can optionally add an entry for *.SysDaemon.* with mode rw to the new ACL. The segment_acl structure described in hcs_$add_acl_entries is used by this entry point.

Usage

```
     declare hcs_$replace_acl entry (char(*), char(*), ptr, fixed bin, bit(1),
          fixed bin(35));

     call hcs_$replace_acl   (dir_name,   entryname,   acl_ptr,   acl_count,
          no_sysdaemon_sw, code);
```

where:

1. dir_name — is the pathname of the containing directory. (Input)

2. entryname — is the entryname of the segment. (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 — is a switch that indicates whether an rw *.SysDaemon.* entry is to be put on the ACL of the segment after the existing ACL has been deleted and before the user-supplied segment_acl entries are added. (Input)
   "0"b adds rw *.SysDaemon.* entry
   "1"b replaces the existing ACL with only the user-supplied segment_acl

6. code — is a storage system status code. (Output)

Notes

If acl_count is zero, then the existing ACL is deleted and only the action indicated (if any) by the no_sysdaemon_sw switch is performed. If 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 in the segment_acl structure is identical.

If the segment is a gate (see "Intraprocess Access Control--Rings" in Section III of the MPM Reference Guide) and if the validation level is greater than ring 1, access is restricted to the same project as that of the user or to the SysDaemon project. If the replacement ACL is in error, then no processing is performed and the subroutine returns the code error_table_$invalid_project_for_gate.

Name: hcs_$replace_dir_acl

The hcs_$replace_dir_acl entry point replaces an entire access control list (ACL) for a directory with a user-provided ACL, and can optionally add an entry for *.SysDaemon.* with mode sma to the new ACL. The dir_acl structure described in hcs_$add_dir_acl_entries is used by this entry point.

Usage

```
     declare hcs_$replace_dir_acl entry (char(*), char(*), ptr, fixed bin,
          bit(1), fixed bin(35));

     call hcs_$replace_dir_acl (dir_name, entryname, acl_ptr, acl_count,
          no_sysdaemon_sw, code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)

2. entryname is the entryname of the directory. (Input)

3. acl_ptr points to a user-supplied dir_acl structure that is to replace the current ACL. (Input)

4. acl_count contains the number of entries in the dir_acl structure. (Input)

5. no_sysdaemon_sw is a switch that indicates whether the sma *.SysDaemon.* entry is put on the ACL of the directory after the existing ACL of the directory has been deleted and before the user-supplied dir_acl entries are added. (Input)
   "0"b adds sma *.SysDaemon.* entry
   "1"b replaces the existing ACL with only the user-supplied dir_acl

6. code is a storage system status code. (Output)

Notes

If acl_count is zero, then the existing ACL is deleted and only the action indicated (if any) by the no_sysdaemon_sw switch is performed. If 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 in the dir_acl structure is identical.

If the replacement ACL is in error, no processing is performed for that ACL entry in the dir_acl structure and the subroutine returns the code error_table_$nam_err or error_table_$invalid_ascii, whichever is appropriate.

Name: hcs_$set_bc

The hcs_$set_bc entry point sets the bit count of a specified segment. It also sets the bit count author of that segment 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 (dir_name, entryname, bit_count, code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)
2. entryname is the entryname of the segment. (Input)
3. bit_count is the new bit count of the segment. (Input)
4. code is a storage system status code. (Output)

Notes

The user must have write access on the segment, but does not need modify permission on the containing directory.

The hcs_$set_bc_seg entry point performs the same function, when a pointer to the segment is provided instead of the pathname.

Name: hcs_$set_bc_seg

The hcs_$set_bc_seg entry point, given a pointer to the segment, sets the bit count of a segment in the storage system. 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 (seg_ptr, bit_count, code);
```

where:

1. seg_ptr is a pointer to the segment whose bit count is to be changed. (Input)

2. bit_count is the new bit count of the segment. (Input)

3. code is a storage system status code. (Output)

Notes

The user must have write access on the segment, but does not need modify permission with respect to the containing directory.

The hcs_$set_bc entry point performs the same function, when provided with a pathname of a segment rather than a pointer.

Name: hcs_$status_

The hcs_$status_ entry point returns various items of information about a specified directory entry.

The main entry point (hcs_$status_) returns the most often needed information about a specified entry.

Usage

```
     declare hcs_$status_ entry (char(*), char(*), fixed bin(1), ptr, ptr,
          fixed bin(35));

     call hcs_$status_ (dir_name, entryname, chase, entry_ptr, area_ptr, code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)

2. entryname is the entryname of the segment, directory, or link. (Input)

3. chase indicates whether the information returned is about a link or about the entry to which the link points. (Input)
   0 returns link information
   1 returns information about the entry to which the link points

4. entry_ptr is a pointer to the structure in which information is returned. See "Entry Information" below. (Input)

5. area_ptr is a pointer to the area in which names are returned. If the pointer is null, no names are returned. See "Notes" below. (Input)

6. code is a storage system status code. See "Access Requirements" below. (Output)

Entry Information

The argument entry_ptr points to the following structure if the entry is a segment or directory:

```
     dcl 1 branch based (entry_ptr) aligned,
          (2 type                   bit(2),
           2 nnames                 fixed bin(15),
           2 nrp                    bit(18),
           2 dtm                    bit(36),
           2 dtu                    bit(36),
           2 mode                   bit(5),
           2 pad                    bit(13),
           2 records                fixed bin(17)) unaligned;
```

where:

1. type — specifies the type of entry:
   "00"b link
   "01"b segment
   "10"b directory

2. nnames — specifies the number of names for this entry.

3. nrp — is a 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 or directory was last modified.

5. dtu — contains the date and time the segment or directory was last used.

6. mode — contains the effective mode of the segment with respect to the current user's validation level. See the hcs_$append_branchx entry point for a description of modes. For directory entries, the 4-bit is 1 (i.e., 00100b).

7. pad — is unused space in this structure.

8. records — contains the number of 1024-word records of secondary storage assigned to the segment or directory.

The argument entry_ptr points to the following structure if the entry is a link:

```
dcl 1 link based (entry_ptr)    aligned,
   (2 type                      bit(2),
    2 nnames                    fixed bin(15),
    2 nrp                       bit(18),
    2 dtem                      bit(36),
    2 dtd                       bit(36),
    2 pnl                       fixed bin(17),
    2 pnrp                      bit(18)) unaligned;
```

where:

1. type is as above.

2. nnames is as above.

3. nrp is 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 specifies the length in characters of the link pathname.

7. pnrp is a pointer (relative to the base of the segment containing the user-specified free storage area) to the link pathname.

Notes

The user must provide the storage space required by the above structures. The hcs_$status_ entry point merely fills them in.

If the area_ptr argument is not null, entrynames are returned in the following structure allocated in the user-specified area:

```
declare names (nnames) char(32) aligned based (np);
```

where np is equal to ptr (area_ptr, entry_ptr->entry.nrp).

The first name in this array is defined as the primary name of the entry.

Link pathnames are returned in the following structure allocated in the user-specified area:

```
declare pathname char(pnl) aligned based (lp);
```

where lp is equal to ptr (area_ptr, entry_ptr->link.nrp).

The user must provide an area that is large enough to accommodate a reasonable number of names.

Access Requirements

The user must have status permission on the containing directory in order to obtain complete information.

If the user lacks status permission but has nonnull access to a segment, the following per-segment attributes can be returned: type, effective mode, bit count, records, and current length. In this instance, if either the hcs_$status_ or hcs_$status_long entry point is called, the code error_table_$no_s_permission is returned to indicate that incomplete information has been returned.

Entry: hcs_$status_long

This entry point returns most user-accessible information about a specified entry. The access required to use this entry point 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 (dir_name, entryname, chase, entry_ptr, area_ptr,
     code);
```

where the arguments are the same as in the hcs_$status_ entry point.

Notes

The entry_ptr argument points to the same structure as described under the hcs_$status_ entry point if the entry is a link. It points to the following structure if the entry is a segment or directory:

```
dcl 1 branch based (entry_ptr) aligned,
     (2 type                   bit(2),
      2 nnames                 fixed bin(15),
      2 nrp                    bit(18),
      2 dtm                    bit(36),
      2 dtu                    bit(36),
      2 mode                   bit(5),
      2 raw_mode               bit(5),
      2 pad1                   bit(8),
      2 records                fixed bin(17),
      2 dtd                    bit(36),
      2 dtem                   bit(36),
      2 pad2                   bit(36),
      2 cur_len                fixed bin(11),
      2 bit_count              bit(24),
      2 did                    bit(4),
      2 pad3                   bit(4),
      2 copy_sw                bit(1),
      2 tpd_sw                 bit(1),
      2 pad4                   bit(9),
      2 rbs (0:2)              fixed bin(5),
      2 uid                    bit(36)) unaligned;
```

where:

1. type is as above.

2. nnames is as above.

3. nrp is as above.

4. dtm is as above.

5. dtu is as above.

6. mode is as above.

7. raw_mode is the mode of the segment with respect to the current user without regard to ring brackets, etc. See the hcs_$append_branchx entry point for a description of modes. For directory entries, the 4-bit is 1 (i.e., 00100b).

8. pad1 is unused space in this structure.

9. records is as above.

10. dtd is the data and time the segment was last dumped.

11. dtem is the date and time the entry was last modified.

12. pad2 is unused space in this structure.

13. cur_len is the current length of the segment in units of 1024-word records.

14. bit_count is the bit count associated with the segment.

15. did specifies the secondary storage device (if any) on which the segment currently resides.

16. pad3 is unused space in this structure.

17. copy_sw contains the setting of the segment copy switch.

18. tpd_sw contains the setting of the segment transparent_paging_device switch. If set, no pages of the segment go on the paging device.

19. pad4 is unused space in this structure.

20. rbs contains the ring brackets of the segment right justified in the 6-bit field.

21. uid is the segment unique identifier.

Entry: hcs_$status_minf

The hcs_$status_minf entry point returns the bit count and entry type given a directory and entryname. Status permission on the directory or nonnull access on the entry is required to use this entry point.

Usage

```
     declare hcs_$status_minf   entry   (char(*),   char(*),   fixed   bin(1),
          fixed bin(2), fixed bin(24), fixed bin(35));

     call hcs_$status_minf (dir_name, entryname, chase, type, bit_count, code);
```

where:

1. dir_name is the same as for the hcs_$status_ entry point above. (Input)

2. entryname is the same as for the hcs_$status_ entry point above. (Input)

3. chase is the same as for the hcs_$status_ entry point above. (Input)

4. type specifies the type of entry. (Output) It can be:
   0 link
   1 segment
   2 directory

5. bit_count is the bit count. (Output)

6. code is a storage system status code. (Output)

Entry: hcs_$status_mins

This entry point returns the bit count and entry type given a pointer to the segment. Status permission on the directory or nonnull access to the segment is required to use this entry point.

Usage

```
     declare hcs_$status_mins entry (ptr, fixed bin(2), fixed bin(24),
          fixed bin(35));

     call hcs_$status_mins (seg_ptr, type, bit_count, code);
```

where:

1. seg_ptr   points to the segment about which information is desired. (Input)
2. type   is as above. (Output)
3. bit_count is as above. (Output)
4. code   is as above. (Output)

Name: hcs_$terminate_file

The hcs_$terminate_file entry point, given the pathname of a segment, terminates all the reference names of that segment and then removes the segment from the address space of the process (makes the segment unknown). For a discussion of reference names, see "Constructing and Interpreting Names" in Section I of the MPM Commands.

Usage

```
     declare hcs_$terminate_file   entry   (char(*),   char(*),   fixed bin(1),
          fixed bin(35));

     call hcs_$terminate_file (dir_name, entryname, seg_sw, code);
```

where:

1. dir_name   is the pathname of the containing directory. (Input)

2. entryname   is the entryname of the segment. (Input)

3. seg_sw   is the reserved segment switch. (Input)
   1   saves segment number in the reserved segment list
   0   does not save segment number

4. code   is a storage system status code. (Output)

Notes

The hcs_$terminate_seg entry point performs the same operation given a pointer to a segment instead of a pathname; the hcs_$terminate_name and hcs_$terminate_noname entry points terminate a single reference name.

The term_ subroutine performs the same operation as the hcs_$terminate_file entry point, but, in addition, causes links to the entry's linkage section to be unsnapped. Use of the term_ subroutine is recommended.

The reference names that are removed are those for which the ring level associated with the name is greater than or equal to the validation level of the process. If any reference names exist that are associated with a ring level less than the validation level of the process, the segment is not made unknown and the code is returned as error_table_$bad_ring_brackets. For a discussion of rings, refer to "Intraprocess Access Control (Rings)" in Section III of the MPM Subsystem Writers' Guide.

Name: hcs_$terminate_name

The hcs_$terminate_name entry point terminates one reference name from a segment and decrements a count of initiated reference names for the segment. If the count of initiated reference names for the given segment is at a system-defined ceiling, the entry point returns the status code error_table_$usage_count_too_large and does not decrement the count of initiated reference names for the given segment. If the hcs_$terminate_name entry point reduces the count of initiated reference names for that segment to zero, the segment is removed from the address space of the process (made unknown). For a discussion of reference names, see "Constructing and Interpreting Names" in Section I of the MPM Commands.

Usage

```
     declare hcs_$terminate_name entry (char(*), fixed bin(35));

     call hcs_$terminate_name (ref_name, code);
```

where:

1. ref_name is the reference name to be terminated. (Input)

2. code is a storage system status code. (Output)

Notes

The hcs_$terminate_noname entry point terminates a null reference name from a specified segment; the hcs_$terminate_file and hcs_$terminate_seg entry points terminate all reference names of a segment and make the segment unknown, given its pathname or segment number, respectively.

The term_$single_refname entry point (see the description of the term_ subroutine) performs the same operation as the hcs_$terminate_name entry point, unsnapping links as well. Use of the term_ subroutine is recommended.

Name: hcs_$terminate_noname

The hcs_$terminate_noname entry point terminates a null reference name from the specified segment and decrements a count of initiated reference names for the segment. If the count of initiated reference names for the given segment is at a system-defined ceiling, the entry point returns the status code error_table_$usage_count_too_large and does not decrement the count of initiated reference names for the given segment. If the hcs_$terminate_noname entry point reduces the count of initiated reference names of the segment to zero, the segment is removed from the address space of the process (made unknown). This entry point is used to clean up after making a segment known and initiating a single null reference name; see also the hcs_$initiate, hcs_$initiate_count, and hcs_$make_seg entry points. For a discussion of reference names, see "Constructing and Interpreting Names" in Section I of the MPM Commands.

Usage

```
     declare hcs_$terminate_noname entry (ptr, fixed bin(35));

     call hcs_$terminate_noname (seg_ptr, code);
```

where:

1. seg_ptr is a pointer to the segment. (Input)

2. code is a storage system status code. (Output)

Note

The hcs_$terminate_name entry point terminates a specified nonnull reference name; hcs_$terminate_file and hcs_$terminate_seg entry points terminate all reference names of a segment and make the segment unknown, given its pathname or segment number, respectively.

Name: hcs_$terminate_seg

The hcs_$terminate_seg entry point, given a pointer to a segment in the current process, terminates all the reference names of that segment and then removes the segment from the address space of the process (makes it unknown). For a discussion of reference names, see the "Constructing and Interpreting Names" in Section I of the MPM Commands.

Usage

```
declare hcs_$terminate_seg entry (ptr, fixed bin(1), fixed bin(35));

call hcs_$terminate_seg (seg_ptr, seg_sw, code);
```

where:

1. seg_ptr   is a pointer to the segment to be terminated. (Input)

2. seg_sw   is the reserved segment switch. (Input)
   1   saves segment number in reserved segment list
   0   does not save segment number

3. code   is a storage system status code. (Output)

Notes

The hcs_$terminate_file entry point performs the same operation given the pathname of a segment instead of a pointer; the hcs_$terminate_name and hcs_$terminate_noname entry points terminate a single reference name.

The term_$seg_ptr entry point (see the term_ subroutine description) performs the same operation as the hcs_$terminate_seg entry point, unsnapping links as well. Use of the term_ subroutine is recommended.

The only reference names that are removed are those for which the ring level associated with the name is greater than or equal to the validation level of the process. If any reference names exist that are associated with a ring level less than the validation level of the process, the segment is not made unknown and the code is returned as error_table_$bad_ring_brackets. For a discussion of rings refer to "Intraprocess Access Control (Rings)" in Section III of the MPM Subsystem Writers' Guide.

Name: hcs_$truncate_file

The hcs_$truncate_file entry point, 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 0´s in the words beyond the specified length.

Usage

```
declare hcs_$truncate_file entry (char(*), char(*), fixed bin(18), fixed
     bin(35));

call hcs_$truncate_file (dir_name, entryname, length, code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)

2. entryname is the entryname of the segment. (Input)

3. length is the new length of the segment in words. (Input)

4. code is a storage system status code. (Output)

Notes

The user must have write access on the segment in order to truncate it.

A directory cannot be truncated.

A segment is truncated as follows: all full pages after the page containing the last word of the new length segment (as defined by the length argument) are discarded. The remainder of the page containing the last word is converted to 0´s.

Bit count is not automatically set by the hcs_$truncate_file entry point. If desired, bit count may be set by using hcs_$set_bc.

The hcs_$truncate_seg entry point performs the same function when given a pointer to the segment instead of the pathname.

Name: hcs_$truncate_seg

The hcs_$truncate_seg entry point, 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 0's in the words beyond the specified length.

Usage

```
declare hcs_$truncate_seg entry (ptr, fixed bin(18), fixed bin(35));

call hcs_$truncate_seg (seg_ptr, length, code);
```

where:

1. seg_ptr 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 of the segment in words. (Input)

3. code is a storage system status code. (Output)

Notes

The user must have write access on the segment in order to truncate it.

A directory cannot be truncated.

A segment is truncated as follows: all full pages after the page containing the last word of the new length (as defined by the length argument) segment are discarded. The remainder of the page containing the last word is converted to 0's.

Bit count is not automatically set by the hcs_$truncate_seg entry point. If desired, bit count may be set by using hcs_$set_bc_seg.

The hcs_$truncate_file entry point performs the same function when given the pathname of the segment instead of the pointer.

Name: ioa_

The ioa_ subroutine is used for formatting a character string from fixed-point numbers, floating-point numbers, character strings, bit strings, and pointers. The character string is constructed according to the control characters entered in a "control string", and a variable list of arguments that are either edited into the output string in character form, or are used in some way to control the formatting of the string. The entire procedure is similar to formatted output in PL/I or FORTRAN.

Several entry points are provided in the ioa_ subroutine to provide various options concerning the formatting and disposition of the resulting string. Since all of the entry points can be called with a variable number of arguments, they must be declared with the following attributes:

```
declare ioa_ entry options (variable);
```

This entry declaration is assumed in all of the entries discussed.

Calls to the ioa_ subroutine normally append a newline character to the end of the string created. In order to be able to suppress this, each type of ioa_ call has a corresponding entry point (with "nnl", for no newline character, at the end of the name) that does the same editing, but does not append the newline character.

Entries: ioa_, ioa_$nnl

These two entry points format the input data according to the control string and write the resulting string on the I/O switch user_output. The resulting string is truncated if it exceeds 256 characters.

Usage

```
call ioa_ (control_string, arg1, ..., argn);
```

where:

1. control_string is a character string (char(*)) of text and control characters that determines how the resulting string is to be formed. (Input)

2. argi are a variable number of arguments (possibly none) that are either edited into the resulting string, or used to control the formatting of it. (Input)

Entries: ioa_$ioa_stream, ioa_$ioa_stream_nnl

These two entries format the resulting string as above, but the string is then written to an I/O switch specified by the switchname argument in the parameter list.

Usage

```
call ioa_$ioa_stream (switchname, control_string, arg1, ..., argn);
```

where:

1. switchname is the name of the I/O switch (char(*)) to which the resulting character string is to be written. (Input)
2. control_string is as above. (Input)
3. argi are as above. (Input)

Entries: ioa_$ioa_switch, ioa_$ioa_switch_nnl

These two entry points are identical to the ioa_$ioa_stream and ioa_$ioa_stream_nnl entry points except that the I/O switch is specified by a pointer to its control block, rather than by name. Since this saves an extra call in the I/O system to locate the control block, these calls are more efficient than ioa_$ioa_stream calls.

Usage

```
call ioa_$ioa_switch (iocb_ptr, control_string, arg1, ..., argn);
```

where:

1. iocb_ptr is a pointer to the switch's control block. (Input)
2. control_string is as described in the ioa_ entry point above. (Input)
3. argi are as described in the ioa_ entry point above. (Input)

Entries: ioa_$rs, ioa_$rsnnl, ioa_$rsnp, ioa_$rsnpnnl

These entry points edit the resulting string as in the above calls, but instead of being written to an I/O switch as the other ioa_ entry points, the string is passed back to the caller. The user program must provide a character string variable into which the string can be returned. This variable may be varying or nonvarying, aligned or unaligned, and of any length. The resulting string is truncated if it exceeds the length of the character string provided.

If the output string is nonvarying, it is padded on the right with spaces if it is not completely filled; however, if the call is to either the ioa_$rsnp or ioa_$rsnpnnl entry points, the padding is not done. Both the ioa_$rsnnl and ioa_$rsnpnnl entry points omit the newline character in the normal way. All of these entry points also return the length of the significant data edited into the string.

Usage

```
     call ioa_$rs (control_string, ret_string, len, arg1, ..., argn);
```

where:

1. control_string is as described in the ioa_ entry point above. (Input)

2. ret_string is a string (char(*) or char(*) varying) into which the output string will be edited. (Output)

3. len is the length of the returned string (fixed bin(17)). (Output)

4. argi are as described in the ioa_ entry point above. (Input)

Entry: ioa_$general_rs

This entry point is used to provide the ioa_ subroutine with a control string and format arguments taken from a previously created argument list to which a pointer has been obtained.

Usage

```
     declare ioa_$general_rs entry (ptr, fixed bin, fixed bin, char(*), fixed
          bin, bit(1) aligned, bit(1) aligned);

     call ioa_$general_rs (arglist_ptr, cs_argno, ff_argno, ret_string, len,
          pad_sw, nl_sw);
```

where:

1. arglist_ptr is a pointer to the argument list from which the control string and format arguments are to be taken. (Input)

2. cs_argno is the argument number of the control string in the argument list pointed to by arglist_ptr. (Input)

3. ff_argno is the argument number of the first format argument in the argument list pointed to by arglist_ptr. (Input)

4. ret_string contains the formatted string. It should be large enough to allow for expansion. (Output)

5. len specifies the number of significant characters in ret_string. (Output)

6. pad_sw is a switch to indicate whether the formatted string is padded. (Input)
"0"b no
"1"b yes

7. nl_sw is a switch to indicate whether a newline character is appended to the formatted string. (Input)
"0"b no
"1"b yes

Control Strings

All calls to the ioa_ subroutine require a control-string argument. This is a character string consisting of either text to be copied, ioa_ control codes, or both. The control codes are always identified by a leading circumflex character (^). Processing by the ioa_ subroutine begins by scanning the control string until a circumflex is found or the end of the string is reached. Any text (including any blanks) passed over is then copied to the output string. The control code is then interpreted and executed. Generally, this results in the next argument being edited into the output string in some character format. The scan then begins again for the next control code. Editing stops when the end of the control string is reached.

The ioa_ subroutine recognizes the following control codes:

^d	^nd	edit a fixed-point number
^i	^ni	edit a fixed-point number (same as ^d)
^f	^nf ^n.df ^.df	edit a floating-point number
^e	^ne	edit a floating-point number in exponential form
^o	^no	edit a fixed-point number in octal
^w	^nw	edit a full machine word in octal
^a	^na	edit a character string in ASCII
^b	^nb ^n.db ^.db	edit a bit string
^A		edit an acc string (ALM ASCII with count)
^p		edit a pointer
^¦	^n¦	insert formfeed character(s)
^/	^n/	insert newline character(s)

^-	^n-	insert horizontal tab character(s)
^x	^nx	insert space character(s)
^^	^n^	insert circumflex character(s)
^R		insert red ribbon shift character
^B		insert black ribbon shift character
^s	^ns	skip argument(s)
^(	^n(	start an iteration loop
^)		end an iteration loop

When n and/or d appear in a control code, they generally refer to a field width or a repetition factor, although the exact meaning depends on the control code with which they appear (see the detailed explanations that follow). The n or d must be specified as unsigned decimal integers, or as the letter "v". If "v" is used, the next argument in the argument list (which must be fixed binary) is used to obtain the actual value. If this argument happens to be negative, 0 is assumed.

When no field width is specified, the ioa_ subroutine uses a field large enough to contain the data to be edited. If a field size is specified that is too small to contain the data, the ioa_ subroutine ignores it and selects a field width of the appropriate size.

The control codes in the control string must correspond to the types of arguments in the argument list. For example, a ^d control code requires a corresponding numeric argument. If there is a mismatch between a control code and the type of the associated argument, the output for that field is a string of asterisks.

An invalid control code, an isolated circumflex character (^), or a control code that requires an argument after the argument list is exhausted, is inserted into the output string unchanged.

The numeric control codes (^d, ^i, ^f, and ^e) take any PL/I numeric data type and use standard PL/I conversion routines, if necessary. (If the argument is complex, only the real part of the argument is used.) It should be understood that these control codes, although similar to standard PL/I and FORTRAN format codes, do not, in general, give the same result. Also, most control codes ignore the field width if the argument is too large to fit into the field provided.

Each of the control codes that result in an argument being edited is explained in detail in the following paragraphs.

^d takes any numeric argument and edits it as a decimal integer. If n is not specified, the number is printed with no leading spaces or 0's. Negative numbers have a leading minus sign. If n is specified, the number is right justified with leading spaces. If the number is too large to fit in the specified field width, the field width is ignored.

^i is the same as ^d, for compatibility with FORTRAN and PL/I formats.

^f takes any numeric argument and edits it as a floating-point number with a decimal point. If n is omitted, P+1 is assumed, where P is the precision of the argument and the extra space is for the decimal point. If the number requires more than n-1 digits to express, it is edited using ^e format. The value d represents the number of digits after the decimal point. If d is omitted, any significant digits after the decimal point are printed, with trailing 0's omitted. If d is specified, the fractional part of the number is truncated, or padded with extra 0's to achieve the desired result. If n is not specified, the number is printed with no leading spaces or 0's (except for a 0 before the decimal point for numbers less than 1). If n is specified, the number is right justified with leading spaces.

^e takes any numeric argument and edits it in floating-point exponential format. The number is always left justified in the field provided, using a standard format. The value n, if used, only has meaning if the edited number is less than n characters in length. In this case, spaces are added to the end of the edited number to fill the field. The standard format that is always used is:

±n.dddde±nn

The first character is a space for positive numbers, or "-" for negative numbers. There is always one digit before the decimal point. The number of digits after the decimal point are enough to express the full precision of the argument. Trailing 0's in the mantissa are omitted. The exponent sign is omitted if positive. Leading 0's in the exponent are also omitted.

^o takes a fixed-point argument and edits it in octal. The format is the same as explained for ^d.

^w takes any argument and edits one machine word in octal. Leading 0's are printed. The word is interpreted as an unsigned 36-bit quantity. If n is omitted, 12 is assumed. If n>12, the number is right-justified with leading spaces. If n<12, the ioa_ subroutine attempts to suppress the first 12-n digits. If any of these digits are nonzero, the ioa_ subroutine chooses a value of n such that all significant digits are printed.

^a edits a character string in ASCII. Trailing spaces in the argument are ignored. If n is specified, the string is left justified and padded on the right with spaces. If the string (without any trailing spaces) is larger than n characters, the field width is ignored.

^b assumes bit string input and converts it to character form. The value d, when specified, is the byte size expressed in bits. It may take on only the values 1 through 4. If d is omitted or less than 1, 1 is assumed. If d is greater than 4, 4 is assumed. A d of 1 results in the string being output in binary; a d of 2 results in quarternary (base 4) output; a d of 3 results in octal output; and a d of 4 results in hexadecimal output. If the field width, n, is omitted, the length of the string divided by d is used. If n is specified, the string is truncated on the right, or padded on the right with spaces, whichever is appropriate.

^A edits an acc string (ALM ASCII with count). The parameter corresponding to the ^A should be a pointer to the string. Trailing spaces are not omitted, and no field width is accepted. This control code is used to print characters in the ALM acc format.

^p edits a pointer, entry variable, or label variable in a standard format, as follows:

```
sss|ooo(bb)
```

where sss is the segment number in octal, ooo is the offset in octal, and bb is the bit offset in decimal, all with leading 0's suppressed. If the bit offset is 0, the (bb) portion of the pointer is omitted.

^s causes the next argument in the parameter list to be ignored. A ^ns causes the next n arguments to be ignored; ^0s does nothing. If n is greater than or equal to the number of arguments remaining, the rest of the argument list is ignored.

^(starts an iteration loop, which must be ended by a corresponding ^). A ^n(specifies that the loop is to be repeated n times. The ^(specifies an indefinite iteration that is repeated until the argument list is exhausted. A ^0(causes everything in the control string up to the corresponding ^) to be ignored. Iterations may be nested up to four deep. The exact rules under which an iteration terminates are explained under ^).

^) marks the end of an iteration loop and either terminates the iteration or causes it to be repeated, depending on the following rules:

1. If n is not specified (the iteration is indefinite), then it is only repeated if there is something in the control string between the ^(and the ^) that requires an argument to be processed (such as ^a, ^v/, etc.), and there are arguments remaining that have not been processed. If either of these conditions are not met, the loop terminates.

2. If n is specified and there is nothing in the control string between the ^(and the ^) that requires an argument to be processed, the iteration is repeated until the repetition count is exhausted. If another repetition requires an argument, the loop is repeated only if arguments remain to be processed, regardless of the value of n.

Array Parameters

The arguments that are edited into the control string by the ioa_ subroutine may be arrays. If this is the case, the ioa_ subroutine selects elements from the array until all array elements are used before going to the next argument in the argument list. All conventions apply to elements of arrays that apply to simple scalar arguments. In particular, the ^s control code skips the next element of an array if the ioa_ subroutine is currently in the process of selecting elements from an array. The arrays are scanned in the order that PL/I allocates the elements, i.e., row major order.

Examples

The following examples illustrate many, but not all, of the features of the ioa_ subroutine. The symbol ƀ is used to represent a space in places where the space is significant.

```
Source:   call ioa_("This is ^a the third of ^a","Mon","July");

Result:   This is Mon the third of July


Source:   call ioa_("date ^d/^d/^d, time ^d:^d",6,20,74,2014,36);

Result:   date 6/20/74, time 2014:36


Source:   call ioa_("overflow at ^p",ptr);

Result:   overflow at 271¦4671


Source:   call ioa_("^2(^2(^w ^)^/^)",w1,w2,w3,w4);

Result:   112233445566 000033004400
          000000000001 777777777777


Source:   bit="110111000011"b;
          call ioa_("^vxoct=^.3b hex=^.4b",6,bit,bit);

Result:   ƀƀƀƀƀƀoct=6703ƀhex=DC3


Source:   call ioa_("^f ^e ^f ^5.2f",1.0,1,1e-10,1);

Result:   1. ƀ1.e0 ƀ1.e-10 ƀ1.00


Source:   call ioa_("^(^d ^)",1,2,56,198,456.7,3e6);

Result:   1 2 56 198 456 3000000
```

```
Source:  abs_sw=0;
         call ioa_$rsnnl("^v(Absentee user ^)^a ^a logged out.",
              out_str,out_cnt,abs_sw,"LeValley","Shop");

Result:  out_cnt=25;
         out_str="LeValley Shop logged out."


Source:  abs_sw=1; /* Using same call to ioa_$rsnnl */
         call ioa_$rsnnl("^v(Absentee user ^)^a ^a logged out.",
              out_str,out_cnt,abs_sw,"LeValley","Shop");

Result:  out_cnt=39;
         out_str="Absentee user LeValley Shop logged out."


Source:  dcl a(2,2)fixed bin init(1,2,3,4);
         call ioa_("^d^s ^d ^w",a);

Result:  1 3 000000000004


Source:  dcl b(6:9)fixed bin init(6,7,8,9);
         call ioa_("^v(^3d ^)",dim(b,1),b);

Result:    6  7  8  9
```

Summary of Entry Points

ioa_, ioa_$nnl

```
call ioa_ (control_string, arg1, ..., argn);
```

ioa_$ioa_stream, ioa_$ioa_stream_nnl

```
call ioa_$ioa_stream (switchname, control_string, arg1, ..., argn);
```

ioa_$ioa_switch, ioa_$ioa_switch_nnl

```
call ioa_$ioa_switch (iocb_ptr, control_string, arg1, ..., argn);
```

ioa_$rs, ioa_$rsnnl, ioa_$rsnp, ioa_$rsnpnnl

```
call ioa_$rs (control_string, ret_string, len, arg1, ..., argn);
```

ioa_$general_rs

```
call ioa_$general_rs  (arglist_ptr,  as_argno,  ff_argno,  ret_string,
     len, pad_sw, nl_sw);
```

Name: iox_

This subroutine performs I/O operations and some related functions. The user should be familiar with the contents of "Multics Input/Output System" and "File Input/Output" in Section V of the MPM Reference Guide. Most of the entry points to the iox_ subroutine are described in the following pages; however, those entry points generally needed only by users who are writing their own I/O modules are described in Section VII of the MPM Subsystem Writers' Guide.

Each entry point documented here has an argument denoting the particular I/O switch involved in the operation. For an entry point that requires the I/O switches to be in the attached state, the description of the entry point's function applies only when the switch is attached to a file or is attached to a device via the I/O module tty_. For the meaning of operations on a switch attached as a synonym, see "Multics Input/Output System" in Section V of the MPM Reference Guide. For other attachments, see the description of the particular I/O module. (The standard system I/O modules are described in Section III of this document.)

When an entry point requires the I/O switch to be opened, and it is not open, the state of the switch is not changed, and the code error_table_$not_open is returned. If the I/O switch is open but not in one of the allowed opening modes, the state of the switch is not changed, and the code that is returned is error_table_$no_operation.

Operations pertaining to files use four position designators for reference: the next byte, the next record, the current record, and the key for insertion. Their use is explained in "File Input/Output" in Section V of the MPM Reference Guide. (Refer to Section V of the MPM Reference Guide for more information on opening modes and how they relate to other I/O operations, file attachments, position designators, file types, and I/O modules.)

Several operations involve the use of a buffer. A buffer is a block of storage provided by the caller of the operation as the target for input or the source for output. A buffer must be byte aligned; i.e., its bit address and bit length must both be evenly divisible by 9.

The code returned by an entry point may be other than a standard status code in cases where the I/O switch is attached via a nonstandard I/O module. (For a list of the most often encountered standard status codes, see Section VII of the MPM Reference Guide.)

Entry: iox_$attach_ptr

This entry point attaches an I/O switch in accordance with a specified attach description. The form of an attach description is given in "Multics Input/Output System" in Section V of the MPM Reference Guide. If the switch is not in the detached state, its state is not changed, and the code error_table_$not_detached is returned.

Usage

```
declare iox_$attach_ptr entry (ptr, char(*), ptr, fixed bin(35));

call iox_$attach_ptr (iocb_ptr, atd, ref_ptr, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. atd is the attach description. (Input)

3. ref_ptr is a pointer to the referencing procedure, used by the search rules to find an I/O module. (Input)

4. code is an I/O system status code. (Output)

Entry: iox_$attach_name

This entry point is the same as the iox_$attach_ptr entry point except that the I/O switch is designated by name and a pointer to its control block is returned. The control block is created if it does not already exist.

Usage

```
declare iox_$attach_name entry (char(*), ptr, char(*), ptr, fixed bin(35));

call iox_$attach_name (switchname, iocb_ptr, atd, ref_ptr, code);
```

where:

1. switchname is the name of the I/O switch. (Input)

2. iocb_ptr points to the switch's control block. (Output)

3. atd is the attach description. (Input)

4. ref_ptr is a pointer to the referencing procedure, used by the search rules to find an I/O module. (Input)

5. code is an I/O system status code. (Output)

Entry: iox_$close

This entry point closes an I/O switch. If the switch is not open, its state is not changed, and the code error_table_$not_open is returned.

Usage

```
declare iox_$close entry (ptr, fixed bin(35));

call iox_$close (iocb_ptr, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. code is an I/O system status code. (Output)

Entry: iox_$control

This entry point performs a specified control order on an I/O switch. The allowed control orders depend on the attachment of the switch. If a control order is not supported for a particular attachment, the code error_table_$no_operation is returned if the switch is open. If the switch is closed, the code error_table_$not_open or error_table_$no_operation is returned, the latter code only by I/O modules that support orders with the switch closed. For details on control orders, see the description of the particular I/O module used in the attach operation.

Usage

```
declare iox_$control entry (ptr, char(*), ptr, fixed bin(35));

call iox_$control (iocb_ptr, order, info_ptr, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. order is the name of the control order. (Input)

3. info_ptr is null or points to data whose form depends on the attachment. (Input)

4. code is an I/O system status code. (Output)

Entry: iox_$delete_record

This entry point deletes the current record from the file to which an I/O switch is attached. The switch must be open for sequential_update, keyed_sequential_update, or direct_update. If the current record is null, the file's position is not changed, and the code error_table_$no_record is returned.

If the file is open for direct_update and the deletion takes place, the current and next record positions are set to null. For keyed_sequential_update, the current and next record positions are set to the record following the deleted record or to end of file (if there is no such record).

Usage

```
     declare iox_$delete_record entry (ptr, fixed bin(35));

     call iox_$delete_record (iocb_ptr, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. code is an I/O system status code. (Output)

Entry: iox_$detach_iocb

This entry point detaches an I/O switch. If the switch is already detached, its state is not changed, and the code error_table_$not_attached is returned. If the switch is open, its state is not changed, and the code error_table_$not_closed is returned.

Usage

```
     declare iox_$detach_iocb entry (ptr, fixed(35));

     call iox_$detach_iocb (iocb_ptr, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. code is an I/O system status code. (Output)

Entry: iox_$find_iocb

This entry point returns a pointer to the control block for an I/O switch. The control block is created if it does not already exist.

Usage

```
     declare iox_$find_iocb entry (char(*), ptr, fixed bin(35));

     call iox_$find_iocb (switchname, iocb_ptr, code);
```

where:

1. switchname is the name of the I/O switch. (Input)

2. iocb_ptr points to the switch's control block. (Output)

3. code is an I/O system status code. (Output)

Entry: iox_$get_chars

This entry point reads 9-bit bytes from the unstructured file or device to which an I/O switch is attached. The switch must be open for stream_input or stream_input_output. The desired number of bytes, n, is specified in the call. Some I/O modules may actually read fewer than n bytes into the buffer, even though n bytes are available from the file or device. In this case the code error_table_$short_record is returned. When this code is returned, the caller may again call the iox_$get_chars entry point to get more bytes. The contents of the buffer beyond the last byte read are undefined.

If the switch is attached to a file, bytes are read beginning with the next byte, and the next byte position designator is advanced by the number of bytes read. If fewer than n bytes remain in the file, the code error_table_$short_record is returned, and the next byte position is set to end of file. If the next byte position is already at end of file, the code error_table_$end_of_info is returned.

Usage

```
     declare iox_$get_chars entry (ptr, ptr, fixed bin(21), fixed bin(21), fixed
          bin(35));

     call iox_$get_chars (iocb_ptr, buff_ptr, n, n_read, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. buff_ptr points to the byte-aligned buffer into which bytes are to be read. (Input)

3. n is the number of bytes to be read where n $\geq$ 0. (Input)

4. n_read is the number of bytes actually read. If code is 0, n_read equals n. (Output)

5. code is an I/O system status code. (Output)

Entry: iox_$get_line

This entry point reads 9-bit bytes from the unstructured file or device to which an I/O switch is attached. The switch must be open for stream_input or stream_input_output. Bytes are read until the input buffer is filled, a newline character is read, or end of file is reached, whichever occurs first. A code of 0 is returned if and only if a newline character is read into the buffer (it will be the last character read). If the input buffer is filled without reading a newline character, the code error_table_$long_record is returned. The contents of the buffer beyond the last byte read are undefined.

If the switch is attached to a file, bytes are read beginning with the next byte, and the next byte position designator is advanced by the number of bytes read. If the next byte is initially at end of file, the code error_table_$end_of_info is returned. Otherwise, if the end of file is reached without reading a newline character, the next byte position designator is set to end of file and the code error_table_$short_record is returned.

Usage

```
declare iox_$get_line entry (ptr, ptr, fixed bin(21), fixed bin(21), fixed
     bin(35));

call iox_$get_line (iocb_ptr, buff_ptr, buff_len, n_read, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. buff_ptr points to a byte-aligned buffer. (Input)

3. buff_len is the length of the buffer in bytes. (Input)

4. n_read is the number of bytes read into the buffer. (Output)

5. code is an I/O system status code. (Output)

Entry: iox_$modes

This entry point is used to obtain or set modes that affect the subsequent behavior of an I/O switch. The switch must be attached via an I/O module that supports modes. If the switch is not attached, the code error_table_$not_attached is returned. If the switch is attached, but modes are not supported, the code error_table_$no_operation is returned for an open switch and the code error_table_$not_open is returned for a closed switch. If the switch is attached and modes are supported, but an invalid mode is given, the code error_table_$bad_mode is returned.

Each mode is a sequence of nonblank characters. A mode string is a sequence of modes, separated by commas and containing no blanks. For a list of valid modes, see the particular I/O module involved.

Usage

```
declare iox_$modes entry (ptr, char(*), char(*), fixed bin(35));

call iox_$modes (iocb_ptr, new_modes, old_modes, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. new_modes is the mode string containing the modes to be set. Other modes are not affected. If this argument is the null string, no modes are changed. (Input)

3. old_modes is the string of modes in force when the call is made. If this argument has length zero, this information is not returned. (Output)

4. code is an I/O system status code. (Output)

This page intentionally left blank.

Entry: iox_$move_attach

This entry point moves an attachment from one I/O switch, s1, to another I/O switch, s2. The s1 switch must be in the attached state and the s2 switch must be in the detached state when the entry point is called. If not, either the code error_table_$not_attached (s1) or error_table_$not_detached (s2) is returned and no change is made to either I/O switch.

Moving the attachment moves the attach description and open description of the s1 switch to the s2 switch. All pointer values and entry values are copied from the control block of the s1 switch to the control block of the s2 switch. (These values are listed in "I/O Control Block" in Section IV of the MPM Subsystem Writers' Guide.) Attach and open data blocks maintained by the I/O module (if the s1 switch is attached) are not affected. Finally, the s1 switch is set to the detached state and iox_$propagate (described in the MPM Subsystem Writers' Guide) is called for both I/O switches.

Usage

```
declare iox_$move_attach entry (ptr, ptr, fixed(35));

call iox_$move_attach (iocb_ptr_1, iocb_ptr_2, code);
```

where:

1. iocb_ptr_1 points to the control block for the I/O switch that is currently attached. This switch is identified as s1 in the discussion above. (Input)

2. iocb_ptr_2 points to the control block for the I/O switch that the user intends to attach. This switch is identified as s2 in the discussion above. (Input)

3. code is an I/O system status code. (Output)

Entry: iox_$open

This entry point opens an I/O switch. The switch must be attached via an I/O module that supports the specified opening mode, and it must be in the closed state. If the switch is not attached, its state is not changed, and the code error_table_$not_attached is returned. If the switch is already open, the code error_table_$not_closed is returned.

If the switch is attached to a file, the appropriate file position designators are established, and an existing file may be replaced by an empty file. This replacement may be avoided by specifying extension of the file in the attach description. See "File Input/Output" in Section IV of the MPM Reference Guide for full details.

Usage

```
declare iox_$open (ptr, fixed bin, bit (1) aligned, fixed bin(35));

call iox_$open (iocb_ptr, mode, unused, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. mode is the number assigned to the mode as shown in Table A-1 in Appendix A, e.g., 1 for stream_input, 2 for stream_output. (Input)

3. unused must be "0"b. (Input)

4. code is an I/O system status code. (Output)

Entry: iox_$position

For an I/O switch attached to a file, this entry point positions to the beginning or end of the file, or skips forward or backward over a specified number of lines (unstructured files) or records (structured files). For an I/O switch attached to a device, this operation reads and discards characters until a specified number of newline characters have been skipped.

The switch must be opened in one of the following modes:

stream_input
stream_input_output
sequential_input
sequential_input_output
sequential_update
keyed_sequential_input
keyed_sequential_update

In addition, for keyed openings, the next record position should not be null. If it is null, the code error_table_$no_record is returned.

Usage

```
     declare iox_$position entry (ptr, fixed bin, fixed bin(21), fixed bin(35));

     call iox_$position (iocb_ptr, type, n, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. type identifies the type of positioning. (Input)
   -1 goes to the beginning of the file
   +1 goes to the end of the file
   0 skips newline characters or records

3. n is the number of lines or records to be skipped (forward skip) or the negative of that number (backward skip). It may be 0. (Input)

4. code is an I/O system status code. (Output)

Notes

Positioning to the beginning of a nonempty file sets the next record position at the first record in the file (sequential and keyed_sequential openings) or sets the next byte position at the first byte in the file (stream openings). Positioning to the end of a file, or to the beginning of an empty file, sets the relevant position designator to the end-of-file position.

Successively skipping records (sequential and keyed_sequential openings) moves the next record position forward or backward by the specified number of records, n, provided that many records exist in the indicated direction. For example, suppose that when the iox_$position entry point is called, the next record is the mth record in the file, and n records are to be skipped. Then for a successful forward skip, the file must contain at least (m+n-1) records, and the next record will be set to record (m+n) (if there are at least m+n records in the file) or to end of file (if there are m+n-1 or fewer records in the file). For a successful backward skip, m must be greater than n, and the next record position is set to record (m-n).

Successively skipping forward over newline characters (stream openings) advances the next byte position over the specified number, n, of newline characters, leaving it at the byte following the nth newline character or at end of file (if the nth newline character is the last byte in the file). Successively skipping backward over n newline characters moves the next byte position backward to the nth preceding newline character and then moves it further backward as far as is possible without encountering another newline character. The effect is to set the next byte position to the first character in a line.

If the relevant part of the file contains too few records or newline characters, the next record position or next byte position is set to the first record or byte (backward skip with nonempty file) or end of file (all other cases), and the code error_table_$end_of_info is returned.

When a call to the iox_$position entry point specifies skipping zero lines or records, the skip is successful, and the next record position is undisturbed.

In openings for update, the current record position is set to the resulting next record or null if the next record is at end of file.

In the case of keyed_sequential_update, the key for insertion is set to null.

Entry: iox_$put_chars

This entry point writes a specified number of 9-bit bytes to the unstructured file or device to which an I/O switch is attached. The switch must be open for stream_output or stream_input_output.

In the case of a file, if the opening is for stream_output, the bytes are simply added at the end of the file. However, if the opening is for stream_input_output, and the next byte position is not at end of file, the file is first truncated so that the byte preceding the next byte becomes the last byte in the file. The bytes being written are then added at the end of the file, and the next byte position is set to end of file.

Truncation can be suppressed in storage system files by specifying an appropriate attach option. See the description of the vfile_ I/O module in Section III for details.

Usage

```
declare iox_$put_chars entry (ptr, ptr, fixed bin(21), fixed bin(35));

call iox_$put_chars (iocb_ptr, buff_ptr, n, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)
2. buff_ptr points to a byte-aligned buffer containing the bytes to be written. (Input)
3. n is the number of bytes to be written where n ≥ 0. (Input)
4. code is an I/O system status code. (Output)

Entry: iox_$read_key

This entry point returns both the key and length of the next record in an indexed file attached to an I/O switch. The switch must be open for keyed_sequential_input or keyed_sequential_update. If the next record position is at end of file, the code error_table_$end_of_info is returned. If the next record position is null, the code error_table_$no_record is returned. The next record position is unchanged and the current record position is set to the next record if the operation is successful; otherwise, the current record position is set to null.

Usage

```
     declare iox_$read_key entry (ptr, char(256) varying, fixed  bin(21),  fixed
          bin(35));

     call iox_$read_key (iocb_ptr, key, rec_len, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)
2. key is the next record's key. (Output)
3. rec_len is the next record's length in bytes. (Output)
4. code is an I/O system status code. (Output)

Entry: iox_$read_length

This entry point returns the length of the next record in a structured file attached to an I/O switch. The switch must be opened in one of the following modes:

```
     sequential_input
     sequential_input_output
     sequential_update
     keyed_sequential_input
     keyed_sequential_update
     direct_input
     direct_update
```

If the next record position is at end of file, the code error_table_$end_of_info is returned. If the next record position is null, the code error_table_$no_record is returned. The next record position is unchanged and the current record position is set to the next record if the operation is successful; otherwise, the current record position is set to null.

Usage

```
     declare iox_$read_length entry (ptr, fixed bin(21), fixed bin(35));

     call iox_$read_length (iocb_ptr, rec_len, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. rec_len is the next record's length in bytes. (Output)

3. code is an I/O system status code. (Output)

Entry: iox_$read_record

This entry point reads the next record in a structured file to which an I/O switch is attached. The switch must be opened in one of the following modes:

sequential_input
sequential_input_output
sequential_update
keyed_sequential_input
keyed_sequential_update
direct_input
direct_update

The read is successful if the next record position is at a record. If the next record position is at end of file, the code error_table_$end_of_info is returned. If the next record position is null, the code error_table_$no_record is returned.

In sequential and keyed_sequential openings, a successful read advances the next record position by one record; an unsuccessful read leaves it at the end of file or null. In direct openings, this operation always sets the next record position to null. In openings for update, a successful read sets the current record position to the record just read; an unsuccessful read sets it to null. In openings for keyed_sequential_update and direct_update, the key for insertion is always set to null.

If the record is too long for the specified buffer, the first part of the record is read into the buffer, and the code error_table_$long_record is returned. As far as setting position indicators is concerned, this is considered a successful read. In all cases, the contents of the buffer beyond the last byte read are undefined.

Usage

```
     declare iox_$read_record entry (ptr, ptr,  fixed  bin(21),  fixed  bin(21),
          fixed bin(35));

     call iox_$read_record (iocb_ptr, buff_ptr, buff_len, rec_len, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. buff_ptr points to a byte-aligned buffer into which the record is to be read. (Input)

3. buff_len is the length of the buffer in bytes. (Input)

4. rec_len is the length of the record in bytes. (Output)

5. code is an I/O system status code. (Output)

Entry: iox_$rewrite_record

This entry point replaces the current record in a structured file to which an I/O switch is attached. The switch must be open for sequential_update, keyed_sequential_update, or direct_update. If the current record position is null, the code error_table_$no_record is returned.

For keyed_sequential_update and sequential_update, this operation sets the next record position to the record immediately following the current record or to end of file (if no such record exists). (It is possible that the next record position may already be at this point). For direct_update, the next record position is set to null. No other changes are made to the position designators.

Usage

```
     declare iox_$rewrite_record entry (ptr, ptr, fixed bin(21), fixed bin(35));

     call iox_$rewrite_record (iocb_ptr, buff_ptr, rec_len, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. buff_ptr points to a byte-aligned buffer containing the new record. (Input)

3. rec_len is the length of the new record. (Input)

4. code is an I/O system status code. (Output)

Entry: iox_$seek_key

This entry point searches for a record with a given key in an indexed file to which an I/O switch is attached. It also serves to define the key for a record to be added by a following write_record operation. The switch must be opened in one of the following modes:

keyed_sequential_input
keyed_sequential_output
keyed_sequential_update
direct_input
direct_output
direct_update

For keyed_sequential_output, the given key should be greater (according to the rules for character-string comparison) than the key of the last record in the file. If it is, the code error_table_$no_record is returned, and the key for insertion is set to the given key. Otherwise, the code error_table_$key_order is returned, and the key for insertion is set to null.

For other openings, this entry point performs as follows:

1. If the file contains a record with the given key, a code of 0 is returned, the record's length is returned, the next record position and current record position are set to the record, and the key for insertion is set to null. (Not all of these position designators are applicable in all openings.)

2. If the file does not contain a record with the given key, the code error_table_$no_record is returned, the next record position and current record position are set to null, and the key for insertion is set to the given key. (Not all of these position designators are applicable in all openings.)

Usage

```
     declare iox_$seek_key entry (ptr, char(256) varying, fixed  bin(21),  fixed
          bin(35));

     call iox_$seek_key (iocb_ptr, key, rec_len, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)

2. key contains the given key. All trailing blanks are removed from key to obtain the given key, and the result may be the null string. (Input)

3. rec_len is the length in bytes of the record with the given key. (Output)

4. code is an I/O system status code. (Output)

Entry: iox_$write_record

This entry point adds a record to a structured file to which an I/O switch is attached. The switch must be opened in one of the following modes:

sequential_output
sequential_input_output
keyed_sequential_output
keyed_sequential_update
direct_output
direct_update

If the switch is open for sequential_output, the record is added at the end of the file. If the switch is open for sequential_input_output, and the next record position is not at the end of the file, the file is truncated so that the record preceding the next record becomes the last record in the file. The new record is then added at the end of the file.

Truncation can be suppressed in sequential_input_output, and write operations can be performed in sequential_update openings of storage system files. See the description of the vfile_ I/O module for details.

If the switch is open for keyed_sequential_output, keyed_sequential_update, direct_output, or direct_update, the key for insertion designator should designate a key. If it does not, the code error_table_$no_key is returned and nothing is changed. If there is a key for insertion, the new record is added to the file with that key and the key for insertion is set to null. For keyed_sequential_update, and sequential_update, the next record position is set to the record immediately following the new record or to end of file (if there is no such record). For keyed_sequential_update, sequential_update, and direct_update, the current record position is set to the new record.

Usage

```
declare iox_$write_record entry (ptr, ptr, fixed bin(21), fixed bin(35));

call iox_$write_record (iocb_ptr, buff_ptr, rec_len, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input)
2. buff_ptr points to a byte-aligned buffer containing the new record. (Input)
3. rec_len is the length of the new record in bytes. (Input)
4. code is an I/O system status code. (Output)

Summary of Entry Points

```
call iox_$attach_ptr (ref_ptr, atd, ref_ptr, code);
call iox_$attach_name (switchname, iocb_ptr, atd, ref_ptr, code);
call iox_$close (iocb_ptr, code);
call iox_$control (iocb_ptr, order, info_ptr, code);
call iox_$delete_record (iocb_ptr, code);
call iox_$detach_iocb (iocb_ptr, code);
call iox_$find_iocb (switchname, iocb_ptr, code);
call iox_$get_chars (iocb_ptr, buff_ptr, n1, n_read1, code);
call iox_$get_line (iocb_ptr, buff_ptr, buff_len, n_read2, code);
call iox_$modes (iocb_ptr, new_modes, old_modes, code);
call iox_$move_attach (iocb_ptr1, iocb_ptr2, code);
call iox_$open (iocb_ptr, mode, unused, code);
call iox_$position (iocb_ptr, type, n2, code);
call iox_$put_chars (iocb_ptr, buff_ptr, n3, code);
call iox_$read_key (iocb_ptr, key1, rec_len1, code);
call iox_$read_length (iocb_ptr, rec_len1, code);
call iox_$read_record (iocb_ptr, buff_ptr, buff_len, rec_len2, code);
call iox_$rewrite_record (iocb_ptr, buff_ptr, rec_len2, code);
call iox_$seek_key (iocb_ptr, key2, rec_len3, code);
call iox_$write_record (iocb_ptr, buff_ptr, rec_len4, code);
```

where:

1. iocb_ptr points to the switch's control block. (Input or Output)

2. atd is the attach description. (Input)

3. ref_ptr is a pointer to the referencing procedure, used by the search rules to find an I/O module. (Input)

4. code is an I/O system status code. (Output)

5. switchname is the name of the I/O switch. (Input)

6. order is the name of the control order. (Input)

7. info_ptr is null or points to data whose form depends on the attachment. (Input)

8. buff_ptr points to the byte-aligned buffer. (Input)

9. n1 is the number of bytes to be read where n1≥0. (Input)

10. n_read1 is the number of bytes actually read. If code is 0, n_read1 equals n1. (Output)

11. buff_len is the length of the buffer in bytes. (Input)

12. n_read2 is the number of bytes read into the buffer. (Output)

13. new_modes is the mode string containing the modes to be set. Other modes are not affected. If this argument is the null string, no modes are changed. (Input)

14. old_modes is the string of modes in force when the call is made. If this argument has length zero, this information is not returned. (Output)

15. iocb_ptr1 points to the control block for the I/O switch that is currently attached. (Input)

16. iocb_ptr2 points to the control block for the I/O switch that the user intends to attach. (Input)

17. mode is the number assigned to the mode as shown in Table A-1 in Appendix A, e.g., 1 for stream_input, 2 for stream_output. (Input)

18. unused must be "0"b. (Input)

19. type identifies the type of positioning. (Input)
-1 goes to the beginning of the file
+1 goes to the end of the file
0 skips newline characters or records

20. n2 is the number of lines or records to be skipped (forward skip) or the negative of that number (backward skip). It may be 0. (Input)

21. n3 is the number of bytes to be written where $n3 \geq 0$. (Input)

22. key1 is the next record's key. (Output)

23. rec_len1 is the next record's length in bytes. (Output)

24. rec_len2 is the length of the record in bytes. (Output)

25. rec_len3 is the length of the new record. (Output)

26. key2 contains the given key. All trailing blanks are removed from key to obtain the given key, and the result may be the null string. (Input)

27. rec_len4 is the length in bytes of the record with the given key. (Output)

This page intentionally left blank.

Name: print_cobol_error_

The print_cobol_error_ subroutine allows the COBOL programmer to display the cause and location of a runtime error. It is meaningful only when called from within a USE procedure in the DECLARATIVE section of a COBOL program. The error information displayed pertains to the error causing the current execution of the USE procedure. This is identical to the messages that would have been printed on the terminal before aborting the program (i.e., signalling the "error" condition) had no USE procedure been provided.

If the main entry point is used, the error information is displayed through the user_output I/O switch.

Usage

```
     call "print_cobol_error_".
```

Entry: print_cobol_error_$switch

This entry point outputs the error information to a specified I/O switch.

Usage

```
     01 switch-name pic x(32).

     call "print_cobol_error_$switch" using switch-name.
```

where switch-name is the name of an I/O switch that is open for output. This includes user_output and error_output, as well as the I/O switch associated with any open external COBOL file, i.e., the internal-file-name as specified in the SELECT clause of the ENVIRONMENT DIVISION. (Input)

Name: random_

The random_ subroutine is a random number generator with entry points that, 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.

There are two sets of entry points to the random_ subroutine. For one set of entry points, each call 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.

The second set of entry points returns an array with a sequence of random numbers. The first element of the array is generated from the input seed. The returned value of the seed is used to generate the next random number of the sequence. The modification of the input seed value occurs once for each element in the array. The programmer can obtain the same result by making one call to an array entry point having n elements or by making n calls to the corresponding single random number entry point.

In addition, for the uniform and normal distributions, there are entry points that produce the negative 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 point.

Entry: random_$uniform

The random_$uniform entry point generates a random number with a value between 0.0 and 1.0. The sequence of random numbers has a uniform distribution on the interval 0 to 1.

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);
```

where:

1. seed — is the optional seed (see "Notes"). (Input or Output)
   - Input — must be a nonzero positive integer; used to generate the random number
   - Output — is the new value (modification of input value); used to generate the next random number of the sequence

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);
```

where:

1. seed — is the optional seed (see "Notes"). (Input or Output)
   - Input — must be a nonzero positive integer; used to generate the first random number in the array
   - Output — is the new value (modification of input value); used to generate the next random number of the sequence; the modification of the input value occurs array_size times

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 the random_no produced by the random_$uniform entry point. 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);
```

where:

1. seed — is the same as in the random_$uniform entry point above. (Input or Output)

2. random_ant — is the random number that is generated. (Output)

Entry: random_$uniform_ant_seq

The random_$uniform_ant_seq entry point returns an array, ant_array, of uniformly distributed random numbers that are negatively correlated with the array produced by the random_$uniform_seq entry point. For any particular value of the seed:

```
(ant_array(i) + array(i)) = 1.0
```

where the range of values for *i* is from 1 to 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);
```

where:

1. seed — is the same as in the random_$uniform_seq entry point above. (Input or 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 random_$normal entry point generates a random number greater than -6.0 and less than 6.0. The sequence of random numbers has an approximately normal distribution with a mean of 0 and a variance of 1. The random number is formed by taking the sum of 12 successive random numbers from the uniformly distributed sequence and then adjusting the sum for a mean of 0 by subtracting 6.0.

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);
```

where the seed and random_no arguments are the same as in the random_$uniform entry point above.

Entry: random_$normal_seq

The random_$normal_seq entry point generates a sequence of random variables with an approximately normal distribution. The sequence contains the number of values specified in the array_size argument.

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);
```

where the seed, array, and array_size arguments are the same as in the random_$uniform_seq entry point above.

Entry: random_$normal_ant

The random_$normal_ant entry point generates a random number, random_ant, that is negatively correlated with the random_no argument produced by the random_$normal entry point. 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);
```

where the seed and random_ant arguments are the same as in the random_$uniform_ant entry point above.

Entry: random_$normal_ant_seq

The random_$normal_ant_seq entry point generates a sequence of array_size, of random variables with approximately normal distribution. The sequence contains the number of values specified in the array_size argument. These variables are negatively correlated with those produced by the random_$normal_seq entry point.

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);
```

where the seed, ant_array, and array_size arguments are the same as in the random_$uniform_ant_seq entry point above.

Entry: random_$exponential

The random_$exponential entry point generates a positive random number. The sequence of random numbers has an exponential distribution with a mean of 1. The random number is generated by taking successive random numbers from the uniformly distributed sequence and applying the VonNeumann method 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);
```

where the seed and random_no arguments are the same as in the random_$uniform entry point above.

Entry: random_$exponential_seq

The random_$exponential_seq entry point 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);
```

where the seed, array, and array_size arguments are the same as in the random_$uniform_seq entry point above.

Entry: random_$get_seed

The random_$get_seed entry point 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);
```

where seed_value is the current value of the internal seed. (Output)

Entry: random_$set_seed

The random_$set_seed entry point 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);
```

where seed_value is the value to which the internal seed is set. This value must be a nonzero positive integer. (Input)

Notes

For all entry points (except random_$set_seed and random_$get_seed), if the optional argument, 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 of 4084114320. Its value is changed only by calls to random_$set_seed or by calls to other entry points in which the optional argument, seed, is not included.

The value of a seed must be a nonzero positive integer so that a valid value will be returned for the seed and the random numbers. If 0 is used for the value of seed, the new value of the seed and the random numbers will be 0. If the value of a seed is negative, the low-order 35 bits of the internal representation are used as the seed. A given seed always produces 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 is 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, for any given input seed, produce different values for the next seed.

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. The algorithm, in terms of the 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).

1. Let register A initially contain the previous random number in bit positions 1 to n with 0 in the sign bit (position 0).

2. Copy register A into register B and then right-shift register B m places.

3. 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 positions 1 through m.)

4. 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 0's in positions m+a through n.)

5. Exclusive-or register B into register A and set register A's sign bit to 0. (Register A now contains all n bits of the new random number.)

6. To obtain a random number between 0.0 and 1.0, 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 the random_ subroutine, a word is considered 36 bits long including the sign bit. This generates 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 the random_ subroutine, a shift of 2 is used, which gives a cycle of (2**35)-1. The random number generating portion of the assembly language code used by the random_ subroutine 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
```

Name: release_temp_segments_

The release_temp_segments_ subroutine is used to return temporary segments (acquired with the get_temp_segments_ subroutine) to the free pool of temporary segments associated with each user process. Through the pool concept, temporary segments can be used more than once during the life of a process. Since the process does not have to create a new segment each time one is needed, overhead costs are decreased.

Usage

```
declare release_temp_segments_ entry (char (*), (*) ptr, fixed bin (35));

call release_temp_segments_ (program_name, ptrs, code);
```

where:

1. program_name is the name of the program releasing the temporary segments. (Input)

2. ptrs is an array of pointers to the temporary segments being released. (Input/Output)

3. code is a standard system status code. (Output)

Notes

A nonzero status code is returned if any segment being released was not assigned to the given program. See the description of the get_temp_segments_ subroutine for a description of how to acquire temporary segments.

The pointers in the ptrs array above are set to the null value after the segments are successfully returned to the free pool. This fact can be used by callers to determine if a given temporary segment has been released.

Name: send_mail_

The send_mail_ subroutine sends one message to a specified user and optionally sends a wakeup with the message.

Usage

```
declare send_mail_ entry (char(*), char(*), ptr, fixed bin(35));

call send_mail_ (destination, message, info_ptr, code);
```

where:

1. destination is a Person_id.Project_id destination. (Input)

2. message is the text of the message to be sent. (Input)

3. info_ptr points to the following structure: (Input)

```
dcl 1 send_mail_info  aligned,
      2 version        fixed bin,
      2 sent_from      char(32) aligned,
      2 switches,
        3 wakeup       bit(1) unal,
        3 mbz1         bit(1) unal,
        3 always_add   bit(1) unal,
        3 never_add    bit(1) unal,
        3 mbz2         bit(1) unal,
        3 acknowledge  bit(1) unal,
        3 mbz          bit(30) unal;
```

where:

version identifies the version of the structure being used. Currently this number must be 1.

sent_from additional information about the sender, e.g., name of anonymous user or name of network site.

wakeup indicates whether a wakeup is sent with the message.
"1"b yes
"0"b no

always_add indicates whether the message is to be added even if a wakeup could not be sent.
"1"b yes
"0"b no

never_add tests whether a wakeup can be sent, without trying to add a message.

"1"b	yes
"0"b	no

acknowledge indicates whether an acknowledgement is requested when the message is read.

"1"b	yes
"0"b	no

mbz1, mbz2, mbz are not used and must be set to "0"b.

4. code is a standard status code. (Output) It may be one of the following:

error_table_$noentry	if the mailbox is not found.
error_table_$no_append	if the sending process has insufficient access to add a message.
error_table_$wakeup_denied	if the sending process has insufficient access to send a wakeup.
error_table_$messages_deferred	if the recipient process is deferring messages.
error_table_$messages_off	if the recipient is not logged in or the recipient process has not been initialized for receiving messages.
error_table_$no_info	if the sending process is not given any information because it has a lower AIM authorization than the recipient process.

This page intentionally left blank.

Name: set_lock_

The set_lock_ subroutine enables cooperating processes to coordinate their use of shared resources. Often, it is necessary to ensure that only one of the cooperating processes at a time executes a critical section of code with respect to a shared resource. For example, if the steps used to modify a shared data base leave it momentarily inconsistent, then while the data is being modified no other process should attempt to modify or examine the data.

A caller-supplied lock word is used for mutual exclusion of processes. This word should be declared as bit(36) aligned and should be set initially to "0"b indicating the unlocked state. When the program is about to enter a critical section of code, it calls the set_lock_$lock entry point. This entry point places a unique lock identifier for the process in the lock word if no other process currently has its lock identifier in the lock word. If the lock word already contains the lock identifier of some other process, the set_lock_$lock entry point waits for that process to unlock the lock word. Since only one process at a time can have its lock identifier in the lock word, that process is assured (subject to the conditions stated below) that it is the only process currently executing the critical section of code. If many critical sections share the same lock word, then only one process can be executing in any of them at a given time. Once the critical section has been completed, the program calls the set_lock_$unlock entry point to reset the lock to "0"b.

Successful use of this subroutine requires that all those processes executing critical sections of code obey the necessary conventions. These conventions are the following:

1. The set_lock_ subroutine is the only procedure that modifies the lock word with the exception of the procedure that initializes the lock word to "0"b before any call to the set_lock_ subroutine is made.

2. All processes issue calls to the set_lock_$lock entry point that result in the lock identifier appearing in the lock word before entering a critical section of code.

3. All processes issue a call to the set_lock_$unlock entry point that results in the lock word being set to 0 after completing execution of a critical section of code.

Entry: set_lock_$lock

This entry point attempts to place the lock identifier of the calling process in the given lock word. If the lock word contains "0"b, then the lock word is set to the lock identifier of the calling process. If the lock word contains a valid lock identifier of another existing process, then the set_lock_$lock entry point waits for this other process to unlock the lock word. If the other process does not unlock the lock word in a given period of time, the set_lock_$lock entry point returns with status. If the lock word contains a lock identifier not corresponding to an existing process, the lock word is overwritten with the lock identifier of the calling process and an indication that an overwriting has taken place is returned; the call is still successful, however.

Relocking an invalid lock implies either a coding error in the use of locks or that a process having a lock set was unexpectedly terminated. In either case, the data being modified can be in an inconsistent state. If the lock word already contains the lock identifier of the calling process, then the set_lock_$lock entry point does not modify the lock word, but returns an indication of the occurrence of this situation. The latter case may or may not indicate a programming error, depending on the programmer's conventions.

Usage

```
declare set_lock_$lock entry (bit(36) aligned, fixed bin, fixed bin);

call set_lock_$lock (lock_word, wait_time, code);
```

where:

1. lock_word — is the word to be locked. (Input)

2. wait_time — indicates the length of real time, in seconds, that the set_lock_$lock entry point should wait for a validly locked lock word to be unlocked before returning unsuccessfully. A value of -1 indicates no time limit. (Input)

3. code — is a standard status code. (Output) It may be one of the following:

   0 — indicates that the lock word was successfully locked because the lock word was previously unlocked

   error_table_$invalid_lock_reset — indicates that the lock word was successfully locked, but the lock word previously contained an invalid lock identifier that was overwritten

   error_table_$locked_by_this_process — indicates that the lock word already contained the lock identifier of the calling process and was not modified

   error_table_$lock_wait_time_exceeded — indicates that the lock word contained a valid lock identifier of another process and could not be locked in the given time limit

Entry: set_lock_$unlock

This entry point attempts to reset a given lock word to "0"b and is successful if the lock word contained the lock identifier of the calling process.

Usage

```
declare set_lock_$unlock entry (bit(36) aligned, fixed bin);

call set_lock_$unlock (lock_word, code);
```

where:

1. lock_word is the lock word to be reset. (Input)

2. code is a standard status code. (Output) It may be one of the following:

0	indicates successful unlocking
error_table_$lock_not_locked	indicates that the lock was not locked
error_table_$locked_by_other_process	indicates that the lock was not locked by this process and therefore was not unlocked

Name: term_

The term_ subroutine terminates the reference names of a segment and removes the segment from the caller's address space and the appropriate combined linkage segment. It also unsnaps any links in the combined linkage segments that contain references to the segment.

Usage

```
     declare term_ entry (char(*) aligned, char(*) aligned, fixed bin(35));

     call term_ (dir_path, entryname, code);
```

where:

1. dir_path is the pathname of the containing directory. (Input)

2. entryname is the entryname of the segment. (Input)

3. code is a standard status code. (Output)

Entry: term_$refname

The term_$refname entry point performs the same function as the term_ entry point given a reference name rather than a pathname.

Usage

```
     declare term_$refname entry (char(*) aligned, fixed bin(35));

     call term_$refname (ref_name, code);
```

where:

1. ref_name is the reference name of the segment. (Input)

2. code is a standard status code. (Output)

Entry: term_$seg_ptr

The term_$seg_ptr entry point terminates reference names for a segment and makes the segment unknown given a pointer to the segment.

Usage

```
declare term_$seg_ptr entry (ptr, fixed bin(35));

call term_$seg_ptr (seg_ptr, code);
```

where:

1. seg_ptr is a pointer to the segment. (Input)

2. code is a standard status code. (Output)

Entry: term_$unsnap

The term_$unsnap entry point unsnaps links to the segment but does not terminate any reference names or make the segment unknown.

Usage

```
declare term_$unsnap entry (ptr, fixed bin(35));

call term_$unsnap (seg_ptr, code);
```

where the seg_ptr and code arguments are the same as above.

Entry: term_$single_refname

The term_$single_refname entry point allows termination of a single reference name. The segment is not made unknown unless the specified reference name was the only reference name initiated for the segment.

Usage

```
     declare term_$single_refname (char(*) aligned, fixed bin(35));

     call term_$single_refname (ref_name, code);
```

where:

1. ref_name is a reference name of the segment. (Input)

2. code is a standard status code. (Output)

Note

The term_ subroutine performs the same operation as certain hcs_ entry points; however, the term_ entry points also unsnap links. The term_ entry points and corresponding hcs_ entry points are:

term_	hcs_$terminate_file
term_$seg_ptr	hcs_$terminate_seg
term_$single_refname	hcs_$terminate_name

Use of the term_ subroutine is preferred to the corresponding hcs_ entry points since the term_ subroutine unsnaps links in addition to terminating the segment.

Name: unique_bits_

The unique_bits_ subroutine returns a bit string that is useful as an identifier. It is obtained by reading the system clock, which returns the number of microseconds elapsed since January 1, 1901, 0000 hours Greenwich mean time. The bit string is, therefore, unique among all bit strings obtained in this manner in the history of this Multics installation.

Usage

```
     declare unique_bits_ entry returns (bit(70));

     bit_string = unique_bits_ ();
```

where bit_string is assigned the unique value. (Output)

Name: unique_chars_

The unique_chars_ subroutine provides a character string representation of a bit string. If the bit string is supplied by the unique_bits_ subroutine, this character string is unique among all character strings generated in this manner in the history of this Multics installation and is therefore useful as an identifier.

Usage

```
     declare unique_chars_ entry (bit(*)) returns (char(15));

     char_string = unique_chars_ (bits);
```

where:

1. char_string is a unique character string. (Output)

2. bits is a bit string of up to 70 bits. See "Notes" below. (Input)

Notes

If the bits argument is less than 70 bits in length, unique_chars_ pads it with 0´s on the right to produce a 70-bit string. If the bits argument equals 0, unique_chars_ calls unique_bits_ to obtain a unique bit string.

The first character in the character string produced is always an exclamation point to identify the string as a unique identifier. The remaining 14 characters that form the unique identifier are alphanumeric, excluding vowels.

Name: user_info_

The user_info_ subroutine allows the user to obtain information concerning his login session. All entry points that accept more than one argument count their arguments and only return values for the number of arguments given.

Entry: user_info_

This entry point returns the user's login name, project name, and account identifier.

Usage

```
declare user_info_ entry (char(*), char(*), char(*));

call user_info_ (person_id, project_id, acct);
```

where:

1. person_id is the user's name from the login line (maximum of 22 characters). (Output)

2. project_id is the user's project identifier (maximum of 9 characters). (Output)

3. acct is the user's account identifier (maximum of 32 characters). (Output)

Entry: user_info_$absentee_queue

This entry point returns the queue number of the absentee queue, for an absentee process. For an interactive process, the number returned is -1.

Usage

```
declare user_info_$absentee_queue entry (fixed bin);

call user_info_$absentee_queue (queue);
```

where queue is the number of the absentee queue. (Output)

Entry: user_info_$absin

This entry point returns the pathname of the absentee input segment for an absentee job. For an interactive user, the pathname is returned as blanks.

Usage

```
     declare user_info_$absin entry (char(*));

     call user_info_$absin (path);
```

where path is the pathname of the absentee input segment (maximum of 168 characters). (Output)

Entry: user_info_$absout

This entry point returns the pathname of the absentee output segment for an absentee job. For an interactive user, the pathname is returned as blanks.

Usage

```
     declare user_info_$absout entry (char(*));

     call user_info_$absout (path);
```

where path is the pathname of the absentee output segment (maximum of 168 characters). (Output)

Entry: user_info_$attributes

This entry point returns a character string containing the name of the user´s attributes, each separated by a comma and a space, and ending in a semicolon. Attributes control such things as the ways in which the user may log in, and the arguments that he is permitted to give when logging in. They are assigned by the project or system administrator. For more information on attributes, see both the Multics Project Administrators´ Manual and the Multics System Administrators´ Manual, Order Nos. AK51 and AK50 respectively.

Usage

```
declare user_info_$attributes entry (char(300));

call user_info_$attributes (attr);
```

where attr is the string containing the names of the user's attributes. (Output)

Entry: user_info_$homedir

The user_info_$homedir entry point returns the pathname of the user's initial working directory.

Usage

```
declare user_info_$homedir entry (char(*));

call user_info_$homedir (hdir);
```

where hdir is the pathname of the user's home directory (maximum of 64 characters). (Output)

Entry: user_info_$limits

This entry point returns the limit values established for the user by the project administrator and also returns the user's spending against these 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);
```

where:

1. mlim — is the dollar amount the user can spend in the month. (Output)

2. clim — is the dollar amount the user can spend (cutoff limit). (Output)

3. cdate is the cutoff date. (Output)

4. crf is the cutoff refresh code. (Output) This indicates what happens 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

5. shlim is an array that shows the dollar amount the user can spend per shift. (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 array of spending against shift limits in dollars. (Output)

Entry: user_info_$load_ctl_info

This entry point returns load control information for the user.

Usage

```
declare user_info_$load_ctl_info entry (char(*), fixed bin, fixed bin(71),
     fixed bin);

call user_info_$load_ctl_info (group, stby, preempt_time, weight);
```

where:

1. group is the name of the load control group. (Output)

2. stby indicates whether a user is a standby user (i.e., one who can be preempted). (Output)
   1 can be preempted
   0 cannot be preempted

3. preempt_time is the clock time after which the user will become standby. (Output)

4. weight is 10 times the user's weight. Weight is a measure of the load placed on the system by the user; most users have a weight of 1. (Output)

Entry: user_info_$login_data

This entry point 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 (person_id, project_id, acct, anon, stby,
     weight, time_login, login_word);
```

where:

1. person_id is the same as in the user_info_ entry point above. (Output)

2. project_id is the same as in the user_info_ entry point above. (Output)

3. acct is the same as in the user_info_ entry point above. (Output)

4. anon indicates whether a user is an anonymous user. (Output)
   1 is anonymous
   0 is not anonymous

5. stby indicates whether a user is a standby user (i.e., one who can be preempted). (Output)
   1 can be preempted
   0 cannot be preempted

6. weight is 10 times the user´s weight. See the user_info_$load_ctl_info entry point above. (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", depending on which command was used to log in. (Output)

Entry: user_info_$logout_data

This entry point returns information about how the user logs out.

Usage

```
     declare user_info_$logout_data entry (fixed bin(71), bit(36) aligned);

     call user_info_$logout_data (logout_channel, logout_pid);
```

where:

1. logout_channel is the event channel over which logouts are to be signalled. (Output)

2. logout_pid is the process identifier of the answering service. (Output)

Entry: user_info_$outer_module

This entry point returns the name of the user's outer module.

Usage

```
     declare user_info_$outer_module entry (char*));

     call user_info_$outer_module (om);
```

where om is the name of the user's outer module (maximum of 32 characters). The outer module is the initial I/O module attached to the user_i/o switch. (Output)

Entry: user_info_$responder

The user_info_$responder entry point returns the name of the user's login responder.

Usage

```
     declare user_info_$responder entry (char(*));

     call user_info_$responder (resp);
```

where resp is the name of the user's login responder (maximum of 64 characters). (Output)

Entry: user_info_$tty_data

This entry point returns information about the terminal on which the user logged in.

Usage

```
declare user_info_$tty_data entry (char(*), fixed bin, char(*));

call user_info_$tty_data (id_code, type, channel);
```

where:

1. id_code is the identifier code of the user's terminal (maximum of four characters). (Output)

2. type is the type of terminal. (Output) It can be:

   0 absentee process or network user
   1 device similar to IBM Model 1050
   2 device similar to IBM Model 2741 (with special modifications)
   3 device similar to Teletype Model 37
   4 device similar to GE TermiNet 300
   5 device similar to Adage, Inc. Advanced Remote Display Station (ARDS)
   6 device similar to IBM Model 2741 (standard)
   7 device similar to Teletype Models 33 or 35
   8 device similar to Teletype Model 38
   9 unused
   10 unused
   11 device similar to a Computer Devices Inc. (CDI) Model 1030 or Texas Instruments (TI) Model 725, or a device with an unrecognized answerback, or a device without an answerback (these devices are collectively termed "ASCII" devices)

3. channel is the channel identification (maximum of eight characters). (Output)

Entry: user_info_$usage_data

This entry point returns user usage data.

Usage

```
     declare   user_info_$usage_data   entry   (fixed bin,    fixed bin(71),
          fixed bin(71), fixed bin(71), fixed bin(71), fixed bin(71));

     call user_info_$usage_data (nproc,  old_cpu,  time_login,  time_create,
          old_mem, old_io_ops);
```

where:

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 same as in the user_info_$login_data entry point above. (Output)

4. time_create — is the same as in the user_info_$login_data entry point above. (Output)

5. old_mem — is the memory usage by previous processes in this login session. (Output)

6. old_io_ops — is the number of terminal I/O operations by previous processes in this login session. (Output)

Entry: user_info_$whoami

The user_info_$whoami entry point is the same as the user_info_ entry point. The name is a mnemonic device added for convenience.

Usage

```
     declare user_info_$whoami entry (char(*), char(*), char(*));

     call user_info_$whoami (person_id, project_id, acct);
```

where person_id, project_id, and acct are the same as in the user_info_ entry point above.

Name: vfile_status_

The vfile_status_ subroutine returns various items of information about a file supported by the vfile_ I/O module.

Usage

```
declare vfile_status_ entry (char(*), char(*), ptr, fixed bin(35));

call vfile_status_ (dir_name, entryname, info_ptr, code);
```

where:

1. dir_name is the pathname of the containing directory. (Input)

2. entryname is the entryname of the file of interest. If the entry is a link, the information returned pertains to the entry to which it points. (Input)

3. info_ptr is a pointer to the structure in which information is to be returned. (See "File Information" below.) (Input)

4. code is a storage system status code. (Output)

File Information

The info_ptr argument points to one of the following self-describing structures, as determined by the type of the file (see "type" below):

```
dcl  1 uns_info            based (info_ptr),  /* structure for
     2 info_version        fixed bin,            unstructured files */
     2 type                fixed bin,
     2 bytes               fixed bin(34),
     2 flags               aligned,,
       3 pad1              bit(2) unal,
       3 header_present    bit(1) unal,
       3 pad2              bit(33) unal,
     2 header_id           fixed bin(35);
```

where:

1. info_version identifies the version of the info structure; this must be set to 1 by the user. (Input)

2. type identifies the file type and the info structure returned:
   1 unstructured
   2 sequential
   3 blocked
   4 indexed

3. bytes gives the file's length, not including the header in bytes.

4. header_present — if set, indicates that an optional header is present.

5. header_id — contains the identification from the file's header, if present. Its meaning is user-defined.

```
     dcl  1  seq_info          based (info_ptr),  /* structure for
          2  info_version      fixed bin,            sequential files */
          2  type              fixed bin,
          2  records           fixed bin(34),
          2  flags             aligned,
             3  lock_status    bit(2) unal,
             3  pad            bit(34) unal,
          2  version           fixed bin;
```

where:

1. info_version — same as in the uns_info structure above.

2. type — same as in the uns_info structure above.

3. records — is the number of records in the file, including those of zero length.

4. lock_status — if zero, indicates that the file's lock is not set; otherwise the file is busy.
   "01"b busy in caller's process
   "10"b busy in another process
   "11"b busy in a defunct process

5. version — identifies the version number of the file and its creating program.

```
     dcl  1  blk_info          based (info_ptr),  /* structure for
          2  info_version      fixed bin,            blocked files */
          2  type              fixed bin,
          2  records           fixed bin(34),
          2  flags             aligned,
             3  lock_status    bit(2) unal,
             3  pad            bit(34) unal,
          2  version           fixed bin,
          2  action            fixed bin,
          2  max_rec_len       fixed bin(21);
```

where:

1. info_version — same as in the uns_info structure above.

2. type — same as in the uns_info structure above.

3. records — same as in the seq_info structure above.

4. lock_status — same as in the seq_info structure above.

5. version — same as in the seq_info structure above.

6. action — if nonzero, indicates an operation in progress on the file:
-1 write in progress
-2 rewrite in progress
-3 delete in progress
+1 truncation in progress

7. max_rec_len — is the maximum record length (in bytes) associated with the file.

```
dcl  1  indx_info           based (info_ptr),  /* structure for
        2  info_version     fixed bin,            indexed files */
        2  type             fixed bin,
        2  records          fixed bin(34),
        2  flags            aligned,
           3  lock_status   bit(2) unal,
           3  pad           bit(34) unal,
        2  version          aligned,
           3  file_version  fixed bin(17) unal,
           3  program_version fixed bin(17) unal,
        2  action           fixed bin,
        2  non_null_recs    fixed bin(34),
        2  record_bytes     fixed bin(34),
        2  free_blocks      fixed bin,
        2  index_height     fixed bin,
        2  nodes            fixed bin,
        2  key_bytes        fixed bin(34),
        2  change_count     fixed bin(35),
        2  num_keys         fixed bin(34),
        2  dup_keys         fixed bin(34),
        2  dup_key_bytes    fixed bin(34),
        2  reserved(1)      fixed bin;
```

where:

1. info_version — same as in the uns_info structure above.

2. type — same as in the uns_info structure above.

3. records — same as in the seq_info structure above.

4. lock_status — same as in the seq_info structure above.

5. file_version — identifies the version number of the file.

6. program_version — identifies the version number of vfile_ that created the file.

7. action — same as in the blk_info structure above.

8. non_null_recs — is a count, not including those of zero length, of the records in the file.

9. record_bytes — is the total length of all records in the file in bytes.

10. free_blocks — is the number of blocks in the file's free space list for records.

11. index_height — is the height of the index tree (equal to zero if file is empty).

12. nodes — is the number of single page nodes in the index.

13. key_bytes — is the total length of all keys in the file in bytes.

14. change_count — is the number of times the file has been modified.

15. num_keys — is the total number of index entries, each associating a key with a record.

16. dup_keys — is the number of index entries with nonunique keys, not including the first instance of each key.

17. dup_key_bytes — is the total length of all duplicate keys in the file, as defined above.

Notes

The user must provide the storage space required by the above structures. Normally, space should be allocated for the largest info structure that might be returned, namely, the one for indexed files.

See the description of the vfile_ I/O module for further details.

Name: virtual_cpu_time_

The virtual_cpu_time_ subroutine returns the CPU time used by the calling process since its creation; this value does not include the time spent handling page faults or system interrupts. It is therefore a measure of the CPU time within a process that is independent of other processes, current configuration, and overhead necessary to implement the virtual memory for the calling process.

Usage

```
     declare virtual_cpu_time_ entry returns (fixed bin(71));

     time = virtual_cpu_time_ ();
```

where time is the virtual CPU time in microseconds, used by the calling process. (Output)

SECTION III

SYSTEM INPUT/OUTPUT MODULES

The Multics input/output (I/O) system, described in detail in Section IV of the MPM Reference Guide, makes use of various I/O modules to perform input and output operations. Earlier versions of Multics used a different, but similar, I/O system, described in different terminology, which may still exist in parts of the system documentation. In particular, the older system used the term "i/o stream" instead of "I/O switch" and the terms "DIM'' and "IOSIM" instead of "I/O module." Also, documentation may describe attaching to a device even though the attachment may be to something other than a peripheral device, e.g., a file in the storage system.

Most programmers need not call the iox_ subroutine (described in Section II) directly. Even when an applications programmer wishes to direct I/O to/from various media (e.g., a terminal, a peripheral device, or a file in the storage system) it is not necessary to call iox_ directly. The I/O module descriptions, however, contain some information of general interest, e.g., the formats of attach descriptions.

The I/O modules described in this section and their functions are:

discard_	provides a sink for output operations
ntape_	supports I/O from/to magnetic tape files
rdisk_	supports I/O from/to removable disk packs
record_stream_	maps stream_calls into record calls or vice versa
syn_	attaches an I/O switch as a synonym for another switch
tape_ansi_	implements the processing of magentic tape files according to standards proposed by the American National Standards Institute (ANSI)
tape_ibm_	implements the processing of magnetic tape files according to standards established by IBM
tty_	supports I/O from/to terminal devices
vfile_	supports I/O from/to files in the storage system

Name: discard_

This I/O module provides a sink for output. It supports output operations, but the operations have no effect.

Entries in the module are not called directly by users; rather the module is accessed through the I/O system.

Attach Description

The attach description has the following form:

```
discard_
```

No options are allowed.

Opening

This module supports the following opening modes:

stream_output
sequential_output
keyed_sequential_output
direct_output

Control Operation

This module supports the control operation when the opening is for stream_output. All orders are accepted; but they have no effect.

Modes Operations

This module supports modes operation when the opening is for stream_output. It always returns a null string for the old modes.

Seek Key Operation

When the opening is for keyed_sequential_output or direct_output, the seek_key operation returns the code error_table_$no_record.

Name: ntape_

This I/O module supports I/O from/to files on magnetic tape.

Entry points in the module are not called directly by users; rather, the module is accessed through the I/O system. See "Multics Input/Output System" for a general description of the I/O system, and see "File Input/Output" for a discussion of files, both in Section IV of the MPM Reference Guide.

Attach Description

The attach description has the following form:

```
ntape_ reel_num -control_arg -optional_args-
```

where:

1. reel_num is the tape reel number. If the tape is 7-track, reel_num must contain ",7track". If the tape is 9-track, reel_num may contain ",9track" (if it contains neither, 9-track is assumed).

2. control_arg must be -raw to indicate that each physical record (block) on the tape represents one logical record.

3. optional_args may be one of the following arguments:

   -write means that the tape is to be mounted with a write ring. This argument must occur if the I/O switch is to be opened for output or input/output.

   -extend specifies extension of the file if it already exists on the tape.

Opening

The opening modes supported are sequential_input, sequential_output, and sequential_input_output. If an I/O switch attached via the ntape_ I/O module is to be opened for output or input_output, the -write argument must occur in the attach description.

Control Operation

This I/O module does not support the control operation.

Modes Operation

This I/O module does not support the modes operation.

Note

Using the -raw control argument, the relation between logical and physical record is as follows:

1. On input, the logical record contains m=4*ceil(n/36) bytes, where n is the number of data bits in the physical record. The first n bits of the input record are the data bits, the last (9+m-n) bits are 0´s.

2. On output, the physical record contains n=k*ceil((36*ceil(m/4))/k) data bits, where k+1 is the number of tracks on the tape, and m is the length of the logical record in characters. The first 9+m data bits of the physical record contain the bits of the logical record (i.e., the output buffer). The last (n-9+m) bits of the physical record are 0´s.

Name: rdisk_

The rdisk_ I/O module supports I/O from/to removable disk packs. Only direct modes are supported.

Entries in this module are not called directly by users; rather, the module is accessed through the I/O system. See the "Multics Input/Output System," for a general description of the I/O system, and "File Input/Output," for a discussion of files, both in Section IV of the MPM Reference Guide.

Attach Description

The attach description has the following form:

```
rdisk_ device_id pack_id -control_args-
```

where:

1. device_id is a character string identifying the model number of the required disk device. Currently, only the DSS191 is supported. The device_id, D191, is used for the DSS191.

2. pack_id is a character string identifying the disk pack to be mounted.

3. control_args may be chosen from the following:

   -write indicates that the disk pack is to be written. If omitted, the operator is instructed to mount the pack with the PROTECT button pressed so that writing is inhibited.

   -size *n* indicates that the value of *n* is to override the value of the buff_len parameter as a record size limit for the read_record operation. (See "Notes" below.)

   -priv indicates that the attachment is being made by a system process and that a disk drive reserved for system functions is to be assigned.

The attachment causes the specified disk pack to be mounted on a drive of the specified type.

Opening

The only opening modes supported are direct_input and direct_update. If an I/O switch attached through rdisk_ is to be opened for update, the -write control argument must occur in the attach description. This operation has no effect on the physical device.

Delete Record Operation

This operation is not supported.

Read Length Operation

This operation is not supported.

Read Record Operation

If the amount of data to be read does not terminate on a sector boundary, the excess portion of the last sector is discarded. A code of 0 is returned in this case. (See "Notes" below.)

Rewrite Record Operation

This operation is the only output operation supported. If the amount of data to be written does not terminate on a sector boundary, the remaining portion of the last sector is filled with binary zeros. A code of 0 is returned in this case. (See "Notes" below.)

Seek Key Operation

This operation returns a status code of 0 for any key that is a valid sector number. The record length returned is always 256 (current physical sector size in characters) for any valid key. The specified key must be a character string that could have been produced by editing through a PL/I picture of "(8)9". (See "Notes" below.)

Control Operation

The following orders are supported when the I/O switch is open, except for getbounds, which is supported while the switch is attached.

changepack — causes the current pack to be dismounted and another pack to be mounted in its place. The info_ptr should point to a variable length character string (maximum of 32 characters) containing the identifier of the pack to be mounted.

getbounds — causes the lowest and highest sector numbers accessible by the caller under the current modes to be returned. The info_ptr should point to a structure of the following form:

```
dcl 1 bounds,
      2 low       fixed bin(35),
      2 high      fixed bin(35);
```

setsize — causes the value of the record size override setting to be reset. The info_ptr should point to an aligned fixed binary(35) quantity containing the new override value.

Modes Operation

The modes operation is supported when the I/O switch is attached. The recognized modes are listed below. Each mode has a complement indicated by the circumflex character (^) that turns the mode off.

label, ^label — specifies that a system-defined number of sectors at the beginning of the pack are reserved for a pack label, and that a seek_key operation is to treat any key within this area as an invalid key. (The default is on.)

alttrk, ^alttrk — specifies that the pack has been formatted with the assignment of alternate tracks, so that a system-defined number of sectors at the end of the pack are reserved for an alternate track area. Therefore, a seek_key operation is to treat any key within that area as an invalid key. (The default is off.)

wrtcmp, ^wrtcmp — specifies that the write-and-compare instruction, rather than the write instruction, is used for the rewrite_record operation. This causes all data written to be read back and compared to the data as it was prior to being written. This mode should be used with discretion, since it doubles the data transfer time of every write. (The default is off.)

Write Record Operation

This operation is not supported.

Closing

The closing has no effect on the physical device.

Detaching

The detachment causes the disk pack to be dismounted.

Notes

This I/O module is a very elementary, physical-device-oriented I/O facility, providing the basic user-level interface to a disk device. All operations are performed through calls to various I/O interfacer (IOI) mechanisms and resource control package (RCP) entries. Certain conditions must be satisfied before a user process can make use of this facility:

1. The system must be configured with one or more disk drives available as I/O disks.

2. The user must have access to assign the disk drive with RCP and access to the IOI gates.

This I/O module allows the user to read or write a caller-specified number of characters to or from a disk pack, beginning at a caller-specified sector number. Currently, the DSS191 is the only device type supported.

The entire disk pack is treated as a keyed direct file, with keys interpreted literally as physical sector numbers. Hence, the only allowable keys are those that can be converted into fixed binary integers that fall within the range of valid sector numbers for the given disk device under the current modes, as returned by the getbounds control operation.

If an attempt is made to read or write beyond the end of the user-accessible area on disk, the code error_table_$device_end is returned. If a defective track is encountered or if any other unrecoverable data transmission error is encountered, the code error_table_$device_parity is returned.

The record length is specified through the buff_len parameter in the read_record operation, and through the rec_len parameter for the rewrite operation, unless overridden by a -size control argument in the attach description. (Since by definition the file consists of the entire pack, the write operation has no meaning in this I/O module.)

The following items must be considered when using this I/O module with language input/output:

1. Device Attachment and File Opening:

   a. PL/I: A file can be attached to a disk pack in PL/I by specifying the appropriate attach description in the title option of an open statement. The open statement should also specify the record and direct attributes plus either the input or update attribute, as is appropriate. After opening, the desired modes should be set, and the current sector bounds should be obtained, through direct calls to iox_$find_iocb, iox_$modes, and iox_$control. These iox_ subroutine entry points are described in Section II.

   b. FORTRAN: It is not possible to attach a file to a disk pack within FORTRAN. Here, the attachment must be made external to the FORTRAN program, e.g., through the io_call command (described in the MPM Commands) or through use of a PL/I subroutine. FORTRAN automatically opens the file with the appropriate attributes. Also, it is impossible to set modes or obtain sector bounds from within FORTRAN. This should be done through use of a PL/I subroutine prior to the first FORTRAN reference to the file.

2. Input:

   a. PL/I: The PL/I read statement with the into and key options is used to read data from a disk pack. The input record length (buff_len) is determined by the size of the variable specified in the into option. The set option should not be used. The key should be a character string containing the character representation of the desired sector number.

   b. FORTRAN: The unformatted, keyed version of the FORTRAN read statement is used. The key must be an integer, whose value is the desired sector number. In FORTRAN, buff_len has no relationship to input variable size. Hence, the -size control argument must be specified in the attach description if the disk pack is to be read through FORTRAN. The size should be set to the length of the longest expected record.

3. Output:

   a. PL/I: To perform output operations to a disk pack, the PL/I rewrite statement must be used with the from and key options specified. The size of the variable referenced in the from option determines the length of the record written to disk. The key should be a character string containing the character representation of the desired sector number.

   b. FORTRAN: The unformatted, keyed version of the FORTRAN write statement must be used to perform output operations to a disk pack. The size of the output record is determined by the amount of data specified in the write list. The key must be an integer whose value is the desired sector number.

Control Operations from Command Level

All control operations may be performed from the io_call command, as follows:

```
io_call control switch order_arg
```

where:

1. switch is the name of the I/O switch.

2. order_arg must be one of the following:

```
changepack newpack
setsize newsize
getbounds
```

where:

newpack is the name of the new pack to be mounted.

newsize is the new record size in words.

Name: record_stream_

This I/O module attaches an I/O switch to a target I/O switch so that record I/O operations on the attached switch are translated into stream I/O operations on the target switch, or so that stream I/O operations on the attached switch are translated into record I/O operations on the target switch. In this way a program that uses only record I/O may process unstructured files and do I/O from/to the terminal. Similarly a program that uses only stream I/O may process some structured files.

Entry points in this module are not called directly by users; rather the module is accessed through the I/O system.

Attach Description

The attach description has the following form:

record_stream_ -switchname- -control_args-

where:

1. switchname — is the name of the target I/O switch. It need not be attached when this attachment is made. If this argument is omitted, the -target control argument must be present.

2. control_args — are chosen from the following to control the transformation of records into a stream of bytes and vice versa, or to control the target attachment:

   -nnl — means that a record is transformed into a stream without appending a newline character.

   -length n — means that the stream of bytes is converted to a sequence of records each of which has length n.

   -target attach_descrip — specifies the attachment of a uniquely named target switch. This control argument must occur if and only if the switchname argument is omitted, and it must be the last control argument in the attach description, if present.

If neither the -nnl nor -length control arguments are given, lines are transformed into records after deleting trailing newlines and records are transformed into lines by appending newlines.

Opening

The attached I/O switch may be opened for stream_input, stream_output, sequential_input, or sequential_output. In addition to the description in "Multics Input/Output System" in Section IV of the MPM Reference Guide, the implications of the opening mode are as follows:

stream_input
: The target I/O switch must be open for sequential_input, open for sequential_input_output, or attached and closed. In the last case, it is opened for sequential_input. The sequence of records read from the target switch is transformed into a stream of bytes that are transmitted to the calling program by the get_line and get_chars operations. The read_record operation is used to read the records from the target switch.

stream_output
: The target I/O switch must be open for sequential_output, open for sequential_input_output, or attached and closed. In the last case, it is opened for sequential_output. The stream of bytes written to the attached switch by the put_chars operation is transformed into a sequence of records that are written to the target switch by use of the write_record operation.

sequential_input
: The target I/O switch must be open for stream_input, open for stream_input_output, or attached and closed. In the last case, it is opened for stream_input. The stream of bytes read from the target switch is transformed into a sequence of records that are transmitted to the calling program by read_record operations. If the attach description specifies the default line to record transformation, the get_line operation is used to read bytes from the target switch. If the attach description specifies the -length control argument, the get_chars operation is used to read bytes from the target switch.

sequential_output
: The target I/O switch must be open for stream_output, open for stream_input_output, or attached and closed. In the last case, it is opened for stream_output. The sequence of bytes written to the attached switch by the write_record operation is transformed into a stream of bytes that are written to the target switch by use of the put_chars operation.

Transformations

The transformation from record to stream form can be described in terms of taking records from a record switch and giving bytes to a stream switch, and similarly for stream to record (a record is a string of bytes). Which switch is the record switch and which the stream switch depends on the opening mode as explained previously under "Opening." The transformation is determined by the control arguments in the attach description. The details are as follows:

Record to stream

(default)
A record is taken from the record switch, a newline character is appended, and the resulting string is given to the stream switch.

-nnl
A record is taken from the record switch and given to the stream switch without modification.

Stream to record

(default)
A line (string of bytes ending with a newline character) is taken from the stream switch, the newline character is deleted, and the resulting string is given to the record switch.

-length *n*
To form a record, n bytes are taken from the stream switch and given to the record switch as one record.

Buffering

The I/O module may hold data in buffers between operations when the switch is opened for stream_output, stream_input, or sequential_input.

Close Operation

The I/O module closes the target switch if and only if the I/O module opened it.

Detach Operation

The I/O module detaches the target switch if and only if the I/O module attached it via the -target control argument.

Position Operation

Only positioning to the beginning of file or end of file and skipping forward are supported, except in the default sequential case, which also permits backward skipping. These operations are only supported to the extent the attachment of the target I/O switch supports them.

Control and Modes Operations

These are supported for open switches in the sense that they are passed along to the I/O module for the target switch.

Input/Output Status

In addition to the I/O status codes specified in the description of the iox_ subroutine for the various I/O operations, this I/O module returns codes returned by the target switch I/O module.

Examples

The following commands would permit sequential input operations from the user's terminal:

```
io_call attach sysin record_stream_ user_input
io_call open sysin sequential_input
```

Each record accessed through sysin corresponds to a line read through user_input, with its trailing newline character deleted.

Consider a PL/I statement of the form:

```
open file(x) title ("record_stream_ -target vfile_ seg") -opening_mode-;
```

The opening_mode option may be:

```
stream input
stream output
sequential input
sequential output
```

Sequential operations on file(x) generate stream operations on seg and vice versa, with lines transformed into records without trailing newlines or records transformed into lines by appending newlines, depending upon the mode of opening.

Consider the command:

```
io_call attach switchxx record_stream_ -target record_stream_ -length 100
     -target vfile_ seg
```

If switchxx is opened for stream_input, seg must be an existing unstructured file. The effect is equivalent to that of inserting a newline after every 100 characters of seg referenced by get_chars, get_line, or position operations through switchxx.

Alternately, switchxx may be opened for sequential_output. In this case, variable length records written through switchxx are given trailing newlines and restructured into 100-character records, which are then transmitted to the sequential file, seg.

Name: syn_

This I/O module may be used to attach an I/O switch, x, as a synonym for another switch, y. Thereafter, performing an operation other than attach or detach on x has the same effect as performing it on y. There is one exception: if the attach description specifies that an operation on y is to be inhibited, performing that operation on x results in an error code.

Entry points in the module are not called directly by users: rather the module is accessed through the I/O system. See "Multics Input/Output System" in Section IV of the MPM Reference Guide for a general description of the input/output system and a discussion of synonym attachments.

Attach Description

The attach description has the following form:

syn_ switchname -control_arg-

where:

1. switchname — is the name of the I/O switch, y, for which the attached switch, x, is to be a synonym.

2. control_arg — may be -inhibit names (or -inh names) to specify that the named I/O operations are to be inhibited. The name(s) must be chosen from the following list:

open
get_line
read_record
rewrite_record
read_length
seek_key
control
close
put_chars
write_record
delete_record
position
read_key
modes

Detach Operation

The detach operation detaches the switch x (the switch attached via syn_). It has no effect on the switch y for which x is a synonym.

Inhibited Operations

An inhibited operation returns the code error_table_$no_operation.

Name: tape_ansi_

The tape_ansi_ I/O module implements the processing of magnetic tape files according to Draft Proposed Revision X3L5/419T of the American National Standards Institute's ANSI X3.27-1969, "Magnetic Tape Labels and File Structure for Information Interchange." This document is referred to below as the DPSR (Draft Proposed Standard Revision). In addition, the I/O module provides a number of features that are extensions to, but outside of, the DPSR. Using these features may produce a nonstandard file, unsuitable for interchange purposes.

Entries in the module are not called directly by users; rather, the module is accessed through the I/O system. See "Multics Input/Output System" in Section IV of the MPM Reference Guide for a general description of the I/O system.

Definition of Terms

For the purpose of this document, the following terms have the meanings indicated. They represent a simplification and combination of the exact and complete set of definitions found in the DPSR.

record — related information treated as a unit of information.

block — a collection of characters written or read as a unit. A block may contain one or more complete records, or it may contain parts of one or more records. A part of a record is a record segment. A block does not contain multiple segments of the same record.

file — a collection of information consisting of records pertaining to a single subject. A file may be recorded on all or part of a volume, or on more than one volume.

volume — a reel of magnetic tape. A volume may contain one or more complete files, or it may contain sections of one or more files. A volume does not contain multiple sections of the same file.

file set — a collection of one or more related files, recorded consecutively on a volume set.

volume set — a collection of one or more volumes on which one and only one file set is recorded.

Attach Description

The attach description has the following form:

tape_ansi_ vn1 vn2 ... vnn -control_args-

where:

1. vni — is a volume specification. A maximum of 64 volumes may be specified. In the simplest (and typical) case, a volume specification is a volume name, which must be six characters or less in length. If a volume name is less than six characters and entirely numeric, it is padded on the left with 0's. If a volume name is less than six characters and not entirely numeric, it is padded on the right with blanks. Occasionally, keywords must be used with the volume name. For a discussion of volume name and keywords see "Volume Specification" below.

   vn1 vn2 ... vnn — comprise the volume sequence list. The volume sequence list may be divided into two parts. The first part, vn1 ... vni, consists of those volumes that are actually members of the volume set, listed in the order in which they became members. The entire volume set membership need not be specified in the attach description; however, the first (or only) volume set member must be specified, because its volume name is used to identify the file set. If the entire membership is specified, the sequence list may contain a second part, vni+1 ... vnn, consisting of potential members of the volume set, listed in the order in which they may become members. These volumes are known as volume set candidates. (See "Volume Switching" below.)

2. control_args — is a sequence of one or more attach control arguments. A control argument may appear only once.

   -create, -cr — specifies that a new file is to be created. (See "Creating a File" below.)

   -name XX, -nm XX — specifies the file identifier of the file where XX is from 1 to 17 characters. (See "Creating a File" below.)

   -number n, -nb n — specifies the file sequence number, the position of the file within the file set, where n is an integer in the range 1 ≤ n ≤ 9999. (See "Creating a File" below.)

   -replace XX, -rpl XX — specifies the file identifier of the file to be replaced, where XX must be from 1 to 17 characters. If no file with file identifier XX exists, an error is indicated. (See "Creating a File" below.)

   -format f, -fmt f — specifies the record format, where f is a format code. (See "Creating a File" below for a list of format codes.)

-record r, -rec r
specifies the record length in characters, where the value of r is dependent upon the choice of record format. (See "Creating a File" below.)

-block b, -bk b
specifies the block length in characters, where the value of b is dependent upon the value of r specified in the -record control argument. (See "Creating a File" below.)

-extend, -ext
specifies extension of an existing file. (See "Extending a File" below.)

-modify, -mod
specifies modification of an existing file. (See "Modifying a File" below.)

-generate, -gen
specifies generation of an existing file. (See "Generating a File" below.)

-mode XX, -md XX
specifies the encoding mode used to record the file data, where XX is the string ascii, ebcdic, or binary. The default is ascii. (See "Encoding Mode" below.)

-expires date, -exp date
specifies the expiration date of the file to be created or generated, where date must be of a form acceptable to the convert_date_to_binary_ subroutine. (See "File Expiration" below.)

-force, -fc
specifies that the expiration date of the file being overwritten is to be ignored. (See "File Expiration" below.)

-device n, -dv n
specifies the maximum number of tape drives that can be used during an attachment, where n is an integer in the range $1 \leq n \leq 63$. (See "Multiple Devices" below.)

-density n, -den n
specifies the density at which the file set is recorded, where n can be either 800 or 1600 bits per inch. (See "File Set Density" below.)

-retain XX, -ret XX
specifies retention of resources across attachments, where XX specifies the detach-time resource disposition. (See "Resource Disposition" below.)

-ring, -rg
specifies that the volume set be mounted with write rings. (See "Write Rings and Write Protection" below.)

The following sections define each control argument in the contexts in which it can be used. For a complete list of the attach control arguments, see "Attach Control Arguments" below.

Creating a File

When a file is created, an entirely new entity is added to the file set. There are two modes of creation: append and replace. In append mode, the new file is added to the file set immediately following the last (or only) file in the set. The process of appending does not alter the previous contents of the file set. In replace mode, the new file is added by replacing (overwriting) an existing file. The replacement process logically truncates the file set at the point of replacement, destroying all files (if any) that follow consecutively from that point.

The -create and -name XX control arguments are required to create a file, where XX is the file identifier. No two files in a file set can have the same file identifier. If the act of creation would cause a duplication, an error is indicated.

If no file having file identifier XX exists in the file set, the new file is appended to the file set; otherwise, the new file replaces the old file of the same name.

If the user wishes to explicitly specify creation by replacement, the particular file to be replaced must be identified. Associated with every file is a name (file identifier) and a number (file sequence number.) Either is sufficient to uniquely identify a particular file in the file set. The -number n and -replace XX control arguments, either separately or in conjunction, are used to specify the file to be replaced. If used together, they must both identify the same file; otherwise, an error is indicated.

When the -number n control argument is specified, if n is less than or equal to the sequence number of the last file in the file set, the created file replaces the file having sequence number n. If n is one greater than the sequence number of the last file in the file set, the created file is appended to the file set. If n is any other value, an error is indicated. When creating the first file of an entirely new file set, the -number 1 control argument must be explicitly specified. (See "Volume Initialization" below.)

The -format f , -record r and -block b control arguments are used to specify the internal structure of the file to be created. They are collectively known as structure attribute control arguments.

When the -format f control argument is used, f must be one of the following format codes, chosen according to the nature of the data to be recorded. (For a detailed description of the various record formats, see "Record Formats" below.)

fb for fixed-length records, blocked. Used when every record has the same length, not in excess of 8192 characters.

db for variable length records, blocked. Used when records are of varying lengths, the longest not in excess of 8188 characters.

sb for spanned records, blocked. Used when the record length is fixed and in excess of 8192 characters, or variable and in excess of 8188 characters. In either case, the record length cannot exceed 1,044,480 characters.

f for fixed-length records, unblocked.

d for variable-length records, unblocked.

s for spanned records, unblocked.

u for undefined records (records undefined in format). Each block is treated as a single record, and a block may contain a maximum of 8192 characters.

NOTE: THE USE OF UNDEFINED RECORDS IS A NONSTANDARD FEATURE.

Records recorded using U format may be irretrievably modified; therefore, the use of U format is strongly discouraged. (See "Block Padding" below.)

Unblocked means that each block contains only one record (f, d) or record segment (s). Blocked means that each block contains as many records (fb, db) or record segments (sb) as possible. The actual number of records/block is either fixed (fb), depending upon the block length and record length, or variable (db, sb), depending upon the block length, record length, and actual records. Because of their relative inefficiency, the use of unblocked formats is discouraged.

When the -record *r* control argument is used, the value of *r* is dependent upon the choice of record format. In the following list, amrl is the actual or maximum record length.

f = fb ¦ f:	*r* = amrl
f = db ¦ d:	$\text{amrl} + 4 \le r \le 8192$
f = sb ¦ s:	$\text{amrl} \le r \le 1044480$
f = u:	*r* is undefined (the -record control argument should not be used.)

When the -block *b* control argument is used, the value of *b* is dependent upon the value of *r*. When the block length is not constrained to a particular value, the largest possible block length should be used.

f = fb:	*b* must satisfy mod (*b*,*r*) = 0
f = f:	$b = r$
f = db:	$b \ge r$
f = d:	$b = r$.
f = sb ¦ s:	$18 \le b \le 8192$
f = u:	$\text{amrl} \le b \le 8192$

In every case, *b* must be an integer in the range $18 \le b \le 8192$.

NOTE: THE USE OF A BLOCK LENGTH IN EXCESS OF 2048 CHARACTERS IS A NONSTANDARD FEATURE.

Because the structure attribute control arguments are extremely interdependent, care must be taken to ensure that specified values are consistent.

Reading a File

The attach description needed to read a file is less complex than the description used to create it. When a file is created, the structure attributes specified in the attach description are recorded in the file's header and trailer labels. These labels, which precede and follow each file section, also contain the file name, sequence number, block count, etc. When a file is subsequently read, all this information is extracted from the labels. Therefore, the attach description need only identify the file to be read; no other control arguments are necessary.

The file can be identified using the -name XX control argument, the -number n control argument, or both in combination. If the -name XX is used, a file with the specified file identifier must exist in the file set; otherwise, an error is indicated. If the -number control argument is used, a file with the specified file sequence number must exist in the file set; otherwise, an error is indicated. If the -name XX and -number n control arguments are used together, they must both refer to the same file; otherwise, an error is indicated.

Output Operations on Existing Files

Three output operations can be performed on an already existing file: extension, modification, and generation. As their functions are significantly different, they are described separately below. They do, however, share a common characteristic. Like the replace mode of creation, an output operation on an existing file logically truncates the file set at the point of operation, destroying all files (if any) that follow consecutively from that point.

Extending a File

File extension is the process of adding records to a file without in any way altering the previous contents of the file.

Because all the information regarding structure, length, etc. can be obtained from the file labels, the attach description need only specify that an extend operation is to be performed on a particular file. The previous contents of the file remain unchanged; new data records are appended at the end of the file. If the file to be extended does not exist, an error is indicated.

The file to be extended is identified using the -name XX control argument, the -number n control argument, or both in combination. The same rules apply as for reading a file. (See "Reading a File" above.)

Recorded in the labels that bracket every file section is a version number, initially set to 0 when the file is created. The version number is used to differentiate between data that have been produced by repeated processing operations (such as extension). Every time a file is extended, the version number in its trailer labels is incremented by 1. When the version number reaches 99, the next increment resets it to 0.

The user may specify any or all of the structure attribute control arguments when extending a file. The specified control arguments are compared with their recorded counterparts; if a discrepancy is found, an error is indicated.

Modifying a File

It is occasionally necessary to replace the entire contents of a file, while retaining the structure of the file itself (as recorded in the header labels). This process is known as modification.

Because all necessary information can be obtained from the file labels, the attach description need only specify that a modify operation is to be performed on a particular file. If a file to be modified does not exist, an error is indicated. The entire contents of the file are replaced by the new data records. The version number in the trailer labels of a modified file is incremented by 1, as described above.

The file to be modified is identified using the -name XX control argument, the -number n control argument, or both in combination. The same rules apply as for reading a file. (See "Reading a File" above.)

If any or all of the structure attribute control arguments are specified, they must match their recorded counterparts; otherwise, an error is indicated.

Generating a File

Recorded in the labels that bracket every file section is a generation number, initially set to 0 when the file is created. The generation number is used to differentiate between different issues (generations) of a file, all of which have the same file identifier. The duplicate file identifier rule (see "Creating a File" above) precludes multiple generations of a file from existing simultaneously in the same file set.

The generation number is a higher order of differentiation than the version number, which is more correctly known as the generation version number. While the process of modification or extension does not change the generation number, the process of generation increments the generation number by 1, and resets the version number to 0. The generation number can only be incremented by rewriting the header labels, and it is in this respect that the processes of generation and modification differ.

Producing a new generation of a file is essentially the same as creating a new file in place of the old; however, the file identifier, sequence number, and structure attributes are carried over from the old generation to the new. The attach description need only specify that a generation operation is to be performed on a particular file. If the file to be generated does not exist, an error is indicated. An entirely new generation of the file is created, replacing (and destroying) the previous generation. The generation number is incremented by 1; the version number is reset to 0. When the generation number reaches 9999, the next increment resets it to 0.

The file to be generated is identified by the -name XX control argument, the -number n control argument, or both in combination. The same rules apply as for reading a file. (See "Reading a File" above.)

If any or all of the structure attribute control arguments are specified, they must match those recorded in the labels of the previous generation; otherwise, an error is indicated.

Encoding Mode

The tape_ansi_ I/O module makes provision for three data encoding modes: ASCII, EBCDIC, and binary. Because the DPSR requires that the data in each record be recorded using only ASCII characters, the default data encoding mode is ASCII. File labels are always recorded using the ASCII character set.

When a file is created, the -mode XX can be used to explicitly specify the encoding mode, where XX is the string ascii, ebcdic, or binary. The default is the string ascii.

NOTE: THE USE OF ENCODING MODES OTHER THAN ASCII IS A NONSTANDARD FEATURE.

If XX is the string ascii, the octal values of the characters to be recorded should be in the range 000 ≤ octal_value ≤ 177; characters in the range 200 to 377 are not invalid, but recording such characters is a nonstandard feature; characters in the range 400 to 777 cause an unrecoverable I/O error. If XX is the string ebcdic, the octal values of the characters to be recorded must be in the range 000 to 177. (See the ascii_to_ebcdic_ subroutine in the MPM Subsystem Writers' Guide for the specific ASCII to EBCDIC mapping used by the I/O module.) If XX is the string binary, any octal value can be recorded.

The tape_ansi_ I/O module records the data encoding mode in a portion of the file labels reserved for system-defined use. If the -mode XX control argument is specified when the file is subsequently extended, modified, or generated, the specified mode must match that recorded in the file labels; otherwise, an error is indicated. When the file is subsequently read, the encoding mode is extracted from the file labels, so the -mode XX control argument need not be specified.

File Expiration

Associated with every file is a file expiration date, recorded in the file labels. If a file consists of more than one file section, the same date is recorded in the labels of every section. A file is regarded as "expired" on a day whose date is later than or equal to the expiration date. Only when this condition is satisfied can the file (and by implication, the remainder of the file set) be overwritten. Extension, modification, generation, and the replace mode of creation are all considered to be overwrite operations.

The expiration date is recorded in Julian form; i.e., yyddd, where yy are the last two digits of the year, and ddd is the day of the year expressed as an integer in the range $1 \leq ddd \leq 366$. A special case of the Julian date form is the value "00000", which means always expired.

The expiration date is set only when a file is created or generated. Unless a specific date is provided, the default value "00000" is used. The -expires date control argument is used to specify an expiration date, where date must be of a form acceptable to the convert_date_to_binary_ subroutine (described in Section II). If the I/O module is invoked through the iox_$attach_ioname entry point or the iox_$attach_iocb entry point (described in Section II), date must be a contiguous string, with no embedded spaces; if invoked through the io_call command, date may be quoted and contain embedded spaces. Julian form, including "00000", is unacceptable. Because overwriting a file logically truncates the file set at the point of overwriting, the expiration date of a file must be earlier than or equal to the expiration date of the previous file (if any); otherwise, an error is indicated.

If an attempt is made to overwrite an unexpired file, the user is queried for explicit permission. (See "Queries" below). The -force control argument unconditionally grants permission to overwrite a file without querying the user, regardless of "unexpired" status.

Volume Specification

The volume name is a six-character identifier physically written on, or affixed to, the volume's reel or container. The volume identifier is a six-character identifier magnetically recorded in the first block of the volume, the VOL1 label. This implementation of the I/O module assumes the volume name and volume identifier to be identical. If this is not the case, volume identifiers must be used in place of volume names.

If a volume name begins with a hyphen (-), the -volume keyword must precede the volume name. Even if the volume name does not begin with a hyphen, it may still be preceded by the keyword. The volume specification has the following form:

```
-volume vni
```

If the user attempts to specify a volume name beginning with a hyphen without specifying the -volume keyword, an error is indicated.

The slot identifier is a six-character string used to identify a volume on a per-installation basis. This implementation of the I/O module assumes the volume name and slot identifier to be identical. If this is not the case, the operator must be provided with the slot identifier of the volume. The volume specification for such a volume must be in the following form:

```
   vni -comment XX
or
   -volume vni -comment XX
```

where the -comment XX keyword and text specify that a given message is to be displayed on the operator's console whenever volume vni is mounted. The message may relate to any subject, not only the slot identifier. XX may be from 1 to 64 characters. If the I/O module is invoked through the iox_$attach_iocb entry point or the iox_$attach_ioname entry point, XX must be a contiguous string, with no embedded spaces; if invoked through the io_call command, XX may be quoted and contain embedded spaces.

Volume Switching

The DPSR defines four types of file set configurations:

single-volume file	a single file residing on a single volume
multivolume file	a single file residing on multiple volumes
multifile volume	multiple files residing on a single volume
multifile multivolume	multiple files residing on multiple volumes

The tape_ansi_ I/O module maintains a volume sequence list on a per-file-set basis, for the life of a process. A minimal volume sequence list contains only one volume, the first (or only) volume set member. If the file set is a multivolume configuration, the sequence list may contain one or more of the additional volume set members, following the mandatory first volume. If the sequence list contains the entire volume set membership (which may be only one volume), it may then contain one or more volume set candidates. Volume set candidates can become volume set members only as the result of an output operation. When an output operation causes the amount of data in the file set to exceed the capacity of the current volume set membership, the first available volume set candidate becomes a volume set member.

When the first attachment to any file in a file set is made, the volume sequence list for the file set is initialized from the attach description. At detach time, the I/O module empirically determines that one or more volumes are volume set members, by virtue of having used them in the course of processing the attached file. The remaining volumes in the sequence list, if any, are considered to be candidates. In subsequent attachments to any file in the file set, the order of volumes specified in the attach description is compared with the sequence list. For those volumes that the I/O module knows to be volume set members, the orders must match; otherwise, an erro' is indicated. Those volumes in the sequence list that the I/O module considers to be candidates are replaced by attach description specifications, if the orders differ. If the attach description contains more volumes than the sequence list, the additional volumes are appended to the list. This implementation maintains and validates the volume set membership on a per-process basis, and maintains a list of volume set candidates that is alterable on a per-attach basis.

Once a volume sequence list exists, subsequent attachments to files in the file set do not require repeated specification of any but the first (or only) volume, which is used to identify the file set. If the I/O module detects physical end of tape in the course of an output operation, it prepares to switch to the next volume in the volume set. An attempt is made to obtain the volume name from the sequence list, either from the sublist of members, or the sublist of candidates. If the list of volume set members is exhausted, and the list of candidates is either empty or exhausted, the user is queried for permission to terminate processing. If the reply is negative, the I/O module queries for the volume name of the next volume, which becomes a volume set member and is appended to the volume sequence list. If a volume name is obtained by either method, it is recorded in a system-defined file label field at the end of the current volume, volume switching occurs, and processing of the file continues.

If the I/O module reaches end of file section (but not of file) in the course of an input operation, it first attempts to obtain the next volume name from the volume sequence list. No distinction is made between the member and candidate sublists, because a volume that ends with a file section must be followed by the volume that contains the next section. If the sequence list is exhausted, the file section's labels are examined for a volume name and, if one is found, it is appended to the sequence list. Should the file labels provide no name, the user is queried, as described above. If any of these three methods results in a volume name, volume switching occurs, and processing of the file continues. This method of searching allows a specified switching sequence to override a sequence recorded in the file labels.

If the volume set is demounted at detach time, all volume set candidates are purged from the volume sequence list.

Multiple Devices

If a volume set consists of more than one volume, the -device n control argument can be used to control device assignment, where n specifies the maximum number of tape drives that can be used during this attachment. n is an integer in the range $1 \leq n \leq 63$. Drives are assigned only on a demand basis, and in no case does the number actually assigned exceed the device limit of the process. The default for an initial attachment to a file in a file set is n equals 1; the default for a subsequent attachment to that (or any other) file in the file set is n equals the previous value of n.

File Set Density

Although the DPSR requires that file sets be recorded at 800 bpi (bits per inch), the I/O module makes provision for two densities: 800, and 1600 bpi. Every file in a file set must be recorded at the same density; otherwise, an error is indicated.

The -density n control argument is used to explicitly specify the file set density, where n specifies the density at which the file set is (to be) recorded. n may be either 800 or 1600 bpi.

NOTE: THE USE OF 1600 BPI IS A NONSTANDARD FEATURE.

The file set density can only be changed in a subsequent attachment if the volume set was demounted by the previous attach.

In the absence of a -density n control argument, the file set density is determined as follows:

open for input: n = density of VOL1 label
open for output, creating new file set: n = 800 bpi
open for output, old file set: n = density of VOL1 label

Opening

The opening modes supported are sequential_input and sequential_output. An I/O switch can be opened and closed any number of times in the course of a single attachment. Such a series of openings may be in either or both modes, in any valid order.

All openings during a single attachment are governed by the same attach description. The following control arguments, all of which pertain to output operations, are ignored when the switch is opened for sequential_input:

-create
-expires
-extend
-force
-generate
-modify
-replace

Resource Disposition

The tape_ansi I/O module utilizes two types of resources: devices (tape drives) and volumes. Once an I/O switch is attached, resources are assigned to the user's process on a demand basis. When the I/O switch is detached, the default resource disposition unassigns all devices and volumes.

If several attaches and detaches to a file set are made in a process, repeated assignment and unassignment of resources is undesirable. Although the processing time required to assign and unassign a device is small, all available devices can be assigned to other processes in the interval between one detach and the next attach. While volumes are not often "competed" for, mounting and dismounting is both time-consuming and expensive.

The -retain XX control argument is used to specify retention of resources across attachments, where XX specifies the detach-time resource disposition. If XX is the string all, all devices and volumes remain assigned to the process. If XX is the string none, all devices and volumes are unassigned. This is the default retention.

The I/O module provides a further means for specifying or changing the resource disposition subsequent to attachment. (See retain_all and retain_none under "Control Operations" below.)

Write Rings and Write Protection

Before a volume can be written on, a write ring (an actual plastic ring) must be manually inserted into the reel. This can only be done before the volume is mounted on a device. When a volume is needed, the I/O module sends the operator a mount message that specifies if the volume is to be mounted with or without a ring.

If the attach description contains any output control argument (-extend, -modify, -generate, or -create), volumes are mounted with rings; otherwise, they are mounted without rings. When a volume set mounted with rings is opened for sequential_input, hardware file protect is used to inhibit any spurious write operations. A volume set mounted without rings cannot be opened for sequential_output.

However, the following sequence of events is possible. An attach description contains none of the output control arguments, but does contain the -retain all control arguments. The volume set is mounted without rings. After one or more (or no) openings for sequential_input, the I/O switch is detached. The volume set remains mounted because of the -retain all control argument. Subsequently, an attach is made whose description contains an output control argument, which requires that the volume set be mounted with rings. However, as rings can only be inserted in unmounted volumes, the entire volume set must be demounted and then remounted.

This situation can be avoided by using the -ring control argument to specify that the volume set be mounted with write rings. If no output control argument is specified in conjunction with -ring, the I/O switch cannot be opened for sequential_output.

When a volume set is mounted with write rings and the I/O switch is opened for sequential_input, the hardware file protect feature is used to safeguard the file set.

Queries

Under certain exceptional circumstances, the I/O module queries the user for information needed for processing to continue or instructions on how to proceed.

Querying is performed by the command_query_ subroutine. The user may intercept one or more types of query by establishing a handler for the command_question condition, which is signalled by the command_query_ subroutine. Alternately, the answer command (described in the MPM Commands) can be used to intercept all queries. The use of a predetermined "yes" answer to any query causes those actions to be performed that attempt to complete an I/O operation without human intervention.

In the following list of queries, status_code refers to command_question_info.status_code. See "Handling Unusual Occurrences" in the MPM Reference Guide for information regarding the command_question condition and the command_question_info structure.

status_code = error_table_$file_aborted

This can occur only when the I/O switch is open for sequential_output. The I/O module is unable to correctly write file header labels, trailer labels, or tapemarks. This type of error invalidates the structure of the entire file set. Valid file set structure can only be restored by deleting the defective file or file section from the file set.

The user is queried for permission to delete the defective file or file section. If the response is "yes", the I/O module attempts deletion. The attempt may or may not succeed; the user is informed if the attempt fails. If the response is "no", no action is taken. The user will probably be unable to subsequently process the file, or append files to the file set; however, this choice permits retrieval of the defective file with another I/O module. In either case, the I/O switch is closed.

status_code = error_table_$unexpired_volume

This can occur only when the I/O switch is open for sequential_output. A volume must be either reinitialized or overwritten; however, the first file or file section on the volume is unexpired.

The user is queried for permission to initialize or overwrite the unexpired volume. If the response is "yes", the volume is initialized or overwritten and processing continues. If the response is "no", further processing cannot continue, and the I/O switch is closed.

status_code = error_table_$uninitialized_volume

This can occur only when the I/O switch is open for sequential_output. A volume requires reinitialization before it can be used to perform any I/O. The I/O module distinguishes among four causes by setting command_question_info.query_code as follows:

query_code = 1	the first block of the tape is unreadable. The tape is either defective, or recorded at an invalid density.
query_code = 2	the first block of the tape is not a valid ANSI VOL1 label. The tape is not formatted as an ANSI volume.

query_code = 3 the volume identifier recorded in the VOL1 label is incorrect. The volume identifier does not match the volume name.

query_code = 4 the density at which the volume is recorded is incorrect. The volume density does not match the specified density.

The user is queried for permission to reinitialize the volume, indicating the causative factor. If the response is "yes", the volume is reinitialized and processing continues. If the response is "no", further processing cannot continue, and the I/O switch is closed.

status_code = error_table_$unexpired_file

This can occur only when the I/O switch is open for sequential_output. A file that must be extended, modified, generated, or replaced is unexpired.

The user is queried for permission to overwrite the unexpired file. If the response is "yes", processing continues. If the response is "no", further processing cannot continue, and the I/O switch is closed.

status_code = error_table_$no_next_volume

This can occur when reading a multivolume file, or when writing a file and reaching physical end of tape. The I/O module is unable to determine the name of the next volume in the volume set.

The user is queried for permission to terminate processing. If the response is "yes", no further processing is possible. If the I/O switch is open for sequential_output, the I/O switch is closed. If the response is "no", the user is queried for the volume name of the next volume. (See status_code = 0 below.)

status_code = 0

This occurs only when the response to the above query is "no". The user is requested to supply the name of the next volume. The response must be a volume name six characters or less in length, optionally followed by a mount message. Even if the volume name begins with a hyphen, it must <u>not</u> be preceded by the -volume control argument. If a mount message is to be specified, the response takes the following form:

```
volume_name -comment XX
```

where XX is the mount message and need not be a contiguous string. See "Volume Specification" above. This is the only query that does not require a "yes" or "no" response. If a preset "yes" is supplied to all queries, this particular query never occurs.

Structure Attribute Defaults

When a file is created, the I/O module can supply a default value for any or all of the file structure attributes. The defaults used are as follows:

1. record format the default is *f* = db

2. block length the default is *b* = 2048

3. record length
   f = u: undefined
   f = fb ¦ f: *r* = block length
   f = db ¦ d: *r* = block length
   f = sb ¦ s: *r* = 1044480

An injudicious combination of explicit specifications and defaults can result in an invalid attribute set. For example, if the control argument -record 12000 is specified, applying the defaults produces the following:

```
     -format db  -block 2048 -record 12000
```

This attribute set is invalid because, in D format (See "Record Formats" below), the record length must be less than or equal to the block length.

Processing Interchange Files

The DPSR makes provision for recording record format, block length, and record length in specific fields of the HDR2 file label. In addition, the I/O module records the encoding mode in a portion of the HDR2 label reserved for system-defined use. Because the DPSR restricts the encoding mode to ASCII, there is no "standard" label field reserved for recording encoding mode. Therefore, if a foreign interchange file (a file not created by this I/O module) uses an encoding mode other than ASCII, the -mode XX control argument must be used to specify the mode.

File sets are almost always recorded with HDR2 file labels, with the exception of those created by "primitive" systems at implementation levels 1 or 2. (See the DPSR for a description of the facilities supported at different implementation levels.) It is therefore rarely necessary to explicitly specify record format, block length, or record length when interchange files are read, extended, modified, or generated. If, however, a file does lack HDR2 labels, explicit attribute specification is required; defaults apply only to file creation.

ASCII Subset

The DPSR suggests that the characters that comprise certain alphanumeric label fields be limited to a 56-character subset of full ASCII. Furthermore, it is suggested that these fields should not contain embedded blanks, nor should they consist entirely of blanks. In particular, the user need only consider file identifiers and volume names.

The 56-character subset includes:

uppercase letters:	ABCDEFGHIJKLMNOPQRSTUVWXYZ
digits:	0123456789
special characters:	<space> " % & ' ( ) * + , - . / : ; < = > ?

These characters were chosen from the center four columns of the code table specified in USA Standard Code for Information Interchange, ANSI X3.4-1968, except for position 5/15 (the underscore (_) character) and those positions where there is provision for alternate graphic representation.

The limitation to this subset is intended to provide maximum interchangeability and consistent printing, especially for international interchange.

Overriding Structure Attributes

Normally, the -format *f*, -block *b*, and -record *r* control arguments are not included in the attach description of an I/O switch that is opened for sequential_input; the structure attributes are extracted from the file labels. However, the I/O module permits the recorded structure attributes to be overridden by explicitly specified attach description control arguments. Because the apparent structure and characteristics of the file can be drastically altered, great care must be taken to ensure that attribute overrides do not produce unexpected and unwanted results.

If a file has the following recorded attributes:

```
-format fb -block 800 -record 80
```

an explicit specification of the -format f and -record 800 control arguments causes each block of ten 80-character records to be treated as a single 800-character record.

If a file has the following recorded attributes:

```
-format fb -block 800 -record 80
```

an explicit specification of the -format f, -block 80, and -record 80 control arguments causes the last 720 characters of every block to be discarded. No error is indicated, because every block of the file contains at least one 80-character record.

Record Formats

ANSI files are structured in one of three record formats: F, D, or S. In addition, the I/O module provides for a fourth format, U. When a file is created, its record format should be chosen in accordance with the nature of the data to be recorded. For example, data consisting of 80-character card images is most economically recorded in F format, fixed-length records. Data consisting of variable length text lines, such as PL/I source code produced by a text editor, is best recorded in D format, variable-length records. Data of arbitrary length (which could exceed the maximum block size) must be recorded in S format, spanned records, so that a lengthy datum can span several blocks.

F, D, and S format files are either blocked or unblocked, blocked being the normal case. Each block of an unblocked file contains just one record, whereas each block of a blocked file can contain several records. Blocking can provide a significant savings of processing time, because several records are accessed with a single physical tape movement. Furthermore, as blocks are separated by distances of blank tape, blocking reduces the amount of tape needed to contain a file.

F FORMAT

In F format, records are of fixed (and equal) length, and files have an integral number (n) of records per block. If the file is unblocked, *n* is equal to 1 and the record length (*r*) is equal to the block length (*b*). If the file is blocked, *n* is greater than 1 and *b* is equal to (*r* * *n*). *n* is known as the blocking factor.

For example, if *r* is equal to 800 and *b* is equal to 800, then the file is unblocked and each block contains just one record.

```
         +------+ +------+ +------+ +------+ +------+ +------+
data     | 800  | | 800  | | 800  | | 800  | | 800  | | 800  |
         +------+ +------+ +------+ +------+ +------+ +------+

         +------+ +------+ +------+ +------+ +------+ +------+
block    | 800  | | 800  | | 800  | | 800  | | 800  | | 800  |
         +------+ +------+ +------+ +------+ +------+ +------+
```

If *r* is equal to 800 and *b* is equal to 2400, then the file is blocked, the blocking factor is 3, and each block contains three records.

```
         +------+  +------+  +------+  +------+  +------+  +------+
data     | 800  |  | 800  |  | 800  |  | 800  |  | 800  |  | 800  |
         +------+  +------+  +------+  +------+  +------+  +------+

         +--------+--------+--------+   +--------+--------+--------+
block    |  800   |  800   |  800   |   |  800   |  800   |  800   |
         +--------+--------+--------+   +--------+--------+--------+
```

The ANSI standard for F format records permits recording a short block only when the last block of a blocked file contains fewer than *n* records and there are no more records to be written when the file is closed.

There are two special cases in which a datum is padded out to length *r*. The first case is that of iobl (the iox_$write_record I/O buffer length; i.e., the number of characters to be written) equals 0: a record of *r* blanks is written. When such a record is subsequently read, it is interpreted as a record of *r* blanks, and not as a zero-length record. The second case is that of 0 < iobl < *r*: the record is padded on the right with blanks to length *r*, and the padded record written. When such a record is read, the original characters plus the padding are returned. The case of iobl is greater than *r* is in error.

NOTE: THE ANSI STANDARD PROHIBITS RECORDING A FIXED-LENGTH RECORD THAT CONSISTS ENTIRELY OF CIRCUMFLEX (^) CHARACTERS.

D FORMAT

In D format, records and therefore blocks may vary in length. Each record is preceded by a four-character record control word (RCW) that contains the total record length (the length of the data plus the length of the RCW itself).

D format files have an integral number (n) of records per block. If blocked, *r* is less than or equal to *b*. For blocked records, the number of records per block varies indirectly with the size of the records.

If *r* equals *b* equals 804 and the file is unblocked, records of up to 800 characters can be written, and each block contains one record.

```
          +--------+ +------+ +---------+ +---------------+
data      |  375   | | 280  | |  610    | |      800      |
          +--------+ +------+ +---------+ +---------------+

          +--------+ +------+ +---------+ +---------------+
          |3|      | |2|    | |6|       | |8|             |
block     |7| 375  | |8|280| |1| 610   | |0|     800     |
          |9|      | |4|    | |4|       | |4|             |
          +--------+ +------+ +---------+ +---------------+
```

If r equals 804, b is greater than or equal to 804, and the file is blocked, records of up to 800 characters can be written.

```
          +--------+ +------+ +---------+ +---------------+
data      |  375   | | 280  | |  610    | |      800      |
          +--------+ +------+ +---------+ +---------------+

          +-----------------+ +---------+ +---------------+
          |3|        |2|    | |6|       | |8|             |
block     |7|  375   |8|280| |1| 610   | |0|     800     |
          |9|        |4|    | |4|       | |4|             |
          +-----------------+ +---------+ +---------------+
```

Each block can contain a maximum of 201 zero-length records (a record written as a four-character RCW containing 0004).

S FORMAT

In S format, a single record is formatted as one or more record segments. A record segment contains either a complete record, the initial portion of a record, a medial portion of a record, or the final portion of a record. No two segments of the same record can be contained in the same block, but a block may contain the segments of several different records. The maximum record length is limited only by the maximum size of a storage system segment, currently 1,044,480 characters.

S format files have an integral number of record segments per block. If the file is unblocked, each block contains only one record segment; if blocked, the number of record segments per block is variable. In either case, r and b are independent of one another.

Each record segment begins with a five-character segment control word (SCW). The SCW contains a four-character record segment length, which includes the length of the SCW itself. The SCW also contains a one-character record segment code, which indicates if the segment contains a complete record, or an initial, medial, or final portion. In the examples below, r equals 1000 and b equals 800.

```
             +-------+  +------------+  +-----------------------------+
   data      ¦ 200   ¦  ¦    400     ¦  ¦            1000             ¦
             +-------+  +------------+  +-----------------------------+

             +-------+  +------------+  +-----------------+  +--------+
             ¦2¦     ¦  ¦4¦          ¦  ¦8¦               ¦  ¦2¦      ¦
   block     ¦0¦ 200 ¦  ¦0¦   400    ¦  ¦0¦     795       ¦  ¦1¦ 205  ¦
             ¦5¦     ¦  ¦5¦          ¦  ¦0¦               ¦  ¦0¦      ¦
             +-------+  +------------+  +-----------------+  +--------+


             +-------+  +------------+  +-----------------------------+
   data      ¦ 200   ¦  ¦    400     ¦  ¦            1000             ¦
             +-------+  +------------+  +-----------------------------+

             +-------+  +------------+  +-----+  +----------+  +-----+
             ¦2¦     ¦  ¦4¦          ¦  ¦1¦   ¦  ¦8¦        ¦  ¦2¦   ¦
   record    ¦0¦ 200 ¦  ¦0¦   400    ¦  ¦9¦185¦  ¦0¦  795   ¦  ¦5¦ 20¦
   segment   ¦5¦     ¦  ¦5¦          ¦  ¦0¦   ¦  ¦0¦        ¦  ¦ ¦   ¦
             +-------+  +------------+  +-----+  +----------+  +-----+

             +--------------------------------+  +----------+  +-----+
             ¦2¦       ¦4¦           ¦1¦      ¦  ¦8¦        ¦  ¦2¦   ¦
   block     ¦0¦  200  ¦0¦    400    ¦9¦185   ¦  ¦0¦  795   ¦  ¦5¦ 20¦
             ¦5¦       ¦5¦           ¦0¦      ¦  ¦0¦        ¦  ¦ ¦   ¦
             +--------------------------------+  +----------+  +-----+
```

U FORMAT

U format files contain records that do not conform to either F, D, or S format. A U format file is always unblocked. The record length is undefined, and b is greater than or equal to iobl. Blocks may vary in length.

NOTE: THE USE OF U FORMAT IS A NONSTANDARD FEATURE

The ANSI block padding convention permits a block (in any format) to be padded out to any length with circumflex characters (^), according to the requirements of the system that produces the file. These characters are ignored on input. (See "Block Padding" below.) In U format, block padding can lead to an ambiguity; i.e., are trailing circumflexes indeed pad characters, or are they actually valid data within the nonpadded portion of the block. The DPSR suggests that a U format block be treated as a single record. In conformance with this suggestion, the I/O module considers trailing circumflexes to be valid data.

The special case of writing a record where iobl is less than 20 characters produces a block padded to length 20 with circumflex characters.

```
          +-------+ +---------------+ +-----+ +------------------+
data      |  60   | |      127      | | 16  | |       156        |
          +-------+ +---------------+ +-----+ +------------------+

          +-------+ +---------------+ +-----+ +------------------+
block     |  60   | |      128      | | 20  | |       156        |
          +-------+ +---------------+ +-----+ +------------------+
```

Record Format Comparison

At first glance, it might appear as if S format were the format of choice, simply because it has the fewest restrictions and the greatest flexibility. Although the latter is certainly true, the former is by no means a valid inference. Increased flexibility is almost invariably accompanied by decreased processing efficiency.

F format requires the least processing time, and should be used if the records are fixed-length. If F format is used with nonfixed-length records the record padding rules apply, so the user must ensure that recorded data is not irretrievably (and perhaps undetectably) modified.

D format, with explicit inclusion of record length in the RCW, is perhaps the "safest" format to use: there are no special padding cases, and the RCW provides an additional validity check. The D format processing overhead is small.

S format permits almost any datum to be recorded, irrespective of length, and further has the "safety" advantage of D format because each segment includes an SCW. While S format records provide maximum flexibility, their use entails considerably more processing time than the use of F or D format.

Block Padding

The DPSR makes provision for extending the recorded length of a block beyond the end of the last (or only) record whenever such padding is deemed necessary or advisable. Padding characters are not considered when computing an RCW or SCW length. Because Multics is implemented on a word-oriented computer, the number of characters in a block must be evenly divisible by four. The I/O module automatically pads every block to the correct length, using from 1 to 3 circumflex characters. In addition, the DPSR does not permit recording a block of fewer than 18 characters. To conform with this requirement, the I/O module pads any block containing fewer than 20 characters out to length 20.

As long as F, D, or S format is used, the presence or absence of block padding characters in a particular block is user-transparent. If U format is used, it is the responsibility of the user to detect and ignore any pad characters that may be generated.

Volume Initialization

The DPSR requires that all volumes be initialized with a VOL1 label and dummy file before they are used for output. The I/O module provides a semiautomatic volume initialization mechanism that performs this operation as an integral part of the output function. The rules that govern permission to initialize a volume are complex, and permission to initialize under most circumstances is specifically denied (by the DPSR) to the application program. The I/O module's mechanism strikes a balance between outright denial and absolute ease. (See "Queries" above.)

It should be noted that a newly initialized volume contains a dummy file. Thus, if a file is created on a newly initialized volume without an explicit specification of the -number 1 control argument, the file is appended to the file set, resulting in a file sequence number of 2, and not 1 as might be expected.

Buffer Offset (Block Prefix)

The DPSR provides for each block of a file being prefixed by from 1 to 99 characters of prefix information, known as the buffer offset. The buffer offset length is recorded in the HDR2 label. If an input file has block prefixes, and the block length is explicitly specified, it must be incremented by the buffer offset length. This calculation should made after the block length has been determined using the normal block-record relationship rules.

The I/O module ignores (skips) buffer offsets on input, and does not provide for writing buffer offsets on output, except when extending or modifying an interchange file with a nonzero buffer offset. In this case, each block written is prefixed with an appropriate number of blanks.

Conformance to Standard

The I/O module conforms to the ANSI standard for level 4 implementations with the following five exceptions:

1. Volume Initialization--The I/O module has a permission-granting mechanism that can be controlled by the application program.

2. Volume and File Accessibility--On input, the I/O module always grants permission to access. On output, the access control fields in the VOL1 and HDR1 labels are always recorded as blank (" ").

3. Overwriting Unexpired Files--The I/O module has a permission-granting mechanism that can be controlled by the application program.

4. User Label Processing--The I/O module ignores user labels on input, and does not provide for writing user labels on output.

5. Buffer Offset Processing--The I/O module ignores buffer offsets on input, and does not provide for writing buffer offsets on output (except as stated above).

Label Processing

VOL1

The label is processed on input and output. The owner-identifier field, character positions (CP) 38 to 51 is recorded using up to 14 characters of the user's Person_id, all lowercase letters translated to uppercase.

UVLa

These labels are not written on output, and ignored on input.

HDR1/EOF1/EOV1

The labels are processed on input and output. The system-code field, CP 61 to 73, is recorded as "MULTICS ANSI ".

HDR2/EOF2/EOV2

The labels are processed on input and output. The reserved-for-system-use field, CP 16 to 50, is recorded as follows:

```
CP 16 to 21 - volume name of next volume (EOV2 only)
CP 22       -  blocking attribute (all)
               "0" = unblocked; "1" = blocked
CP 23       -  data encoding mode (all)
               "1" = ASCII, 9 mode
               "2" = EBCDIC, 9 mode
               "3" = binary
```

HDR3/EOF3/EOV3 - HDR9/EOF9/EOV9

These labels are not written on output and are ignored on input.

UHLa/UTLa

These labels are not written on output and are ignored on input.

Error Processing

If an error occurs while reading, the I/O module makes 25 attempts to backspace and reread. If an error occurs while writing, the I/O module makes 10 attempts to backspace, erase, and rewrite. Should an unrecoverable error occur while reading or writing the, I/O module "locks" the file so that no further I/O is possible. If an unrecoverable error occurs while writing file labels or tapemarks, the user is queried about preserving the defective file versus file set consistency. (See "Queries" above.) If an unrecoverable error occurs during certain phases of volume switching or label reading, the I/O switch may be closed. The overriding concern of the error recovery strategy is:

1. to maintain a consistent file set structure
2. to ensure the validity of data read or written

Close Operation

The I/O switch must be open.

Control Operation

The I/O module supports eight control operations.

hardware_status
status
volume_status
file_status
feov
close_rewind
retain_all
retain_none

In the descriptions below, info_ptr is the information pointer specified in an iox_$control entry point call.

hardware_status OPERATION

This operation returns the 72-bit IOM status string generated by the last tape I/O operation. The I/O switch must be open. The substr argument (IOM_bits, 3, 10) contains the major and minor status codes generated by the tape subsystem itself. (See MTS500 Magnetic Tape Subsystem, Order No. DB28, for an explanation of major and minor status.) The variable to which info_ptr points is declared as follows:

```
declare  IOM_bits bit(72) aligned;
```

status OPERATION

This operation returns a structure that contains an array of status codes, providing an interpretation of the IOM status string generated by the last tape I/O operation. These codes may be used in calls to the com_err_ subroutine, or may be converted to printable strings by calling the convert_status_code_ subroutine. (See the description of the com_err_ subroutine in this manual. and the description of the convert_status_code_ subroutine in the MPM Subsystem Writers' Guide.) The I/O switch must be open. The structure to which info_ptr points, device_status.incl.pl1, is declared as follows:

```
dcl  dstat_ptr         pointer;
dcl 1 device_status    based (dstat_ptr),
    2 IOM_bits         bit(72) aligned,    /* IOM status */
    2 n_minor          fixed bin,          /* number of minor codes */
    2 major            fixed bin(35),      /* major status code */
    2 minor            (10) fixed bin(35); /* minor status codes */
```

volume_status OPERATION

This operation returns a structure that contains the status of the current volume. If the I/O switch is open, the current volume is the volume on which the file section currently being processed resides. If the switch has never been opened, the current volume is the first (or only) volume in the volume set. If the switch was opened, but is now closed, the current volume is that on which the last file section processed resides. If the switch was closed by the I/O module as the result of an error while writing file header labels, trailer labels, or tapemarks, the current volume is the last (or only) volume in the volume set. The structure to which info_ptr points, tape_volume_status.incl.pl1, is declared as follows:

```
dcl  tvstat_ptr           pointer;
dcl 1 tape_volume_status  based (tvstat_ptr),
    2 volume_name         char(6),      /* volume name */
    2 volume_id           char(6),      /* from VOL1 label */
    2 volume_seq          fixed bin,    /* order in volume set */
    2 tape_drive          char(8),      /* tape drive name */
                                        /* "" if not mounted */
    2 read_errors         fixed bin,    /* read error count */
    2 write_errors        fixed bin;    /* write error count */
```

In the current implementation of the I/O module, read_errors and write_errors are always zero. Eventually, the resource control package (RCP) will supply these values.

file_status OPERATION

This operation returns a structure that contains the current status of the file specified in the attach description. If the I/O switch has never been opened, no information can be returned; this situation is indicated by file_status.state = 0. If the switch was opened, but is now closed, the current status of the file is its status just prior to closing. If the switch was closed by the I/O module as the result of an error while writing file header labels, trailer labels, or tapemarks, the entire file may have been deleted. In this case, the structure contains the current status of the previous file in the file set, if any. The spructure to which info_ptr points, tape_file_status.incl.pl1, is declared as follows:

```
dcl  tfstat_ptr            pointer;
dcl 1 tape_file_status     based (tfstat_ptr),
     2 state               fixed bin,       /* 0 - no information */
                                            /* 1 - not open */
                                            /* 2 - open, no events */
                                            /* 3 - open, event lock */
     2 event_code          fixed bin(35),   /* error_table_ code if
                                               state = 3 */
     2 file_id             char(17),        /* file identifier */
     2 file_seq            fixed bin,       /* order in file set */
     2 cur_section         fixed bin,       /* current or last
                                               section processed */
     2 cur_volume          char(6),         /* volume name of volume
                                               on which cur_section
                                               resides */
     2 generation          fixed bin,       /* generation number */
     2 version             fixed bin,       /* version of generation */
     2 creation            char(5),         /* Julian creation date */
     2 expiration          char(5),         /* Julian expiration date */
     2 format_code         fixed bin,       /* 1 - U format */
                                            /* 2 - F format */
                                            /* 3 - D format */
                                            /* 4 - S format */
     2 blklen              fixed bin,       /* block length */
     2 reclen              fixed bin(21),   /* record length */
     2 blocked             bit(1),          /* "0"b - no ¦ "1"b - yes */
     2 mode                fixed bin,       /* 1 - ASCII */
                                            /* 2 - EBCDIC */
                                            /* 3 - binary */
     2 cur_blkcnt          fixed bin(35);   /* current block count */
```

The "event" referenced in tape_file_status.state, above, is defined as an error or circumstance that prevents continued processing of a file. For example, parity alert while reading, reached end of information, no next volume available, etc.

feov OPERATION

This operation forces the end of a volume when writing a file. The switch must be open for sequential output. The operation is equivalent to detection of the end of tape reflective strip. The info_ptr should be a null pointer.

close_rewind OPERATION

This operation specifies that the current volume is to be rewound when the I/O switch is next closed. The info_ptr should be a null pointer. The switch need not be open when the operation is issued. The operation effects only one close; subsequent closings require additional control calls.

retain_all OPERATION

This operation causes all devices and volumes to remain assigned at detach time. The I/O switch need not be open. The info_ptr should be a null pointer.

retain_none OPERATION

This operation causes all devices and volumes to be unassigned at detach time. The I/O switch need not be open. The info_ptr should be a null pointer.

Detach Operation

The I/O switch must be closed. If the I/O module determines that the membership of the volume set might have changed, the volume set members are listed before the set is demounted; volumes not listed are available for incorporation into other volume sets.

Modes Operation

This I/O module does not support the modes operation.

Position Operation

The I/O switch must be open for sequential_input. The I/O module does not support skipping backwards. In the course of a position operation, events or errors may occur that invoke the query mechanism. (See "Queries" above.) An unrecoverable error locks the file, and a severe error causes the I/O module to close the I/O switch.

Read Length Operation

The I/O switch must be open for sequential_input. In the course of a read_length operation, events or errors may occur that invoke the query mechanism. (See "Queries" above.) An unrecoverable error locks the file, and a severe error causes the I/O module to close the I/O switch.

Read Record Operation

The I/O switch must be open for sequential_input.

Write Record Operation

The I/O switch must be open for sequential_output.

Examples

In the following examples, it must be emphasized that an attach description describes a potential operation, and in and of itself does nothing to the file. Depending upon the sequence of openings in various modes, one attach description can perform diverse functions.

```
tape_ansi_ 042381 -nm ARD21 -cr -fmt sb -ret all
```

A file named ARD21 is to be appended to the file set whose first volume is 042381. If a file named ARD21 already exists in the file set, openings for sequential_input access that file, and openings for sequential_output create new files replacing the old. If no file named ARD21 already exists in the file set, openings for sequential_input prior to the first opening for sequential_output fail. The first opening for sequential_output creates the file by appending it to the end of the file set. Subsequent openings for sequential_input will access the newly created file, and subsequent openings for sequential_output replace it. Spanned records are specified; the block length defaults to 2048, the record length to 1044480, and the encoding mode to ASCII. The density defaults to 800 bpi, and the maximum number of devices defaults to 1. The volume set and devices are retained after detachment.

```
tape_ansi_ 042381 -nm fargo.pl1 -nb 2 -cr -force -fmt fb -bk 800 -rec 80
```

A file named fargo.pl1 is created at position 2 in the file set. If a file named fargo.pl1 already exists at position 2, openings for sequential_input prior to the first opening for sequential_output access that file. The first opening for sequential_output creates a new file, and subsequent openings for sequential_input access the new file. If no file named fargo.pl1 exists at position 2, openings for sequential_input prior to the first opening for sequential_output fail. If a file exists at position 2, it is replaced irrespective of its expiration date.

```
tape_ansi_ 042381 -nm zbx -rpl zbx -cr -md binary -bk 6000 -exp 2weeks
```

A file named zbx is to be created, replacing a file of the same name. Openings for sequential_input prior to the first opening for sequential_output will access the old file. Each opening for sequential_output will create a new file, and each subsequent opening for sequential_input will access the most recently created file. The specified encoding mode is binary. The record format defaults to D, blocked, and the record length defaults to 6000 because the block length is specified as 6000. The file is protected from overwriting for a period of two weeks, so each opening for sequential_output subsequent to the initial opening for sequential_output causes the user to be queried for permission to overwrite.

```
tape_ansi_ 042381 -nb 14 -gen -dv 3 -expires 12/31/77
```

A new generation of the file at position 14 in the file set is to be created, replacing the old generation. If the old generation is not expired, the user is queried for permission to overwrite. Each opening for sequential_input accesses the current generation. Each opening for sequential_output creates a new generation. The new generation has an expiration date of December 31, 1977. The maximum number of devices that can be used is three.

```
tape_ansi_ 042381 042382 042383 -nm THESIS -rg
```

A file named THESIS is to be read. The I/O switch can only be open for sequential_input. The volume set consists of at least three volumes, and they are mounted with write rings. Only one device can be used.

```
tape_ansi_ 042381 -nm FF -nb 3 -ext -dv 4 -ret all
```

A file named FF at position 3 in the file set is to be extended. Each opening for sequential_input accesses the current version. Each opening for sequential_output produces a new version. A maximum of four devices can be used, and resources are retained after detachment.

```
tape_ansi_ 042381 -vol -COS -com in_slot_000034 -nb 6 -mod -fc
```

The file at position 6 in the file set is to be modified, irrespective of its expiration date. Each opening for sequential_input accesses the current version. Each opening for sequential_output produces a new version. The second volume of the volume set has volume identifier -COS, and can be found in slot 000034.

Attach Control Arguments

The following is a complete list of all valid attach control arguments in both long and short forms:

-block b	-bk b	18 ≤ b ≤ 8192
-create	-cr	
-density n	-den n	n = 800 ¦ 1600
-device n	-dv n	1 ≤ n ≤ 63
-expires date	-exp date	valid date
-extend	-ext	
-force	-fc	
-format f	-fmt f	f = fb ¦ f ¦ db ¦ d ¦ sb ¦ s ¦ u
-generate	-gen	
-mode XX	-md XX	XX = ascii ¦ ebcdic ¦ binary
-modify	-mod	
-name XX	-nm XX	XX ≤ 17 characters
-number n	-nb n	1 ≤ n ≤ 9999
-record r	-rec r	1 ≤ r ≤ 1044480
-replace	-rpl	XX ≤ 17 characters
-retain XX	-ret XX	XX = all ¦ none
-ring	-rg	

The following is a list of positional keywords:

-comment XX	-com XX	XX ≤ 64 characters
-volume vni	-vol vni	vni ≤ 6 characters

Name: tape_ibm_

The tape_ibm_ I/O module implements the processing of magnetic tape files in accordance with the standards established by the following IBM publications: *OS Data Management Services Guide*, Release 21.7, GC26-3746-2; *IBM System 360 Disk Operating System Data Management Concepts*, GC24-3427-8; and *OS Tape Labels*, Release 21, GC28-6680-4. These documents are collectively referred to below as the Standard.

Entries in the module are not called directly by users; rather, the module is accessed through the I/O system. See "Multics Input/Output System" in Section IV of the MPM Reference Guide for a general description of the I/O system.

Definition of Terms

record — related information treated as a unit of information.

block — a collection of characters written or read as a unit. A block may contain one or more complete records, or it may contain parts of one or more records. A part of a record is a record segment. A block does not contain multiple segments of the same record.

file — a collection of information consisting of records pertaining to a single subject. A file may be recorded on all or part of a volume, or on more than one volume.

volume — reel of magnetic tape. A volume may contain one or more complete files, or it may contain sections of one or more files. A volume does not contain multiple sections of the same file.

file set — a collection of one or more related files, recorded consecutively on a volume set.

volume set — a collection of one or more volumes on which one and only one file set is recorded.

Attach Description

The attach description has the following form:

```
tape_ibm_ vn1 vn2 ... vnn -control_args-
```

where:

1. vn*i* — is a volume specification. A maximum of 64 volumes may be specified. In the simplest (and typical) case, a volume specification is a volume name, which must be six characters or less in length. If a volume name is less than six characters and entirely numeric, it is

padded on the left with 0's. If a volume name is less than six characters and not entirely numeric, it is padded on the right with blanks. Occasionally, keywords must be used with the volume name. For a discussion of volume name and keywords see "Volume Specification" below.

vn1 vn2 ... vnn

comprise what is known as the volume sequence list. The volume sequence list may be divided into two parts. The first part, vn1 ... vni, consists of those volumes that are actually members of the volume set, listed in the order in which they became members. The entire volume set membership need not be specified in the attach description; however, the first (or only) volume set member must be specified, because its volume name is used to identify the file set. If the entire membership is specified, the sequence list may contain a second part, vni+1 ... vnn, consisting of potential members of the volume set, listed in the order in which they may become members. These volumes are known as volume set candidates. (See "Volume Switching" below.)

2. control_args

is a sequence of one or more attach control arguments. A control argument may appear only once.

-create, -cr

specifies that a new file is to be created. (See "Creating a File" below.)

-name XX, -nm XX

specifies the file identifier of the file, where XX is from 1 to 17 characters. (See "Creating a File" below.)

-number n, -nb n

specifies the file sequence number, the position of the file within the file set, where n is an integer in the range $1 \leq n \leq 9999$. (See "Creating a File" below.)

-replace XX,
-rpl XX

specifies the file identifier of the file to be replaced, where XX must be from 1 to 17 characters. If no file with file identifier XX exists, an error is indicated. (See "Creating a File" below.)

-format f, -fmt f

specifies the record format, where f is a format code. (See "Creating a File" below for a list of format codes.)

-record r, -rec r

specifies the record length in characters, where the value of r is dependent upon the choice of record format. (See "Creating a File" below.)

-block b, -bk b

specifies the block length in characters, where the value of b is dependent upon the value of r specified in the -record r control argument. (See "Creating a File" below.)

-dos

specifies that a file was produced by, or is destined for, a DOS installation. (See "DOS Files" below.)

-extend, -ext

specifies extension of an existing file. (See "Extending a File" below.)

-modify, -mod

specifies modification of an existing file. (See "Modifying a File" below.)

-mode XX, -md XX specifies the encoding mode used to record the file data, where XX is the string ebcdic or ascii, The default is ebcdic. (See "Encoding Mode" below.)

-expires date,
-exp date specifies the expiration date of the file to be created or generated where date must be of a form acceptable to the convert_date_to_binary_ subroutine. (See "File Expiration" below.)

-force, -fc specifies that the expiration date of the file being overwritten is to be ignored. (See "File Expiration" below.)

-device n, -dv n specifies the maximum number of tape drives that can be used during an attachment, where n is an integer in the range 1 ≤ n ≤63. (See "Multiple Devices" below.)

-density n,
-den n specifies the density at which the file set is recorded, where n can be either 1600 or 800 bits per inch. (See "File Set Density" below.)

-retain XX,
-ret XX specifies retention of resources across attachments, where XX specifies the detach-time resource disposition. (See "Resource Disposition" below.)

-ring, -rg specifies that the volume set be mounted with write rings. (See "Write Rings and Write Protection" below.)

-no_labels, -nlb specifies that unlabeled tapes are to be processed. (See "Unlabeled Tapes" below.)

The following sections define each control argument in the contexts in which it can be used. For a complete list of the attach control arguments see "Attach Control Arguments" below.

Creating a File

When a file is created, an entirely new entity is added to the file set. There are two modes of creation: append and replace. In append mode, the new file is added to the file set immediately following the last (or only) file in the set. The process of appending does not alter the previous contents of the file set. In replace mode, the new file is added by replacing (overwriting) a particular previously existing file. The replacement process logically truncates the file set at the point of replacement, destroying all files (if any) that follow consecutively from that point.

The -create and -name XX control arguments are required to create a file, where XX is the file identifier. Except when creating a file, XX must be 17 characters or less. When creating a file, XX must be eight characters or less; the first character must be an uppercase letter or national character (@, #, or $) and the remaining characters must be uppercase letters, national characters, or the digits 0 to 9. No two files in a file set can have the same file identifier. If the act of creation would cause a duplication, an error is indicated.

If no file having file identifier XX exists in the file set, the new file is appended to the file set; otherwise, the new file replaces the old file of the same name.

If the user wishes to explicitly specify creation by replacement, the particular file to be replaced must be identified. Associated with every file is a name (file identifier) and a number (file sequence number). Either is sufficient to uniquely identify a particular file in the file set. The -number n and -replace XX control arguments either separately or in conjunction, are used to specify the file to be replaced. If used together, they must both identify the same file; otherwise, an error is indicated.

When the -number n control argument is specified, if n is less than or equal to the sequence number of the last file in the file set, the created file replaces the file having sequence number n. If n is one greater than the sequence number of the last file in the file set, the created file is appended to the file set. If n is any other value, an error is indicated. When creating the first file of an entirely new file set, the -number 1 control argument must be explicitly specified. (See "Volume Initialization" below.)

The -format f, -record r and -block b control arguments are used to specify the internal structure of the file to be created. They are collectively known as structure attribute control arguments. When the -format f control argument is used, f must be one of the following format codes, chosen according to the nature of the data to be recorded. (For a detailed description of the various record formats, see "Record Formats" below.)

fb for fixed-length records. Used when every record has the same length, not in excess of 8192 characters.

vb for variable-length records. Used when records are of varying lengths, the longest not in excess of 8184 characters.

vbs for spanned records. Used when the record length is fixed and in excess of 8192 characters, or variable and in excess of 8184 characters. In either case, the record length cannot exceed 1,044,480 characters. (See "DOS Files" below.)

f for fixed-length records, unblocked.

v for variable-length records, unblocked.

vs for spanned records, unblocked. (See "DOS Files" below.)

NOTE: BECAUSE OF PADDING REQUIREMENTS RECORDS RECORDED USING VS FORMAT MAY BE IRREVERSIBLY MODIFIED. (See "Padding" below.)

Unblocked means that each block contains only one record (f, v) or record segment (vs). Because of their relative inefficiency, the use of unblocked formats in general is discouraged. Blocked means that each block contains as many records (fb, vb) or record segments (vbs) as possible. The actual number of records/block is either fixed (fb), depending upon the block length and record length, or variable (vb, vbs), depending upon the block length, record length, and actual records.

u for undefined records. U format records are undefined in format. Each block is treated as a single record, and a block may contain a maximum of 8192 characters.

When the -record $\underline{r}$ control argument is used, the value of $\underline{r}$ is dependent upon the choice of record format. In the following list, amrl is the actual or maximum record length.

$\underline{f}$ = fb ¦ f:	$\underline{r}$ = amrl
$\underline{f}$ = vb ¦ v:	amrl + 4 $\leq \underline{r} \leq$ 8188
$\underline{f}$ = vbs ¦ vs:	amrl $\leq \underline{r} \leq$ 1044480
$\underline{f}$ = u:	$\underline{r}$ is undefined (the -record control argument should not be used.)

When the -block $\underline{b}$ control argument is used, the value of $\underline{b}$ is dependent upon the value of $\underline{r}$. When the block length is not constrained to a particular value, the largest possible block length should be used.

$\underline{f}$ = fb:	$\underline{b}$ must satisfy mod ($\underline{b}$,$\underline{r}$) = 0
$\underline{f}$ = f:	$\underline{b} = \underline{r}$
$\underline{f}$ = vb:	$\underline{b} \geq \underline{r}$ + 4
$\underline{f}$ = v:	$\underline{b} = \underline{r}$ + 4
$\underline{f}$ = vbs ¦ vs:	20 $\leq \underline{b} \leq$ 8192
$\underline{f}$ = u:	amrl $\leq \underline{b} \leq$ 8192

In every case, $\underline{b}$ must be an integer in the range 20 $\leq \underline{b} \leq$ 8192, and, when the I/O switch is opened for sequential_output, must satisfy mod ($\underline{b}$,4) = 0.

Because the structure attribute control arguments are interdependent, care must be taken to ensure that specified values are consistent.

Padding

Because Multics is implemented on word-oriented hardware, records recorded in any format are subject to block and/or record padding. On output, the hardware requires that the number of characters in a block be evenly divisible by 4; i.e., only words can be written. The I/O module therefore requires that mod ($\underline{b}$,4) = 0, and pads a record, if necessary, to meet this requirement. The following rules govern padding on output:

$\underline{f}$ = fb: if iobl (the I/O buffer length in an iox_$write_record call; i.e., the number of characters to be written) is less than $\underline{r}$, the record is padded on the right with blanks to length $\underline{r}$. The last (or only) record of the file may be padded on the right with $\underline{n}$ blanks, where 0 $\leq \underline{n} \leq$ 19 is sufficient to satisfy $\underline{b} \geq$ 20, and mod ($\underline{b}$,4) = 0.

$\underline{f}$ = f: if iobl is less than $\underline{r}$, the record is padded on the right with blanks to length $\underline{r}$. Because the specified value of $\underline{b}$ must satisfy $\underline{b} \geq$ 20, mod ($\underline{b}$,4) = 0, and $\underline{r} = \underline{b}$, there are no other padding possibilities.

f = vb: the last (or only) record in every block is padded on the right with n blanks, where $0 \leq n \leq 12$ is sufficient to satisfy $b \geq 20$, and mod $(b,4) = 0$. Because the number of records in a block is variable, it is difficult to determine which records of a file are padded, if any.

f = v: every record is padded on the right with n blanks, where $0 \leq n \leq 12$ is sufficient to satisfy $b \geq 20$, and mod $(b,4) = 0$.

f = vbs: the last (or only) record of the file is padded on the right with n blanks, where $0 \leq n \leq 12$ is sufficient to satisfy $b \geq 20$, and mod $(b,4) = 0$.

f = vs: every record or record segment is padded on the right with n blanks, where $0 \leq n \leq 12$ is sufficient to satisfy $b \geq 20$, and mod $(b,4) = 0$.

NOTE: THIS REQUIREMENT CAN RESULT IN AN INDETERMINATE NUMBER OF BLANKS BEING INSERTED INTO A RECORD AT ONE OR MORE INDETERMINATE POSITIONS.

f = u: every record is padded on the right with n blanks, where $0 \leq n \leq 12$ is sufficient to satisfy $b \geq 20$, and mod $(b,4) = 0$.

Reading a File

The attach description needed to read a file is less complex than the description used to create it. When a file is initially created by the I/O module, the structure attributes specified in the attach description are recorded in the file's header and trailer labels. These labels, which precede and follow each file section, also contain the file name, sequence number, block count, etc. Files created by OS installations also record the structure attributes in the file labels. (See "DOS Files" below.) When a file is subsequently read, all this information is extracted from the labels. Therefore, the attach description need only identify the file to be read; no other control arguments are necessary.

The file can be identified using the -name XX control argument, the -number *n* control argument, or both in combination. If the -name XX control argument is used, a file with the specified file identifier must exist in the file set; otherwise, an error is indicated. If the -number *n* control argument is used, a file with the specified file sequence number must exist in the file set; otherwise, an error is indicated. If the -name XX and -number *n* control arguments are used together, they must both refer to the same file; otherwise, an error is indicated.

DOS Files

Files created by DOS installations differ from OS files in one major respect--DOS does not record HDR2 labels, which contain the structure attributes. It is therefore necessary to specify all of the structure attributes whenever a file created by a DOS installation is to be processed.

It is further necessary to distinguish between OS and DOS files recorded in VBS or VS format. The segment descriptor word (SDW) of a zero-length DOS spanned record has a high-order null record segment bit set, while a zero-length OS spanned record does not. (See "V(B)S Format" below, for an explanation of the SDW.)

The -dos control argument must be used when writing a VBS or VS file destined for a DOS installation, or when reading a VBS or VS file written by a DOS installation. In the interest of clarity, however, it is recommended that the control argument always be specified when DOS files are processed, regardless of record format.

Output Operations on Existing Files

There are two output operations that can be performed on an already existing file: extension, and modification. As their functions are significantly different, they are described separately below. They do, however, share a common characteristic. Like the replace mode of creation, an output operation on an existing file logically truncates the file set at the point of operation, destroying all files (if any) that follow consecutively from that point. Because the block length is constrained to mod(b,4) = 0 for output operations, a file whose block length does not satisfy this criterion cannot be extended or modified.

Extending a File

It is often necessary to add records to a file without in any way altering the previous contents of the file. This process is known as extension.

Because all the information regarding structure, length, etc., can be obtained from the file labels, the attach description need only specify that an extend operation is to be performed on a particular file. (See "DOS Files" above.) If the file to be extended does not exist, an error is indicated. New data records are appended at the end of the file; the previous contents of the file remain unchanged.

The file to be extended is identified using the -name XX control argument, the -number n control argument, or both in combination. The same rules apply as for reading a file. (See "Reading a File" above.)

The user may specify any or all of the structure attribute control arguments when extending a file. The specified control arguments are compared with their recorded counterparts; if a discrepancy is found, an error is indicated.

Modifying a File

It is occasionally necessary to replace the entire contents of a file, while retaining the structure of the file itself. This process is known as modification.

Because all necessary information can be obtained from the file labels, the attach description need only specify that a modify operation is to be performed on a particular file. (See "DOS Files" above.) If a file to be modified does not exist, an error is indicated. The entire contents of the file are replaced by the new data records.

The file to be modified is identified using the -name XX control argument, the -number n control argument, or both in combination. The same rules apply as for reading a file. (See "Reading a File" above.)

If any or all of the structure attribute control arguments are specified, they must match their recorded counterparts; otherwise, an error is indicated.

Encoding Mode

The I/O module makes provision for two data encoding modes: EBCDIC, and ASCII. The default data encoding mode is EBCDIC. File labels are always recorded using the EBCDIC character set.

When a file is created, the -mode XX can be used to explicitly specify the encoding mode, where XX is the string ascii or ebcdic.

If XX is the string ascii, the octal values of the characters to be recorded must be in the range 000 ≤ octal_value ≤ 377; otherwise, an unrecoverable I/O error occurs. If XX is the string ebcdic, the octal values of the characters to be recorded must be in the range 000 ≤ octal_value ≤ 177. (See the ascii_to_ebcdic_ subroutine in the MPM Subsystem Writers' Guide for the specific ASCII to EBCDIC mapping used by the I/O module.)

Because the data encoding mode is not recorded in the file labels, the -mode ascii control argument must always be specified when subsequently processing an ASCII file.

File Expiration

Associated with every file is a file expiration date, recorded in the file labels. If a file consists of more than one file section, the same date is recorded in the labels of every section. A file is regarded as "expired" on a day whose date is later than or equal to the expiration date. Only when this condition is satisfied can the file (and by implication, the remainder of the file set) be overwritten. Extension, modification, and the replace mode of creation are all considered to be overwrite operations.

The expiration date is recorded in Julian form; i.e., yyddd, where yy are the last two digits of the year, and ddd is the day of the year expressed as an integer in the range 1 ≤ ddd ≤ 366. A special case of the Julian date form is the value "00000", which means always expired.

The expiration date is set only when a file is created. Unless a specific date is provided, the default value "00000" is used. The -expires date control argument is used to specify an expiration date where date must be of a form acceptable to the convert_date_to_binary_ subroutine. If the I/O module is invoked through the iox_$attach_ioname entry point or the iox_$attach_iocb entry point, date must be a contiguous string, with no embedded spaces; if invoked through the io_call command, date may be quoted and contain embedded spaces. Julian form, including "00000", is unacceptable. Because overwriting a file logically truncates the file set at the point of overwriting, the expiration date of a file must be earlier than or equal to the expiration date of the previous file (if any); otherwise, an error is indicated.

If an attempt is made to overwrite an unexpired file, the user is queried for explicit permission. (See "Queries" below). The -force control argument unconditionally grants permission to overwrite a file without querying the user, regardless of "unexpired" status.

Volume Specification

The volume name is a six-character identifier physically written on, or affixed to, the volume's reel or container. The volume identifier is a six-character identifier magnetically recorded in the first block of the volume, the VOL1 label. This implementation of the I/O module assumes the volume name and volume identifier to be identical. If this is not the case, volume identifiers must be used in place of volume names.

If a volume name begins with a hyphen (-), the -volume keyword must precede the volume name. Even if the volume name does not begin with a hyphen , it may still be preceded by the -volume keyword. The volume specification has the following form:

```
     -volume vni
```

If the user attempts to specify a volume name beginning with a hyphen without specifying the -volume keyword, an error is indicated.

The slot identifier is a six-character string used to identify a volume on a per-installation basis. This implementation of the I/O module assumes the volume name and slot identifier to be identical. If this is not the case, the operator must be provided with the slot identifier of the volume. The volume specification for such a volume must be in the following form:

```
     vni -comment XX
  or
     -volume vni -comment XX
```

where the -comment XX keyword and text specify that a given message is to be displayed on the operator's console whenever volume vni is mounted. The message may relate to any subject, not only the slot identifier. XX may be from 1 to 64 characters. If the I/O module is invoked through the iox_$attach_iocb entry point or the iox_$attach_ioname entry point, XX must be a contiguous string, with no embedded spaces; if invoked through the io_call command (described in the MPM Commands), XX may be quoted and contain embedded spaces.

Volume Switching

The Standard defines four types of file set configurations:

single-volume file	a single file residing on a single volume
multivolume file	a single file residing on multiple volumes
multifile volume	multiple files residing on a single volume
multifile multivolume	multiple files residing on multiple volumes

The I/O module maintains a volume sequence list on a per-file-set basis, for the life of a process. A minimal volume sequence list contains only one volume, the first (or only) volume set member. If the file set is a multivolume configuration, the sequence list may contain one or more of the additional volume set members, following the mandatory first volume. If the sequence list contains the entire volume set membership (which may be only one volume), it may then contain one or more volume set candidates. Volume set candidates can become volume set members only as the result of an output operation. When an output operation causes the amount of data in the file set to exceed the capacity of the current volume set membership, the first available volume set candidate becomes a volume set member.

When the first attachment to any file in a file set is made, the volume sequence list for the file set is initialized from the attach description. At detach time, the I/O module empirically determines that one or more volumes are volume set members, by virtue of having used them in the course of processing the attached file. The remaining volumes in the sequence list, if any, are considered to be candidates. In subsequent attachments to any file in the file set, the order of volumes specified in the attach description is compared with the sequence list. For those volumes that the I/O module knows to be volume set members, the orders must match; otherwise, an error is indicated. Those volumes in the sequence list that the I/O module considers to be candidates are replaced by attach description specifications, if the orders differ. If the attach description contains more volumes than the sequence list, the additional volumes are appended to the list. This implementation maintains and validates the volume set membership on a per-process basis, and maintains a list of volume set candidates that is alterable on a per-attach basis.

Once a volume sequence list exists, subsequent attachments to files in the file set do not require repeated specification of any but the first (or only) volume, which is used to identify the file set. If the I/O module detects physical end of tape in the course of an output operation, it prepares to switch to the next volume in the volume set. An attempt is made to obtain the volume name from the sequence list, either from the sublist of members, or the sublist of candidates. If the list of volume set members is exhausted, and the list of candidates is either empty or exhausted, the user is queried for permission to terminate processing. If the reply is negative, the I/O module queries for the volume name of the next volume, which becomes a volume set member and is appended to the volume sequence list. If a volume name is obtained by either method, it is recorded in a system-defined file label field at the end of the current volume, volume switching occurs, and processing of the file continues.

If the I/O module reaches end-of-file section (but not of file) in the course of an input operation, it first attempts to obtain the next volume name from the volume sequence list. No distinction is made between the member and candidate sublists, because a volume that ends with a file section must be followed by the volume that contains the next section. If the sequence list is exhausted, the file section's labels are examined for a volume name, and if one is found, it is appended to the sequence list. Should the file labels provide no name, the user is queried as described above. If any of these three methods results in a volume name, volume switching occurs and processing of the file continues. This method of searching allows a specified switching sequence to override a sequence recorded in the file labels.

If the volume set is demounted at detach time, all volume set candidates are purged from the volume sequence list.

Multiple Devices

If a volume set consists of more than one volume, the -device n control argument can be used to control device assignment, where n specifies the maximum number of tape drives that can be used during this attachment. n is an integer in the range $1 \leq n \leq 63$. Drives are assigned only on a demand basis, and in no case does the number actually assigned exceed the device limit of the process. The default for an initial attachment to a file in a file set is n equals 1; the default for a subsequent attachment to that file or any other in the file set is n equals the previous value of n.

File Set Density

The I/O module makes provision for two densities: 1600, and 800 bpi (bits per inch). Every file in a file set must be recorded at the same density; otherwise, an error is indicated.

The -density n control argument used to explicitly specify the file set density where n specifies the density at which the file set is (to be) recorded. n may be either 1600 or 800 bpi. The file set density can only be changed in a subsequent attachment if the volume set was demounted by the previous attach.

In the absence of a -density n control argument, the file set density is determined as follows:

open for input: n = density of VOL1 label
open for output, creating new file set: n = 1600 bpi
open for output, old file set: n = density of VOL1 label

Opening

The opening modes supported are sequential_input and sequential_output. An I/O switch can be opened and closed any number of times in the course of a single attachment. Such a series of openings may be in either or both modes, in any valid order.

All openings during a single attachment are governed by the same attach description. The following control arguments, all of which pertain to output operations, are ignored when the switch is opened for sequential_input:

```
-create    -force
-expires   -modify
-extend    -replace
```

Resource Disposition

The I/O module utilizes two types of resources: devices (tape drives), and volumes. Once an I/O switch is attached, resources are assigned to the user's process on a demand basis. When the I/O switch is detached, the default resource disposition unassigns all devices and volumes.

If several attaches and detaches to a file set are made in a process, repeated assignment and unassignment of resources is undesirable. Although the processing time required to assign and unassign a device is small, all available devices can be assigned to other processes in the interval between one detach and the next attach. While volumes are not often "competed" for, mounting and demounting is both time-consuming and expensive.

The -retain XX control argument is used to specify retention of resources across attachments, where XX specifies the detach-time resource disposition. If XX is the string all, all devices and volumes remain assigned to the process. If XX is the string none, all devices and volumes are unassigned. This is the default retention.

The I/O module provides a further means for specifying or changing the resource disposition subsequent to attachment. (See retain_all and retain_none under "Control Operations" below.)

Write Rings and Write Protection

Before a volume can be written on, a write ring (an actual plastic ring) must be manually inserted into the reel. This can only be done before the volume is mounted on a device. When a volume is needed, the I/O module sends the operator a mount message that specifies if the volume is to be mounted with or without a ring.

If the attach description contains any of the output control arguments (-extend, -modify, or -create), volumes are mounted with rings; otherwise, they are mounted without rings. When a volume set mounted with rings is opened for sequential_input, hardware file protect is used to inhibit any spurious write operations. A volume set mounted without rings cannot be opened for sequential_output.

However, the following sequence of events is possible. An attach description contains none of the output control arguments, but does contain the "-retain all" control argument. The volume set is mounted without rings. After one or more (or no) openings for sequential_input, the I/O switch is detached. The volume set remains mounted because of the "-retain all" control argument. Subsequently, an attach is made whose description contains an output control argument, which requires that the volume set be mounted with rings. However, as rings can only be inserted in a demounted volume, the entire volume set must be demounted and then remounted.

This situation can be avoided by using the -ring (-rg) control argument to specify that the volume set be mounted with write rings. If no output control argument is specified in conjunction with -ring, the I/O switch cannot be opened for sequential_output.

When a volume set is mounted with write rings and the I/O switch is opened for sequential_input, the hardware file protect feature is used to safeguard the file set.

Queries

Under certain exceptional circumstances, the I/O module queries the user for information needed for processing to continue or instructions on how to proceed.

Querying is performed by the command_query_ subroutine. The user may intercept one or more types of query by establishing a handler for the command_question condition, which is signalled by the command_query_ subroutine. Alternately, the answer command (described in the MPM Commands) can be used to intercept all queries. The use of a predetermined "yes" answer to any query causes those actions to be performed that attempt to complete an I/O operation without human intervention.

In the following list of queries, status_code refers to command_question_info.status_code. See "Handling Unusual Occurrences" in the MPM Reference Guide for information regarding the command_question condition and the command_question_info structure.

status_code = error_table_$file_aborted

This can occur only when the I/O switch is open for sequential_output. The I/O module is unable to correctly write file header labels, trailer labels, or tapemarks. This type of error invalidates the structure of the entire file set. Valid file set structure can only be restored by deleting the defective file or file section from the file set.

The user is queried for permission to delete the defective file or file section. If the response is "yes", the I/O module attempts deletion. The attempt may or may not succeed; the user is informed if the attempt fails. If the response is "no", no action is taken. The user will probably be unable to subsequently process the file, or append files to the file set; however, this choice permits retrieval of the defective file with another I/O Module. In either case, the I/O switch is closed.

status_code = error_table_$unexpired_volume

This can occur only when the I/O switch is open for sequential_output. A volume must be either reinitialized or overwritten; however, the first file or file section on the volume is unexpired.

The user is queried for permission to initialize or overwrite the unexpired volume. If the response is "yes", the volume is initialized or overwritten and processing continues. If the response is "no", further processing cannot continue, and the I/O switch is closed.

status_code = error_table_$uninitialized_volume

This can occur only when the I/O switch is open for sequential_output. A volume requires reinitialization or initialization before it can be used to perform any I/O. The I/O module distinguishes among four causes by setting command_question_info.query_code as follows:

query_code = 1	the first block of the tape is unreadable. The tape is either defective, or recorded at an invalid density.
query_code = 2	the first block of the tape is not a valid IBM VOL1 label. The tape is not formatted as an IBM SL volume.
query_code = 3	the volume identifier recorded in the VOL1 label is incorrect. The volume identifier does not match the volume name.
query_code = 4	the density at which the volume is recorded is incorrect. The volume density does not match the specified density.

The user is queried for permission to reinitialize or initialize the volume, indicating the causative factor. If the response is "yes", the volume is reinitialized or initialized and processing continues. If the response is "no", further processing cannot continue, and the I/O switch is closed.

status_code = error_table_$unexpired_file

This can occur only when the I/O switch is open for sequential_output. A file which must be extended, modified, or replaced is unexpired.

The user is queried for permission to overwrite the unexpired file. If the response is "yes", processing continues. If the response is "no", further processing cannot continue, and the I/O switch is closed.

status_code = error_table_$no_next_volume

This can occur when reading a multivolume file, or when writing a file and reaching physical end of tape. The I/O module is unable to determine the name of the next volume in the volume set.

The user is queried for permission to terminate processing. If the response is "yes", no further processing is possible. If the I/O switch is open for sequential_output, the I/O switch is closed. If the response is "no", the user is queried for the volume name of the next volume. (See status_code = 0 below.)

status_code = 0

This occurs only when the response to the above query is "no". The user is requested to supply the name of the next volume. The response must be a volume name 6 characters or less in length, optionally followed by a mount message. Even if the volume name begins with a hyphen, it must <u>not</u> be preceded by the -volume control argument. If a mount message is to be specified, the response takes the following form:

```
volume_name -comment XX
```

where XX is the mount_message and need not be a contiguous string. See "Volume Specification" above. This is the only query that does not require a "yes" or "no" response. If a preset "yes" is supplied to all queries, this particular query never occurs.

<u>Structure Attribute Defaults</u>

When a file is created, the I/O module can supply a default value for any or all of the file structure attributes. The defaults used are as follows:

1. record format - the default is <u>f</u> = vb

2. block length - the default is <u>b</u> = 8192

3. record length <u>f</u> = u: undefined
   <u>f</u> = fb ¦ f: <u>r</u> = block length
   <u>f</u> = vb ¦ v: <u>r</u> = block length - 4
   <u>f</u> = vbs ¦ vs: <u>r</u> = 1044480

An injudicious combination of explicit specifications and defaults can result in an invalid attribute set. For example, if -record 12000 is specified, applying the defaults produces the following:

```
-format vb  -block 8192 -record 12000
```

This attribute set is invalid because, in vb format (see "Record Formats" below) the record length must be less than or equal to the block length minus 4.

Overriding Structure Attributes

Normally, the -format f, -block b, and -record r control arguments are not included in the attach description of an I/O switch that is opened for sequential_input; the structure attributes are extracted from the file labels. However, the I/O module permits the recorded structure attributes to be overridden by explicitly specified attach description control arguments. Because the apparent structure and characteristics of the file can be drastically altered, great care must be taken to ensure that attribute overrides do not produce unexpected and unwanted results.

If a file has the following recorded attributes:

```
-format fb -block 800 -record 80
```

an explicit specification of the -format f and -record 800 control arguments causes each block of ten 80-character records to be treated as a single 800-character record.

If a file has the following recorded attributes:

```
-format fb -block 800 -record 80
```

an explicit specification of the -format fb, -block 80, and -record 80 control arguments causes the last 720 characters of every block to be discarded. No error is indicated, because every block of the file contains at least one 80-character record.

Record Formats

Files are structured in one of four record formats: F(B), V(B), V(B)S, or U. When a file is created, its record format should be chosen in accordance with the nature of the data to be recorded. For example, data consisting of 80-character card images is most economically recorded in FB format, blocked fixed-length records. Data consisting of variable length text lines, such as PL/I source code produced by a text editor, is best recorded in VBS format, blocked spanned records, so that blanks are not inserted except after the last line.

With the exception of U format, files are either blocked or unblocked, blocked being the usual case. Each block of an unblocked file contains just one record, whereas each block of a blocked file can contain several records. Blocking can provide a significant savings of processing time, because several records are accessed with a single physical tape movement. Furthermore, as blocks are separated by distances of blank tape, blocking reduces the amount of tape needed to contain a file.

F(B) FORMAT

In F format, records are of fixed (and equal) length, and files have an integral number (n) of records per block. If the file is unblocked, n equals 1 and the record length (r) equals the block length (b). If the file is blocked, n > 1 and b equals (r * n) where n is known as the blocking factor.

For example, if r equals 800 and b equals 800, then the file is unblocked and each block contains just one record.

```
         +-----+  +-----+  +-----+  +-----+  +-----+  +-----+
data     | 800 |  | 800 |  | 800 |  | 800 |  | 800 |  | 800 |
         +-----+  +-----+  +-----+  +-----+  +-----+  +-----+

         +-----+  +-----+  +-----+  +-----+  +-----+  +-----+
block    | 800 |  | 800 |  | 800 |  | 800 |  | 800 |  | 800 |
         +-----+  +-----+  +-----+  +-----+  +-----+  +-----+
```

If r equals 800 and b equals 2400, then the file is blocked, the blocking factor is 3, and each block contains three records.

```
         +-----+  +-----+  +-----+  +-----+  +-----+  +-----+
data     | 800 |  | 800 |  | 800 |  | 800 |  | 800 |  | 800 |
         +-----+  +-----+  +-----+  +-----+  +-----+  +-----+

         +-------+-------+-------+  +-------+-------+-------+
block    |  800  |  800  |  800  |  |  800  |  800  |  800  |
         +-------+-------+-------+  +-------+-------+-------+
```

The Standard for F format records permits recording short blocks. A short block is a block that contains fewer than n records, when n is greater than 1. Although the I/O module can read this variant of F format, it writes a short block in only one case. The last block of a blocked file can contain fewer than n records if there are no more records to be written when the file is closed. Therefore, blocked F format files written by the I/O module are always in FBS (fixed blocked standard) format.

There are two special cases in which a datum is padded out to length r. The first case is that of iobl (the number of characters to be written) equals 0: a record of r blanks is written. When such a record is subsequently read, it is interpreted as a record of r blanks, and not as a zero-length record. The second case is that of 0 is less than iobl is less than r: the record is padded on the right with blanks to length r, and the padded record written. When such a record is read, the original characters plus the padding are returned. The case of iobl is greater than r is in error.

V(B) FORMAT

In V format, records and therefore blocks may vary in length. Each record is preceded by a four-character record descriptor word (RDW) that contains the actual record length in binary, including the length of the RDW itself. Each block is preceded by a four-character block descriptor word (BDW) that contains the actual block length in binary, including the length of the BDW itself.

V format files have an integral number of records per block, n. If the file is unblocked, b = r + 4; if blocked, b ≥ r + 4; For blocked records, the number of records per block varies indirectly with the size of the records.

If r equals 804, b equals 808, and the file is unblocked, records of up to 800 characters can be written, but each block can contain only one record.

```
         +----------+  +--------+  +-------------------+
data     |   375    |  |  280   |  |        800        |
         +----------+  +--------+  +-------------------+

         +----------+  +--------+  +-------------------+
         |3|3|      |  |2|2|    |  |8|8|               |
block    |8|8| 376  |  |8|8|280|  |0|0|      800      |
         |4|0|      |  |8|4|    |  |8|4|               |
         +-+-+------+  +-+-+----+  +-+-+---------------+
```

If r equals 804, b equals 808, and the file is blocked, records of up to 800 characters can be written. Each block can contain a maximum of 201 zero-length records (a record written as a 4-character RDW containing the binary value 4).

```
         +----------+  +--------+  +-------------------+
data     |   375    |  |  280   |  |        800        |
         +----------+  +--------+  +-------------------+

         +-----------------------+  +-------------------+
         |6|3|        |2|        |  |8|8|               |
block    |6|8|  376   |8|  280   |  |0|0|      800      |
         |8|0|        |4|        |  |8|4|               |
         +-+-+--------+-+--------+  +-+-+---------------+
```

V(B)S FORMAT

In V(B)S format, a single record is formatted as one or more record segments. A record segment contains either a complete record, the initial portion of a record, a medial portion of a record, or the final portion of a record. No two segments of the same record can be contained in the same block, but a block may contain the segments of several different records. The maximum record length is limited only by the maximum size of a storage system segment, currently 1,044,480 characters.

V(B)S format files have an integral number of record segments per block. If the file is unblocked, each block contains only one record segment; if blocked, the number of record segments per block is variable. In either case, r and b are independent of one another.

Each record segment begins with a four-character segment descriptor word (SDW). The SDW contains a four-character record segment length in binary, which includes the length of the SDW itself. (See "DOS Files" above.) The SDW also contains a one-character record segment code in binary, which indicates if the segment contains a complete record, or an initial, medial, or final portion. In the examples below, r equals 1000 and b equals 800.

```
          +--------+  +------------+  +------------------------------+
data      |  200   |  |    400     |  |             1000             |
          +--------+  +------------+  +------------------------------+

          +-+-+----+  +-+-+--------+  +-+-+-----------+  +-+-+-------+
          |2|2|    |  |4|4|        |  |8|8|           |  |2|2|       |
block     |0|0|200 |  |0|0|  400   |  |0|9|    792    |  |1|1|  208  |
          |8|4|    |  |8|4|        |  |0|6|           |  |6|2|       |
          +-+-+----+  +-+-+--------+  +-+-+-----------+  +-+-+-------+

          +-------+  +------------+  +------------------------------+
data      | 200   |  |    400     |  |             1000             |
          +-------+  +------------+  +------------------------------+

          +-------+  +------------+  +-----+  +----------+  +------+
          |2|     |  |4|          |  |1|   |  |7|        |  |2|    |
record    |0| 200 |  |0|   400    |  |8|184|  |9|  792   |  |8| 24 |
segment   |4|     |  |4|          |  |8|   |  |6|        |  | |    |
          +-------+  +------------+  +-----+  +----------+  +------+

          +-------------------------------+  +----------+  +------+
          |8|2|     |4|          |1|   |  |8|7|        |  |3|2|  |
block     |0|0| 200 |0|   400    |8|184|  |0|9|  792   |  |2|8|24|
          |0|4|     |4|          |8|   |  |0|6|        |  | | |  |
          +-------------------------------+  +----------+  +------+
```

U FORMAT

U format files contain records that do not conform to either F(B), V(B), or V(B)S format. A U format file is always unblocked. The record length is undefined, and the block length must equal or exceed the maximum record length. Blocks may vary in length. The special case of writing a record of less than 20 characters produces a block padded to length 20 with blanks.

```
          +-------+  +--------------+  +----+  +------------------+
data      |  60   |  |     127      |  | 16 |  |       156        |
          +-------+  +--------------+  +----+  +------------------+

          +-------+  +--------------+  +----+  +------------------+
block     |  60   |  |     128      |  | 20 |  |       156        |
          +-------+  +--------------+  +----+  +------------------+
```

Volume Initialization

The Standard requires that all volumes be initialized with VOL1 and dummy HDR1 labels before they are used for output. The I/O module provides a semiautomatic volume initialization mechanism that performs this operation as an integral part of the output function. It should be noted that, as stated above, a newly initialized volume contains a dummy HDR1 label, but not a dummy file. If a file is created on a newly initialized volume without an explicit specification of the -number 1 control argument, the I/O module attempts to append it to the file set, resulting in an error.

Conformance to Standard

With two exceptions, the I/O module conforms to the Standard: the I/O module cannot process block lengths in excess of 8192 characters; and the I/O module ignores the data set security field in the HDR1 label on input, and records it as 0 on output.

Label Processing

VOL1

The label is processed on input and output. The owner-name and address-code-field, character positions (CP) 42 to 51, is recorded using up to 10 characters of the user's Person_id, all lowercase letters translated to uppercase.

UVL1 - UVL8

These labels are not written on output and ignored on input.

HDR1/EOF1/EOV1

The labels are processed on input and output. The system-code-field, CP 61 to 73, is recorded as "MULTICS IBM ".

HDR2/EOF2/EOV2

The labels are processed on input and output. The 17-character job/job-step-identification-field, CP 18 to 34, is recorded as follows:

```
"MULTICS /" || Julian creation date || "   "
```

HDR3/EOF3/EOV3 - HDR8/EOF8/EOV8

These labels are not written on output and are ignored on input.

UHL1/UTL1 - UHL8/UTL8

These labels are not written on output and are ignored on input.

Error Processing

If an error occurs while reading, the I/O module makes 25 attempts to backspace and reread. If an error occurs while writing, the I/O module makes 10 attempts to backspace, erase, and rewrite. Should an error while reading or writing data prove to be unrecoverable, the I/O Module "locks" the file, and no further I/O is possible. If an unrecoverable error occurs while writing file labels or tapemarks, the user is queried as to preserving the defective file versus file set consistency. (See "Queries" above.) If an unrecoverable error occurs during certain phases of volume switching or label reading, the I/O switch may be closed. The overriding concern of the error recovery strategy is:

1. to maintain a consistent file set structure
2. to ensure the validity of data read or written

Close Operation

The I/O switch must be open.

Control Operation

The I/O module supports eight control operations.

hardware_status
status
volume_status
file_status
feov
close_rewind
retain_all
retain_none

In the descriptions below, info_ptr is the information pointer specified in an iox_$control call.

hardware_status OPERATION

This operation returns the 72-bit IOM status string generated by the last tape I/O operation. The I/O switch must be open. The substr argument

(IOM_bits, 3, 10) contains the major and minor status codes generated by the tape subsystem itself. (See MTS500 Magnetic Tape Subsystem, Order No. DB28 for an explanation of major and minor status.) The variable to which info_ptr points is declared as follows:

```
     declare IOM_bits bit(72) aligned;
```

status OPERATION

This operation returns a structure that contains an array of status codes, providing an interpretation of the IOM status string generated by the last tape I/O operation. These codes may be used in calls to the com_err_ subroutine, or may be converted to printable strings by calling the convert_status_code_ subroutine. (See the description of the convert_status_code_ subroutine in the MPM Subsystem Writers' Guide.) The I/O switch must be open. The structure to which info_ptr points, device_status.incl.pl1, is declared as follows:

```
     dcl  dstat_ptr           pointer;
     dcl 1 device_status      based (dstat_ptr),
         2 IOM_bits           bit(72) aligned,    /* IOM status */
         2 n_minor            fixed bin,          /* number of minor codes */
         2 major              fixed bin(35),      /* major status code */
         2 minor              (10) fixed bin(35); /* minor status codes */
```

volume_status OPERATION

This operation returns a structure that contains the status of the current volume. If the I/O switch is open, the current volume is the volume on which the file section currently being processed resides. If the switch has never been opened, the current volume is the first (or only) volume in the volume set. If the switch was opened, but is now closed, the current volume is that on which the last file section processed resides. If the switch was closed by the I/O module as the result of an error while writing file header labels, trailer labels, or tapemarks, the current volume is the last (or only) volume in the volume set. The structure to which info_ptr points, tape_volume_status.incl.pl1, is declared as follows:

```
     dcl  tvstat_ptr          pointer;
     dcl 1 tape_volume_status based (tvstat_ptr),
         2 volume_name        char(6),    /* volume name */
         2 volume_id          char(6),    /* from VOL1 label */
         2 volume_seq         fixed bin,  /* order in volume set */
         2 tape_drive         char(8),    /* tape drive name */
                                          /* "" if not mounted */
         2 read_errors        fixed bin,  /* read error count */
         2 write_errors       fixed bin;  /* write error count */
```

In the current implementation of the I/O module, read_errors and write_errors are always zero. Eventually, the resource control package (RCP) will supply these values.

file_status OPERATION

This operation returns a structure that contains the current status of the file specified in the attach description. If the I/O switch has never been opened, no information can be returned; this situation is indicated by file_status.state = 0. If the switch was opened, but is now closed, the current status of the file is its status just prior to closing. If the switch was closed by the I/O module as the result of an error while writing file header labels, trailer labels, or tapemarks, the entire file may have been deleted. In this case, the structure contains the current status of the previous file in the file set, if any. The structure to which info_ptr points, file_status.incl.pl1, is declared as follows:

```
dcl  tfstat_ptr         pointer;
dcl 1 tape_file_status  based (tfstat_ptr),
    2 state             fixed bin,       /* 0 - no information */
                                         /* 1 - not open */
                                         /* 2 - open, no events */
                                         /* 3 - open, event lock */
    2 event_code        fixed bin(35),   /* error_table_ code if
                                            state = 3 */
    2 file_id           char(17),        /* file identifier */
                                         /* "" if -no_labels */
    2 file_seq          fixed bin,       /* order in file set */
    2 cur_section       fixed bin,       /* current or last
                                            section processed */
    2 cur_volume        char(6),         /* volume name of volume
                                            on which cur_section
                                            resides */
    2 pad1              fixed bin,       /* not used */
    2 pad2              fixed bin,       /* not used */
    2 creation          char(5),         /* Julian creation date */
                                         /* "00000" if -no_labels */
    2 expiration        char(5),         /* Julian expiration date */
                                         /* "00000" if -no_labels */
    2 format_code       fixed bin,       /* 1 - U format */
                                         /* 2 - F(B) format */
                                         /* 3 - V(B) format */
                                         /* 4 - V(B)S format */
    2 blklen            fixed bin,       /* block length */
    2 reclen            fixed bin(21),   /* record length */
    2 blocked           bit(1),          /* "0"b - no ¦ "1"b - yes */
    2 mode              fixed bin,       /* 1 - ASCII */
                                         /* 2 - EBCDIC */
    2 cur_blkcnt        fixed bin(35);   /* current block count */
```

The "event" referenced in tape_file_status.state, above, is defined as an error or circumstance that prevents continued processing of a file. For example, parity alert while reading, reached end of information, no next volume available, etc.

feov OPERATION.

This operation forces end of volume when writing a file. The switch must be open for sequential output. The operation is equivalent to detection of the end of tape reflective strip. The info_ptr should be a null pointer.

close_rewind OPERATION

This operation specifies that the current volume is to be rewound when the I/O switch is next closed. info_ptr should be a null pointer. The switch need not be open when the operation is issued. The operation effects only one close; subsequent closings require additional control calls.

retain_all OPERATION

This operation causes all devices and volumes to remain assigned at detach time. The I/O switch need not be open. The info_ptr should be a null pointer.

retain_none OPERATION

This operation causes all devices and volumes to be unassigned at detach time. The I/O switch need not be open. The info_ptr should be a null pointer.

Detach Operation

The I/O switch must be closed. If the I/O module determines that the membership of the volume set may have changed, the volume set members are listed before the set is demounted; volumes not listed are available for incorporation into other volume sets. If the volume set is unlabeled, only the name of the last volume processed is listed.

Modes Operation

This I/O module does not support the modes operation.

Position Operation

The I/O switch must be open for sequential_input. The I/O module does not support skipping backwards. In the course of a position operation, events or errors may occur that invoke the query mechanism. (See "Queries" above.) An unrecoverable error locks the file, and a severe error causes the I/O module to close the I/O switch.

Read Length Operation

The I/O switch must be open for sequential_input. In the course of a read_length operation, events or errors may occur that invoke the query mechanism. (See "Queries" above.) An unrecoverable error locks the file, and a severe error causes the I/O module to close the I/O switch.

Read Record Operation

The I/O switch must be open for sequential_input.

Write Record Operation

The I/O switch must be open for sequential_output.

Unlabeled Tapes

The I/O module supports basic processing of unlabeled tapes that are structured according to the OS Tape Labels document mentioned at the beginning of this description. DOS leading tape mark (LTM) unlabeled format tapes cannot be processed.

The -no_labels control argument specifies that unlabeled tapes are to be processed. The -no_labels control arguments and any of the following control arguments are mutually exclusive:

```
-name       -extend
-replace    -modify
-expires    -dos
-force
```

Volume switching is handled somewhat differently for unlabeled tapes. When the I/O module detects a tape mark in the course of an input operation, it determines whether or not any volumes remain in the volume sequence list. If another volume appears in the list, volume switching occurs and processing continues on the next volume. If the list is exhausted, the I/O module assumes that end of information has been reached. Detection of end of tape during an output operation is handled in much the same way as it would be for a labeled tape. (See the OS Tape Labels document for a complete description of unlabeled volume switching strategy.)

Examples

In the following examples, it must be emphasized that an attach description describes a potential operation, and in and of itself does nothing to the file. Depending upon the sequence of openings in various modes, one attach description can perform diverse functions.

```
tape_ibm_ 042381 -nm ARD21 -cr -fmt vbs -ret all
```

A file named ARD21 is to be appended to the file set whose first volume is 042381. If a file named ARD21 already exists in the file set, openings for sequential_input access that file, and openings for sequential_output replace the old file of that name. If no file named ARD21 already exists in the file set, openings for sequential_input prior to the first opening for sequential_output fail. The first opening for sequential_output creates the file by appending it to the end of the file set. Subsequent openings for sequential_input access the newly created file, and subsequent openings for sequential_output replace it. Spanned records are specified; the block length defaults to 8192, the record length to 1044480, and the encoding mode to EBCDIC. The density defaults to 1600 cpi, and the maximum number of devices defaults to 1. The volume set and devices are retained after detachment.

```
tape_ibm_ 042381 -nm fargo.pl1 -nb 2 -cr -force -fmt fb -bk 800 -rec 80
```

A file named fargo.pl1 is created at position 2 in the file set. If a file named fargo.pl1 already exists at position 2, openings for sequential_input prior to the first opening for sequential_output access that file. The first opening for sequential_output creates a new file, and subsequent openings for sequential_input access the new file. If no file named fargo.pl1 exists at position 2, openings for sequential_input prior to the first opening for sequential_output fail. If a file exists at position 2, it is replaced irrespective of its expiration date.

```
tape_ibm_ 042381 -nm zbx -rpl zbx -cr -md ascii -bk 6000 -exp 2weeks
```

A file named zbx is created, replacing a file of the same name. Openings for sequential_input prior to the first opening for sequential_output access the old file. Each opening for sequential_output creates a new file, and each subsequent opening for sequential_input access the most recently created file. The specified encoding mode is ascii. The record format defaults to VB, and the record length defaults to 5996 because the block length is specified as 6000. The file is protected from overwriting for a period of two weeks, so each opening for sequential_output subsequent to the initial opening for sequential_output causes the user to be queried for permission to overwrite.

```
tape_ibm_ 042381 042382 -nb 14 -nlb -cr -dv 3
```

A file is to be created at position 14 on volume 042381. If a file already exists at position 14, an opening for sequential_input prior to the first opening for sequential_output accesses that file; otherwise, an error is indicated. Openings for sequential_output create new files, and openings for sequential_input subsequent to the first opening for sequential_output access the most recent creation. The default record format is VBS, the default block length 8192, and the default record length 1044480. The volume set is unlabeled. If the file exceeds the capacity of volume 042381, it is continued on volume 042382. If it then exceeds the capacity of volume 042382, the user is queried for instructions. A maximum of three devices can be used.

```
tape_ibm_ 042381 042382 042383 -nm THESIS -ring
```

A file named THESIS is to be read. The I/O switch can only be open for sequential_input. The volume set consists of at least three volumes, and they are mounted with write rings. Only one device can be used.

```
tape_ibm_ 042381 -nm FF -nb 3 -ext -dv 4 -ret all
```

A file named FF at position 3 in the file set is to be extended. Each opening for sequential_input accesses the current version. Each opening for sequential_output produces a new version. A maximum of four devices can be used. Resources are retained after detachment.

```
tape_ibm_ 042381 -vol -COS -com in_slot_000034 -nb 6 -mod -fc
```

The file at position 6 in the file set is to be modified, irrespective of its expiration date. Each opening for sequential_input accesses the current version. Each opening for sequential_output produces a new version. The second volume of the volume set has volume identifier -COS, and can be found in slot 000034.

Attach Control Arguments

The following is a complete list of all valid attach control arguments in both long and short forms:

-block b	-bk b	20 ≤ b ≤ 8192 mod (b,4) = 0 if open for sequential_output
-create	-cr	
-density n	-den n	n = 1600 ¦ 800
-device n	-dv n	1 ≤ n ≤ 63
-dos		
-expires date	-exp date	valid date
-extend	-ext	
-force	-fc	
-format f	-fmt f	f = fb ¦ f ¦ vb ¦ v vbs ¦ vs ¦ u
-mode XX	-md XX	XX = ebcdic ¦ ascii
-modify	-mod	
-name XX	-nm XX	XX ≤ 17 characters ≤ 8 characters (restricted subset) with -create
-no_labels	-nlb	
-number n	-nb n	1 ≤ n ≤ 9999
-record r	-rec r	1 ≤ r ≤ 1044480
-replace XX	-rpl XX	XX ≤ 17 characters
-retain XX	-ret XX	XX = all ¦ none
-ring	-rg	

The following is a list of positional keywords:

-comment XX	-com XX	XX ≤ 64 characters
-volume vni	-vol vni	volume name ≤ 6 characters

Name: tty_

This I/O module supports I/O from/to devices that can be operated in a typewriter-like manner, e.g., the user's terminal.

Entry points in the module are not called directly by users; rather the module is accessed through the I/O system. See "Multics Input/Output System" in Section IV of the MPM Reference Guide for a general description of the I/O system.

Attach Description

The attach description has the form:

```
tty_ device
```

where device identifies the particular device. Normally the user is only interested in his own terminal, and this is attached when the process is initialized.

Opening

The opening modes supported are: stream_input, stream_output, and stream_input_output.

Editing

On both input and output, data is automatically edited as described in "Typing Conventions" in Section V of the MPM Reference Guide. To control the editing, use the modes operation. Details on the various modes are given below.

Buffering

In general, this I/O module reads input data into an intermediate buffer as the device makes it available. The operations get_line and get_chars get the data from the buffer later. Similarly, output data is stored in a buffer and then transmitted to the device. This allows the process to proceed without waiting for the device.

The amount of buffering is unpredictable. To discard pending I/O from the buffers, use the control operation with the order resetread, resetwrite, or abort.

Interrupted Operations

When an I/O operation, except detach, being performed on a switch attached by this I/O module is interrupted by a signal, other operations may be performed on the switch during the interruption. The effect, as seen by the user, is that the interrupted operation is completely performed before the interruption or is not started until after the interruption.

Control Operation

The following orders are supported when the I/O switch is open. Except as noted, the info_ptr should be null.

abort flushes the input and output buffers.

resetread flushes the input buffer.

resetwrite flushes the output buffer.

hangup disconnects the telephone line connection of the terminal, if possible.

listen sends a wakeup to the process if the line associated with this device identifier is dialed up.

info returns information about the device. The info_ptr should point to the following structure that is filled by the call:

```
dcl 1 info_structure    aligned,
    2 id                char(4) unaligned,
    2 baud_rate         fixed bin unaligned,
    2 reserved          bit(54) unaligned,
    2 type              fixed bin;
```

where:

1. id is the identifier of the specific device as told to Multics by the device when the device is initialized.

2. baud_rate is the baud rate at which the device is running.

3. reserved is space reserved for compatibility purposes.

4. type identifies the type of device:

1 device similar to IBM Model 1050
2 device similar to IBM Model 2741
3 device similar to Teletype Model 37
4 device similar to GE TermiNet 300
5 device similar to Adage, Inc. Advanced Remote Display Station (ARDS)

6 device similar to IBM Model 2741 with correspondence keyboard and 015 typeball

7 device similar to Teletype Models 33 or 35

8 device similar to Teletype Model 38

9 unused

10 unused

11 device similar to a Computer Devices Inc. (CDI) Model 1030 or Texas Instruments (TI) Model 725, or a device with an unrecognized answerback, or a device without an answerback (these devices are collectively termed "ASCII" devices)

read_status tells whether or not there is any type-ahead input waiting for a process to read. The info_ptr should point to the following structure that is filled in by the call:

```
dcl 1 info_structure         aligned,
    2 ev_chan                fixed bin(71),
    2 input_available        bit(1);
```

where:

1. ev_chan is the event channel used to signal the arrival of input.

2. input_available indicates whether input is available.
"0"b no input
"1"b input

quit_enable causes quit signal processing to be enabled for this device. (Quit signal processing is initially disabled.)

quit_disable causes quit signal 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 terminal to be temporarily disabled if it is physically possible for the terminal to do so; if it is not, the status code error_table_$action_not_performed is returned. (See "Note" below.)

printer_on causes the printer mechanism of the terminal to be reenabled. (See "Note" below.)

wru initiates the transmission of the device's answerback, if it is so equipped. This operation is allowed only for the process that originally attached the device (generally the initializer process). The answerback may subsequently be read by means of the get_chars input/output operation.

store_id stores the answerback identifier of the terminal for later use by the process. The info_ptr should point to a char(4) variable, which contains the identifier.

set_line_type sets the line type associated with the terminal to the value supplied. The info_ptr should point to a fixed bin variable containing the new line type. Line types can be:

1 7-bit ASCII using Bell 103A-type modem protocol
2 IBM Model 1050
3 IBM Model 2741, with or without auto EOT inhibit
4 ARDS-type protocol using Bell 202C6-type modem
5 Direct connect ASCII synchronous, no protocol
6 Direct connect ASCII synchronous, Model G-115 remote computer protocol
7 Dialup ASCII synchronous, Model G-115 remote computer protocol
8 GE TermiNet 1200 protocol using Bell 202C5/6-type modem

This operation is not permitted for a line that is dialed up.

set_type sets the device type associated with the channel to one of the types described under the info control operation. The info_ptr should point to a fixed bin variable, which contains the type.

start_xmit_hd causes the channel to remain in a transmitting state at the completion of the next block of output, rather than starting to accept input. The line will then remain in a transmitting state until the stop_xmit_hd control operation is issued. This operation is valid only for ARDS-like devices.

stop_xmit_hd causes the channel to resume accepting input from the terminal (after the completion of current output, if any). This operation is only valid for ARDS-like terminals and is used only to counteract a preceding start_xmit_hd operation.

Modes Operation

The modes operation is supported when the I/O switch is open. The recognized modes are listed below. Some modes have a complement indicated by the circumflex character (^) that turns the mode off (e.g., ^erkl). For these modes the complement is displayed along with the mode.

erkl, ^erkl performs "erase" and "kill" processing on input. (Default is on.)

can, ^can performs standard canonicalization. (Default is on.)

rawi, ^rawi reads the data specified from the device directly without any conversion or processing. (Default is off.)

rawo, ^rawo writes data to the device directly without any conversion or processing. (Default is off.)

tabs, ^tabs inserts tabs in output in place of spaces when appropriate. If tabs mode is off, any tab characters are mapped into the appropriate number of spaces. (Default is off for ASCII devices and Teletype Models 33, 35, and 38; default is on for all other terminal types.)

edited, ^edited
suppresses printing of characters for which there is no defined Multics equivalent on the device referenced. If edited mode is off, the 9-bit octal representation of the character is printed. (Default is off.)

esc, ^esc
enables escape processing (see "Typing Conventions" in Section III of the MPM Reference Guide) on all input read from the device. (Default is on.)

red, ^red
sends red and black shifts to the terminal. (Default is off for devices similar to GE TermiNet 300s, ASCII devices, and for all terminals without an answerback identifier; default is on for all other terminals.)

vertsp, ^vertsp
performs the vertical tab and form feed functions, and sends appropriate characters to the device. Otherwise, such characters are escaped. (The default is off for all devices.)

crecho, ^crecho
echoes a carriage return when a line feed is typed. (Default is off; this mode is only functional with devices similar to Teletype Models 33, 35, 37, and 38, GE TermiNet 300s, or ASCII devices.)

lfecho, ^lfecho
echoes and inserts a line feed in the user's input stream when a carriage return is typed. (Default is off; the same restriction applies as for crecho.)

tabecho, ^tabecho
echoes the appropriate number of spaces when a horizontal tab is typed. (Default is off; the same restriction applies as for crecho.)

echoplex, ^echoplex
echoes all characters typed on the terminal. (Default is off; the same restriction applies for crecho.)

fulldpx, ^fulldpx
allows the terminal to receive and transmit simultaneously. (Default is off; this mode is automatically turned on and off when echoplex is turned on and off.)

capo, ^capo
outputs all lowercase letters in uppercase. If edited mode is on, uppercase letters are printed normally; if edited mode is off and capo mode is on, uppercase letters are preceded by an escape (\) character. (Default is off.)

replay, ^replay
prints any partial input line that is interrupted by output at the conclusion of the output, and leaves the carriage in the same position as when the interruption occurred. (Default is off.)

polite, ^polite
does not print output sent to the terminal while the user is typing input until the carriage is at the left margin, unless the user allows 30 seconds to pass without typing a newline. (Default is off.)

lln

specifies the length in character positions of a terminal 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 devices similar to IBM 1050s, 125 for IBM 2741s, 88 for Teletype Model 37, 118 for GE TermiNet 300s, 80 for ARDS, 72 for Teletype Models 33 and 35, 132 for Teletype Model 38, and 79 for ASCII devices.)

pln

specifies the length in lines of a page. When an attempt is made to exceed this length, a warning message is printed. When the user types a form-feed character, the output continues with the next page. If the page length is zero, end-of-page checking is disabled. (Default page length is 50 for ARDS-like terminals, and zero for all other terminals.)

hndlquit, ^hndlquit

echoes a newline character and performs a resetread of the associated stream when a quit signal is detected. (Default is on.)

default

is a shorthand way of specifying erkl, can, ^rawi, ^rawo, and esc. The settings for other modes are not affected.

ctl_char, ^ctl_char

specifies that ASCII control characters that do not cause carriage or paper motion are to be accepted as input. If the mode is off, all such characters are discarded. (Default is off.)

Note

The status code error_table_$action_not_performed is returned by the printer_on and printer_off control operations if the control tables currently in effect indicate that this terminal cannot perform the printer_on or printer_off operation. A code of zero is returned otherwise.

Control Operations from Command Level

All control operations may be performed from the io_call command, as follows:

```
io_call control switch order_arg
```

where:

1. switch is the name of the I/O switch.

2. order_arg can be any one of the control orders described under "Control Operation" above. The store_id, set_type, or set_line_type orders must be specified as follows:

   ```
   store_id id
   set_type type
   set_line_type line_type
   ```

   where:

   id is a new answerback identifier.

   type is a new device type; device types are explained under the info control order above.

   line_type is a new line type; line types are explained in the set_line_type control order description above.

This page intentionally left blank.

Name: vfile_

This I/O module supports I/O from/to files in the storage system. All logical file types are supported.

Entry points in this module are not called directly by users; rather, the module is accessed through the I/O system. See "Multics Input/Output System" and "File Input/Output" in Section V of the MPM Reference Guide for a general description of the I/O system and a discussion of files, respectively.

Attach Description

The attach description has the following form:

```
vfile_ path -control_args-
```

where:

1. path — is the absolute or relative pathname of the file.

2. control_args — may be chosen from the following:

   -extend — specifies extension of the file if it already exists. This control argument is only meaningful with openings for output or input_output; otherwise, it is ignored.

   -share -wtime- — allows an indexed file to be open in more than one process at the same time, even though not all openings are for input. (See "Multiple Openings" below.) The wtime, if specified, is the maximum time in seconds that this process will wait to perform an operation on the file. A value of -1 means the process may wait indefinitely. If no wtime is given, a default value of 1 is used.

   -blocked -n- — specifies attachment to a blocked file. If a nonempty file exists, n is ignored and may be omitted. Otherwise, n is used to set the maximum record size (bytes).

   -no_trunc — indicates that a put_chars operation into the middle of an unstructured file (stream_input_output) is permitted, and no truncation is to occur in such cases. Also prevents the truncation of an existing file at open and in stream_input_output openings causes the next byte position to be initially set to beginning of file.

   -append — in input_output openings, this causes put_chars and write_record operations to add to end of file instead of truncating when the file position is not at end of file. Also the position is initially set to beginning of file, and an existing file is not truncated at open.

-header -n- for use with unstructured files, this control argument indicates that a header is expected in an existing file, or is to be created for a new file. If a header is specified, it contains an optional identifying number, which effectively permits user-defined file types. If n is given and the file exists, the file identifier must be equal to n; a new file takes the value of n, if given, as its identifier. The header is maintained and becomes invisible only with the explicit use of this control argument.

-old indicates that a new file is not to be created if an attempt is made to open a nonexisting file for output, input_output, or update.

-ssf restricts the file to a single segment. If specified, an attempt to open a multisegment file or to expand a file beyond a single segment is treated as an error. The file must not be indexed.

-dup_ok indicates that the creation of duplicate keys is to be permitted. The file must be indexed. (See "Duplicate Keys" below.)

The -extend, -append, and -no_trunc control arguments conflict; only one may be specified.

To form the attach description actually used in the attachment, the pathname is expanded to obtain an absolute pathname.

Opening and Access Requirements

All opening modes are supported. For an existing file, the mode must be compatible with the file type. (See "File Input/Output" in Section V of the MPM Reference Guide.) The mode must be compatible with any control arguments given in the attach description.

An existing file is not truncated at open if its safety switch is on and its bit count is nonzero.

If the opening is for input only, only read access is required on the file. In all other cases, rw access is required on the file.

Position Operation

An additional type of positioning is available with unstructured and blocked files that are open for input, input_output, or update. When the type argument of the iox_$position entry point is 2, this specifies direct positioning to the record or byte whose ordinal position (0, 1, 2, ...) is given. The zero position is just beyond the file header, if a header is present.

Write Operation

In blocked and sequential files open for update, this operation is supported. Its effect is to append a record to the file or replace the next record, depending on the next record position.

Rewrite Operation

If the file is a sequential file, the new record must be the same length as the replaced record. If not, the code returned is error_table_$long_record or error_table_$short_record.

In a blocked file, no record may be rewritten with a record whose length exceeds the maximum record length of the file. Attempting to do so causes the code, error_table_$long_record, to be returned.

Delete Operation

If the file is a sequential file, the record is logically deleted, but the space it occupies is not recovered.

Deletions are not supported in blocked files. If the user attempts to delete a record in a blocked file, the code, error_table_$no_operation is returned.

Modes Operation

This operation is not supported.

Control Operation

The following orders are supported by the vfile_ I/O module.

read_position
seek_head
set_wait_time
truncate
max_rec_len

add_key
delete_key
get_key
min_block_size
reassign_key
record_status
set_file_lock

The five orders in the first column are described below. The remaining orders, documented in the vfile_ I/O module in the MPM Subsystem Writers' Guide, implement various features of indexed files that require somewhat more knowledge of internal file structure than is expected of most users.

read_position

The read_position order is accepted when the I/O switch is open and attached to a nonindexed file. The operation returns the ordinal position (0, 1, 2, ...) of the next record (byte for unstructured files), and that of the end of file, relative to the file base. The file base is just beyond the header, if a header is present.

For this order, the info_ptr argument must point to a structure of the following form:

```
dcl 1 info             based (info_ptr),
      2 next_position  fixed(34),  /*output*/
      2 last_position  fixed(34);  /*output*/
```

seek_head

The seek_head order is accepted when the I/O switch is open for keyed_sequential_input or keyed_sequential_update. For this order the info_ptr argument must point to a structure of the following form:

```
dcl  1  info           based (info_ptr),
       2  relation_type fixed,
       2  n            fixed,
       2  search_key   char (0 refer (n));
```

The order locates the first record with a key whose head has the specified relation with the given search_key. The next record position and (for keyed_sequential_update) the current record position are set to the record. If no such record exists, the code error_table_$no_record is returned.

The head of a record's key is the first n characters of the key, the key being extended by blanks if it has fewer than n characters. The allowed values for info.relation_type are:

0	head = search_key
1	head >= search_key
2	head > search_key

set_wait_time

The set_wait_time order is accepted when the I/O switch is open and attached to an indexed file with the -share control argument. For this order the info_ptr argument must point to a structure of the following form:

```
dcl new_wait_time float based(info_ptr);
```

This order specifies a limit on the time that the user's process will wait to perform an order when the file is locked by another process. The interpretation of new_wait_time is the same as that described earlier for the wtime limit used with the -share control argument.

truncate

The truncate order is accepted when the I/O switch is attached to a nonindexed file open for input_output or update. The operation truncates the file at the next record (byte for unstructured files). If the next position is undefined, the code error_table_$no_record is returned.

No info structure is required for this order.

max_rec_len

The max_rec_len order is accepted when the I/O switch is open and attached to a blocked file. The operation returns the maximum record length (bytes) of the file. A new maximum length can be set by specifying a nonzero value for the second argument. In this case the file must empty and open for modification, or the code error_table_$no_operation is returned.

For this order the info_ptr argument must point to a structure of the following form:

```
dcl 1 info          based (info_ptr),
    2 old_max_recl  fixed(21),   /*output*/
    2 new_max_recl  fixed(21);  /*input*/
```

*

Duplicate Keys

By default, the vfile_ I/O module prevents the user from associating a single key with more than one record in the same indexed file. This restriction is removed when the -dup_ok control argument is used or if the file's statistics indicate that duplicate keys are already present.

Duplicate keys can be created via either the write_record operation or the add_key or record_status control orders. When duplications are permitted, the key for insertion is defined as the key of the current record, if it exists.

With this extension, the notion of an "index entry" becomes more basic than that of a single key in the index. An index entry is an association between a string of characters (key) and a number (record descriptor).

Index entries are ordered by key. Within multiple occurrences of the same key, the order is identical to the order in which the entries were created. A seek_key or seek_head operation locates the first instance of a set of duplicate keys. A write_record operation advances the file position beyond the last instance of the key for insertion, if the key already exists in the index.

The next record position is best thought of as corresponding to the next index entry. Operations that can advance the next record and position (i.e., read_record; rewrite_record; and position, with a type argument of 0) permit one to locate intermediate instances of duplicate keys.

Multiple Openings

It is possible to have or attempt to have multiple openings of the same file, that is, to have two or more open I/O switches attached to the same file. These switches might be in the same process or in different processes. With respect to the effects of multiple openings, the various opening modes can be divided into four classes (explained below). Multiple openings in which the opening modes are in more than one class are invalid, as are multiple openings within certain classes. The vfile_ I/O module prevents some cases of multiple opening. In these cases, error_table_$file_busy is returned by the open operation. In cases where an invalid multiple opening does occur, I/O operations will cause unpredictable errors in the processes involved, and the contents of the files may be damaged.

The classes of multiple openings are:

1. Openings for input without the -share control argument.
   Any number of openings in this class are allowed. The existence of an opening in this class never causes damage to the file. When this class of opening is attempted, the existence of all class 2 and 3 openings and some class 4 openings will be detected for structured files.

2. Openings for output or input_output without the -extend control argument.
   Only one opening is allowed. The existence of another opening is never detected when this class of opening is attempted. The file is simply replaced by an empty file of the appropriate type. If the file was already open with an opening of any class except class 1, the contents of the new file will probably be damaged.

3. Openings for update without the -share control argument and for output or input_output without the -share control argument and with the -extend control argument.
   Only one opening of this class is allowed. For structured files, multiple openings within the class are detected. An invalid multiple opening involving an opening of this class and other openings of class 4 may be detected. If not, the only effect is that the class 3 opening locks the file for the entire opening.

4. Openings with the -share control argument.
   Any number of openings of this type are allowed. When a process performs an update on the file, the file is locked. Other processes attempting an operation while the file is locked will wait up to the limit specified by wtime in the -share control argument or from the last set_wait_time order. If the operation is not carried out because of the wtime limit, the code error_table_$file_busy is returned.

   There are two codes that pertain only to class 4 openings: error_table_$asynch_deletion and error_table_$asynch_insertion. The first is returned when there is an attempt to reference a record located by the previous operation, but the record has been deleted in some other opening. The second is returned by write_record when a record with the key for insertion (defined by a seek_key operation) has already been inserted (by some other opening).

Interrupted Openings

If a process opens a file and terminates without closing the file, the file may be left in an intermediate state that prohibits normal I/O operations on the file. The exception is openings for input only. The details depend on the particular type of file as follows:

1. Unstructured file.
   In general, the bit count of the file's last segment will not be properly set. This condition is not detected at subsequent openings, and part of the file's contents may be overwritten or ignored.

2. Sequential file.
   In general, certain descriptors in the file and the bit count of the file's last segment will not be properly set. This condition is detected at a subsequent open, and either the file is automatically adjusted or (if the opening is input only) the code error_table_$file_busy is returned.

3. Blocked File.
In general, the file's bit count and record count will not be correct. This condition is detected at a subsequent open, and either the file is automatically adjusted or (if the opening is input only) the code error_table_$file_busy is returned.

4. Indexed file.
In general, the bit counts of the file's segments will not be properly set, and the file contents will be in a complex intermediate state (e.g., a record, but not its key in the index, will be deleted). This situation is detected at a subsequent open or at the beginning of the next operation, if the file is already open with the -share control argument. Unless the opening is for input only, the file is automatically adjusted; otherwise, the code error_table_$file_busy is returned.

   When an indexed file is adjusted, the interrupted operation (write_record, rewrite_record, delete_record, etc.), if any, is completed. For rewrite_record, however, the bytes of the record may be incorrect. (Everything else will be correct.) In this case, an error message is printed on the terminal. The user can rewrite or delete the record as required. The completion of an interrupted write operation may also produce an incorrect record, in which case the defective record and its key are automatically deleted from the file.

Any type of file may be properly adjusted with the vfile_adjust command (described in the MPM Commands), if an interrupted opening has occurred.

Inconsistent Files

The code error_table_$bad_file (terminal message: "File is not a structured file or is inconsistent") may be returned by operations on structured files. It means that an inconsistency has been detected in the file. Possible causes are:

1. The file is not a structured file of the required type;

2. A program accidentally modified some words in the file.

Obtaining File Information

The type and various statistics of any of the four vfile_ supported file structures may be obtained with the vfile_status command or vfile_status_ subroutine (described in the MPM Commands and Subroutines respectively).

INDEX

CUT ALONG

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

TITLE	SERIES 60 (LEVEL 68) MULTICS PROGRAMMERS' MANUAL SUBROUTINES	ORDER NO.	AG93, REV. 1
		DATED	MAY 1975

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

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

FROM: NAME ______________________ DATE ____________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

CUT ALONG LINE

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WELLESLEY HILLS,
MA 02181

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

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
60 WALNUT STREET
WELLESLEY HILLS, MA 02181

ATTENTION: PUBLICATIONS, MS 050

FOLD ALONG LINE

Honeywell

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

TITLE	MULTICS PROGRAMMERS' MANUAL – SUBROUTINES ADDENDUM A	ORDER NO.	AG93A, REV. 1
		DATED	SEPTEMBER 1975

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

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

FROM: NAME ______________________ DATE ______________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

CUT ALONG LINE

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WELLESLEY HILLS,
MA 02181

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

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
60 WALNUT STREET
WELLESLEY HILLS, MA 02181

ATTENTION: PUBLICATIONS, MS 050

FOLD ALONG LINE

Honeywell

CUT ALONG LIN.

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

TITLE: SERIES 60 (LEVEL 68)
MULTICS PROGRAMMERS' MANUAL -
SUBROUTINES ADDENDUM B

ORDER NO. AG93B, REV. 1

DATED MARCH 1976

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

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

FROM: NAME ____________________ DATE __________

TITLE ____________________

COMPANY ____________________

ADDRESS ____________________

____

CUT ALONG LINE

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

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

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

FOLD ALONG LINE

Honeywell

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS

Technical Publications Remarks Form

TITLE: SERIES 60 (LEVEL 68)
MULTICS PROGRAMMERS' MANUAL – SUBROUTINES
ADDENDUM C

ORDER NO. AG93C, REV. 1

DATED JULY 1976

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

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

FROM: NAME ________________________ DATE ____________

TITLE ________________________

COMPANY ________________________

ADDRESS ________________________

____

CUT ALONG LINE

FOLD ALONG LINE

FIRST CLASS
PERMIT NO. 39531
WALTHAM, MA
02154

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

Postage Will Be Paid By:

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTENTION: PUBLICATIONS, MS 486

FOLD ALONG LINE

Honeywell